Techniques and Tools for Supporting Maintenance of Node.js Programs

Martin Toldam Torp

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark
Techniques and Tools for Supporting Maintenance of Node.js Programs

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Martin Toldam Torp
August 31, 2021
Abstract

JavaScript has become one of the most used programming languages. Much of its popularity can be attributed to the Node.js JavaScript runtime, which is used extensively for writing back-end web applications. A key strength of Node.js is the ease with which third-party code is bundled in packages and managed using the Node Package Manager (npm). Since packages can be reused many times, the development of Node.js applications is cheap and fast. However, the heavy use of third-party packages has resulted in several maintenance-related challenges: (i) How do third-party package developers ensure that backward-incompatible changes are not introduced unintentionally? Failing to identify all breaking changes may cause failures in client applications. (ii) How do client application developers keep third-party dependencies up-to-date? Developers neglecting this task may miss out on bug fixes, security improvements, and new features. (iii) How do client and package developers keep their applications and packages free of security vulnerabilities?

In this thesis, we present program analysis-based techniques and tools that help address these maintenance-related challenges. We present a dynamic analysis that generates API models of Node.js modules, and then compares models across versions to detect breaking changes. For adapting client applications to breaking changes in dependency updates, we develop a static analysis that both identifies source locations affected by breaking changes and patches them to restore compatibility. To detect usages of third-party packages with known vulnerabilities in client applications, we propose a static call graph analysis and report only those vulnerabilities where the affected part of the package API is reachable in the call graph. Finally, we present a dynamic analysis for generating taint specifications of third-party packages, allowing a static taint analysis to find injection vulnerabilities in large real-world Node.js applications.
Resumé


Acknowledgments

First a foremost a big thanks to Anders for motivating me to pursue the PhD degree and for providing unconditional guidance and support throughout my studies. Even from my brief exposure to academia, I know that few advisors are as invested in the academic success and well-being of their PhD students. I also want to express an extra level of gratitude to Benjamin for being an outstanding research partner for the past couple of years, and for his invaluable feedback on this thesis. Few people are as dedicated and hard-working as Benjamin. Also a big thanks to Gianluca for his guidance during the first years of my PhD studies. Thanks to Michael and all of his PhD students for hosting me in Stuttgart and for making my stay a fun and pleasant experience despite all the corona restrictions. Finally, I’m grateful to all current and former PhDs and postdocs in the PL and LogSem groups at Aarhus University for all the entertaining coffee break discussions and for creating a friendly and welcoming work environment.

Martin Toldam Torp,
Aarhus, August 31, 2021.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Resume</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Contents</td>
<td>vii</td>
</tr>
</tbody>
</table>

## 1 Overview

### 1.1 Introduction

#### 1.1.1 Research Challenges

#### 1.1.2 Contributions

#### 1.1.3 Outline

### 1.2 JavaScript and Node.js

#### 1.2.1 Reflective Features

#### 1.2.2 Node.js Package Manager

#### 1.2.3 Semantic Versioning

### 1.3 Breaking Change Detection

#### 1.3.1 Existing Work

#### 1.3.2 Type Regression Testing

#### 1.3.3 Discussion

### 1.4 Breaking Change Adaptation

#### 1.4.1 Existing Work

#### 1.4.2 Breaking Change Adaptation for Node.js

#### 1.4.3 Discussion

### 1.5 Node.js Application Security

#### 1.5.1 Existing Work

#### 1.5.2 Security Scanning

#### 1.5.3 Taint Analysis Specifications
## CONTENTS

5.4 Discussion ................................................. 47

6 Conclusions ................................................. 51

II Publications ............................................... 55

7 Type Regression Testing to Detect Breaking Changes in Node.js Libraries 57
   7.1 Introduction ........................................... 58
   7.2 A Preliminary Experimental Study of Breaking Changes ........................................... 60
   7.3 Motivating Example ..................................... 62
   7.4 Overview ............................................... 63
   7.5 Public API Discovery .................................... 64
   7.6 Type Regression Testing ............................... 72
   7.7 Evaluation ............................................ 73
   7.8 Related Work .......................................... 80
   7.9 Conclusion ........................................... 82

8 Model-Based Testing of Breaking Changes in Node.js Libraries 83
   8.1 Introduction ........................................... 83
   8.2 Motivating Example ..................................... 85
   8.3 Overview ............................................... 87
   8.4 Phase I: Model Generation ............................ 89
   8.5 Phase II: Type Regression Testing .................. 93
   8.6 Evaluation ............................................ 95
   8.7 Related Work .......................................... 103
   8.8 Conclusion ........................................... 105

9 Detecting Locations in JavaScript Programs Affected by Breaking Library Changes 107
   9.1 Introduction ........................................... 107
   9.2 Motivating Example ..................................... 110
   9.3 Preliminary Study ...................................... 113
   9.4 The TAPIR Approach .................................... 116
   9.5 A Pattern Language for Describing API Access Points ........................................... 117
   9.6 A Static Analysis for Finding Uses of API Access Points in Client Code .................. 121
   9.7 Evaluation ............................................ 129
   9.8 Related Work .......................................... 135
   9.9 Conclusion ........................................... 136

10 Semantic Patches for Adaptation of JavaScript Programs to Evolving Libraries 139
   10.1 Introduction ........................................... 140
**CONTENTS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2 Motivating Example</td>
<td>141</td>
</tr>
<tr>
<td>10.3 Overview</td>
<td>143</td>
</tr>
<tr>
<td>10.4 A Semantic Patch Language</td>
<td>144</td>
</tr>
<tr>
<td>10.5 Interactive Phase</td>
<td>149</td>
</tr>
<tr>
<td>10.6 Evaluation</td>
<td>151</td>
</tr>
<tr>
<td>10.7 Related Work</td>
<td>160</td>
</tr>
<tr>
<td>10.8 Conclusion</td>
<td>161</td>
</tr>
<tr>
<td>10.9 Additional Examples</td>
<td>162</td>
</tr>
<tr>
<td><strong>11 Extracting Taint Specifications for JavaScript Libraries</strong></td>
<td>165</td>
</tr>
<tr>
<td>11.1 Introduction</td>
<td>166</td>
</tr>
<tr>
<td>11.2 Motivating Example</td>
<td>168</td>
</tr>
<tr>
<td>11.3 Taint Specifications for Modules</td>
<td>169</td>
</tr>
<tr>
<td>11.4 Inferring Taint Specifications via Dynamic Analysis</td>
<td>174</td>
</tr>
<tr>
<td>11.5 Using Taint Specifications</td>
<td>180</td>
</tr>
<tr>
<td>11.6 Evaluation</td>
<td>181</td>
</tr>
<tr>
<td>11.7 Discussion</td>
<td>185</td>
</tr>
<tr>
<td>11.8 Related Work</td>
<td>187</td>
</tr>
<tr>
<td>11.9 Conclusion</td>
<td>188</td>
</tr>
<tr>
<td><strong>12 Modular Call Graph Construction for Security Scanning of Node.js Applications</strong></td>
<td>189</td>
</tr>
<tr>
<td>12.1 Introduction</td>
<td>189</td>
</tr>
<tr>
<td>12.2 Motivating Example</td>
<td>191</td>
</tr>
<tr>
<td>12.3 Key Challenges</td>
<td>193</td>
</tr>
<tr>
<td>12.4 Module Summary Construction</td>
<td>195</td>
</tr>
<tr>
<td>12.5 Call Graph Construction</td>
<td>198</td>
</tr>
<tr>
<td>12.6 Security Scanning</td>
<td>206</td>
</tr>
<tr>
<td>12.7 Evaluation</td>
<td>208</td>
</tr>
<tr>
<td>12.8 Related work</td>
<td>212</td>
</tr>
<tr>
<td>12.9 Conclusion</td>
<td>214</td>
</tr>
</tbody>
</table>

**Bibliography** 215
Part I

Overview
Chapter 1

Introduction

Software is ubiquitous in the modern world. Anything that consumes electricity is likely to contain a microchip that executes software. That includes less-obvious items such as light bulbs, kitchen appliances, and electric toothbrushes. Software development is therefore big business today. The revenue of the software industry in 2021 is projected to be $580 billion and is estimated to grow to $772 billion by 2025.¹ That figure even excludes products that are only partially software such as cars, computers, airplanes, and so on, making the real figure much larger. It is therefore undeniably the case that any measures, which reduces the time it takes to produce software and improve its quality, are of huge economical and societal benefit. Since the primary cost of software development is human resources, developer productivity is tightly coupled with costs. Those terms are even used interchangeably in the literature.

Broadly speaking, we can categorize a software development task into two major phases: the production phase, and the maintenance phase. The production phase comprises all tasks leading up to the point where the product is first shipped to customers. That phase includes specification of requirements, design of database schemas, implementation work, testing, and so on. Any task that follows the initial release is considered maintenance. An obvious example of a maintenance task is fixing a bug reported by a customer, but we also consider adding features to meet new customer requirements as maintenance.

Surprisingly, multiple studies have found that the maintenance phase is often more costly than the production phase [13, 37]. The maintenance phase is typically much longer than the production phase: a product produced in a few months or years may require maintenance for decades. It is therefore imperative that research is devoted towards developing tools and processes that can aid developers with maintenance-specific issues. For example, fixing bugs as they are reported, integrating new versions of dependencies, dealing with newly discovered security problems, and many other tasks.

Software development practices are constantly evolving with new languages,

¹https://www.statista.com/outlook/tmo/software/worldwide
frameworks, development workflows, and so on. One trend that has been particularly strong in the past decade is using JavaScript as an all-purpose language. JavaScript originally emerged as a language for the browser: a position which it has dominated since the deprecation of Java applets and the discontinuation of Flash Player in most modern browsers. Recently, WebAssembly was added as an alternative language for the browser, but it has still only seen limited use, and there are many features, such as interacting with the HTML, that only work in JavaScript. With the emergence of the Node.js server-side JavaScript runtime, JavaScript has also steadily increased its market share as a back-end development language. In 2015 less than 0.1% of web applications used Node.js compared to around 1.3% in 2021, but the market share is much greater among enterprise level applications from large companies such as LinkedIn, Netflix, and Uber.²³

There are multiple reasons that may explain JavaScript's recent growth in popularity. The fact that JavaScript is used both on the front-end and the back-end of an application potentially makes it easier for developers, who were previously considered front-end only, to now also work on the back-end. The event-driven model of JavaScript is also particularly well-suited for web applications, and much simpler to use correctly than the explicit multi-threading of competitive languages, such as Java and C++. Some argue that the dynamic and very forgiving nature of JavaScript, where the developer does not have to deal with complicated compiler errors and where uncaught runtime exceptions do not halt an application completely, is appealing to new developers, who may feel more productive than in alternative statically typed languages.

Several languages have also emerged with JavaScript as their primary compile target. Developers who are not happy with the JavaScript language itself, may still develop JavaScript applications by choosing one of these other languages that suite their language preferences better. For example, developers who prefer a static typing discipline may write their applications in TypeScript, Elm, or Nim.

One feature that sets Node.js apart from server-side runtimes for other languages is the extremely large degree of code reusability it accommodates. Node.js libraries and frameworks are distributed as packages. The Node Package Manager (npm) allows developers to easily add and manage packages through the package.json configuration file. Furthermore, the npm registry contains more than 1.6 million third-party and open source packages, which developers can use for free. Node.js also allows multiple versions of the same package to co-exist in the dependency structure, preventing the infamous “dependency hell” complication that haunts other languages [65]. This advantage has spurred a tendency where developers rely on third-party packages for even the most miniscule set of features. An extreme example is the nice-try package, which is downloaded more than 16 million times per week, and is nothing but a

² https://w3techs.com/technologies/history_overview/web_server/ms/y
³ https://medium.com/quick-code/node-js-and-fortune-500-companies-fewer-efforts-more-rewards-282db19160c0
⁴ http://www.modulecounts.com/
⁵ https://www.npmjs.com/package/nice-try
try-catch block wrapped around a function call. A consequence of this tendency is that third-party code now comprises more than 90% of the average Node.js application’s code base \cite{77}. A package may in turn also depend on other packages, such that an application with just a few direct dependencies may have many transitive dependencies as illustrated by Figure 1.1.

1.1 Research Challenges

While this growth in the popularity of JavaScript and Node.js arguably has had positive benefits for developer productivity, it has also brought along a new set of challenges, many of which revolve around dependency management.

A package developer may want to update the package API with additional features or refactor it to improve the usability of the package. In this process, the package developer is responsible for ensuring that clients know exactly which changes must be made to the client code to adapt it to the new version of the package. Changes to the package code that require clients to modify their usage of the package are known as breaking changes. An undocumented breaking change can have dire consequences for the client since failing to address it may result in the client crashing or misbehaving. Client developers typically want to update their dependencies to benefit from the various improvements introduced in the new version (including security fixes).

The package developer has to be aware of security issues and bugs that are discovered after the package is released. These issues must be fixed, and the fixed version must be shipped to clients as fast as possible. Client developers are responsible for
ensuring that their applications’ security is not compromised by outdated vulnerable dependencies. This leads to many research challenges.

C1 How can Node.js package developers determine if and where breaking changes are introduced? Node.js package developers introduce breaking changes when updating their packages, sometimes inadvertently because it can be difficult to determine which changes are breaking. Ideally a program analysis tool should be able to tell developers not only if, but also where breaking changes occur in a package update. With such a tool, package developers would be able to write complete changelogs that describe every breaking change in the update. The clients would also be able to confidently switch to the new package version without worrying about undocumented breaking changes introduced in the update.

C2 How can Node.js client developers identify the source locations that are affected by breaking changes? Most package updates come with a changelog describing which parts of the package API that are affected by breaking changes. It is the responsibility of the client developer to identify the locations in the client code that use the changed package API. Since package updates may contain hundreds of breaking changes, it is a laborious and error-prone task to manually examine the client code for affected locations. Ideally, a tool should be able to tell the client developer exactly which source locations are affected by breaking changes.

C3 How can Node.js client developers determine how to adapt the source locations affected by breaking changes? After identifying the source locations affected by breaking changes in a package update, the client developer has to adapt the code to these changes. Some high quality changelogs may provide suggestions for how the client should adapt to the changes, but that is not the general case. Since the required changes are often the same across all clients, a tool would ideally automate this process for the client.

C4 How can Node.js client developers determine which dependency vulnerabilities are relevant to them? New vulnerabilities are continuously discovered in npm packages. Reports about these vulnerabilities are stored in public databases, such as the npm advisories database. Several tools known as security scanners, such as Snyk, Dependabot, and npm audit will notify clients if they depend on packages with known vulnerabilities. This approach for handling vulnerabilities is in theory good since clients are notified quickly and can take preventative action before they are exploited. However, clients tend to rely on many packages, and new vulnerabilities are discovered frequently, so many

\[\text{https://www.npmjs.com/advisories} \]
\[\text{https://snyk.io/} \]
\[\text{https://dependabot.com/} \]
\[\text{https://docs.npmjs.com/cli/v7/commands/npm-audit} \]
clients are simply overwhelmed with alarms about potential vulnerabilities. In addition, many vulnerability alarms are irrelevant since the client may not even use the exploitable part API of the vulnerable dependency. Thereby, clients may experience notification fatigue, where new alarms are neglected leaving the clients in a vulnerable state. To reduce the notification fatigue and improve the security of clients, they should ideally only be notified about the vulnerabilities that are exploitable.

C5 How can Node.js package developers find vulnerabilities in their code? Discovering security vulnerabilities in Node.js packages is often a race between package developers and white hat security researchers on one side and black hat malicious attackers on the other. When the attackers succeed in discovering a package vulnerability before the security community, the vulnerability may end up exploited before the package developer has a chance to fix it. To reduce the probability of that scenario, package developers will benefit from automated tools that find vulnerabilities. A particularly popular type of tool for detecting security problems is a taint analysis, which looks for dataflow between untrusted sources (user input) and sinks (problematic APIs). Static taint analyses are often favored over dynamic taint analyses for their potential soundness property ensuring that they detect all problematic taint flows. Scaling static taint analyses for JavaScript applications is, however, difficult due to some JavaScript language constructs that are hard to analyze statically.

1.2 Contributions

This thesis contributes with novel techniques for mitigating several of the maintenance-related challenges that developers in the Node.js/npm ecosystem face. We present techniques and prototype implementations of tools for addressing the challenges described above by identifying breaking changes in package updates, detecting and fixing locations in client code affected by breaking changes, performing fine-grained security scanning of Node.js applications, and pre-computing module summaries for helping static taint analyses scale to large real-world Node.js applications. The contributions made by this thesis, for addressing the challenges described above, are as follows.

C1 How can Node.js package developers determine if and where breaking changes are introduced? We present the type regression testing technique implemented in the tools NoRegrets (Chapter 7) and NoRegrets+ (Chapter 8). Using a dynamic analysis that leverages client test suites, type regression testing computes a model of package’s API before and after an update of the package. Certain changes across the models known as type regressions are indicators of breaking changes, which should be clearly documented for the clients. Whereas NoRegrets computes a model for both the pre-update and post-update version of the package, NoRegrets+ uses the pre-update model to generate tests for the
post-update version, thereby considerably increasing the performance of type regression testing. Type regression testing discovers many breaking changes in package updates, which cannot be detected with previous techniques.

C2 How can Node.js client developers identify the source locations that are affected by breaking changes? In Chapter 9, we present a novel static analysis that finds usages of APIs affected by breaking changes. The affected APIs are described using a novel type of pattern known as a detection pattern. The analysis, which is implemented in the TAPIR tool, takes as input a set of detection patterns and a Node.js program, and produces as output the locations in the code of the client corresponding to the API usages described by the detection patterns. The detection patterns are meant as a supplement to changelogs. For example, package developers may provide the detection patterns with the changelog to ease the update process for their clients. In the evaluation of Chapter 9, we show that the detection patterns are sufficiently expressive to detect almost all breaking changes in changelogs of top npm packages, and that TAPIR misses no affected locations in practice and is very precise. Finally, we show that TAPIR is extremely fast, using on average only around 1 second to analyze a client application.

C3 How can Node.js client developers determine how to adapt the source locations affected by breaking changes? In Chapter 10, we present the JSFIX tool, which augments the TAPIR analysis from Chapter 9 with the ability to also adapt client code to the breaking changes at the affected locations. We present the notion of a semantic patch consisting of a detection pattern (as described in Chapter 9) and a code template, where the latter define the transformation necessary to adapt the client code. Two essential properties that JSFIX must satisfy are that it patches all of the affected locations, and that no patches are applied to unaffected locations. If those two properties are not satisfied, then the updated client code may no longer work after the JSFIX patches are applied, and manual adjustments to the code are required. To ensure the first property holds, TAPIR will over-approximate the affected locations. An interactive phase, where users are asked about properties of their code that the analysis could not infer, is then used by JSFIX to remove false positive location matches, which ensures that the second property holds. We were able to write code templates for more than 90% of the detection patterns written for TAPIR, meaning that JSFIX can handle most breaking changes encountered in practice. 31 pull requests updating dependencies based on JSFIX patches have been accepted, strengthening the confidence in the correctness of the JSFIX patches.

C4 How can Node.js client developers determine which dependency vulnerabilities are relevant to them? Only a fraction of the audit reports presented to Node.js client developers are relevant since, in many cases, the vulnerable part of the affected dependency’s API is not used by the client. In Chapter 12, we present JAM, a lightweight modular call graph analysis for Node.js, which
1.2. CONTRIBUTIONS

can be used to determine if the vulnerable part of an API is reachable from the entry point of the client application. An evaluation shows that JAM reduces the number of false positives by 81% compared to a dependency scanner like npm audit, and that no reachable vulnerabilities (true positives) are missed by JAM.

C5 How can Node.js package developers find vulnerabilities in their packages? In Chapter [11] we present the dynamic analysis TASER, which infers taint specifications for packages. TASER leverages test suites of Node.js packages to track how values flow in and out of package APIs. From these observations, taint specifications are generated. When a static taint analysis encounters a use of some API for which a taint specification has been inferred by TASER, the taint analysis can use the specification instead of analyzing the code. Since reading the specification is much faster than analyzing the code, the taint analysis can scale to applications much larger than if no specifications were available. We show in our evaluation that the state-of-the-art static taint analysis LGTM[^10] can find previously undetectable security vulnerabilities using specifications inferred by TASER.

The five main contributions of this thesis are based on six published research papers listed below. The papers are included in Part II of this thesis (Chapters 7–12) with only minor layout adjustments to accommodate the rest of this thesis.

- **Type regression testing to detect breaking changes in Node.js libraries.** Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Published in Proc. 32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 2018 [96]. The paper is included in Chapter 7.

- **Model-based testing of breaking changes in Node.js libraries.** Anders Møller and Martin Toldam Torp. Published in Proc. Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering ESEC/FSE, August 2019 [100]. The paper is included in Chapter 8.

- **Detecting locations in JavaScript programs affected by breaking library changes.** Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. Published in Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), November 2020 [101]. The paper is included in Chapter 9.

- **Semantic Patches for Adaptation of JavaScript Programs to Evolving Libraries.** Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Published in Proc. 42nd International Conference on Software Engineering (ICSE), May 2021 [107]. The paper is included in Chapter 10.

[^10]: https://lgtm.com/
CHAPTER 1. INTRODUCTION


- Modular Call Graph Construction for Security Scanning of Node.js Applications. Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Published in Proc. 30th International Symposium on Software Testing and Analysis, July 2021 [106]. The paper is included in Chapter [12]

The author of this thesis has contributed to all aspects (idea generation, implementation, evaluation, and paper writing) of all the research projects presented in this thesis. The NOREGRETS tool was implemented in collaboration with Gianluca Mezzetti, TASER in collaboration with Christian-Alexandru Staicu, and TAPIR, JSFIX, and JAM in collaboration with Benjamin Barslev Nielsen. The NOREGRETS+ tool was implemented exclusively by the author of this thesis.

1.3 Outline

This thesis is divided into two parts. Part I presents an overview of the research conducted by the author during his PhD studies, and discusses its implications and potential for future work. Part II contains the full text of the papers on which part I is based.

The structure of Part I is as follows. In Chapter 2 the JavaScript language, Node.js runtime, and npm ecosystem are introduced. All the research presented later in this thesis addresses problems in Node.js and npm related to maintenance. Chapter 3 focuses on detecting breaking changes in package updates. The chapter first motivates the need for automated techniques for finding breaking changes and then describes the limitations with previous techniques. Next, the chapter provides a brief description of the type regression testing technique as implemented in the NOREGRETS and NOREGRETS+ tools. The chapter concludes with a discussion on limitations with type regression testing and research challenges for future work. Chapter 4 also keeps the focus on breaking changes but instead addresses the problem of adapting client code to breaking changes introduced in dependency updates. The chapter begins with an overview of the problem, including a description of how client developers currently handle breaking changes. It then provides a brief introduction to the TAPIR and JSFIX tools that partially automate the update process. Finally, it discusses how JSFIX may help change both client and dependency developers’ attitudes toward breaking changes. Chapter 5 considers two major security issues affecting many Node.js applications: the high false positive rate of security scanners and the challenge of scaling static taint analyses to real-world Node.js applications. First, the chapter discusses the scope of the problems, with examples on why existing security scanners have a low precision and why scaling static taint analyses for Node.js is hard. Then it presents the JAM tool, which performs security scanning by computing a call graph of an
application, and then only warns about vulnerable API usages reachable in the call graph, thereby achieving a much higher precision than existing security scanners. Afterward, it presents TASER, a dynamic analysis that infers taint analysis models for Node.js packages. Static taint analyses may gather relevant facts from these models instead of analyzing the package code, which increases the scalability of the analyses. The chapter ends with a discussion on the implications of JAM and TASER on Node.js security and potential directions for future work. Finally, Chapter 6 contains the conclusions of the thesis.
Chapter 2

JavaScript and Node.js

JavaScript is a highly popular language, and is also the language targeted by all of the techniques presented later in this thesis. While JavaScript is used in many different settings, this thesis focuses especially on the Node.js server-side runtime, which has seen extensive growth in popularity over the past decade. The strongest selling point of Node.js is the Node Package Manager (npm) system consisting of both a package manager application and a package registry. Using npm, developing Node.js applications with a high degree of code reuse is extremely easy compared to other languages such as Java, where package management is a more involved process. In this section, we consider both JavaScript, Node.js and the npm ecosystem. We will not regard every language feature of JavaScript and Node.js, but instead focus on the ones that are most important for the comprehension of the remainder of this thesis.

JavaScript is a dynamically typed programming language originally intended as a client-side (browser) web programming language. Web pages can use JavaScript to manipulate the HTML, send HTTP requests, store and load cookies, and in general make webpages more dynamic. Despite “Java” appearing in its name, JavaScript is quite different from the Java language. The name was primarily chosen for marketing purposes [139].

While originally designed as a front end-only language, JavaScript has since evolved into a general-purpose language through the creation of the Node.js server-side JavaScript runtime. Node.js is a combination of a JavaScript virtual machine (based on V8 as also used in Chrome), and an extension of the language itself with a library for performing system-facing operations. JavaScript has limited built-in support for I/O operations, so this library was crucial for making JavaScript a practical general-purpose language.

A Node.js program is comprised of a set of modules and each module always corresponds to exactly one file. One module may load another module through a call to require(id), where id determines which module to load. The exact algorithm used

---

1In a recent edition of JavaScript, the import statement was added to the language, which allows for the loading of specific properties from a module.
to resolve the id to a specific file is fairly complicated.\footnote{A pseudo-code version of the algorithm is provided here \url{https://nodejs.org/api/modules.html#modules_all_together}} However, the most important takeaway from the algorithm is that if id is a non-built-in module name and not a path, then require looks for the specified module in a special node_modules folder. For example, if calling require("lodash"), then a module stored in node_modules/lodash is loaded. The exact module is determined by the package.json file (see Section 2.2) in node_modules/lodash, but the default file is index.js. The node_modules folder is also where npm installs third-party dependencies as explained in Section 2.2.

A module exposes its public API by writing to properties on the module.exports object. Properties written to this object become available on the object returned by a require call. For example, in the fs module, the readFile function is exposed as follows:

```javascript
1 module.exports.readFile = function () { ... }
```

This function is then available for other modules as illustrated below:

```javascript
2 const fs = require('fs');
3 fs.readFile(...);
```

JavaScript code that is in the outer-most scope of a Node.js module is executed when the module is loaded. Node.js wraps this code in an implicit function that is executed upon a call to require or when using import. We can think of this function as the main function of the module. A Node.js program is then started by providing the Node.js runtime with a path to the initial module, e.g., the command below runs the index.js module.

```
$ node index.js
```

Other modules are loaded from index.js through calls to require or using the import statement.

### 2.1 Reflective Features

One of the traits that separates JavaScript from other languages is the way reflective features are an integrated and broadly used part of the language: it is possible to dynamically both overwrite and delete properties on an object; property names can be computed dynamically; JavaScript has the eval function for interpreting a, potentially dynamically constructed, string as JavaScript code; arguments to require can be dynamically computed, which means that it might not even be possible to tell which modules an application depends on statically.

While other languages, such as Java, also support reflection, the key difference in JavaScript is that many of the reflective mechanisms are not niche features primarily used for special-purpose tools such as frameworks and parsers, but instead essential features used in average basic applications. For example, until recently JavaScript did not include a type specifically for map data structures, so standard objects are
commonly used as maps. It is therefore normal to see JavaScript programs where properties are added, deleted, or overwritten dynamically from objects [121].

Because of this reflective trait in the JavaScript language, it is well-known to be a particularly hard language to analyze statically [80, 105, 133]. A small loss of precision may lead to an explosion in the number of spurious paths. A common example of a problematic situation is when an imprecise abstract string is used in a dynamic property read or write. In such a situation, the analysis will have to assume that every property of the object is read or written if the analysis should remain sound. Another challenging example is when a correlated read/write pair is used to copy properties from one object to another [20, 104].

**Example 1** A contrived correlated read/write pair is shown below:

```javascript
const x = {
    f: () => {...},
    g: () => {...}
};
...
const y = {};
for (var p in x) {
    y[p] = x[p]
}
}...
y.f();
```

On line [12] each property of x is written to y. In particular, the f property is written to y, such that when the y.f() method call is invoked on line [15] it is the function defined on line [6] that is called. For a static analysis to resolve the call on line [15] it has to precisely analyze the assignment on line [12] A state-of-the-art static analysis can handle the example above, but if the dynamic read and write are placed further apart, then it may be too challenging for even the most advanced analysis [104].

Many of the tools and techniques, which we present later in this thesis, are designed with this observation about the JavaScript language in mind. We do not attempt to create fully sound whole-program static analyses for JavaScript since we know that such analyses will struggle to handle real-world JavaScript. Instead, we either rely on dynamic techniques or lightweight static analyses, where the static analyses are built with certain assumptions about how programmers write JavaScript programs. While the static analyses may not be theoretically sound, they are designed to be sound for the programs where the soundness assumptions hold. The soundness assumptions are explained in detail in the Part II chapters.

### 2.2 Node.js Package Manager

As explained in the previous section, the Node.js module system makes it easy to separate a program into logically self-contained units. In addition, Node.js also has its own package manager npm (Node Package Manager) that makes it easy to import and manage third-party packages. A package is a collection of Node.js modules. Some packages are intended to be used as libraries or frameworks, while others are
Figure 2.1: Package.json example

```json
{
    "name": "myPackage",
    "version": "1.0.0",
    "dependencies": {
        "lodash": "4.0.0"
    },
    "devDependencies": {
        "eslint": "^7.0.0"
    },
    "author": "",
    "license": "ISC"
}
```

standalone applications. Packages can either come from a custom registry, GitHub or one of many other places. However, most commonly, packages are stored in the npm registry, which, as of this writing, contains more than 1.6 million packages and grows with almost a thousand new packages per day.1

A Node.js developer manages third party packages through the `dependencies` and `devDependencies` fields in a JSON file called package.json. Figure 2.1 shows an example of a package.json file for the application `myPackage`, which has two dependencies: one runtime dependency (`lodash`) and one developer dependency (`eslint`). The developer dependencies are only required during development, for example to compile or lint the application, whereas the runtime dependencies are required to run the application. When the developer runs the `npm install` command, npm will download the dependency packages and add them to the `node_modules` subfolder of the application.

Each dependency is accompanied with a version constraint. For example, on line 20, the constraint specifies that exactly version 4.0.0 of `lodash` is required. Version constraints can also allow for a range of versions. For example, the constraint on line 23 requires `eslint` to be at minimum version 7.0.0 but lower than version 8.0.0. Version 7.31.0 is thus allowed, but not version 8.0.0. This type of version constraint permits automatic updates of minor and patch versions but not major versions (see semantic versioning below).

As a summary of the methodology used when talking about Node.js applications: A package is a collection of Node.js modules. A package may be a standalone application or a library designed to be used by other packages. If package A uses package B, then we say that B is a dependency of A and that A is a client or dependent of B. A client does not necessarily use every module in each of its dependencies. A package may serve as a dependency to some client and itself also be a client of another package as illustrated by Figure 1.1. We call the transitive closure of a package’s dependencies for its transitive dependencies.

http://www.modulecounts.com/
2.3 Semantic Versioning

Semantic versioning is an approach for assigning version numbers to program updates according to specific rules about the impact of these updates on clients. Every version number is a triple consisting of a major, a minor, and a patch number separated by dots. For example, for the triple 1.2.4, the major number is 1, the minor number is 2, and the patch number is 4.

Whenever a developer of a package wishes to release a new version of the package, the developer must carefully choose a new version number. Following semantic versioning, the developer must increment the major number if the update contains backwards incompatible changes (breaking changes), the minor number if new backwards compatible features are added, and otherwise, increment the patch number. The minor and patch numbers are reset to 0 when the major version is incremented, and similarly, the patch number is reset when the minor number is incremented. Updates are named according to the number incremented, e.g., if the patch number is incremented, then the update is known as a patch update. Patch updates are typically used for bug fixes, security enhancements, and performance improvements.

The primary advantage of semantic versioning is that clients can update dependencies automatically, not having to consider potentially backwards incompatible breaking changes, as long as the updates are marked as minor or patch. At least, that is the intention of semantic versioning. It is generally the responsibility of the update developer to decide which semantic versioning category the update belongs to. The developer may intend for an update to be backwards compatible, but due to some mistake, or some edge case that was not considered, the update may actually contain breaking changes. Some languages that follow semantic versioning have tooling available that partially automates the decision process. For example, the elm compiler can detect changes in the static types across versions that are backwards incompatible thus indicating breaking changes. Semantic versioning is recommended for npm packages, but npm does not include any tooling that helps developers categorize updates.

An important criticism of semantic versioning is that incompatibility is not easily formally defined, and different communities may have different views on what changes should be considered incompatible. There are certain incompatible changes, where one can provide reasonable arguments that they are too unlikely to cause problems for clients, to warrant a major version. For example, minor formatting changes in the message of an exception will only cause trouble in clients that have very specific requirements to the structure of the message. We will discuss this issue further in Chapter 3.

[^semver.org]: https://semver.org/
[^elm-lang.org]: https://elm-lang.org/
Chapter 3

Breaking Change Detection

Most Node.js packages require ongoing maintenance. Over time bugs are reported, security issues are discovered, and the Node.js runtime is updated. Package maintainers are expected to react to these events and decide if changes to the packages they maintain are required. A package may also have dependencies that should be updated for similar reasons. Other factors may also drive package maintainers to modify their packages. For example, a package maintainer may wish to refactor a package’s API to ease future maintenance and improve its usability for clients. We will use the terms package and dependency interchangeably in the remainder of this chapter (see Section 2.2 for a detailed explanation about these terms).

If changes to the package code are required, a new version of the package is created and submitted to the npm registry where it will be available for clients to install. Package developers are expected to follow the semantic versioning scheme (see Section 2.3) when assigning version numbers to package updates. If the package update contains breaking changes, then the package developer should increment the major number, or otherwise, increment either the patch or minor number. For major updates, the package developer should also document the breaking changes in a changelog in sufficient detail to allow client developers to address them.

Determining if an update contains breaking changes is often a non-trivial task. Our study has shown that at least 5% of all packages have been affected by a breaking change in a patch or minor update of a dependency [96]. Several studies have also shown that more than a third of Java dependency updates that are marked as minor or patch contain breaking changes [33, 118]. We will demonstrate why some breaking changes are missed by package developers through the changes to the createConnection function in Figure 3.1. If we examine only the signature changes on line 2 and 3, the update seems to be non-breaking. A new port parameter has been inserted after options, but in JavaScript, it is allowed to pass fewer arguments than there are parameters, which implicitly results in undefined being used for the missing arguments. As long as the case where port is undefined is handled as in the pre-update version, this change is non-breaking, and by line 8, the same default port value of 80.
CHAPTER 3. BREAKING CHANGE DETECTION

Figure 3.1: Green (+) lines are added and red (-) lines are removed.

```javascript
// inside package http-tools
module.exports.createConnection = function (host, options) {
  if (host === 'localhost') {
    return this.localConnectionObject;
  }
  host = '${host}:80';
  return new Connection(host, options.timeout, options.debug);
}
```

is used as in the pre-update version.

However, the package developer has also removed a path through the method where `this.localConnectionObject` was returned for calls to `createConnection` where host had the value 'localhost' (see lines 4–6). It might also seem like a non-breaking change at first glance, but it causes a `TypeError` exception to be thrown at the read of `options.timeout` or `options.debug` on line 9 when `options` is undefined. It was valid to set the `options` parameter to undefined in the pre-update version for the calls where host was set to 'localhost' due to the fast path where `localConnectionObject` was immediately returned before any properties on `options` were read. Since this second change is subtle and probably also unintentional, it is quite likely that it will not be documented in the changelog.

The changes shown in Figure 3.1 demonstrate the disguised nature of some breaking changes. Especially breaking changes that are introduced unintentionally can be hard to discover. For example, had the developer of the `createConnection` function been aware of the type errors potentially arising on line 9, he could have used the optional chaining operator (`options?.timeout` and `options?.debug`), which causes the reads to return `undefined` if `options` is null or undefined.

Breaking Changes

Before continuing the discussion on how to identify breaking changes, it is important to consider how to formally define a breaking change. A naive first attempt is to define a breaking change as: “A change in an API where it is possible to construct a client that uses the API and whose behavior is altered by the change”. However, that definition is too broad. The word “behavior” may comprise both semantic behavior as in “two functions have the same behavior if their result and side-effects are equivalent” but also other types of non-functional behavior such as running time and memory use. A change that decreases the running time may be breaking to some clients where the order in which events are triggered is affected by the change in performance. Similarly, a change that increases memory usage may cause clients to run out of

\[ x \| y \text{ results in } y \text{ if } x \text{ is a falsy value such as undefined} \]
3.1. EXISTING WORK

memory. Arguably it is both rare and bad practice for clients to have very specific requirements to the running time and memory usage of its dependencies. We should therefore limit our definition of breaking changes to only include semantic behavioral changes, i.e., changes to the semantics of the program.

Our second attempt at a breaking change definition is “A change in an API where it is possible to construct a client that uses the API and whose semantic behavior is altered by the change”. This second attempt is, however, too imprecise since it forces us to also categorize bug fixes and security fixes as breaking changes. Consider a client developer who discovers a bug in a dependency, and then immediately reports it to the dependency developer. Not wanting to disable the dependency while the bug is being fixed, the client developer decides to write a workaround in the client code that handles the case where the bug appears. Since the workaround now expects the bug to be present, it may well be that when the dependency developer later releases a version of the dependency without the bug, the client breaks.

The third attempt at a formal definition then becomes: “A change in an API where it is possible to construct a client that uses the API, adheres to the API as specified in the API documentation, and whose semantic behavior is altered by the change”. The “adheres to the API as specified in the API documentation” is an attempt to ensure that the client is reasonable, i.e., that it does not include workarounds for bugs or deliberately misuses the API. Unfortunately, it is an informal requirement since documentation is often missing or incomplete leaving us in an uncomfortable situation, where we cannot give a precise definition of a breaking change that will allow us to unambiguously classify every change as breaking or non-breaking. Nevertheless, this definition is as good as it gets, and for most real-world cases, it should allow us to mark the changes as breaking or non-breaking with high confidence. Considering the changes in Figure 3.1, the signature change is non-breaking since clients that use the pre-update version of the API correctly are unaffected by this change. The second change related to the timeout and debug property on the options object is, according to our definition, only breaking if the documentation of the pre-update API allows options to be undefined.

3.1 Existing Work

Very little attention has been put on developing tools for detecting breaking changes in JavaScript or Node.js dependency updates prior to our work on type regression testing (see Section 3.2). To the best of our knowledge, the only other tool for Node.js is the

---

2 It is reasonable for clients to expect changes in dependencies to not significantly degrade performance without such a change being marked as breaking. Since this issue is orthogonal to semantic behavioral changes, and not as relevant in the Node.js and npm context, we will not discuss it further in this thesis.


4 https://xkcd.com/1172/
The *dont-break* tool, which works as follows. For a package update of $A$ from version $x$ to $y$, *dont-break* gathers a set of clients (also known as dependents) of $A$ that use version $x$. It will first run the test suites of the clients using version $x$ of $A$. Afterward, it changes the clients to use version $y$ of $A$, and then reruns the test suites. The test failures appearing after changing to version $y$ must be caused by breaking changes introduced in version $y$. While *dont-break* is conceptually simple and precise (all reported warnings are true positives modulo flaky tests), it, unfortunately, has a low recall. Many breaking changes do not necessarily result in test failures, either because the client tests do not cover the affected part of the dependency API, or because the breaking change is not sufficiently critical to cause a test failure in that specific case.

Several commercial tools have been built on the *dont-break* approach. Dependabot (now acquired by GitHub) creates pull request updating dependencies whenever new versions are available. Dependabot keeps track of how many CI pipelines fail or succeed with the new dependency version, and uses this statistic to assign a compatibility score to each dependency version. Dependency versions with a high compatibility score are less likely to contain breaking changes. Greenkeeper (now acquired by Snyk) also keeps track of dependencies, and creates pull requests when new versions are released. Greenkeeper tracks if pull requests are breaking tests, i.e., if they contain breaking changes.

More advanced breaking change detection tools also exist for other languages. All of these tools attempt to find backward-incompatible changes in the types across dependency versions. Common to all of these tools is that they target statically typed languages, where the types are easy to extract.

The APIDiff [17], Clirr [9] and RevAPI [10] tools find either source or binary incompatibilities across versions of Java APIs. Source incompatibilities comprise those changes that cause the compilation of target programs to fail. For example, adding an extra argument to a method signature is clearly going to cause compilation failures for existing clients. Binary incompatibilities comprise those changes that will not result in compilation errors for clients, but may still cause failures at runtime if the compiled class files of the old version are swapped with compiled class files of the new version. Binary incompatibilities are not relevant in languages like JavaScript where no Ahead of Time (AOT) compilation takes place, so we will not discuss them further here.

The Elm [11] and Go [12] languages (both statically typed) also include tooling for automatic detection of breaking changes based on changes in static types across versions.

---
[8] https://pkg.go.dev/golang.org/x/exp/apidiff
3.2 Type Regression Testing

Type regression testing (TRT) is the first technique for detecting breaking changes in Node.js package updates by identifying changes in the types of the public APIs of packages. A detailed description of the type regression testing technique is provided in Chapter 7 and Chapter 8.

Because JavaScript is dynamically typed, it is infeasible to statically infer the types of a module’s public API. The members of a public API can even change at runtime making it difficult to statically name the members (properties) that are part of the public API. Therefore, TRT relies on a dynamic analysis for gathering the required type information. While this choice means that TRT is by construction unsound, TRT still improves significantly on the only existing breaking change detection tool for Node.js, which is dont-break.

TRT can be seen as an extension to the dont-break approach. In fact, all breaking changes detected by dont-break are by construction also detected by TRT. TRT is implemented in the NoRegrets (Chapter 7) and NoRegrets+ (Chapter 8) tools, where NoRegrets+ includes a set of enhancements to the technique making it both faster and able to detect more breaking changes. For the remainder of this section, we describe TRT as implemented in NoRegrets. In the end of this section, we provide an overview of some of the enhancements made in the NoRegrets+ implementation.

Like dont-break, TRT exploits the test suites of existing client applications. However, while dont-break relies entirely on test failures to detect breaking changes, TRT builds a fine-grained model of the package’s public API. The key observation here is that “While client test suites are not generally designed to test dependencies of the client, parts of the dependency code are exercised as a side-effect of testing the client.” This observation illustrates both the weakness of dont-break and the strength of TRT. Because client test suites are not generally designed to test the dependency code, there is a high probability that breaking changes in a dependency will not cause test failures in clients, resulting in a low recall for dont-break. On the other hand, the fact that some dependency code is exercised lets TRT gather observations about the interface between the client and the dependency. As our evaluations reveal (see Chapter 7 and 8), these observations are sufficient to discover many breaking changes.

The TRT technique computes a model of a dependency’s public API. The model assigns types to every point in the public API, where the points are identified using a special-purpose access path mechanism (see Example 2). In the NoRegrets tool, a public API model is computed for both the pre-updated and post-update versions of the package. A special typing relation is then used to compare these models. Incompatible differences between the models, which we refer to as type regressions, indicate that breaking changes were made between the pre-update and post-update versions of the package.

To create the public API models, a membrane-based dynamic analysis is used.

Unsoundness is here used with the same meaning as when talking about static analyses, i.e., that an unsound analysis may miss some true alarms.
CHAPTER 3. BREAKING CHANGE DETECTION

Figure 3.2: Membrane-based analysis principle of type regression illustrated. Observations about the values are only collected when the values are on the opposite of where they were created.

(a) A value created by Client flows to Dependency and back again.

(b) A value created by Dependency flows to Client and back again.

Conceptually, the analysis can be seen as an extension to a JavaScript/Node.js interpreter that adds instrumentation, such that the interpreter creates the API model dynamically as it is running. In the NoRegrets tools, the proxy mechanism of ECMAScript 6 is used to emulate this interpreter. The membrane-based analysis principle is illustrated in Figure 3.2. The membrane sits at the barrier between the client and the dependency code. When a value is passed from the side on which it was created to the opposite side of the barrier, any future usage of that value (at least until it passes the barrier again) will provide information about the dependency’s public API. TRT observes when a value passes the barrier from the side on which it was created. When the value is on this side, all actions performed on the value are recorded in the public API model. For example, if an object is created on the client-side and passed to the dependency as the first argument of a function \( f \), and the dependency reads a property on this object, we know that the dependency’s behavior is probably dependent on whether this property is present on the first argument to \( f \). We also know from the value stored in this property what type it should have. Similarly, a call to a method \( m \) on an object that is created in the dependency and returned to the client by the dependency function \( g \), tells us that \( m \) shall be present on objects returned by \( g \). Bits of information like these are like pieces in a puzzle that comprise the types of the dependency’s public API. While TRT does not guarantee that it can create the full puzzle (infer all public API types), it will gradually learn more and more facts about the API as more information is gathered from more clients. Furthermore, the part of the public API that TRT infers by running the test suites of real clients is also the part

3.2. TYPE REGRESSION TESTING

Table 3.1: Partial models for the pre-update and post-update version of http-tools created for Figure 3.1

<table>
<thead>
<tr>
<th>access path</th>
<th>pre-update type</th>
<th>post-update type</th>
</tr>
</thead>
<tbody>
<tr>
<td>require(http-tools).createConnection</td>
<td>function</td>
<td>function</td>
</tr>
<tr>
<td>require(http-tools).createConnection(2)0</td>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>require(http-tools).createConnection(2)1</td>
<td>object</td>
<td>object</td>
</tr>
<tr>
<td>require(http-tools).createConnection(2)1.timeout</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>require(http-tools).createConnection(2)1.debug</td>
<td>undefined</td>
<td>undefined</td>
</tr>
</tbody>
</table>

that is most likely to be relevant for real clients. Breaking changes in largely unused corners of a dependency’s API that no clients use are not as critically important to avoid or document.

Example 2 Let us consider how TRT can detect the breaking change caused by the options.timeout read on line 9 in Figure 3.1. We assume that there is a client of http-tools which, when running its test suite, issues the following call to the createConnection function: httpTools.createConnection('localhost',{}). The important thing to notice here is that this call does not trigger the type error since the options object is not undefined. In other words, using the dont-break approach, this client would not reveal the breaking change.

Running the client’s test suite twice (once with the pre-update dependency version and once with the post-update version) using NOREGRETS, two public API models are created. A model is a map from special-purpose access paths to types. In Table 3.1, parts of the models created for http-tools are shown. An access path identifies a specific point in the public API of the dependency, and the type it maps to covers all values observed at that API point. For example, require(http-tools).createConnection refers to the value stored in createConnection property on the module object returned by require('http-tools'), and require(http-tools).createConnection(2) refers to the return value of the function stored in this property when it is called with two arguments. Since require(http-tools).createConnection maps to function, all values observed at the API point identified by that path were of type function. Every model is a total map, in the sense that for a path where no values were observed, the path maps to the type ◦, which illustrates that no values were observed for that path.

Going back to the models computed for http-tools as shown in Table 3.1, we see that the only paths where the pre-update and post-update models disagree on the type are:

- require(http-tools).createConnection(2)1.timeout
- require(http-tools).createConnection(2)1.debug

These paths represent the read of the timeout and debug properties on the first argument of createConnection when createConnection is called with two arguments. For both paths, the pre-update type is ◦, indicating that no observations were made in connection to these access paths in the pre-update version of http-tools, and the post-update type
is undefined, indicating that NOREGRETS observed at least one read of timeout and debug where their value was undefined. Following the Liskov substitution principles, a function is allowed to relax its parameter types without causing type errors. However, in this case, the requirement is strengthened (createConnection now expects the timeout and debug properties to be present), so a type regression is produced for each of the two paths. The type regressions show that type-related breaking changes have occurred, and the user of NOREGRETS is notified accordingly.

The type regression testing technique as implemented in NOREGRETS improves considerably on the dont-break approach. NOREGRETS detects 26 breaking changes in 167 minor and patch updates of high-quality npm packages, where only 4 of those breaking changes are detectable using dont-break (see Section 7.7). Furthermore, the type regressions often indicate exactly where the breaking change has occurred (as demonstrated in Example 2) as opposed to the test failures of dont-break, where the dependency developer often has to inspect the client code to understand the test failure.

An improved version of the TRT technique was implemented in the NOREGRETS+ tool (see Chapter 8). In the improved version, only the pre-update model is generated. NOREGRETS+ then looks for type regressions in a post-update version of the package by using the model to drive an exploratory phase through the package’s public API. In this phase, NOREGRETS+ will assert that no type regressions have occurred between the pre-update and post-update versions. The model is augmented with additional information compared to NOREGRETS to allow NOREGRETS+ to synthesize values, e.g., function arguments, that are necessary for exploring the API.

The NOREGRETS+ model checking phase is more than 25 times faster than the model generation phase of NOREGRETS, and NOREGRETS+ finds more breaking changes due to more fine-grained API models. NOREGRETS+ also relaxes the constraints on which client test suites that may be used for model generation, which makes it more suitable for packages that have fewer clients.

### 3.3 Discussion

Type regression testing is the first technique for inferring breaking changes in Node.js packages at the level of API types. It provides package developers with the NOREGRETS and NOREGRETS+ tools that allow them to identify the specific parts of package APIs affected by breaking changes. Previously, package developers had to rely on their instinct, which if wrong, could result in major updates incorrectly being marked as minor or patch, or in breaking changes not being documented in the changelog.

However, both tools are imperfect, and only partially solve the problem of detecting breaking changes. Since the tools are based on dynamic analysis, they may produce false negatives, i.e., they may fail to detect all breaking changes. As we argued above, it is generally difficult to create sound static analyses for JavaScript, which is also why we designed TRT as a dynamic technique. However, if incorporating certain assumptions about the analyzed program, static analyses may scale well
at the cost of some soundness that means little in practice \cite{43, 101, 106}. For future work, evolving TRT to a static technique may be possible using some of these ideas for improving scalability.

Some breaking changes fall into the category of behavioral breaking changes (also called semantic breaking changes in Chapter 7 and 8), which neither TRT nor any other current technique is able to detect. As described above, previous work has shown that behavioral changes are rare compared to the type-related changes TRT can detect. However, the fact that some breaking changes remain undetectable by TRT means that developers can only use the \texttt{NoRegrets} results as a supplement to their own judgement. Evolving TRT to a technique that detects all behavioral changes is not possible since it effectively becomes a matter of checking program equivalence, which is undecidable. It might be possible to improve the technique for finding at least a subset of the behavioral changes. Because it is a dynamic technique based on test suites of real clients, it is also likely that the behavioral changes detected by such a tool, are the ones that are most likely to affect real clients. A possible way of adding this extension is to record primitive values instead of types in the API models with some mechanism to filter or abstract random and time-specific values.
Chapter 4

Breaking Change Adaptation

In the previous chapter, we discussed breaking changes primarily from the package developer’s perspective. Our goal was to help package developers identify all breaking changes that are introduced in a package update. With this information, the package developer can remove the breaking changes from the update before releasing it to the clients of the package. Alternatively, the package developer can mark the update as a major release and document the breaking changes accordingly.

In this chapter, we keep our focus on breaking changes, but take the perspective of client applications. That is, we now focus on the scenario where a package update containing breaking changes (a major update) has been released, and client applications are forced to address these breaking changes if they want to apply the update.

The first question to ask about this scenario is why should client applications even care to update their dependencies at all? If a client C depends on version 1.0.0 of the package P, and a new major release bumping P from version 1.0.0 to 2.0.0 is released, why should C care to update P to version 2.0.0 if C works well with version 1.0.0? There are multiple possible answers to this question. Maybe version 1.0.0 of P contains security vulnerabilities that are fixed in version 2.0.0. Version 2.0.0 of P could also come with performance enhancements or usability improvements that ease maintenance of its clients. It may also be that version 1.0.0 is incompatible with a new version of Node.js. A recent study by Brito et al. surveyed Java developers about their reasons for introducing breaking changes and found that they were motivated “by the need to implement new features, by the desire to make the APIs simpler and with fewer elements, and to improve maintainability” [18]. However, even if none of the previous cases apply or are irrelevant for C, a much more subtle problem still persists if C decides not to upgrade. What if in two years a new security vulnerability of P is discovered that affects all versions up to and including 2.0.0 of P? What is likely to happen in such a scenario is that the maintainer of P fixes the vulnerability in a patch or minor update in the most recent major range. For example, the maintainer of P may fix the problem in a patch update marked as version 2.0.1. Since this patch update builds on top of version 2.0.0, C, who still depends on version 1.0.0, is not able
to apply this update without considering the breaking changes introduced in version 2.0.0. Even worse, if $C$ has neglected several major updates of $P$, the $C$ developer needs to address breaking changes from multiple major updates before receiving the vulnerability fix [82]. Since fixing breaking changes is currently a time-consuming manual process, $C$ is inevitably left in the vulnerable state for longer than if it already depended on the latest version of $P$.

Node.js developers typically neglect updating dependencies because of the manual effort required and the high risk of mistakes. As one developer responded in a survey by Mirhosseini and Parnin “it [dependency management] is one of the most significantly painful problems with development” [98]. In general the surveyed developers emphasized the three biggest hurdles as breaking changes, understanding the implications of changes, and migration effort. These are exactly the challenges we will address in this chapter (breaking changes from the dependency developer’s perspective were covered in the previous chapter).

To illustrate why developers find dependency management difficult, let us consider the typical update process for a developer of a client $C$ once she has decided to upgrade a dependency $P$, to a version that contains breaking changes. First, she should consult the changelog of $P$, where the developer of $P$ should have provided a comprehensive list of both the breaking and non-breaking changes in the update. From this list, she must find the source locations in $C$ that are affected by the breaking changes. For most breaking changes, this task is a matter of identifying usages of specific parts of $P$’s API [101]. For example, if a breaking change affects calls to the function $f$, then she must find all calls to $f$ in $C$. How should a Node.js developer go about with such a task? The $C$ application may consist of thousands of lines of code, and the $C$ developer may not remember exactly where $P$ is used. She may also be one of many contributors to $C$ meaning that she is probably unfamiliar with large parts of the codebase. In the most naive approach she scans, for each breaking change, through the entire codebase of $C$ trying to identify source locations affected by the breaking change. Of course, that approach is rarely used in practice since tools like grep can help reduce the search space. After identifying the affected locations, she has to determine what should be done to address the breaking change. Sometimes the changelog will provide suggestions, but in other cases, the developer must envision a solution. There are thus two challenges the $C$ developer must overcome: identifying the source locations in $C$’s codebase affected by breaking changes, and taking appropriate action to address the breaking changes.

Example 3  Let us consider the change made to the `createConnection` function of `http-tools` in Figure 3.1. We assume that the developer of `http-tools` used NOREGRETS, learned about the breaking change related to the `timeout` and `debug` properties, and decided to use the `?` operator to avoid the type error. However, after inspecting the

---

1Creating this changelog is a non-trivial and error-prone task. Tools like NOREGRETS described in Chapter 3 may help, but it is not uncommon that some breaking changes are missing from changelogs. For most of this chapter, we ignore this issue and instead assume that changelogs are complete. In the end, we well provide a discussion about incomplete changelogs.
other changes, the developer was left unsatisfied with the way the `port` parameter had been introduced. In the rest of the `http-tools` API (and in most other packages), the convention is to have `options` as the last parameter. So while adding `port` as the last parameter preserves backward compatibility, it has the undesirable side-effect that `options` is not the last parameter. The developer decides to switch the order of `options` and `port`, document the breaking change, and mark the update as major. The change to the signature of `createConnection` is as shown below:

```javascript
1 - module.exports.createConnection = function (host, options, port) {
2 + module.exports.createConnection = function (host, port, options) {
```

The changelog of `http-tools` will read “Breaking change affecting `createConnection`: Users should specify the port through the second parameter. Using undefined for port results in the value 80 being used. Move the `options` parameter to the third position.”

We now switch the perspective to a client developer who wishes to update to the newest version of `http-tools`. The client developer reads the changelog and notices the breaking change affecting `createConnection`. She reads through the client codebase, and eventually finds a single call to `createConnection`. After reconsulting the changelog about the introduced breaking change, she determines that she can preserve the old behavior by inserting 80 as the second argument\(^2\) and move the `options` argument to the third position:

```javascript
3 // load the http-tools library
4 const httpTools = require('http-tools');
5 // establish the connection
6 - httpTools.createConnection('localhost', { timeout: 5000 });
7 + httpTools.createConnection('localhost', 80, { timeout: 5000 });
```

By applying the changes to the `createConnection` call as shown in line 6 and 7, she can upgrade `http-tools` without the breaking change affecting the client.

## 4.1 Existing Work

Adapting client code to breaking changes in dependencies is a challenge that has received some attention in the literature. Coccinelle is a tool that modifies drivers written in C to become compatible with new versions of the Linux kernel \([108][109]\). The term *semantic patch* was also first introduced in relation to Coccinelle. A semantic patch is a formal description (written in some domain-specific language) that specifies both the specific API usage affected by a breaking change and how this usage is adapted to the breaking change. Coccinelle has since been updated to also work with general Java programs \([70]\).

The Coccinelle tool is not easily adapted to work with JavaScript. Coccinelle takes advantage of the often very idiomatic usage of the Linux API to identify specific API usages. For example, a typical usage of some Linux kernel API may look like this:

\(^2\)Using `undefined` would also preserve the semantics, but the code is more readable when the port is explicit.
The declaration of `xyz` followed by the call to `some_linux_api`, followed by a conditional to handle errors from the call strongly suggests which part of the Linux API is used. Coccinelle can thus detect usages of this API by searching for that pattern. Contrast this example with a typical Node.js API usage:

```javascript
function f(x)
{
  x.map(...)
}
```

Is `x` here a JavaScript array or is it an object created by the lodash utility library? We cannot answer this question just by looking at the body of `f`, so a different approach is required. A tempting solution might be to try to develop a whole-program static type inference analysis. However, as we explained in Section 2.1, scaling static analyses for JavaScript is challenging.

One approach for creating scalable static analyses for JavaScript is to use a so-called field-based analysis as first demonstrated by Feldthaus et al. [42]. A field-based analysis abstracts object fields by their name. Consequently, if the analysis encounters a write `x.f = function () { ... }` and later `y.f = function () { ... }` followed by a call of the form `z.f()`, then the analysis will abstractly assume that the callee is both of the functions previously written to the property `f`. The advantage of this approach is that no points-to analysis is necessary and the analysis does not have to be context sensitive. Both of these properties improve scalability considerably. The primary disadvantage is the risk of spurious dataflow caused by non-unique property names. However, as the evaluation of our field-based analyses demonstrate (see Section 4.2 and Chapter 5) in most real-world programs, property names are sufficiently distinct for a field-based analysis to produce useful results. Field-based analyses generally also handle dynamic property reads and writes unsoundly. For example, for a write like `y[p] = x[p]` on line 12 in Section 2.1, a field-based analysis may not know the value of `p`, so to analyze this statement soundly, it would have to assume that `p` could be any property. Clearly that will cause an explosion in spurious dataflow at all dynamic property reads and writes, and is therefore not a reasonable approach. However, what Feldthaus et al. found is that most dynamic property reads and writes are used to copy properties that were already assigned elsewhere. For example, considering Example 1, the copy on line 12 copies the properties assigned to the `x` object during its initialization on line 6 and line 7. So if the analysis just ignores the dynamic read and write on line 12, it will still correctly resolve the call on line 15 to the function defined on line 6. As a consequence: while ignoring dynamic property reads and writes is technically unsound, it does, according to our experiments, not lead to a noticeable degree of unsoundness in real-world code.
4.2 Breaking Change Adaptation for Node.js

We have developed two tools that help Node.js client application developers update dependencies that contain breaking changes (See Chapter 9 and 10). The first tool, TAPIR (Chapter 9), identifies source locations in the client application’s codebase affected by breaking changes (from here on called “affected locations”). The second tool, JSFIX (Chapter 10), patches the source code surrounding the affected locations, ensuring the semantics of the client application are preserved when applying the update. While TAPIR can theoretically be used independently of JSFIX for users who only want to identify the affected locations, JSFIX requires the results from TAPIR to perform its transformations. For the remainder of this section, we focus on the JSFIX tool, which can be viewed as a 3-phased tool where TAPIR serves as the first phase.

The 3 phases of JSFIX are illustrated in Figure 4.1. In the example illustrated in the figure, we have a client application where some dependency lib is upgraded from version 1.0.0 to 2.0.0.

The JSFIX tool requires a set of semantic patches to update lib to version 2.0.0. A semantic patch consists of a detection pattern for identifying affected locations and a code template for patching the affected locations. The notion of a semantic patch as a mechanism for describing how to identify and patch affected locations was first introduced in relation to the Coccinelle tool [108]. While semantic patches used with Coccinelle serve the same purpose as the ones used with JSFIX, there is otherwise little in common between the two semantic patch languages. Semantic patches are currently entirely hand-written, which may seem perplexing for a tool that claims to automate the update process. However, the key advantage of JSFIX is that the semantic patches can be reused to patch infinitely many clients. The time spent writing semantic patches is easily justified when seen in relation to the accumulated time savings that JSFIX brings to all client developers.

In the first phase of JSFIX (Analysis phase in Figure 4.1), the TAPIR analysis is used to find all affected locations in the client application. TAPIR uses a lightweight field-based static analysis to compute a set of access paths for each expression in the client application. The field-based approach ensures scalability as explained in Section 4.1. The access paths are matched against the detection patterns resulting in a set of potentially affected source locations. The “potentially” part is from the fact that TAPIR is designed to be overapproximating. It is critical that all affected locations in the client’s source code are detected. An unpatched affected location entering production may result in misbehavior or crashes of the client application. Because TAPIR is overapproximating, it may also mark too many locations as affected resulting in false positives. A false positive affected location causes JSFIX to patch an unaffected location, which may result in misbehavior or crashes for the client application. It is thus crucial that TAPIR produces neither false positives nor false negatives. By Rice’s theorem such a requirement cannot be satisfied in practice, which brings us to the next phase of JSFIX (Interactive phase in Figure 4.1).

The goal of the interactive phase is to remove the false positive affected locations produced by the analysis phase (TAPIR). Since the analysis phase is overapproximat-
Figure 4.1: The 3 phases of JSFIX illustrated.

ing, we can ignore the issue of false negatives. JSFIX removes the false positives by asking the user (typically the client application developer) yes/no questions about the client application code. There is a very important characteristic about these questions, which should be emphasized: Even though the breaking changes are in the dependency, the questions concern only client code. This property holds since it is insufficient information about the client code that leads to uncertainty in JSFIX.

JSFIX does not need to ask questions about every potentially affected location. The analysis phase marks each potentially affected location as a “high confidence” or “low confidence” match. If an affected location has high confidence, then the analysis phase managed to gather enough information with enough precision to know that the affected location is a true positive match. For low confidence locations, TAPIR either lacks precision or information to confidently determine if the location is a true positive or a true negative. In those cases, the missing information or missing precision is recovered by asking the user one or more questions. Our evaluation shows that JSFIX asks on average 2.8 questions per client, which means that significantly less effort is required to answer questions compared to the substantial workload of a completely manual update process. Another key observation is “without JSFIX, the client developer would instead be forced to understand the implications of every breaking change on the client code” [107]. In other words, the questions asked during the interactive phase are only a subset of the questions the client developer would implicitly ask himself during a manual update process.

In the third and final phase of JSFIX the (now true positive) affected locations are patched (Transformation phase in Figure 4.1). The code template, which accompanies

\[^3\]

TAPIR is designed to be what we call “practically sound”. It means that it is only sound under some assumptions that seem to hold in almost all real programs. We discuss the soundness of TAPIR in Section 9.6.
4.2. BREAKING CHANGE ADAPTATION FOR NODE.JS

the detection pattern that matched the affected location, is used as a blueprint for the required transformation. A code template is a small meta-program, which may contain AST reference variables that are interpolated with expressions from the client application in the transformation phase. The AST node corresponding to the affected location is replaced with the instantiated (interpolated) code template. JSFIX also updates the package.json file, such that the client uses the new version of the dependency.

Example 4 Let us consider how to write a semantic patch for patching the breaking change from Example 3. To write the detection pattern, we need to identify what part of the http-tools API that is affected by the breaking change and under which conditions. In this case, the breaking change affects the createConnection function. It affects all calls to createConnection where an options object is passed. If the options argument is mandatory, it will affect all calls and if it is not, then it will affect only those calls where options is passed, i.e., calls to createConnection with two arguments. For the sake of this example, let us assume that options is not mandatory. The detection pattern will then read:

call <http-tools>.createConnection [2,2]

It matches calls to the createConnection method on the module object loaded from http-tools. The [2,2] part is a call filter specifying that the call must take exactly 2 arguments.

To write the accompanying code template, we must consider the changes necessary to preserve the old behavior after applying the update. We can see from the changelog that options must be moved from the second to the third position. For the second argument, either undefined or 80 can be used. Since using 80 is arguably the more readable option, we choose this option, and the resulting code template becomes:

$callee($1, 80, $2)

Say the detection pattern has matched the call to createConnection from Example 3.

httpTools.createConnection('localhost', { timeout: 5000 })

Instantiating the template according to this affected location, the $callee AST reference specifies that JSFIX should insert the callee from the affected location (httpTools.createConnection). JSFIX should then insert a begin parenthesis, followed by $1, which refers to the first argument ('localhost') of the matched call. Next, a comma followed by the literal 80, followed by a comma again. Finally, $2 referring to the second argument ({ timeout:5000 }) of the affected location, i.e., the options object, is inserted together with an end parenthesis. The instantiated template is equivalent to the statement shown on line 7.

In our evaluation of JSFIX, we created a total of 298 semantic patches for handling almost all breaking changes in 12 major updates of top npm packages. Only a few breaking changes are not supported by JSFIX, either because the detection pattern
language is insufficiently expressive to identify the affected API, or because we could not write a correct patch using the code template language. For the latter case, many breaking changes fall into a category where no general patch is possible (discussed further in Section \[10.6\]).

Based on these 298 semantic patches, \texttt{JSFIX} was able to patch 82 of 89 clients, where their test suites were failing due to a major dependency update, such that their test suites went from failing to passing. We also generated pull requests based on \texttt{JSFIX} patches, of which 31 have been accepted so far.

As specified earlier, the interactive phase of \texttt{JSFIX} generates only 2.8 questions per client, and \texttt{JSFIX} uses an average of less than 3 seconds to update a client (including the analysis phase).

### 4.3 Discussion

The \texttt{TAPIR} and \texttt{JSFIX} tools provide both Node.js client and dependency developers with tooling that automates and streamlines the dependency update process. Without such tooling, the update process is more complicated for both dependency and client developers.

Dependency developers tend to be hesitant about introducing breaking changes since it may lead to unsatisfied clients. While this tendency is weaker in Node.js compared to other environments, dependency developers still strive to create backward-compatible updates \[13\]. For example, 96\% of updates are marked as minor or patch \[\text{143}\], and changelogs include statements like “Many internal use utilities like isArray are now hidden under rxjs/internal, they are implementation details and should not be used” \[\text{4}\] trying to justify the breaking changes. One problem with backward compatibility is that it reduces flexibility and increases the maintenance burden for dependency developers. For example, dependency developers may feel inclined to deprecate a function which they would ideally remove, or they may abstain from performing refactorings that would improve the maintainability and usability of the package. Since \texttt{JSFIX} reduces the workload required to deal with breaking changes, and reduces the probability of unaddressed breaking changes entering production, it may eventually help alleviate the hesitancy to introduce breaking changes, and thereby ease the maintenance burden for dependency developers.

For client developers, the benefits of \texttt{JSFIX} are more obvious. \texttt{JSFIX} addresses the three core hurdles of dependency management (breaking changes, understanding the implications of changes, and migration effort) as identified by Mirhosseini and Parnin \[\text{98}\]. It reduces the workload of updating dependencies and most likely also reduces the probability of not addressing all affected locations. It may also increase security since it makes it easier to keep dependencies up-to-date and thereby free of known vulnerabilities.

In this chapter, we have so far assumed that changelogs are complete. In other words, if the semantic patch creator (for example, the dependency developer) creates

\[\text{https://github.com/ReactiveX/rxjs/blob/master/CHANGELOG.md}\]
the semantic patches for an update based on information retrieved from the changelog, then the semantic patches cover all breaking changes introduced in the update. Of course, changelogs are rarely complete in practice. They may not provide sufficient information as in this lodash version 4 breaking change “Removed thisArg params from most methods because they were largely unused” that does not mention the specific set of methods affected. In other cases, some breaking changes are completely undocumented as evident from the 11 breaking changes we discovered in 5 of the 12 library updates while conducting the experiments for TAPIR. For the former of these cases, JSFIX is helpful since it forces the semantic patch writer to be explicit about the affected part of the API. For the latter, tools like NoRegrets may help find some of the undocumented breaking changes. However, as discussed in Chapter 7, more research is required on identifying breaking changes in Node.js dependency updates.

Somewhat related to the issue of detecting breaking changes is also the question of whether semantic patches can be inferred. Previous work has looked at using machine learning to learn patches for migrating an Android application between two versions of the Android operating system. The patches are learned from Android applications that have already migrated. Since we can view the Android operating system as a dependency of an Android app, a similar approach may work for other dependencies. Whether a similar approach can be used for Node.js dependencies is an open question for future work. It may also be possible to infer the detection patterns directly from source code differences between a pre-update and post-update version of a dependency. However, this challenge is not easily tackled since it is hard to statically compute the API of a Node.js application.

JSFIX has proven an effective tool for addressing breaking changes in the source code of client applications. However, there is not much that prevents JSFIX from evolving into a more generalized program transformation or refactoring tool. Certain features, which have proven unnecessary for breaking change adaptation, may be useful for more general transformations. For example, TAPIR currently only matches a single statement, which means that JSFIX cannot swap the order of two statements. There is also no way to only match a call if its argument is only sometimes null or undefined. Adding support for these features would require expanding the detection pattern language to include more constructs from the JavaScript language. Exactly how these extensions should be added remains an open question for future work.

---

[1] https://github.com/lodash/lodash/wiki/Changelog#v400
Chapter 5

Node.js Application Security

Security vulnerabilities is one of problems that receives the most attention in the Node.js community. As a case in point, the Snyk company that started in 2015 primarily focusing on vulnerability detection, has received more than $750 million in funding already.\(^1\) It is not surprising that this level of attention is put on application security: Security breaches can cost companies millions and damage their reputation irreversibly.\(^2\)

There are numerous research challenges related to security. In this chapter, we will focus on two problems that are extremely important for real-world Node.js application security.

The first challenge we consider in this chapter is not about any specific type of vulnerability. Rather we consider number 9 on the OWASP top 10 application security risks, which is about "Using Components with Known Vulnerabilities".\(^4\) This issue is especially prevalent in the Node.js ecosystem where applications tend to depend on tens if not hundreds of third-party packages. In fact, this problem is so pervasive that npm has a built-in tool called npm audit\(^5\) for warning application developers about known vulnerabilities in their applications. The npm audit tool belongs to a category of applications we call security scanners. A security scanner is a conceptually simple tool that compares a list of installed dependencies against a database containing information about known vulnerabilities. The user is notified if one of the installed dependencies has a known vulnerability. A security scanner may also provide suggestions for how to remove the vulnerability by showing the user how to upgrade to a newer non-vulnerable version of the affected dependency.

A big disadvantage of current security scanners is that their rate of false positives is high. Previous work has shown that the false positive rate may be as high as

\(^1\)https://www.crunchbase.com/organization/snyk/company_financials
\(^3\)https://www.barrons.com/articles/facebook-s-big-data-breach-could-cost-it-over-1-billion-1538417905
\(^4\)https://owasp.org/www-project-top-ten/
\(^5\)https://docs.npmjs.com/cli/v7/commands/npm-audit
73% \[42\]. A vulnerability in a dependency typically only affects a small part of the dependency. For example, a prototype pollution vulnerability in lodash prior to version 4.17.11 only affects the `merge`, `mergeWith`, and `defaultsDeep` functions, but the lodash API consists of hundreds of other functions.\[6\] The issue is illustrated in Figure 5.1, where the client \(c\) transitively depends on the vulnerable dependency \(d_2\), but the vulnerable function in \(d_2\) is not reachable from \(c\). A relevant question to ask is why are these false positives even a problem? If developers address all alarms, then clearly they will also address all of the true positives. However, because Node.js applications have so many dependencies, and vulnerabilities are relatively common, the sheer number of audit alarms quickly become overwhelming for developers. This phenomenon is often referred to as notification fatigue and is well-described in the literature \[98\]. The consequence of notification fatigue is that users ignore new alarms, which defeats the purpose of using a security scanner. For similar reasons, others have also argued that the Node.js community needs better vulnerability detection tools \[22, 84\]. In Section 5.2, we present JAM a call graph-based security scanner that reduces the number of false positives considerably. A more detailed description of JAM is provided in Chapter 12.

The second challenge is about detecting so-called injection vulnerabilities. This category of vulnerabilities encompasses all types of vulnerabilities where a malicious user can carefully create an input to an application that causes it to misbehave. Typical

\[6\]https://snyk.io/vuln/SNYK-JS-LODASH-73638
5.1. **EXISTING WORK**

Examples are SQL injection, regular expression denial of service (ReDoS), cross-site scripting (XSS), and directory traversal vulnerabilities. Vulnerabilities in this category can be generalized as shown in Figure 5.2. There is a place in the application where users can provide input. We call such a place a *source*. Examples of sources are HTTP request handlers, command-line input parsers, and WebSocket message handlers. Likewise, there is a usage of a potentially dangerous API in the application source code. We call such a usage a *sink*. Examples of sinks are SQL queries, regular expressions, file-related APIs, the eval function, and the child_process API for creating and managing child processes. If there is a dataflow path from a source to a sink (as in Figure 5.2), then there is a potential for a vulnerability. Often this potential can be removed by inserting a *sanitizer* on the path, which prevents data from flowing to the sink if it does not have an acceptable structure. For example, a sanitizer may remove paths containing directory traversals (..) before these paths are sent to a file read API.

Injection vulnerabilities are commonly uncovered using a taint analysis. The taint analysis looks for dataflow from sources to sinks and may sometimes try to filter dataflow that goes through a sanitizer to avoid false positive alarms. Like for most other analysis challenges, both static and dynamic taint analyses are possible. Dynamic analyses usually offer high precision at the cost of a low recall and the opposite is true for static analyses. When it comes to taint analyses, static analyses are often favored since a high recall is critical for finding as many vulnerabilities as possible. However, even the most advanced static analysis still cannot handle large JavaScript applications as explained in Section 2.1. In Section 5.3, we describe a new dynamic analysis TASER that infers taint specifications at the level of Node.js modules, which frees an existing static taint analysis from having to analyze these modules, and thereby increases its scalability. A more detailed description of TASER is provided in Chapter 11.

### 5.1 Existing Work

Node.js security has received attention from both academia, and several successful companies have been built with Node.js security as one of their core competences.

Several papers have considered specific security problems, e.g., prototype pollution [75, 88], command injection attacks [130], ReDoS attacks [129], and supply chain attacks [35]. Some papers have also looked at more general approaches for enhancing the security of Node.js applications [77, 105, 130].

Synode is a protection mechanism that, at runtime, prevents injection attacks targeting `eval` and `exec` [130]. It first uses a static analysis to compute templates modelling the structure of non-malicious inputs to these APIs. At runtime, dynamic checks will ensure that all values adhere to the templates. While Synode has been shown to provide good protection against attacks targeting `eval` and `exec`, it will unfortunately also, in rare cases, reject valid input. The runtime overhead of dynamic checks is also often not acceptable for applications running in production.
Nodest is a taint analysis built on top of the sound whole-program abstract interpreter TAJS [105]. Since TAJS currently does not scale to large Node.js programs, Nodest uses a heuristic where it only analyzes packages on-demand as it learns that they are relevant for some taint flow. Nodest is able to detect some injection vulnerabilities in real-world Node.js applications, and is much more scalable than the normal TAJS analysis. However, it is still a relatively slow analysis, and when the taint passes through modules which TAJS struggles to analyze, Nodest will also timeout.

Mininode is a static analysis for Node.js that uses a call graph to find unused code in an application and determine which built-in modules are required by the application [77]. It then both removes the unused code and restricts the require function such that unused built-in modules cannot be loaded. The goal of the approach is to limit the capabilities of an attacker who performs a code injection attack. For example, if Mininode has marked the fs module as unused, then Mininode will reject attacker injected code that uses the fs module, and thereby prevent the attacker from reaching the file system.

The Snyk company provides a security scanner for Node.js similar to npm audit. [https://snyk.io/](https://snyk.io/)
Snyk also includes their own vulnerability database and employs security researchers to constantly monitor the npm system for new vulnerabilities. Snyk has received more than $750 million in funding, and claims to have more than 2.2 million users. Some of their customers pay thousands of dollars per month for their service, showing that Node.js security is a top priority for many organizations.

Other companies that have received millions of dollars in funding include Debricked (another security scanner), Semmle (general-purpose static analysis with security as one of its core focus areas), and R2C (a lightweight static analysis focusing on detecting security vulnerabilities).

5.2 Security Scanning

We have developed JAM, a modular call graph analysis for Node.js applications. The primary application of JAM, which we will cover in this section, is a security scanner. Because the JAM security scanner only warns about usages of vulnerable APIs that are reachable from the entry point of the scanned application, the false positive rate is much lower than for classic security scanning. Whereas classic security scanners warn about all vulnerabilities in all dependencies (whether the vulnerabilities are reachable or not), the only source of false positives in JAM is spurious edges in the call graph. For the application illustrated in Figure 5.1 JAM does not warn the developer since the vulnerable API is unreachable in the call graph.

The JAM analysis is modular since the call graph of an application can be computed from its dependencies bottom-up. For example, if two applications both depend on lodash version 4.0.0, then JAM can precompute the call graph for lodash, and then start the call graph analysis for both applications with the lodash call graph precomputed. Our evaluation shows that JAM, which is already very fast using less than 4 seconds to compute a call graph for a full application on average, is more than 10 times faster if the call graph is precomputed for all dependencies.

For a call graph to be useful for security scanning, it should ideally be sound. An unsound graph may result in undetected security vulnerabilities, which leads to vulnerable applications entering production. However, as is generally well-known, creating sound and scalable static analyses for JavaScript is difficult (see Section 2.1). Instead, we aim for a practically sound analysis, i.e., an analysis, which, under a set of assumptions that hold in most real-world JavaScript applications, is sound. For example, the analysis ignores dynamic module loads and dynamic code generation (eval).

The key observation on which we have designed JAM is that Node.js modules can be viewed as independent components with only limited possibilities for cross-module communication. The JAM analysis first precomputes a module summary for
CHAPTER 5. NODE.JS APPLICATION SECURITY

each module in the application (including its dependencies) in a module summary construction phase, and then combines the module summaries to a call graph in a call graph construction phase. The call graph construction phase is entirely based on the module summaries, and does not need any access to the source code.

A module summary is comprised of three components: a function call summary, a function return summary, and an object property summary. All of these components are maps to sets of access paths.

Example 5  The following access path describes a call to $f$ on the object returned by require('lib'), e.g., the lib module object.

µ<lib>.f()

and this access path describes the function defined on line 42 in the file index.js.

Fun(index.js, 42)

The function call summary maps a function definition (identified by its source location) to a set of access paths corresponding to the call expressions in the body of the function. The return call summary maps a function definition to a set of access paths corresponding to the expressions appearing in return statements in the body of the function. Finally, the object property summary maps a property name to access paths corresponding to the expressions assigned to properties with that name. The module summaries are constructed using a lightweight field-based static alias and access path analysis. The field-based approach (as also used by the TAPIR analysis introduced in Chapter 4) ensures scalability of the analysis (see Section 4.1).

The module summaries from all modules in an application provide the required information to create the call graph for the application. The goal of the call graph construction algorithm is to resolve each call access path to a function definition. The algorithm will resolve each type of access path as one would intuitively expect. For example, for an access path of the form $x()()$, where we have a call to the value returned by $x$, the algorithm will first resolve the call to $x$, and then retrieve the access paths corresponding to its return values from its function return summary. These access paths are then resolved iteratively.

One key idea that allows the algorithm to scale well is that it uses the field information stored in the object property summaries to conservatively resolve method calls, when the origin of the receiver object is unknown. In other words, for a call $x.m()$: if the algorithm does not know where $x$ comes from, it will assume that all functions written to a property named $m$ in the application can be the callee. This approach works well for the average application since property names do not collide frequently.

Once the call graph has been constructed, it can be used for security scanning. Unlike a classic security scanner, where a vulnerability is identified by a dependency

---

12 A call graph here means a graph with a node for each function, and an edge from the node representing the function $f$ to the node representing the function $g$ if there is a call to $g$ in the body of $f$. 
name and a version range, the JAM security scanner also needs to know specifically which part of the dependency’s API is vulnerable. For this purpose, we also use an access path mechanism that resembles the access paths created by the access path analysis during module summary construction.

Example 6 If the `map` function of the lodash library is vulnerable, then the following access path is used to notify JAM about the vulnerability.

```javascript
<lodash>.map
```

JAM converts these access paths into concrete nodes in the call graph following an approach that resembles the procedure for resolving access paths during the call graph construction. The alternative approach of identifying vulnerable functions by specifying their source location in the package code does not scale well since the source location may change across vulnerable versions of the affected package.

Finally, once JAM has created both the call graph and identified the nodes corresponding the vulnerable functions, the security scan is performed. For this step, the user specifies the entry point in the application. Typically some “main module” of the application (see Section 5.4 for a discussion about this step). JAM then performs a reachability analysis from this entry point. For each vulnerable node that is reachable, JAM displays a warning to the user. Unlike classic security scanners, JAM also displays a stack trace corresponding to the sequence of calls required to reach the vulnerable function. Thereby, users of JAM can easily inspect warnings and ignore potential false positives.

We evaluated JAM on 12 randomly selected Node.js packages, where npm audit reports at least 1 warning for each. Manual inspection of the 34 alarms reported by npm audit for these packages showed that only 8 are true positives. The JAM analysis finds all 8 true positive alarms, but reports only 5 false positives thereby reducing the number of false positives by 81% compared to npm audit for these benchmarks. As mentioned previously, it takes on average less than 4 seconds for JAM to create a call graph for a Node.js application (including the module summary construction), and with precomputed call graphs for all dependencies, the average call graph construction time is just 0.14 seconds. The low running time of JAM is promising for its potential future use in various real-time IDE integrated tools, such as a code navigation or refactoring tool.

5.3 Taint Analysis Specifications

In the beginning of this chapter, we described the challenge of scaling static taint analyses to real-world applications. One solution is to simply ignore dependencies since they comprise more than 90% of the code in an average Node.js application, thus reducing the amount of code to be analyzed considerably [77]. An obvious problem with that approach is that it is very unsound, resulting in poor analysis results.

An alternative is to provide hand-written specifications for dependencies. The shape of these specifications depends on the specific analysis type. For a taint analysis,
they may describe how dataflow propagates through the dependency API, and which parts of its API that flow to sinks. It is typically much faster for a static analysis to gather relevant facts from the specification than it is to gather the same facts from the source code itself. Specifications are sometimes only written for the most important and commonly used dependencies and omitted for the remaining dependencies, leading to a still unsound but better performing analysis.

A huge disadvantage of using hand-written specifications is the manual labor required to both write and maintain the specifications. Especially in Node.js, where packages evolve quickly, maintaining the specifications is an insurmountable task. In this section, we describe the TASER analysis that infers taint specifications using a dynamic analysis. As mentioned previously, dynamic analyses generally suffer from a low recall due to their limited coverage. The key observation on which TASER is built is that we can mitigate the coverage issue, by creating a high-quality taint specification of a module based on executions from many different clients of that module. In particular, TASER builds on the same principle as the NoRegrets analyses (see Chapter 7 and 8), where the test suites of client applications are used to drive the analysis. As the test suites execute, TASER creates taint specifications recording two types of taint information. First, it records how data propagates from an entry point of a module to an exit point. We call a record of this type a propagation summary. It also records how data propagates from an entry point of a module to some existing known sink. We call a record of this type an additional sink.

Example 7 Consider the contrived Node.js module depicted below:

```javascript
1 // library 'util'
2 import { exec } from 'child_process';
3
4 module.exports.map = function(xs, f) {
5   const res = [];
6   for (const x of xs) {
7     res.push(f(x));
8   }
9   return res;
10 }
11
12 module.exports.loadPage = function(url, cb) {
13   exec('curl -L ' + url, cb);
14 }
```

If the `map` function is invoked by a client during the execution of its test suite, e.g., `map([1,2,3], (x) => 2 * x)`, then the following propagation summary is created:

```
(member * (parameter 0 (member map (root util))))
→
(member * (return (member map (root util))))
```

A propagation summary consists of two access paths: the entry access path first and the exit access path last. An access path refers to a place in the barrier between a module and its client, which we call a contact point. Intuitively, a propagation summary specifies that values that enter the module at the contact point of the entry
5.4. DISCUSSION

Path are used in the computation of values that exit the module at the contact point of the exit path. In the propagation summary above, the entry path refers to every property (member *) of the first parameter (parameter 0) in a call to map (member map) of the util module (root util). The exit path refers to every property of the return value of the map function. So, in summary, the propagation summary specifies that the properties of the first parameter are used to compute the properties on the return value as we would expect from a map function.

Similarly, if the loadPage function is invoked by a client during the execution of its test suite, e.g., loadPage('https://example.com', (x) => console.log(x)), then the following additional sink is created:

\[(\text{parameter 0 (member loadPage (root util))})\]

An additional sink only consists of a single access path. It specifies that values that enter the module through the contact point referenced by the access path are used in the computation of a value that flows to a known sink. For the example above, the first parameter of loadPage is used to construct a string that is passed to exec. Since exec is a known sink, an additional sink is created for this parameter.

Informally, the way the taint specifications are consumed by a static analysis is as follows: Let us consider the scenario of Figure 5.2 where the client is the module m1 that depends on m2 and m3. At some point, the analysis encounters the call from m1 to a function in m2 (corresponding to the arrow between the circles 1 and 2). Instead of analyzing the code of m2, the analysis considers its taint specification. A propagation summary specifies that data flowing from 1 to 2 flows back to 4. Because of this propagation summary, the analysis immediately learns about this dataflow without considering the call between 2 and 3. When the analysis later encounters the call between 6 and 7, a lookup in the taint specification of m3 reveals 7 is an additional sink. The analysis therefore only has to consider the code of m1 as it retrieves all the relevant facts about m2 and m3 from their taint specifications.

Our evaluation shows that the existing state-of-the-art static taint analysis LGTM detects 136 new alarms using taint specifications inferred by TASER. Manual investigation of 30 of these alarms shows that only 6 are false positives and that these are due to imprecision in the LGTM analysis.

5.4 Discussion

Reducing the false positive rate of security scanners and improving static taint analyses are two related problems. Provided with a perfect taint analysis, a security scanner is mostly obsolete since a taint analysis can uncover most security vulnerabilities. However, no perfect taint analysis exists for Node.js, and given the challenges with statically analyzing JavaScript, no such analysis is likely to exist any time soon.

13Remember that in JavaScript, arrays are just a special type of object where the property names are numeric indices.
CHAPTER 5. NODE.JS APPLICATION SECURITY

Instead, we should view security scanners and taint analyses as two types of tools that can coexist and assist each other. The taint analysis helps build the database of known vulnerabilities, and the security scanner reports to clients where a vulnerable API is used.

One challenge that security scanners currently do not handle well is warning about usages of APIs that are vulnerable by design. For example, the fs-extra\textsuperscript{14} package provides extensions to the built-in fs filesystem API. Most methods provided by fs-extra are technically vulnerable to directory traversals, i.e., attacks where a malicious user uses directory traversals (the .. path on Unix systems) to read/write/delete some file on the host system, which was not intended to be available to the user. The fs-extra package does not prevent users from providing paths with directory traversals to its remove method, so if user input flows to this method, a malicious user may be able to delete files on the host system. It would not be reasonable to prevent directory traversals in the arguments to fs-extra methods since there are legitimate use cases for wanting to use directory traversals. Yet, security scanners do not warn about the usage of fs-extra, which may give developers a false sense of security. Taint analyses are better suited for handling this type of vulnerability since they do not care if a sink was intended to be a sink or not. With the precision improvements compared to classic security scanning that JAM achieves, it should be feasible to also warn developers about usages of intentionally vulnerable APIs without overloading them with security warnings. Whether this hypothesis holds in practice is an open question for future work.

A limitation of the JAM security scanner is that it requires an entry point to the analyzed application. A warning is reported to the developer if a usage of a vulnerable API is reachable in the call graph starting from this entry point. For a typical application, the entry point is some module that serves as a “main function” of that application. However, some packages have multiple different entry points or no well-defined entry point. For example, a library typically does not have a single entry point. Instead, client applications can load one or more modules provided by the library, and call functions exposed by these modules. Since there is no way to precisely determine which modules and functions are private and which are public, JAM is currently not well-suited for scanning libraries. That is not a problem if JAM is used to scan every client since the library is only usable with clients. However, scanning every client is not realistic, so it would be beneficial if JAM could be used as a security scanner for libraries directly. It remains an open question whether there exists a satisfying solution to this problem. One possible solution is to use the membrane-based approach of NOREGRETS and TASER to identify functions that are part of a library’s public API, and then instantiate JAM using these functions as entry points.

The JAM analysis builds on a set of soundness assumptions. For example, that dynamic code generation and dynamic module loads are sparsely used in practice, and that dynamic property writes can safely be ignored since they are primarily used

\textsuperscript{14}https://www.npmjs.com/package/fs-extra
to copy properties that were previously written at some other place in the application. These assumptions hold to a degree where JAM at least did not produce any false positives during the evaluation. However, it remains an open question to precisely quantify how realistic these assumptions are for real-world Node.js applications. If it can be shown that the assumption about dynamic property writes is sound in practice, then it justifies using the field-based static analysis approach for other problem domains. It would also increase confidence in analyses like TAPIR and JSFIX (see Chapter 4) that also build on the field-based principle.

The TASER analysis is designed to infer propagation summaries and additional sinks. For future work, it is also relevant to consider inferring additional sources. For example, TASER might learn that whenever a user calls the `get` method on an express application object, then user input may flow into the callback passed to `get`. Such information may aid static taint analyses since it allows them to skip the analysis of the express library, and thereby scale better. Such an extension to TASER will require the TASER analysis to track dataflow from existing sources to exit points of the library.

express is a frequently used lightweight web framework for Node.js.
Chapter 6

Conclusions

The Node.js ecosystem has grown considerably in popularity since its initial release in 2009. It allows developers to write back-end applications in JavaScript, which is convenient for developers already familiar with front-end development in the browser. The millions of free Node.js packages stored in the npm package registry also accommodate a high degree of code reusability. In an average Node.js application, more than 90% of the source code comes from third-party packages. However, while using third-party code boosts productivity, it has amplified several maintenance-related problems that are less prevalent in other ecosystems.

In this thesis, we have identified and considered some of the primary maintenance challenges that face both application and dependency developers of the Node.js ecosystem. We consider breaking change detection, which is the challenge of identifying backward-incompatible changes in a dependency’s public API across two versions of the dependency. Breaking change adaptation is the challenge of identifying the source locations in client applications affected by breaking changes in a dependency update, and then patching these locations to restore compatibility with the dependency. Finally, with Node.js application security we consider both the challenge of warning application developers about known vulnerabilities in their packages and identifying new vulnerabilities using taint analysis.

In summary, the thesis makes the following contributions in the area of breaking change detection.

- The type regression testing (TRT) technique for identifying breaking changes in packages update, and an implementation in the form of the NOREGRETS tool (Chapter 7). TRT computes models of the package’s public API for both the pre-update and post-update version of the package using test suites of clients to drive a dynamic analysis. Certain types of differences in these models, called type regressions, indicate that breaking changes occurred. The breaking changes identified by TRT all belong to the category of type-related breaking changes, i.e., breaking changes affecting the types of the package’s public API, which is the most common type of breaking change [34, 96]. Our evaluation shows that NOREGRETS finds significantly more breaking changes than previous techniques.
• An improved version of TRT implemented in the NoREGRETS+ tool (Chapter 8). In this version, TRT only computes a pre-update model of the package. From this pre-update model, a test is then derived for finding type regressions in future versions of the package. Because running this test is considerably faster than generating a model, NoREGRETS+ is much faster than NoREGRETS. NoREGRETS+ also includes several improvements to the models and can derive models from more clients.

In the area of breaking change adaptation, the thesis makes the following contributions.

• The TAPIR analysis (Chapter 9) for identifying source locations in Node.js client applications affected by breaking changes introduced in dependency updates. TAPIR is supplied with a set of detection patterns, written in a domain-specific language, describing the affected parts of the dependency’s API. TAPIR uses a lightweight field-based static analysis to match the source code of the client against the detection patterns. The field-based principle allows the analysis to scale to large applications. It is designed to overapproximate the affected locations, and then mark each affected location as either a high confidence or a low confidence match. All false positives belong to the low confidence category so only those affected locations need to be manually vetted by the user.

• The JSFIX analysis (Chapter 10) that semi-automatically adapts affected source locations in client applications to preserve semantics in the presence of breaking changes during a dependency update. JSFIX is a 3 phased tool where the first analysis phase consists of the TAPIR analysis. In a second interactive phase, false positive affected locations are removed by asking the client developer questions that resolve the uncertainties from the analysis phase. Finally, in a third transformation phase, the affected locations are transformed. The transformations are described using code templates written in a domain-specific language. Our evaluation shows that JSFIX is able to patch most affected locations in real Node.js applications and that the patches are of a high enough quality for real developers to accept them as pull requests.

Finally, in the area of Node.js application security, the thesis makes the following contributions.

• The JAM analysis (Chapter 12) that performs security scanning on Node.js applications for detecting usages of dependencies with known vulnerabilities. Unlike previous security scanners, which simply scans for dependencies with known vulnerabilities, JAM first computes a call graph of the application, and then only reports a warning if the vulnerable part of the dependency API is reachable in this call graph. This improvement is crucial since three-quarters of alarms produced by existing security scanners are false positives, and notification fatigue may occur when the false positive rate is so overwhelmingly high. Like TAPIR, JAM also scales to large real-world applications since it uses
a lightweight field-based analysis to construct call graphs. While JAM is not theoretically sound, the sources of unsoundness are rare in practice, and our evaluation shows that JAM misses no real vulnerabilities.

- The TASER analysis (Chapter 11) that infers taint specifications of Node.js modules using a dynamic analysis driven by test suites of clients. These taint specifications can be used in place of the module source code when running a static taint analysis on an application that uses the module. Since gathering facts from the model is much faster than analyzing the module source code, the static taint analysis can scale to much larger applications. Our evaluation shows that the state-of-the-art static taint analysis LGTM can detect many new injection vulnerabilities when using taint specifications inferred by TASER.

These contributions significantly improve on breaking change detection, breaking change adaptation, and Node.js application security in the Node.js ecosystem. However, there are still numerous challenges left untackled.

For breaking change detection, the type regression testing and NOREGRETS tools only consider type-related breaking changes. While it has been shown that this type of breaking change is by far the most common [34, 96], behavioral breaking changes that change behavior while preserving types is not entirely uncommon either. Addressing behavioral changes completely is not viable using an automatic technique since it becomes a matter of checking program equivalence. However, it might be possible to adjust the NOREGRETS techniques to gather more fine-grained information from the client test suite executions, and thereby detect, at least, a subset of the behavioral changes. A disadvantage of type regression testing is that it, as a dynamic technique, relies on high coverage from client test suites to produce high-quality models. Using lightweight field-based analysis, as recently proven effective in both TAPIR and JAM, type regression testing may potentially be adapted to a static analysis.

In the context of breaking change adaptation, TAPIR and JSFIX currently rely on hand-written semantic patches. While the manual labor required to write semantic patches is low in relation to the accumulated time savings it brings to thousands of client application developers, convincing package developers to write semantic patches may still be difficult. An interesting challenge for future work is looking at automatic inference of either just detection patterns or detection patterns and code templates. A possible solution is to use client test suites to identify package API changes across different versions of the package (similar to NOREGRETS), and then derive detection patterns from this information.

For application security, JAM can currently only analyze applications (not libraries) since it requires an entry point in the call graph from which the reachability analysis is conducted. Finding the correct entry points of a library statically is challenging, but a membrane-based dynamic analysis, similar to NOREGRETS and TASER, may help with this issue. TASER currently only attempts to infer additional taint sinks and propagation summaries. However, external modules may also contain taint sources, which the static analysis should ideally know about. Future work may consider approaches for adding taint source inference to the TASER analysis.
Both JAM and TAPIR are unsound analyses which, under a set of assumptions that seem to hold for most real-world Node.js programs, produce sound results. For example, that dynamic property reads and writes are primarily used to copy properties that were previously assigned elsewhere, and can therefore safely be ignored in a field-based analysis [43]. Whether all of these assumptions hold is currently only supported by the results from our evaluations. It remains an interesting direction for future work to conduct a large-scale study to determine to which degree these assumptions are reasonable. If they are proven to hold for most real-world applications, using field-based analyses may also be an effective approach to create scalable static Node.js and JavaScript analyses for other problem domains.
Part II

Publications
Chapter 7

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

By Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Published in Proc. 32nd European Conference on Object-Oriented Programming (ECOOP), July 2018.

Abstract

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

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

Experimental results on 12 widely used libraries show that the technique can identify type-related breaking changes with high accuracy. It fully automatically classifies at least 90% of the updates correctly as either major or as minor or patch, and it detects 26 breaking changes among the minor and patch updates.
CHAPTER 7. TYPE REGRESSION TESTING TO DETECT BREAKING CHANGES IN NODE.JS LIBRARIES

7.1 Introduction

The world’s largest software repository, npm[1] hosts 475,000 Node.js JavaScript packages as of January 2018 and is used by millions of software developers. Most packages are libraries, and many are frequently updated, so versioning is essential to ensure that the components of a software system are compatible and up-to-date. The npm system encourages the use of semantic versioning[117], which distinguishes between patch, minor and major version updates: patch and minor are incremental updates that are not supposed to break any library client, whereas major version updates do not have any restrictions.

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

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

In statically typed programming languages like Java, even without versioning systems, the type system is helpful for detecting many situations where an upgrade of a library causes an application to break. For example, if the type signature of a library method has changed, the application code no longer compiles. Using access modifiers enables library developers to encapsulate private parts of the library, so that internal data representations and operations can be changed without affecting client code. Additionally, annotations about deprecated functionality signal to the application developer that attention is needed. Java’s binary compatibility conditions[50, Chapter 13] and tools like JAPICC[114] make it possible to detect type-related breaking changes in libraries even without involving the client code. In the world of JavaScript and npm, there is no static type system or compilation to binary code, so incompatibility issues are often not detected until runtime.

We distinguish between two main categories of breaking changes. Type-related breaking changes are modifications of a library that affect the presence or types of functions or other properties in the library interface. Such changes include renaming

7.1. INTRODUCTION

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

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

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

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

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

• We first present a preliminary experimental study on the prevalence and the kinds of breaking changes in the npm repository. We find that at least 5% of all packages have been affected by a breaking change in a minor or patch update of a dependency, and that a majority of these breaking changes are due to changes in the public API of the package.
• We propose type regression testing as a mechanism for leveraging the preexisting test suites of npm packages that depend on a JavaScript library of interest, to learn models of the library API and detect likely breaking changes.

• At the core of the type regression testing mechanism, library APIs are modeled using dynamic access paths and types that provide information about how the library and the clients interact. We define precisely how these models are obtained and compared. The possible breaking changes are identified by type regressions: changes in the type signatures of the library APIs that are incompatible with the mutual expectations of client and library developers.

• We report results from an experimental evaluation of the approach on 12 of the most depended upon libraries in the npm system, demonstrating that it can detect type-related breaking changes with high accuracy. Our implementation, named NOREGRETS, classifies at least 90% of the updates correctly as either major or as minor or patch, and it also successfully identifies 26 breaking changes among the minor and patch updates. Moreover, in cases where a likely breaking change is detected, the warning message produced by the tool pinpoints the involved part of the library, which aids diagnosis.

7.2 A Preliminary Experimental Study of Breaking Changes

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

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

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

For each dependency specified in the package.json file, we ran the test suites using each version of the dependency going from the oldest to the newest version satisfying the semantic versioning constraint. We removed versions which, although satisfying
7.2. A PRELIMINARY EXPERIMENTAL STUDY OF BREAKING CHANGES

the semantic versioning constraint, would never have been installed by npm in practice since a newer version was already available at the point where the constraint was created. An update was flagged as potentially breaking whenever the test suite of the client went from all tests passing to at least one test failing after applying the update. We did not count at the granularity of individual test cases, since it is technically difficult to do because of nested and parameterized tests.

This amounts to 75913 executions of test suites of the 4616 packages. Of these, 430 fail, meaning that breaking changes are detected. Whenever we had two versions of the same client appearing among the failures, we discarded the oldest one of them to avoid duplicates. We also discarded major updates and updates containing a pre-release identifier. A pre-release identifier is a hyphen followed by a tag added to the end of a version number used to indicate, for example, release candidates and beta versions. Updates containing a pre-release identifier should be treated as a major updates according to the semantic versioning principle. Furthermore, we chose to exclude 94 failures that could not be reproduced consistently due to flaky tests in the clients. With these filters, the total number of failures was reduced to 263 affecting 259 different clients. None of these 263 breaking changes should have appeared if semantic version had worked as intended and the library developers and the client developers had a common agreement on what is the public interface of the libraries!

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

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

As result we found that among the 263 failures, 176 were type-related, 37 were semantic, and the remaining 50 were marked as unknown. Some type-related breaking changes are easy to detect. In particular, sometimes simply attempting to load a library module fails, because it has been relocated or because its initialization code consistently crashes after an update. An example of the latter is the bug in the debug package mentioned in Section 7.1, which alone accounts for 101 of the failures. Even if we only count the occurrence of this bug once, we still find that at least 46% of the breaking changes are type-related.

This preliminary study motivates the need for tool support to detect breaking changes in Node.js libraries before the developers publish new versions of their libraries. It also justifies focusing on breaking changes that are type-related, which are more amenable to automated detection than the semantic ones.
CHAPTER 7. TYPE REGRESSION TESTING TO DETECT BREAKING CHANGES IN NODE.JS LIBRARIES

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

7.3 Motivating Example

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

```javascript
1 // lodash 3.2.0
2 function isIterateeCall(value, index, object) {
3     ....
4     return prereq && object[index] === value;
5 }

6 // lodash 3.3.0
7 function isIterateeCall(value, index, object) {
8     ....
9     var other = object[index];
10    return prereq && (value === value ? value === other : other !== other);
11 }
```

The variable `prereq` is computed in the same way in both versions. The important difference is that the `object[index]` property lookup is only executed when `prereq` is true in version 3.2.0, due to short-circuiting of `&&`, whereas it is always executed in version 3.3.0. If `index` is an object, then the `toString` method is implicitly called on `index` to coerce it to a string such that it can be used in the property lookup. However, it is possible that there is no `toString` method on `index` if, for example, the `index` value was created using `Object.create(null)`. Consequently, a type error exception will be thrown when the coercion is attempted.

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

12 // client

\(^4\)https://www.npmjs.com/package/dont-break
The `oBad` object is passed as the `index` parameter to `_.iterateeCall`, and a runtime type error appears when using `lodash` 3.3.0 but not when using the older version. The problem has been confirmed by the developers who fixed it in `lodash` 3.3.1 as mentioned in the changelog.

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

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

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

7.4 Overview

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

(1) **Public API Discovery** In the first phase, all the packages with tests that depend on the old library version are retrieved from npm. The type regression in the motivating example can be spotted by using any client whose tests cause the invocation of the `lodash merge` function, for example `strong-params` at version 0.7.0. Then, a public API model of both the old and the new library version is built using an instrumented interpreter that runs the tests of all the collected dependents. A model $\pi$, which we formally define in Section 7.5 is a map from dynamic access paths to types.

Intuitively, a dynamic access path is an abstraction of the sequence of actions that are performed to obtain a value. Types are an abstraction over values that is
more specific than the ordinary notion of runtime types in JavaScript. For example, the path \( p = \text{require}(lodash).\text{merge}(3)_4.\text{toString} \) exhibits the regression in the example from Section 7.3. This path denotes any value obtained by accessing the \text{toString} property of the third argument passed to the \text{merge} function when \text{merge} is invoked with four arguments. No type is associated with this path in the execution of the tests of \textit{strong-params} when using \textit{lodash} 3.2.0, but a function is observed when switching to \textit{lodash} 3.3.0. In the model \( \pi \) of the public API of \textit{lodash} 3.2.0, we have \( \pi(p) = \circ \), denoting the fact that no value has been observed for \( p \). In the model \( \pi' \) of the public API of \textit{lodash} 3.3.0, we instead have \( \pi'(p) = \text{function} \land \text{object} \), which means that the value is an instance of \text{function} and \text{object} (formally \( \pi'(p) \) is an intersection type).

This phase is fully detailed in Section 7.5.

(2) Type Regression Detection In the second phase, we compare the models \( \pi \) and \( \pi' \) obtained from the old and the new library version to report type regressions, which are indications of type-related breaking changes. Type regressions are detected by comparing \( \pi(p) \) and \( \pi'(p) \) for every dynamic access path \( p \), using a notion of subtyping, which we define in Section 7.5. In the example above, the type regression is reported because the type \( \text{function} \land \text{object} \) is not a supertype of \( \circ \). This phase is fully detailed in Section 7.6.

7.5 Public API Discovery

The workflow described in Section 7.3 requires two models of the public API to be built, one for the old and one for the new library version. It is important that the API models only capture the publicly available API, so that the comparison is not susceptible to changes in the private parts of the library where modifications are always allowed.

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

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

\[ \text{http://expressjs.com} \]
Dynamic Access Paths

Dynamic access paths, hereafter also shortened to paths, are used in the model to refer to values that are part of the interface between the client and the library. This mechanism ignores the syntactical structure of the library code and only considers how the library is being accessed dynamically by the client code.

**Definition 7.5.1** (Dynamic Access Path). A dynamic access path \( p \in \text{Path} \) is a possibly empty sequence of actions \( \alpha \) that abstractly represents a set of values, as defined by the grammar below. We indicate integers by the letters \( i, j \) and strings (library names and property names) by the letter \( n \).

\[
p ::= \varepsilon \mid \text{require}(n) \mid p \alpha \quad p \in \text{Path} \\
\alpha ::= .n \mid ()_i \mid \text{new}(i)_i \mid (j)_i \mid .* \\
\]

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

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

**Example 7.5.1.** The following listing shows a client that uses a hypothetical library \text{twice}, which has a single method \text{twice} that takes an object and returns an object with the property \( \text{res} \) that contains the doubled value of the \( x \) property of the argument object.

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


Consider which parts of the `twice` library are part of its public API. The value of the
property `b.res` must be part of the public API, because `b` is coming from the library
and `res` is accessed by the client on line 27. Intuitively, the client expects the `res`
property to be available on the return value of the call on line 26. This value is given
the path `require(twice).twice().res`. Likewise, the value of the property `a.x` is also part
of the public API with the path `require(twice).twice(1).x`. The `twice` method reads the
`x` property, so it should be present on the argument passed to `twice`. The value of the
property `a.y`, instead, is not part of the library API since it is never read by the `twice`
method (even though `a` is passed to `twice`). We describe in detail our mechanism for
distinguishing between public and private parts of the API in Section 7.5.

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

Types

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

**Definition 7.5.2 (Types).** A type \( t \in \text{Type} \) is a term in the following grammar:

\[
\begin{align*}
t &::= \circ | b \\
b &::= b \lor b' | b \land b' | \text{undefined} | \text{string} | \text{boolean} | \text{number} | \text{object} \\
& \quad | \text{function} | \text{array} | \text{set} | \text{map} | \text{event-emitter} | \text{stream} | \text{throws}
\end{align*}
\]

To simplify the presentation, we only show a representative subset of the Node.js
types. The type \( b \lor b' \) is a union type, while \( b \land b' \) is an intersection type. The type \( \circ \)
has a special use: If a path \( p \) is ascribed the type \( \circ \) in a model (i.e., \( \pi(p) = \circ \)), then
no values represented by the path have been observed during client test execution, therefore they are assumed not to be part of the public interface of the library. (Note that the special type \( \circ \) and the special path \( \varepsilon \) are both used for identifying the library API; \( \circ \) is used in the generated models, and \( \varepsilon \) is used for tagging values in our instrumented interpreter as explained in Section 7.5.) The special type \( \text{throws} \) is ascribed to functions that throw exceptions. Considering exceptions as a special type is unusual, but as we demonstrate in Section 7.7 it fits our setting well.

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

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

As mentioned in Section 7.4, we use subtyping to detect changes in the public API of a library that are breaking. The intuition, according to the Liskov substitution principle, is that subtyping should satisfy substitutability. Informally, if \( t' \) is a subtype of \( t \), denoted \( t' <: t \), then values of type \( t \) may be replaced with values of type \( t' \) without affecting the desirable properties of the program [89]. In our case, if a library method returns a value of type \( t \) in the old version and a value of type \( t' \) in the new version where \( t' <: t \), then behavioral subtyping tells us that there is no type-related breaking change.

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

**Definition 7.5.3 (Subtyping).** The subtyping relation \( <: \) among types is the relation given by the reflexive, transitive closure of the following rules.

\[
\begin{align*}
    b &: b \lor b' \\
    b' &: b \lor b' \\
    b \land b' &: b \\
    b \land b' &: b' \\
    b &: b'' \land b' <= b'' \\
    b' &: b'' \land b' <= b'' \\
    b'' &: b \\
    b'' &: b \\
    b &: b'' \land b'' \land b' <= b' \\
    \text{object} &: \text{undefined} \\
    t &: \circ
\end{align*}
\]

The rules for intersection and union types are standard. Note that the subtyping relation, because of union and intersection types, is not anti-symmetric [64]: for

example \(b \land b' <: b' <: b \land b'\) whenever \(b' <: b\). Consequently, types are not an order under the subtyping but only a preorder. Therefore, the least upper bound (also called join), which is needed for inference and checking, is not unique. For this reason, we implicitly work on the quotient order constructed from the preorder whenever we use the join operator \(\sqcup\). The rule \(\text{object} <: \text{undefined}\) is motivated by the following example, and \(t <: \circ\) is relevant for Example 7.6.1.

Example 7.5.2. Consider the patch update of the \textit{express} library from 3.0.1 to 3.0.2:

```javascript
28 // express 3.0.1
29 app.use = function(route, fn){
30 ...
31 return this._router.route.apply(this._router, args);
32 }
33 // express 3.0.2
34 app.use = function(route, fn){
35 ...
36 this._router.route.apply(this._router, args);
37 return this;
38 }
```

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

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

### Instrumented Interpreter

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

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

We describe the evaluation of a representative subset of JavaScript language constructs in A-normal form, i.e., where every subterm has been evaluated to a value \([45]\). For simplicity, we focus on a core language and do not explain the instrumentation of binary operators, constructor invocations, writes to local variables, exceptions, functions with multiple parameters, etc. The purpose of the following definitions is to explain the API discovery mechanism as an instrumentation of an existing JavaScript interpreter; in particular we are here not interested in all the details of JavaScript semantics, which are described elsewhere \([12, 53]\).

**Definition 7.5.4** (A-Normal Forms). The A-normal forms considered are the ones below; \(v\) denotes values, and \(c\) denotes constants.

\[
\begin{align*}
e & ::= c \quad \text{(literal constant)} & e \in \text{Exp} \\
x & \quad \text{(variable read)} \\
v[v] & \quad \text{(property access)} \\
v[v] = v & \quad \text{(property update)} \\
v(v) & \quad \text{(function call)}
\end{align*}
\]

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

The instrumented semantics \(\langle e, \rho, \sigma, \pi \rangle \Downarrow^I \langle v, \sigma', \pi' \rangle\) extends the semantics with the \(\pi, \pi'\) components where \(\pi'\) is the computed public API model. The initial model \(\pi\), used at the beginning of the program execution, associates the type \(\circ\) with each path \(p\). Similarly to information flow analyses, the instrumented interpreter uses tagged values \(v^p\) where \(p \in \text{Path}\) is a path, in place of the original values \([10]\). Values \(v^p\) and \(v^{p'}\) are indistinguishable by JavaScript programs for any \(p, p'\). Recall that \(\varepsilon\) is called the empty path and is intended to represent values that are not part of the public API.

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

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

**Example 7.5.3.** Suppose the library `foo` is used by a client as follows:

```javascript
39 var lib = require(foo);
40 var x = new lib.c();
41 lib.m(x);
```

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

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

The semantic rules for the instrumented interpreter are shown in [Figure 7.1](#). We use the notation `{p ↦→ t}` meaning the model that maps the path `p` to the type `t` and all other paths to `◦`. The join `π⊔π'` of two models `π, π'` is the pointwise join of the types for each path. By a slight abuse of notation, when joining the type `◦` with another type we assume that it behaves as the unit, i.e. if `π(p) = ◦` and `π'(p) = t` then `(π⊔π')(p) = t`.

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

At property accesses, the value of a property needs to be retrieved from a target object, and the model should be updated to reflect the type of the property accessed whenever the target object is part of the public library API. The rule `ACCESS` uses an auxiliary function `Lookup` to handle the actual lookup of the property, possibly walking the prototype chain, to retrieve the value `v_p` of the property `v_f` of the object `v_t` in the store `σ`. (We omit formal definitions of this and other auxiliary functions since they are not important for the tagging mechanism.) The rule distinguishes three cases for the path `p'_f` that is used as tag for the resulting value `v_f`. Assume the property access occurs in the client code. In the first case, `p_f` is nonempty and `p'_f` is empty, meaning that the object `v_f` comes from the library, and the library code that wrote the property value `v_f` to the object did not obtain that value from the client, so in this case the path `p'_f` is set to `p_f`α. The recorded action `α` is `.v_f` where `v_f` is the name of the property accessed, unless `v_f` is an array, in which case the special array access action `∗` is used. In the second case, `p_f` and `p'_f` are both nonempty. This means that we are accessing a property of a library object, and the library code that wrote the property value to the object obtained that value from the client, so we are in a situation similar
7.5. PUBLIC API DISCOVERY

\[\langle c, \rho, \sigma, \pi \rangle \Downarrow \langle c^f, \sigma, \pi \rangle\]

**VAR**

\[\rho(x) = v^p\]

\[\langle x, \rho, \sigma, \pi \rangle \Downarrow \langle v^p, \sigma, \pi \rangle\]

**ACCESS**

\[\text{Lookup}(v^p_t, v^p_f, \sigma) = v^p_r\]

\[p'_r = \begin{cases} p_t \alpha & \text{if } p_t \neq \epsilon \land p_r = \epsilon \\ \epsilon & \text{if } p_t \neq \epsilon \land p_r \neq \epsilon \\ p_r & \text{if } p_t = \epsilon \end{cases}\]

\[\pi' = \pi \sqcup \{p'_r \mapsto \text{type}(v_r)\}\]

\[\langle v^p_t[v^p_f], \rho, \sigma, \pi \rangle \Downarrow \langle v^p_t, \sigma, \pi' \rangle\]

**UPDATE**

\[p'_{a} = \begin{cases} \epsilon & \text{if } p_t \neq \epsilon \\ p_a & \text{if } p_t = \epsilon \end{cases}\]

\[\text{Update}(v^p_t, v^p_f, v^p_a, \sigma) = \sigma'\]

\[\langle v^p_t[v^p_f] = v^p_a, \rho, \sigma, \pi \rangle \Downarrow \langle v^p_a, \sigma', \pi' \rangle\]

**CALL**

\[p'_a = \begin{cases} p_f(1) & \text{if } p_f \neq \epsilon \land p_a = \epsilon \\ \epsilon & \text{if } p_f \neq \epsilon \land p_a \neq \epsilon \\ p_a & \text{if } p_f = \epsilon \end{cases}\]

\[\text{Call}(v^p_f, v^p_a, \sigma, \pi) = \langle v^p_r', \sigma', \pi' \rangle\]

\[p'_r = \begin{cases} \text{require}(v_a) & \text{if } v_f = l_{\text{require}} \\ p_f(1) & \text{if } v_f \neq l_{\text{require}} \land p_f \neq \epsilon \land p_r = \epsilon \\ \epsilon & \text{if } v_f \neq l_{\text{require}} \land p_f \neq \epsilon \land p_r \neq \epsilon \\ p_r & \text{if } v_f \neq l_{\text{require}} \land p_f = \epsilon \end{cases}\]

\[\pi'' = \pi' \sqcup \{p'_a \mapsto \text{type}(v_a)\} \sqcup \{p'_r \mapsto \text{type}(v_r)\}\]

\[\langle v^p_f[v^p_a], \rho, \sigma, \pi \rangle \Downarrow \langle v^p_r, \sigma', \pi'' \rangle\]

Figure 7.1: Semantic rules of the instrumented interpreter.
to Example [7.5.3] and we set $p'_r$ to $\varepsilon$ accordingly. In the third case, $p_t$ is empty, which means that the object $v_t$ comes from the client, so we simply keep the existing path $p_r$ for the resulting value. If the property access instead occurs in the library code, the reasoning is the same, but with the roles of client and library swapped. In either case, the $\pi$ component is updated to reflect that a value of type $\text{type}(v_r)$ has been observed for the path $p'_r$.

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

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

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

### 7.6 Type Regression Testing

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

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

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

$$
\begin{align*}
    t < t' & \quad \alpha = (j), \quad t' <;_{p'} t \\
    t <;_p t' & \quad \alpha \neq (j), \quad t <;_{p'} t'
\end{align*}
$$

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

**Definition 7.6.1 (Type Regression).** Let $\pi$ and $\pi'$ be models of an old and a new library version, respectively. A nonempty path $p$ exhibits a type regression whenever $\pi'(p) \not<;_p \pi(p)$.

**Example 7.6.1.** Continuing the example from Section 7.3, the path where the type regression is detected is $p = \text{require(lodash).merge(3).toString}$. In the old model $\pi(p) = \circ$, while in the new model $\pi'(p) = \text{function} \land \text{object}$. By definition, the path $p$ exhibits a type regression: $\text{function} \land \text{object} \not<;_p \circ$ because $\circ <; \text{function} \land \text{object}$ does not hold.

Note that if the opposite change was made to the library, such that toString is read in the old version of the library but not in the new version, then the rule $t <; \circ$ (see Definition 7.3.3) would prevent the type regression, as desired.

### 7.7 Evaluation

To evaluate the type regression testing technique, we developed a tool, **NOREGRETS**\footnote{node.js type regression tester}. The implementation consists of 1800 lines of TypeScript code and 6400 lines of
Scal code. The TypeScript part implements the instrumentation described in Section 7.5 using ES6 proxies. The Scala part fetches test suites for the npm packages from GitHub and post-processes the generated models to detect type regressions as described in Section 7.6.

Our primary hypothesis is that the type regression testing technique can help a developer decide if an update should be marked as either a major update or as a minor or patch update. We test this hypothesis by considering the tool as a binary classifier [127, 138], that is, a decision procedure that, given an input (in our case, a library update), returns one of two possible outcomes (breaking or non-breaking). Our secondary hypothesis is that the tool improves the dont-break approach for finding breaking changes, by amplifying the ability of the client test suites to reveal breaking changes, and by providing accurate and meaningful type regression reports. These hypotheses lead to the following research questions:

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

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

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

**Benchmarks** We randomly selected 12 among the most depended upon libraries from npm[10] as benchmarks, listed in Table 7.1. Since the development and release process of those libraries is usually under scrutiny of many skilled developers, they can be considered high-quality libraries: their changelogs are usually accurate, minor updates rarely introduce breaking changes, and major updates are usually reserved for those situations where breaking changes have been introduced.

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

Table 7.1 contains additional details on the selected libraries: the name and the major version of the library, the number of lines of code in the major version of the library, the number of minor/patch and major updates of the library considered, the number of clients with test suites, and the size of the public API (counted as the number of non-o paths in the inferred model, averaged over the different library versions). All the data collected comes from a snapshot of the npm repository taken in April 2017. To simplify the experiments we only consider clients that use the

[10]https://www.npmjs.com/browse/depended
Table 7.1: Node.js libraries used in the experimental evaluation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>Minor/Patch</th>
<th>Major</th>
<th>Client Test Suites</th>
<th>Model Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug 2.0.0</td>
<td>226</td>
<td>19</td>
<td>1</td>
<td>63</td>
<td>33</td>
</tr>
<tr>
<td>async 2.0.0</td>
<td>1682</td>
<td>5</td>
<td>0</td>
<td>64</td>
<td>3316</td>
</tr>
<tr>
<td>lodash 3.0.0</td>
<td>5225</td>
<td>16</td>
<td>1</td>
<td>42</td>
<td>4661</td>
</tr>
<tr>
<td>moment 2.0.0</td>
<td>1041</td>
<td>31</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>express 3.0.0</td>
<td>1011</td>
<td>95</td>
<td>1</td>
<td>4</td>
<td>54</td>
</tr>
<tr>
<td>chalk 1.0.0</td>
<td>169</td>
<td>4</td>
<td>0</td>
<td>93</td>
<td>105</td>
</tr>
<tr>
<td>bluebird 3.0.0</td>
<td>4827</td>
<td>29</td>
<td>0</td>
<td>16</td>
<td>503</td>
</tr>
<tr>
<td>react 15.0.0</td>
<td>41685</td>
<td>11</td>
<td>1</td>
<td>5</td>
<td>31</td>
</tr>
<tr>
<td>commander 2.0.0</td>
<td>370</td>
<td>12</td>
<td>0</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>request 2.0.0</td>
<td>626</td>
<td>98</td>
<td>0</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>body-parser 1.0.0</td>
<td>89</td>
<td>55</td>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>q 1.0.0</td>
<td>1152</td>
<td>9</td>
<td>1</td>
<td>8</td>
<td>277</td>
</tr>
</tbody>
</table>

Mocha testing framework, and to reduce noise we omit test suites that do not succeed consistently on major releases.

Our open-source implementation of NOREGRETS and all benchmarks and experimental data are available at [http://brics.dk/noregrets](http://brics.dk/noregrets).

RQ1 (accuracy as binary classifier)

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

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

RQ2 (comparison with the dont-break approach)

To answer the second research question, we manually inspected each type regression reported by NOREGRETS on minor and patch updates. To reduce the time spent, we focused on 5 benchmarks (debug, async, lodash, moment, and express).
Table 7.2: Breaking changes found.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Changelog</th>
<th>Test Failure</th>
<th>Synthetic Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug 2.0.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>async 2.0.0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>lodash 3.0.0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>moment 2.0.0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>express 3.0.0</td>
<td>0</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>26</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RQ2.1 We consider a type regression on a path as introducing a breaking change whenever (i) the developers reference the change in the changelog, (ii) the breaking change can be witnessed by a test failure of one of the selected clients, or (iii) we can construct a synthetic client that crashes because of the change. If none of these conditions are satisfied, then the type regression is classified as a false positive. The breaking changes in the second category are the only ones that can be identified by the dont-break approach. The synthetic clients in the third category may not be representative of typical clients, but they nevertheless witness breaking changes. Also, as demonstrated by our motivating example and by Example 7.7.1, breaking changes often cause problems exactly because clients use libraries in ways that the library developers did not anticipate.

Example 7.7.1. An example of a breaking change in the first category is the one introduced by moment 2.5.1 where the library started using the method `hasOwnProperty`. Unfortunately, the method is only available on non-host objects in older browsers. It took until version 2.8.2 for developers to realize this fact and fix the problem.11

The main results of our inspection of the reported regressions are shown in Table 7.2. The Changelog column contains the number of breaking changes that are confirmed by the changelog, and the Test Failure and Synthetic Client columns show how many are witnessed by a failing preexisting test or a synthetic client, respectively. Remarkably, NOREGRETS is able to find 26 breaking changes in minor updates of high-quality libraries. It is also notable that this is accomplished with few client tests; for example, the two breaking changes in moment are detected using only 5 client test suites.

Moreover, only 4 of the breaking changes that we have found could also be identified by a test failure, demonstrating that our approach amplifies client tests compared to the dont-break approach. Going back to Example 7.7.1, note that detecting the breaking change by the dont-break approach would not just require a test that triggers the invocation of `hasOwnProperty` on non-host objects, but the test should also be run in a specific browser. Instead, NOREGRETS reports a type regression indicating that the object passed to moment should have a function `hasOwnProperty`.

11https://github.com/moment/moment/pull/1874
RQ2.2 On the 5 benchmarks, \textsc{NoRegrets} reports a total of 168 type regressions across 167 library updates. By manually inspecting these 168 reports we find that 96 indicate actual breaking changes. (Some reports have the same root cause, which is how we arrive at a total of 26 breaking changes.) Thus, the number of false positives is acceptable: on average around one warning is reported per library update, and the majority of the warnings indicate actual breaking changes. Moreover, in the situations where multiple warnings are reported at a library update, we find that investigating the cause of one warning often quickly shows that other warnings have the same cause and therefore can be dismissed with little effort.

Example 7.7.2. In \textit{lodash} 3.10.0, the developers changed the behavior of the public method \texttt{isPlainObject} to use a different heuristic for recognizing so-called plain objects. Quoting from the implementation: “In most environments an object’s own properties are iterated before its inherited properties”. The method was changed accordingly to inspect all properties of the given object using a for-in loop, which causes 44 type regressions to be reported, one for each property of the objects being passed to \texttt{isPlainObject}. We classified this case as a false positive: the type regressions do not identify a breaking change since the property values are not used for anything but equality checks. We only had to inspect one library code location to understand that the other type regression reports had the same cause.

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

As already mentioned, 26 of our investigations resulted in the identification of an actual breaking change. To give some examples, the type regressions for 5 of these 26 breaking changes are listed in the first five rows of Table 7.3. The \textit{lodash} and \textit{moment} examples have already been explained in Section 7.3 and in Example 7.7.1. The \textit{debug} example, which shows the type regression for the breaking change mentioned in Section 7.1, and the \textit{async} example both involve the special \texttt{throws} type. In the \textit{async} example, the call to \texttt{require} returns an object in version 2.0.1 but throws an exception in version 2.1.0 because the module has been moved. Notice that none of type rules in Section 7.5 explicitly involve the \texttt{throws} type. Thereby, it is a type error if a function either starts throwing exceptions or stops throwing exceptions after a library update, which the \textit{async} example motivates well; moving a public module clearly affects the public API. Likewise in the \textit{debug} example, a spelling error results in an excepting being thrown when the \textit{debug} module is loaded, which breaks the public API. The \textit{express} example is similar to the \textit{moment} and \textit{lodash} examples: the property \texttt{readable} of the function argument is not accessed in version 3.14.0 of \textit{express}, but it is accessed in version 3.15.0, demonstrating that the type signature of the library function has changed.

\textsc{NoRegrets} also reported 72 type regressions that we categorized as false positives. Using the same approach as when investigating the true positives, we found that
CHAPTER 7. TYPE REGRESSION TESTING TO DETECT BREAKING CHANGES IN NODE.JS LIBRARIES

these false positives had 10 separate causes. Many of these were due to technical limitations of NoRegrets rather than of the type regression testing technique itself.

ES6 introduces default exports of objects and functions, where the default exports are automatically read by Node.js when reading properties of the required object, but from the perspective of dynamic access paths it still looks like the properties are read from the default object, e.g., \texttt{require(foo).default.p} instead of just \texttt{require(foo).p}. In the 2.1.2 patch update of the async library, it started to use default exports resulting in 3 reports about breaking changes that are false positives. These are only false positives because a fallback mechanism is included to handle old installations that do not support default exports. Therefore, it is fair to assume that an inconsiderate developer, who did not include a fallback mechanism, may still have benefited from these warnings. Another cause of false positives is that \texttt{hasOwnProperty} is not being instrumented, which is due to limitations of ES6 proxies.

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

Based on our experience with the dont-break approach in the preliminary study (Section 7.2) and the NoRegrets approach in the experiment for RQ2.1, we find that type regression reports greatly simplify the investigations of the breaking changes. The library developer does not need to understand what the client tests are doing, and can focus exclusively on the changes in the library’s codebase that have resulted in the changes of the public API. Since NoRegrets additionally records the call-stack at the point when a new type observation is created, dynamic access paths can easily be correlated with actions performed deep down in the private code of the library. A typical example is the one discussed in Section 7.3 where the path \texttt{require(lodash).merge(3).toString} shows that the coercion performed in the private \texttt{isIterateeCall} function is actually performed on the third argument of the \texttt{merge} function, and such information is not available if using the dont-break approach.

The last three rows of Table 7.3 show examples of type regressions found by NoRegrets in major updates. The first two examples show that the \texttt{pluck} function was removed from \texttt{lodash} in version 4.0.0 and that the \texttt{mime} property was removed from \texttt{express} in version 4.0.0. The last example, involving the \texttt{forEach} function of \texttt{lodash}, is a little more subtle. The \texttt{forEach} function takes 3 parameters in version 3.10.1, a collection, a callback function, which is applied to each element in the collection, and an object that is used as the \texttt{this} object in the callback. However, in version 4.0.0, the ability to set the \texttt{this} object of the callback is removed from the \texttt{forEach} function. After the update, reading a property of \texttt{this} in the callback causes a type error since \texttt{this} is now undefined.
Table 7.3: Type regression examples.

<table>
<thead>
<tr>
<th>Library Update</th>
<th>Path</th>
<th>Type Regression</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug 2.3.3 → 2.4.0</td>
<td>require(debug)!1</td>
<td>throws ( \text{object} \neq \text{function} \text{\object} )</td>
</tr>
<tr>
<td>async 2.0.1 → 2.1.0</td>
<td>require(async/asyncify)</td>
<td>throws ( \text{object} \neq \text{function} \text{\object} )</td>
</tr>
<tr>
<td>lodash 3.2.0 → 3.3.0</td>
<td>require(lodash).merge(3)_toString</td>
<td>( \text{\object} \neq \text{undefined} )</td>
</tr>
<tr>
<td>moment 3.5.0 → 3.5.1</td>
<td>require(moment)(1)_hasOwnProperty</td>
<td>( \text{\object} \neq \text{function} \text{\object} )</td>
</tr>
<tr>
<td>express 3.14.0 → 3.15.0</td>
<td>require(express)(1)[2].readable</td>
<td>( \text{\object} \neq \text{boolean} )</td>
</tr>
<tr>
<td>lodash 3.10.1 → 4.0.0</td>
<td>require(lodash).pluck</td>
<td>( \text{\object} \neq \text{function} \text{\object} )</td>
</tr>
<tr>
<td>express 3.21.2 → 4.0.0</td>
<td>require(express).mime</td>
<td>( \text{\object} \neq \text{\object} )</td>
</tr>
<tr>
<td>lodash 3.10.1 → 4.0.0</td>
<td>require(lodash).forEach(2)[3]</td>
<td>( \text{\object} \neq \text{throws} \text{\object} )</td>
</tr>
</tbody>
</table>

Discussion

The experimental evaluation is based on relatively few client test suites, which limits the fraction of the public APIs that are modeled and thereby reduces NOREGRETS’s ability to detect breaking changes. The libraries used in the experiments have thousands of clients, but our current implementation uses a fairly simple technique to locate clients and retrieve their test suites. In particular, it currently does not look for clients on GitHub but only uses npm. Also, tests are rarely published together with packages on npm, so NOREGRETS requires that the packages.json file for each client contains a URL to a GitHub repository and that this repository contains a git tag matching the client version that is required. Many clients do not include tags in their repositories, so we chose to discard those clients. In principle, this issue could be alleviated by comparing the source code in all the repository commits with the source of the client published on npm, but we leave that to future work. Additionally, as mentioned we exclude test suites that do not succeed consistently on major releases. This is a well-known problem: In a recent study of 373 popular JavaScript applications, 41 of the packages had tests that failed or froze, and 3 had build or deployment issues [38]. This problem could in principle be mitigated by using a more fine-grained approach where NOREGRETS looks at test failures at the granularity of single tests rather than entire test suites. Furthermore, the ES6 proxy mechanism used by NOREGRETS sometimes interferes with tests causing them to fail, so we have to disregard those too to avoid noise. This is a known problem with opaque ES6 proxies, which has already been addressed by Thiemann et al. [73]. We could similarly solve this problem by modifying Node.js such that proxies becomes transparent, but this is again a technical limitation of our current implementation that could be alleviated with further implementation work. Still, the experimental results obtained with our current proof-of-concept implementation suffice to demonstrate the potential of the type regression testing idea.

Another opportunity for improvements is to investigate extensions of our notion of types that could arguably enable NOREGRETS to better fit specific programming constructs. For example, Andreasen et al. [5] show that parametric polymorphism and recursive types could be beneficial to type-JavaScript functional programming constructs used in practice. Other possible extensions include representations of
CHAPTER 7. TYPE REGRESSION TESTING TO DETECT BREAKING CHANGES IN NODE.JS LIBRARIES

tuple types, polymorphic functions, and variadic parameters. Although these are theoretically interesting ideas, and could easily be implemented in NoRegrets, in our evaluation we have not yet encountered concrete use-cases to justify the technical effort to introduce them.

7.8 Related Work

Studies of npm Several experimental studies have investigated the npm repository [38, 81]. A study on JavaScript repositories showed that regression testing is a common practice, with an average of 78% of the packages having at least one test [38]. Two studies focus on the structure of the npm dependency network [74, 140]. In one of the studies, it is shown that the mean number of direct dependencies is 6, and that this number seems to be growing rapidly [140]. The same study also showed that the percentage of packages that are depended upon by other packages is only 27.5%, and a few popular packages are widely used by other packages. This should not be seen as a threat to the general applicability of our technique. If a package has no dependencies, then it matters little that the packages developer adheres to the semantic versioning principle. Another study has shown that the number of transitive dependencies is 10 times the number of direct dependencies and confirms that the number of dependencies of packages is growing exponentially, with a 60% increase in 2016 [74].

Studies of library updates and breaking changes Our preliminary study is the first published study of the prevalence of breaking changes in the npm repository. So far, research on breaking changes has focused on other ecosystems, mainly Java. An experimental study by Derr et al. [33], involving 203 developers, analyzed the reasons behind many Android applications using outdated libraries versions. More than 50% of the participants indicated that one of the reasons was to “prevent incompatibilities”. The authors developed a tool to compute the difference between the public API of two Java library versions, which they used to show that as many as 39% of minor and patch updates should have been flagged as major, which justifies the skepticism about the guarantees of semantic versioning expressed by the 203 developers.

The study from Raemakers et al. [118] addressed the use and misuse of semantic versioning in the Maven repository for Java packages. They conclude that “one third of all releases introduces at least one breaking change, and that this figure is the same for minor and major releases, indicating that version numbers do not provide developers with information in stability of interfaces”, showing that breaking changes are prevalent in Maven repositories. A similar study by Jezek et al. [68] on 109 Java open-source libraries discovered that every library introduces at least one breaking change of the public API in non-major updates.

Other studies are concerning the relation between library updates and breaking changes for JavaScript libraries. Mirhosseini and Parnin [98] showed that breaking changes, understanding the implications of changes, and migration effort are among the top concerns of JavaScript developers. A small user study among npm package
7.8. RELATED WORK

Maintainers showed that package updates are mostly coordinated by personal communications between developers [14]. A follow-up study, comparing 8 npm library developers to Eclipse and R/CRAN developers, showed that “npm developers were more willing than developers of other platforms to perform breaking changes in the name of progress” [15]. A study of the prevalence of client side vulnerabilities in web applications also showed that, like in the Maven system, many applications are using outdated libraries [84]. Zerouali et al. also found that many dependencies in the npm system are outdated due to too strict version constraints, and conclude that developers are reluctant to update dependencies since they want to avoid incompatible changes [143]. Wittern et al. [140] showed that 29% of all package.json version constraints specify a fixed version, while 68% of the constraints allow either all minor and patch updates or just all patch updates. The remaining 3% are free ranging constraints that also allow major updates.

**Type inference for dynamic languages** Dynamic inference of types for dynamic languages is a widely studied topic [1, 3, 5, 116, 123]. However, this paper is the first one to also distinguish between private and public parts of a library’s API. The use of runtime traces to learn types has already been exploited for Ruby [3], JavaScript [5, 123], and Dart [1]. One notable difference with our approach is that we do not ascribe types to syntactical elements of the programs, but instead to our notion of dynamic access paths. TypeDevil [116] uses a dynamic analysis to gather runtime type information where types are either primitive or records of types. Inconsistencies of the observed types are reported as potential bugs. Other forms of dynamic analysis for JavaScript are discussed in a recent survey [6].

**Detection of breaking changes** The only other tool that also aims at detecting breaking changes in npm package updates is the `dont-break` tool. Unfortunately, we were not able to make `dont-break` work properly, but we applied the same methodology in the preliminary study as discussed in Sections 7.2 and 7.7.

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

Java’s binary compatibility conditions [50, Chapter 13] and tools like JAPICC [114] make it possible to automatically detect type-related breaking changes in Java libraries. A disciplined set of guidelines for upgrading library releases have also been developed within the IBM’s System Object Model to guarantee binary compatibility [47].

---

1 https://greenkeeper.io/
7.9 Conclusion

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

We have implemented type regression testing in the tool NOREGRETS. Our evaluation shows that NOREGRETS is capable of detecting 26 breaking changes in 167 minor and patch updates of 5 high quality npm packages, and most of those breaking changes could not have been detected by existing techniques. We also find that NOREGRETS reports only a small number of false positives, and that the reported type regressions make it easy for the developer to determine the causes of the breaking changes. Furthermore, NOREGRETS correctly classifies at least 90% of the updates as either major or as minor or patch.
Chapter 8

Model-Based Testing of Breaking Changes in Node.js Libraries


Abstract

Semantic versioning is widely used by library developers to indicate whether updates contain changes that may break existing clients. Especially for dynamic languages like JavaScript, using semantic versioning correctly is known to be difficult, which often causes program failures and makes client developers reluctant to switch to new library versions.

The concept of type regression testing has recently been introduced as an automated mechanism to assist the JavaScript library developers. That mechanism is effective for detecting breaking changes in widely used libraries, but it suffers from scalability limitations that make it slow and also less useful for libraries that do not have many available clients.

This paper presents a model-based variant of type regression testing. Instead of comparing API models of a library before and after an update, it finds breaking changes by automatically generating tests from a reusable API model. Experiments show that this new approach significantly improves scalability: it runs faster, and it can find breaking changes in more libraries.

8.1 Introduction

An important challenge in software maintenance is how library developers can make updates without unintentionally breaking the existing clients of the libraries. Library developers commonly use the semantic versioning scheme to indicate if an update contains backward incompatible changes, also called breaking changes. With semantic
CHAPTER 8. MODEL-BASED TESTING OF BREAKING CHANGES IN
NODE.JS LIBRARIES

versioning, updates are marked as major when they are backward incompatible and
minor or patch otherwise. Generally, library developers should strive toward creating
backward compatible updates since clients often apply such updates automatically,
and instant rollout of updates can be critical for security fixes.

A considerable weakness of semantic versioning is that library developers mostly
rely on their own estimates when deciding which semantic versioning category an
update belongs to. Previous work has shown that developers often incorrectly classify
updates as minor or patch despite breaking changes [16, 46, 68, 96, 119]. This
is especially problematic for dynamically typed languages, like JavaScript, where
mismatches between the library and the client code are not detected until run-time.
JavaScript application programmers use libraries extensively; the npm repository
contains more than 750000 modules, mostly libraries, many of which have thousands
of daily downloads and are frequently updated.

A few tools exist for helping developers detect breaking changes before an update
is released to the clients. Examples include APIDiff, Clirr, and Revapi for Java [67],
the elm diff tool for elm, and NoRegrets [96] and dont-break for JavaScript. A
common property of these tools is that they compute the changes to the types of
the public API of the library for a given update, and then identify the changes that
may break clients. Although this approach can only detect type-related breaking
changes, not semantic changes that affect the library functionality but preserve the
types, previous work has shown that it is strong enough to catch most breaking changes
in practice [34, 96].

The existing techniques NoRegrets and dont-break for JavaScript require running
the test suites of a library’s clients to detect breaking changes when the library has
been updated. That approach has several disadvantages. First, installing the client test
suites may consume a considerable amount of storage, and running them often takes
significant time, although typically only a small part of those test suites is relevant
for the library. The dont-break tool simply reports breaking changes whenever a
client test fails with the updated version of the library. In contrast, NoRegrets uses a
technique called type regression testing. It performs a dynamic analysis of the client
test executions to infer models of the library API before and after the library update,
which leads to more errors being detected and to more actionable error reports for
the library developer. However, an important limitation of NoRegrets is that it can
only use those clients whose dependencies include the current version of the library.
For example, after a new major release of the library, the clients cannot be used by
NoRegrets until they have been updated to the new version. (We explain this technical
limitations of NoRegrets in more detail in section 8.7.) As a consequence, we find
that NoRegrets does not work well on libraries that only have few available clients.

In this paper, we present a new technique for finding breaking changes in Node.js
library updates, which does not suffer from these limitations of existing tools and

https://www.npmjs.com
https://package.elm-lang.org/
https://www.npmjs.com/package/dont-break
yet finds more breaking changes. The new technique is implemented in the tool \textsc{NoRegrets}+. It borrows the concept of dynamically computed API models introduced by \textsc{NoRegrets}, however, \textsc{NoRegrets}+ does not need to re-run all the client tests at every new release candidate of a library. Instead, from a single execution of the client tests it computes an API model that can be used for checking multiple subsequent updates of the library. It does so by using the model to guide a dynamic exploration of the library, while checking that the types of the values that flow between the library and the clients are compatible with the model.

Since \textsc{NoRegrets}+ only uses the client tests to generate the initial model, it avoids running the irrelevant code of the client tests in the checking phase, which makes it considerably faster than \textsc{NoRegrets}. The models are typically not very large, so they are also more easily stored than the whole set of clients. Additionally, this new approach is less sensitive to the versioning constraints in the client dependencies, which makes it useful even for libraries with relatively few clients.

In summary, this work makes the following contributions:

- We present a new model-based approach to type regression testing, designed to overcome the main practical limitations of the \textsc{NoRegrets} technique.

- We demonstrate by an experimental evaluation of our implementation \textsc{NoRegrets}+ that it is able to find more breaking changes than \textsc{NoRegrets}, an order of magnitude faster and requiring less space, and that the new approach works better for libraries where relatively few clients are available. Specifically, applying \textsc{NoRegrets}+ to a total of 1914 minor or patch updates of 25 Node.js libraries with varying numbers of clients detects 84 breaking changes, where \textsc{NoRegrets} in comparison only finds 28.

The tool \textsc{NoRegrets}+ is available at \url{https://brics.dk/noregrets/}.

8.2 Motivating Example

To illustrate the practical limitations of the existing techniques for detecting breaking changes in JavaScript libraries, consider the \texttt{big-integer} library for arbitrary precision integer arithmetic\footnote{\url{https://www.npmjs.com/package/big-integer}}.

Example 1 The patch update of \texttt{big-integer} from version 1.4.6 to version 1.4.7 introduced a new representation of integers that are small enough to fit in a primitive number, based on a new constructor named \texttt{SmallInteger}. The library internally uses a function \texttt{parseValue} to create a representation of a big integer from some user-supplied input, for example, a string representation of the integer in decimal form. The update contains the following changes:

\begin{verbatim}
1 //big-integer 1.4.6
2 function parseValue (v) {
3 ...
\end{verbatim}
The new `SmallInteger` constructor is used instead of `BigInteger` when the user-supplied value is small enough (lines 8-10). The `SmallInteger` constructor internally uses a primitive number to represent its value, which makes it more efficient than the array of numbers used by `BigInteger`. To make the underlying representation transparent to the users, the update also includes operations on `SmallInteger` objects mirroring the existing functionality of `BigInteger`. All the operations performed on these types are overloaded, for example, it is possible to seamlessly multiply a `SmallInteger` with a `BigInteger`. With this optimization, the `big-integer` library became much faster at processing smaller integers with the release of version 1.4.7.

However, the `valueOf` method behaves differently. On `BigInteger` it returns a best-effort conversion to a primitive number, while on `SmallInteger` it instead returns a reference to the `SmallInteger` object itself. Because of this difference, the update contains a breaking change that should not have been introduced in a patch update.

As mentioned in Section 8.1, the `dont-break` tool works by running the test suites of clients of the library before and after the update. One such client is the `deposit-iban` library, which contains the following code:

```javascript
const bigInt = require('big-integer');
export function isValidIban(iban) {
    ... const bban = ... // '620000000202102329006182700';
    const checkDigitBigInt = bigInt(bban);
    let checkDigitNumber = String(98 - checkDigitBigInt.mod(bigInt('97')));
    ... }
```

Before the upgrade of `big-integer`, in line 20 the `mod` method returns a `BigInteger` object whose `valueOf` method is invoked implicitly at the `-` operator. After the upgrade, `mod` instead returns a `SmallInteger` object with the different `valueOf` method, which returns the `SmallInteger` object instead of a primitive number. This means that at the `-` operator, JavaScript implicitly now also invokes `SmallInteger`'s `toString` method, which returns a string that in turn is coerced into a primitive number. The test

[https://www.npmjs.com/package/deposit-iban](https://www.npmjs.com/package/deposit-iban)
suite of deposit-iban does reach the isValidIban function and the different behavior in line 20. Nevertheless, all the tests still succeed with the broken version 1.4.7 of big-integer because the JavaScript runtime coerces the result of the mod call to the same primitive number as in version 1.4.6, even though the behavior of valueOf has changed. As a consequence, dont-break misses the breaking change.

In contrast, the NOREGREGS tool can detect this breaking change using deposit-iban’s test suite. The API model produced by NOREGRETS for big-integer version 1.4.6 will state that valueOf returns a number, whereas the model of version 1.4.7 will state that valueOf returns an object. Clearly, these two types are not interchangeable, so a breaking change is reported. However, NOREGRETS still runs all of deposit-iban’s test suite, which consists of 45 separate tests where only some use big-integer. That test suite was naturally developed to test the logic of deposit-iban rather than that of big-integer, so even for those tests that do use big-integer, most of the work is irrelevant from the perspective of determining whether the API of the big-integer library has changed.

With our new approach, NOREGRETS+, the test suites of the clients are still required to infer the initial API model of big-integer. However, once this initial model has been constructed, NOREGRETS+ checks the types of the library’s API by dynamically exploring it based on the information in the model. Specifically, for the aforementioned breaking change, all NOREGRETS+ needs to do is to load the big-integer library, call the mod function with the right arguments, call valueOf on the result, and assert that the type is compatible with the type in the model. Expressed as JavaScript code, this corresponds to executing the following test:

```javascript
23 const bigInt = require('big-integer');
24 assert(typeof (bigInt('620000000201202329006182700').mod(bigInt('97')).valueOf())
25 === "number")
```

With this approach there is no need for storing the entire deposit-iban client and its test suite (and similarly for all the other clients of big-integer), and the breaking change detection phase is much faster since the irrelevant work is avoided.

8.3 Overview

The purpose of NOREGRETS+ is to help Node.js library developers determine if a modification of a library results in breaking changes in the types of the library’s API.

The intended usage is as follows. First, the library developer uses the model generation phase of NOREGRETS+ that automatically fetches publicly available clients and their tests from GitHub, and then runs the tests and simultaneously records the interactions with the library to form a model of the library’s API. When the library developer is later ready to release an update, NOREGRETS+ is run in the type regression testing phase on the updated version of the library code, and a set of non-backward-

---

6Using the terminology introduced by Mezzetti et al. [96], a type regression is a change in the type signatures of the library API that is incompatible with the mutual expectations of the client and the
CHAPTER 8. MODEL-BASED TESTING OF BREAKING CHANGES IN

compatible differences in the API types is reported. If the set is empty, then the library developer can confidently mark the update as either minor or patch, since the API types of the library probably did not change. On the other hand, a nonempty set indicates changes to the API. If a manual inspection of the causes of the warnings produced by NoRegrets+ shows that the differences are unlikely to cause problems in practice, then the developer can go ahead and release the new code as a minor or patch update. If instead the warnings reveal more serious breaking changes, then the developer can either release the changes as a major update (and appropriately document the breaking changes), or, if the changes were unintended, choose to fix the library code and rerun the checking phase of NoRegrets+ to check that the type regressions are gone and that no new type regressions were introduced in the process. The checking phase is fast enough to be integrated into the library’s integration test suite, such that NoRegrets+ can be used continuously to check for type regressions during the development cycle.

Because of the dynamic nature of JavaScript, the API models produced by NoRegrets+ are of course not perfect, so the tool should be used as a supplement, not a substitute for the developer’s understanding of the library code. However, as shown in previous work [96] and in the experimental evaluation of NoRegrets+ (section 8.6), library developers often overlook breaking changes, and NoRegrets+ can catch many of them.

Example 2   Continuing Example[1] NoRegrets+ will first generate an API model for version 1.4.6 of big-integer, by running the test suite of deposit-iban while dynamically analyzing the interactions between the client and the library. The main constituent of an API model is a map from dynamic access paths to types, which we define formally in section 8.4. Intuitively, a dynamic access path (or path, for short) refers to the value that appears as result of performing a sequence of operations, for example, a call from the client to a library function, or a write within the library to an object originating from the client. Types include the ordinary JavaScript types, such as string and number, and also concrete primitive values. For example, the following paths expose the problem from Example[1]

\[
p_1 : \text{require(big-integer)} \xrightarrow{a} \text{ARG}_0 \\
p_2 : \text{require(big-integer)} \xrightarrow{b} \text{ARG}_0 \\
p_3 : \text{require(big-integer)}() \\
p_4 : \text{require(big-integer)}() \xrightarrow{a} \text{ARG}_0 \\
p_5 : \text{require(big-integer)}() \xrightarrow{a} \text{ARG}_0 \\
\]

A model that includes these paths (and many others) is generated when using the client test code shown in lines[14-22]. For line[18] when the client calls bigInt, the path \( p_1 \) refers to the value being read by the library function when accessing argument number 0, in this case the string ‘620000000202102329006182700’. For the second call to bigInt in line[20], \( p_2 \) similarly refers to the string ‘97’, and \( p_3 \) refers to the return value. The path \( p_4 \) refers to the value read by the mod library function when it reads
8.4 Phase I: Model Generation

We obtain realistic executions of the library of interest by leveraging the publicly available test suites of clients of the library. Running the test suites using program instrumentation with ES6 proxies, NoRegrets+ can monitor the flow of values between the clients and the library, which makes it possible to build a model of the public API of the library. Although this phase of NoRegrets+ is conceptually very close to NoRegrets, for completeness we briefly explain NoRegrets+’s notion of API models, and we point out the important differences.

API models An API model is a triple $(\pi, \sigma, \rho)$. We first explain $\pi$, which is map of the form $\pi : \text{Path} \to \text{Type}$ that associates types with elements of a library API. The set $\text{Path}$ consists of dynamic access paths, each being a sequence of actions, as described in the following grammar by $p$ and $\alpha$, respectively.

$$
p ::= \varepsilon \mid \text{require}(n) \mid p \alpha
$$

$$
\alpha ::= .n \mid ()^\kappa \mid \text{new}()^\kappa \mid \text{ARG}_j \mid .n \rightarrow
$$

Dynamic access paths can be thought of as references to elements of the library’s API. Each kind of action corresponds to a JavaScript operation, and a path corresponds to a sequence of operations. All paths begin with a $\text{require}(n)$ action, where $n$ is the name of a Node.js module. The $\text{require}(n)$ action can be followed by a sequence of property reads (denoted $n$ where $n$ is a property name), function and constructor applications (denoted $(\kappa)^\kappa$ and $\text{new}()^\kappa$ where $\kappa$ is explained below) and argument reads (denoted $\text{ARG}_j$ where $j$ indicates the zero-indexed position of the argument). We refer to Mezzetti et al. [96] for further description of these different kinds of actions that also appear in NoRegrets.

---

7Node.js libraries are loaded via the built-in $\text{require}$ function, as shown in section 8.1
CHAPTER 8. MODEL-BASED TESTING OF BREAKING CHANGES IN
NODE.JS LIBRARIES

In NoRegrets\(^+\), paths can additionally contain write actions (denoted \( \cdot n \rightarrow \), where \( n \) is the property being written), for modeling side-effects of the client and library functions in the API models. The \( \kappa \) label in the actions is used to distinguish calls to the same function.\(^8\) In an argument read action, \( \rightarrow \text{ARG}_j \), the label \( \kappa \) identifies the function call for which the argument is being read. The purpose of these modifications to the \textit{Path} mechanism becomes clear when we explain the type regression testing phase in section 8.5.

As an example, the \( \texttt{qs} \) library has a method named \texttt{parse} that in version 2.2.1 unintentionally writes to the \texttt{value} property of the object given as argument (this error is described in more detail in section 8.6). We can refer to the value being written using the path \( \texttt{require(qs).parse\rightarrowARG}_0\cdot\texttt{value} \rightarrow \). This path describes the following actions: load the library using \( \texttt{require('qs')} \), invoke its \texttt{parse} method (with an argument obtained via another part of the model), and then write to the \texttt{value} property of its argument. (The action label \( a \) is not relevant in this example.) The position of an action in the path shows whether it appears in client code or in library code: every argument read or write action corresponds to switching side, as indicated by the \( \rightarrow \) symbols. For this specific path, invoking \( \texttt{require('qs')} \) and accessing its \texttt{parse} method happens in client code, but reading the method argument and writing to its \texttt{value} property happens in library code. Since the property write happens on an object that comes from the client code, the value being written by the library is visible on the client side, as indicated by the last \( \rightarrow \) symbol. We say that a path is \textit{covariant} if the value described by the path flows from library to client, corresponding to an even number of \( \rightarrow \) symbols, and \textit{contravariant} in the opposite case.

A type \( t \in \text{Type} \) can be a standard JavaScript runtime type (number, boolean, object, etc.), a Node.js specific type like \texttt{stream} or \texttt{event-emitter}, or the default type \( \circ \) which we use for paths that do not belong to the library’s public API.

\[
t ::= \circ | \text{undefined} | \text{string} | \text{boolean} | \text{number} | \text{object} | \text{function} | \text{array} | \text{set} | \text{map} | \text{event-emitter} | \text{stream} | \text{throws} | \text{prim}
\]

Unlike in NoRegrets, a type can also be a JavaScript primitive value (denoted \textit{prim}), similar to how primitive values can be used as types in TypeScript.\(^9\) This extension is made because NoRegrets\(^+\) needs to reconstruct arguments for library functions in the type regression testing phase.\(^1\) We do not need traditional record types or function types, because the different properties of an object or parameters of a function are represented by different paths.

\(^8\) Because of the introduction of the \( \kappa \) labels, NoRegrets\(^+\) does not need to track the number of arguments at calls as done by NoRegrets. The array access abstraction, which is used in NoRegrets to model reads of array indices, is also not needed in NoRegrets\(^+\). Instead the property read action \( \cdot n \) is used where \( n \) is the array position being read.

\(^9\) For readers who are familiar with NoRegrets: NoRegrets\(^+\) does not use intersection types nor union types. NoRegrets uses intersection types to model JavaScript prototype chains, however, for NoRegrets\(^+\) to reconstruct the client arguments in tests, it must know exactly on which object in a

\(\text{https://www.npmjs.com/package/qs}\)

\(\text{https://www.typescriptlang.org/docs/handbook/advanced-types.html}\)
The second and third components of the model triple, $\sigma$ and $\rho$, are new to \texttt{NOREGRETS}+. The second component, $\sigma$, is a partial map $\sigma : \text{Path} \rightarrow \mathbb{N}$ that associates a unique number with each path $p$ where $\pi(p) \neq \varnothing$. It has the following property: for any two paths $p$ and $p'$, $\sigma(p) < \sigma(p')$ if and only if $p$ is encountered before $p'$ in the model generation phase described below. This information is needed by the testing phase to be able to invoke the library functions in the same order as the client on which the model is based, which we will later demonstrate in Example 4. For paths that are encountered multiple times during the model generation, we always use the observations from the first one.

The third component, $\rho$, is a binary relation $\rho \subseteq \text{Path} \times \text{Path}$. This relation is used to track how values flow from one path into another; for example, if a value returned by a library function call, represented by the path $p$, is later passed back to the library as an argument to a library function, where the argument is represented by the path $p'$, then $(p, p') \in \rho$.

**Model generation** To generate an API model $(\pi, \sigma, \rho)$ of a given library based on a collection of client test suites, \texttt{NOREGRETS}+ instruments the loaded module with ES6 proxies, runs the client test suites, and records the interactions between the library and the clients. The details of how this instrumentation works are explained by Mezzetti et al. [96], except for some straightforward adjustments to accommodate our new variant of API models.

One of the adjustments involves extending the $\pi$ component with a new path $p$. The type associated with $p$ now depends on the variance of $p$: if $p$ is contravariant and the value $v$ observed at $p$ is of a primitive type $t$, then $v$ is used as the type instead of $t$. For example, if the value is the string ‘foo’ and $p$ is contravariant then the type is ‘foo’, otherwise it is ‘string’. Thereby we ensure that the type regression testing phase of \texttt{NOREGRETS}+ has values available for library function arguments, and the model compression mechanism, which we will describe shortly, is not restricted by too specific types.

Another adjustment involves extending the $\rho$ relation whenever a value flows from one path to another. In Example 2, the value created by the $\texttt{BigInt}$ call in line 20 represented by the path $p_3$ flows into the argument of the $\texttt{mod}$ call represented by the path $p_4$, resulting in $(p_3, p_4)$ being added to $\rho$.

**Example 3** For the following simplistic library and client, \texttt{NOREGRETS}+ constructs the model shown in Figure 8.1.

```javascript
//library ‘lib’
module.exports.f = function (flag) {
  if (flag) { return { p : 42 }; }
  else { return {};
}
}
module.exports.g = function (x) {
  return 87;
}
```

Prototype chain a property resides, so extended paths such as $p$.\texttt{prototype}$ are used instead to refer to the prototype of a path $p$. Union types are used by \texttt{NOREGRETS} to model polymorphic functions, but are not needed in \texttt{NOREGRETS}+ since different calls are distinguished using the $\kappa$ labels.
### CHAPTER 8. MODEL-BASED TESTING OF BREAKING CHANGES IN NODE.JS LIBRARIES

<table>
<thead>
<tr>
<th>$p \in \text{Path}$</th>
<th>$\pi(p)$</th>
<th>$\sigma(p)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>require(lib)</td>
<td>object</td>
<td>1</td>
</tr>
<tr>
<td>require(lib).f</td>
<td>function</td>
<td>2</td>
</tr>
<tr>
<td>require(lib).f $\xrightarrow{\text{ARG}_0}$</td>
<td>true</td>
<td>3</td>
</tr>
<tr>
<td>require(lib).f($)</td>
<td>object</td>
<td>4</td>
</tr>
<tr>
<td>require(lib).f($) $\xrightarrow{\text{false}}$</td>
<td>false</td>
<td>5</td>
</tr>
<tr>
<td>require(lib).f($) $\xrightarrow{\text{true}}$</td>
<td>object</td>
<td>6</td>
</tr>
<tr>
<td>require(lib).g($)</td>
<td>number</td>
<td>7</td>
</tr>
<tr>
<td>require(lib).g($)</td>
<td>function</td>
<td>8</td>
</tr>
<tr>
<td>require(lib).g($) $\xrightarrow{\text{ARG}_0}$</td>
<td>object</td>
<td>9</td>
</tr>
<tr>
<td>other paths</td>
<td>$\odot$</td>
<td>undefined</td>
</tr>
</tbody>
</table>

$\rho = \{(\text{require(lib).f($) $\xrightarrow{\text{true}}$}, \text{require(lib).g($) $\xrightarrow{\text{ARG}_0}})\}$

#### Figure 8.1: API model for Example 3

```javascript
35 // client test suite
36 const lib = require('lib');
37 const o1 = lib.f(true);
38 const o2 = lib.f(false);
39 assert(o1.p === 42);
40 assert(lib.g(o2) === 87);
```

The client code loads the library `lib`, calls the `f` method with the argument `true` and stores the result in `o1`, then it calls `f` with the argument `false` and stores the result in `o2`. Finally it checks that `o1.p` is 42 and that `lib.g` called with `o2` as argument returns 87.

The paths and types of every operation taking place at the boundary of the client and the library are recorded in $\pi$: the read of the $f$, $p$, and $g$ properties, the two calls to $f$, the call to $g$, and finally the argument reads at the three calls. Notice how the two calls to $f$ are distinguished using the labels $a$ and $b$ in the paths. If we were to abstractly refer to both calls using just one path, then there would be no way to determine if the $p$ property should be present on the return value only when $f$ is called with the argument `true`, when it is called with the argument `false`, or in both cases. The fact that the argument passed to $g$ is the value returned by the call to $f$ in line 38 is indicated by the single entry in $\rho$.

### Model compression

The action label $\kappa$ is used to distinguish different calls to the same library function, as mentioned above. Because of these labels, models may become much larger than in NOREGRETS if the same library function is called many times. To mitigate this model size explosion problem, we add a simple compression mechanism. The idea is to only include one call of a polymorphic function for each of its possible return types since that suffices for full coverage of the types. We first identify pairs of paths $q = p\xrightarrow{\text{f}}$ and $q' = p\xrightarrow{\text{f}'}$ where $q$ and $q'$ are covariant paths representing two calls to the same function only separated by different labels, $a \neq a'$. If all paths $s = qr$ and $s' = q'r$, where $r$ is a sequence of actions that does not begin with an argument read action, the types are equal, i.e. $\pi(s) = \pi(s')$, and $s$ and $s'$...
8.5 Phase II: Type Regression Testing

The key novelty of NoRegrets+ is the use of model-based testing, based on the automatically generated models. When the library developer has obtained an API model of one version of a library and later wishes to release an update, NoRegrets+ uses the model to perform a dynamic exploration of the updated library while testing for type regressions relative to the model.

The dynamic exploration consists of two primary steps:

1. For covariant paths \( p \) where \( \pi(p) \neq \circ \), NoRegrets+ executes the actions described by \( p \) and checks that the type of the resulting value is compatible with the type \( \pi(p) \) as explained below.

2. For contravariant paths \( p \) whose actions happen to be executed as a consequence of step 1, NoRegrets+ checks that \( \pi(p) \neq \circ \).

Intuitively, the first step corresponds to checking the types of the values that are passed by the library to its client. For example, a library method call that returned a string before the update should not return a number after the update. The second step corresponds to checking that the requirements of the values supplied by the client to the library are not strengthened in the update. For example, after the update, a library function should not read more properties of an object that has been supplied by the client.

If any of the checks performed in these two steps fail, then it is an indicator that the API of the library has changed in a way that could be breaking clients.

API exploration In the type regression testing phase, NoRegrets+ mimics clients by performing the computations corresponding to the actions of the covariant paths in the model. These computations sometimes require values from other paths, which is handled by a synthesis procedure described below.

Example 4 To call \( g \) in line 40 in Example 3, we first need to call \( f \) as done in line 38 since its return value is used as argument to \( g \).

It is common for paths to have shared prefixes, for example, all paths in Figure 8.1 have \( \text{require(lib)} \) as a prefix. For such paths, the value obtained for the prefix is reused for all of them, to ensure that potential side-effects in the library functions are handled correctly.

To accommodate these requirements, NoRegrets+ represents a model \( \pi \) as a tree \( \tau \). Every node \( x \) in \( \tau \) is a triple \( x = (p_x, C_x, v_x) \) consisting of a path \( p_x \in \text{Path} \), a set \( C_x \)
of child nodes, and a JavaScript value $v_x$ that is assigned when $x$ has been processed as explained below. The tree has one node for each path $p$ where $\pi(p) \neq \varnothing$. A node $x$ is child of $x'$ if $p_x = p_{x'} \alpha$ for some action $\alpha$. In the exploration of the API, NoRegrets$^+$ traverses $\tau$ starting at the root, and when a node $x$ has been processed, the resulting value is stored in the tree as $v_x$. A child is never processed until its parent has been processed. When NoRegrets$^+$ has to choose between two nodes $x$ and $x'$ to process next, it picks $x$ if $\sigma(p_x) < \sigma(p_{x'})$. Thereby the nodes are processed in the same order as they were added to $\pi$ in the model generation phase.

In the process of exploring the API, NoRegrets$^+$ needs to convert actions into their corresponding JavaScript operations. To process a node $x$ whose parent is $x'$, NoRegrets$^+$ performs a pattern match of $p_x$ and executes the associated operations:

- **require($n$):** Load the module by calling `require($n$).
- **$p.n$:** Fetch the value $v_{x'}$ (corresponding to $p$) and read its $n$ property.
- **$p()$:** First, fetch the value $v_{x'}$, which is the function to be called. Next, construct the arguments. Each argument has its own node $x'$ whose path is $p_x = p \xrightarrow{\text{ARG}}$, which is a child of $x'$. The argument at position $i$ is constructed by invoking the synthesis procedure described below for the node $x'$. Finally, invoke $v_{x'}$ with the synthesized arguments to obtain the result value.
- **$p\cdot n\rightarrow$:** Invoke the synthesis procedure for $x$ to produce a value, and then write that value to the property $n$ of $v_{x'}$.

Paths ending in argument read actions are handled by the synthesis procedure described next.

**Synthesis of values** The synthesis procedure is used above to construct arguments for library function calls and to construct values for writes to library objects. The procedure is invoked with a node $x$ as argument. If there exists a node $x'$ such that $(p_{x'}, p_x) \in \rho$ then the desired value originates from an earlier interaction with the library represented by a path $p_{x'}$, so the value $v_{x'}$ is returned. Otherwise, we proceed according to the type $\pi(p_x)$ of $x$:

- If the type is a primitive value then that value is returned.
- If the type is `object` or one of the Node.js-specific types, then NoRegrets$^+$ creates a new empty object and wraps it in a proxy object, which is then returned. The purpose of the proxy is twofold. If the proxy is later used as an argument to a function, then that function might read one of its properties, $q$, in which case the proxy looks for a node $x'$ where $p_q = p_{x'q}$ among the children of $x$. If $x'$ is found, then the proxy recursively invokes the synthesis procedure with argument $x'$. Thereby, the properties of object arguments are constructed by need. If no node
is found, then the proxy reports a type regression indicating that the library is now trying to read a property that it did not previously read, cf. step 2. Writes by the library to the proxy are handled similarly to calls from the library to client functions, as described next.

- If the type is function then \texttt{NoRegrets}+ creates a new function \( f \) that behaves as follows when called. If \( x \) has a child \( x' \) in \( \tau \) such that \( p_{x'} = p_x \rightarrow \text{ARG} \), i.e., that path ends in a call action, then a value \( v_{x'} \) for \( x' \) is obtained by a recursive call to the synthesis procedure. This value is then used as the return value of \( f \). Furthermore, the API exploration mechanism described above is invoked recursively on each argument passed to \( f \). For each argument at position \( i \), the API exploration checks that it recursively satisfies the type of \( x_i \) where \( p_{x_i} = p_x \rightarrow \text{ARG} \). On the other hand, if no child of \( x \) with a call action is found, then \texttt{NoRegrets}+ reports a type regression to indicate that a previously uncalled callback is now being called, cf. step 2. The \( f \) function is also wrapped in a proxy since functions can also have properties, which may later be read if \( f \) is used as an argument.

**Type checking** During the API exploration, \texttt{NoRegrets}+ checks type compatibility of the values obtained for the covariant paths, as mentioned above for step 1. If \( v \) is the value obtained through the application of the actions of the covariant path \( p \), then \( v \) must satisfy \( \text{type}(v) <: \pi(p) \) where \( \text{type}(v) \) denotes the type of \( v \). A violation of this property indicates a breaking change in the library’s API at \( p \). The subtyping relation \( <: \) expresses which type changes are permitted. In particular, functions are subtypes of objects, i.e. \texttt{function} \( <: \texttt{object} \), since JavaScript functions are basically callable objects. We also define \( t <: o \), meaning that everything is a subtype of \( o \), thereby permitting clients to read additional properties of library supplied objects. The subtype relation additionally includes a few rules stating that some of the Node.js specific types are subtypes of \texttt{object} and/or \texttt{function}.

For the contravariant paths in step 2, we do not use \( <: \) but simply check \( \pi(p) \neq o \) as explained above, because the values represented by such paths are generated by \texttt{NoRegrets}+, not by the library.

### 8.6 Evaluation

As explained in section 8.1, the overall goal of \texttt{NoRegrets}+ is to mitigate the scalability issues of \texttt{NoRegrets}. To assess how well \texttt{NoRegrets}+ reaches this goal, we conducted an experiment designed to answer the following research questions.

**RQ1** How many breaking changes does \texttt{NoRegrets}+ detect compared to \texttt{NoRegrets} in widely used Node.js libraries, specifically those used in the evaluation of \texttt{NoRegrets} [96 Section 7]?

**RQ2** How much faster is \texttt{NoRegrets}+, and how much space does it require compared to \texttt{NoRegrets} when testing for breaking changes in a library update?
CHAPTER 8. MODEL-BASED TESTING OF BREAKING CHANGES IN NODE.js LIBRARIES

RQ3 Can NoRegrets+ find breaking changes in libraries with fewer clients compared to NoRegrets?

We omit a direct comparison with dont-break as it finds strictly fewer breaking changes than NoRegrets [96].
Table 8.1: Experimental comparison of NoRegrets+ vs. NoRegrets.

| Benchmark          | NoRegrets+ |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--------------------|------------|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Experimental Setup

We sampled 25 npm packages from three segments of the npm repository as listed in Table 8.1. The first five packages are among the top 10 most depended upon npm packages and are also the packages used in the evaluation of NOREGRETS. Then we have a set of 10 packages sampled around the top 100 most depended upon packages, and a set of 10 packages sampled around the top 1000 most depended upon package. The less depended upon packages have fewer available clients with test suites, which both NOREGRETS and NOREGRETS$^+$ need for API model generation, but all of the packages are widely used. Most of them have more than 100000 weekly downloads, so breaking changes in non-major updates can have severe consequences. We skipped packages whose newest version was less than 1.0.0 since such packages are not required to follow semantic versioning. We also skipped very small packages with trivial APIs, such as is-stream and make-dir, since their update rate is low and their APIs are unlikely to change.

We selected the first major version of every package and applied NOREGRETS$^+$ and NOREGRETS to every patch and minor update up to the newest version (as of January 2019). For reasons discussed in section 8.7, NOREGRETS$^+$ is able to use more clients than NOREGRETS when generating API models, however, when comparing the execution times of the two tools, we constrained NOREGRETS$^+$ to use the same set of clients as NOREGRETS to ensure a fair comparison. Because finding clients for many libraries is a time consuming process, we limited the client retrieval phase to consider at most 2000 packages. For NOREGRETS, we built the API model only for the first version, and then reused this model in the test of every update. The execution time of NOREGRETS$^+$ is measured as the time it takes to execute the type regression testing phase, whereas for NOREGRETS it is the time it takes to generate the post-update model and compare it with the pre-update model. In both cases, this reflects the work done when testing a new update of a library for breaking changes. The time required to generate a model is the same for the two tools.

For every type regression reported by the two tools at minor or patch updates, we manually inspected the type regression to identify its cause and determine if it is an actual breaking change (meaning that the type of some element of the library API has changed) or a false positive. It is common for one breaking change to result in multiple type regression warnings, for example, if the return type changes for a function with many call actions then a type regression is reported for every call. Such related regressions are easy to identify by their common structure, so we group them and only count them as one breaking change.

Results and Discussion

We present the results of running NOREGRETS and NOREGRETS$^+$ on the 25 benchmarks in Table 8.1. The columns contain left to right: the benchmark name and the major version on which the testing was started, lines of code in the initial version excluding tests, total number of direct dependents in npm, numbers of minor and patch updates,
8.6. EVALUATION

the number of clients found by the client detection phase of \textsc{NoRegrets}+, the average \textsc{NoRegrets}+ model size per client, the statement coverage of \textsc{NoRegrets}+ in the initial benchmark version, the number of breaking changes (BC) found by \textsc{NoRegrets}+, the number of clients found by the client detection phase of \textsc{NoRegrets}, the average \textsc{NoRegrets} client size, the statement coverage of \textsc{NoRegrets} in the initial benchmark version, the number of breaking changes found by \textsc{NoRegrets}, and the average speed-up ratio of \textsc{NoRegrets}+ compared to \textsc{NoRegrets}. For both tools, the reported numbers of breaking changes are only counting true positives, and excluding duplicates with same root cause as explained above.

**RQ 1** Looking at the first 11 rows, which are the benchmarks where \textsc{NoRegrets} has a non-empty set of usable clients, we see that \textsc{NoRegrets}+ finds at least as many breaking changes as \textsc{NoRegrets} for all benchmarks apart from mime and \textsc{express}. \textsc{NoRegrets}+ detects 84 breaking changes, whereas \textsc{NoRegrets} only detects 28.

The breaking changes found by \textsc{NoRegrets}+ include 11 of those found by \textsc{NoRegrets}. There are two reasons why the \textsc{NoRegrets} breaking changes sometimes go undetected by \textsc{NoRegrets}+. First, the clients used by \textsc{NoRegrets} are not necessarily a subset of the clients used by \textsc{NoRegrets}+. The reason is that \textsc{NoRegrets}+ will always use the newest version of a client that has the library as a dependency since it is more likely to utilize more of the library than earlier versions, however, for reasons we describe in section 8.7, \textsc{NoRegrets}+ will always pick a version of the client that satisfies the pre-update version constraint. Second, for some benchmarks \textsc{NoRegrets}+ is not able to synthesize values with enough precision to faithfully reconstruct the library-client interaction on which the model is based. In our experiments, this situation occurs because the model generation phase of \textsc{NoRegrets}+ uses ES6 proxies to monitor the interaction between the client and the library, but some values do not tolerate proxification well. For example, \textsc{ServerResponse} objects, which are commonly used with the HTTP library of Node.js, will crash Node.js if they are wrapped in proxies at certain places in the HTTP library. Therefore, \textsc{NoRegrets}+ must avoid using proxies on such objects, which means that their exact structure cannot by synthesized by \textsc{NoRegrets}+ in the checking phase, so \textsc{NoRegrets}+ has to use default values instead. This problem is especially prevalent in the \textsc{express} benchmark since it uses the HTTP library of Node.js extensively. (With a further implementation effort it might be possible to mitigate such problems; we plan to investigate this in future work.)

In addition to the 84 breaking changes detected by \textsc{NoRegrets}+, the tool emitted 4 false positives (not shown in the table). False positives may appear due to, for example, the issues with the proxy mechanism described earlier. Some of the correctly detected breaking changes are of course more serious than others; we show some examples as case studies below.

In summary, \textsc{NoRegrets}+ successfully detects more than twice as many breaking changes compared to \textsc{NoRegrets}.

**RQ 2** Looking at the speed-up column of Table 8.1 we see that \textsc{NoRegrets}+ on average runs the type regression testing phase 25x faster than \textsc{NoRegrets} generates
and checks the post-update model. For some libraries, for example the debug library, \textsc{NoRegrets}+ is 38x faster than \textsc{NoRegrets}, whereas for lodash the speed-up is only 1.46x. The relatively large difference in the speed-ups is explained by various factors, for example, how much irrelevant code (non-library code) is run by the client tests.

The actual time it takes to check an update for type regressions naturally depends on the size of the generated model and the complexity of the client test suites. The mean time it takes \textsc{NoRegrets} to check an update on the 11 benchmarks where clients are available is 96 seconds, compared to only 15 seconds for \textsc{NoRegrets}+. Excluding the outliers lodash and async, \textsc{NoRegrets}+ checks each update in less than 6 seconds.

The numbers also show that \textsc{NoRegrets}+ requires substantially less space than \textsc{NoRegrets}. The average size of a library model produced by \textsc{NoRegrets}+ is 367 kB per client used for the model construction, whereas \textsc{NoRegrets} requires 82x as much space.

In summary, \textsc{NoRegrets}+ is more than an order of magnitude faster than \textsc{NoRegrets} when testing a library update for breaking changes, and it requires substantially less space to run, which makes it feasible to use \textsc{NoRegrets}+ in library integration test suites.

\textbf{RQ 3} For the second segment of the benchmarks (i.e., the libraries sampled around top 100), \textsc{NoRegrets}+ finds breaking changes in 7 out of 10 benchmarks, and for the third segment (i.e., the libraries sampled around top 1000) it finds breaking changes in 6 out of 10 benchmarks. In comparison, \textsc{NoRegrets} only finds breaking changes in 2 of the benchmarks from the second segment and in none of the benchmarks in the third segment. This shows that \textsc{NoRegrets}+ scales much better for libraries with fewer clients.

For most libraries where \textsc{NoRegrets}+ finds no type regressions, the generated tests cover on average 50% of the statements in the library, which provides some indication that those updates are in fact non-breaking. One exception is react-onclickoutside where all of the models generated by \textsc{NoRegrets}+ are empty. That package is a plugin for the browser UI library react, which means that it is unlikely that any clients have automated tests that use it.

\textbf{Case studies} To give some insight into the nature of the breaking changes that \textsc{NoRegrets}+ can detect, we describe some representative examples.

\textit{Example 5} The qs package is a library for parsing query strings. As an example, \texttt{qs.parse("p=foo")} returns the object \{\texttt{p : "foo"}\}. A special feature of the package is that it supports parsing of objects where some on the properties are query strings that are parsed recursively, for example, \texttt{qs.parse({‘a[b]’: ‘c’})} returns the nested objects\{\texttt{a : \{b : "c" \}}\}.

In the update of qs to version 2.2.1, a mistake was introduced that resulted in the parse function sometimes overwriting existing properties on object arguments. This mistake is revealed by \textsc{NoRegrets}+ through a type regression on the path

\[ p = \text{require(qs).parse} \rightarrow \text{ARG0} \rightarrow \text{value} \rightarrow \]
where \( \pi(p) = o \) but a value of type `string` is written in version 2.2.1. For most cases this overwrite in benign because `parse` overwrites the property with its existing value, however, specifically for buffers, `parse` writes the result of calling `toString` on the buffer.

A well-known problem with semantic versioning is that it requires a specification of the library’s API\( ^{12} \) typically in the form of documentation, such that a client knows exactly what the library expects and what it produces, and this is often an unrealistic requirement [14]. Without such a specification, any change to the library that breaks a client might as well be classified as the client not using the library as the library developer intended. With NoRegrets\( ^+ \), we assume that the clients used in the model generation phase adhere to the library’s specification. For clients where this is not the case, NoRegrets\( ^+ \) may produce type regressions, which the library developer could rightly classify as caused by incorrect usage of the library. Nevertheless, we still believe that such warnings can be beneficial, since they may point to ambiguities and underspecified points in the documentation; each such warning reveals that a client developer has misunderstood the specification.

**Example 6** The `async` package is a widely used library that provides a large set of utility functions for working with asynchronous functions. One of these functions is `each`, which takes a collection (typically an array), an asynchronous iterator function, and a callback function as arguments. It then asynchronously runs the iterator on every element in the collection, eventually calling the callback when the iteration is done. The iterator function also takes a callback, which it can call with any argument to signal an error, which in turn calls the callback of `each` with the error value. A typical use of the `each` function is demonstrated by the following example where the client asynchronously performs some computation on an array of files.

```javascript
41 async.each(['file1.txt', 'file2.txt'],
42     function(file, cb) {
43         var err = ... //async operation
44         if (err) { cb(err) }
45         cb();
46     },
47     function(err) {
48         console.log("error processing files");
49     });
```

In the update of `async` from version 2.0.0 to 2.0.1, the `each` function was changed slightly to improve its performance when the collection is an array. While this update is non-breaking when the iterator function is asynchronous, it unfortunately changed the behavior of `each` when the iterator is synchronous. In version 2.0.0 when a synchronous iterator function calls its callback with an error value, the `each` call is directly terminated potentially leaving some elements in the array uniterated. However, in version 2.0.1 the iteration is not terminated on an error, so all elements will always be processed. This breaking change is detected by NoRegrets\( ^+ \) as a type regression.

\( ^{12} \)“Software using Semantic Versioning MUST declare a public API” — rule 1 of the SemVer specification, [https://semver.org/](https://semver.org/)
on the path

\texttt{\text{require(async).each-}^a\text{\rightarrow\text{ARG}_0.1}}

which refers to the element at index 1 in the array passed to the \texttt{each} call. The model states that this path is not read, nevertheless, \texttt{N}o\texttt{R}egrets\texttt{+} detects a read of this path in version 2.0.1 resulting in a type regression being reported. Upon inspection of this type regression, we find that the iterator function fails when processing the first element of the array, but that does not stop \texttt{each} from also beginning the processing of the second element and thereby causing the unexpected read.

Notice that this can only break clients that use \texttt{each} with synchronous functions, which is not allowed according to the \texttt{async} specification. However, due to either a misunderstanding of the specification or a general lack of knowledge of how asynchrony works in Node.js, many clients use \texttt{async} with synchronous functions. A search for "\texttt{RangeError: Maximum call stack size exceeded}", which is an error caused by the incorrect use of synchronous functions, on \texttt{async}'s issue tracker results in no less than 22 results. Furthermore, the first point in the “Common pitfalls” section of the \texttt{async} documentation page mentions the use of synchronous functions as a pitfall.

This example demonstrates that library developers may benefit from warnings reported by \texttt{N}o\texttt{R}egrets\texttt{+} even in situations where the changed library behavior is intended by the library developer, because many clients fail to follow the library specification and are thereby affected by the change.

While most type regressions reported by \texttt{N}o\texttt{R}egrets\texttt{+} are true positives, some of them are unlikely to cause problems in practice if the library developer is cautious, as demonstrated by the following example.

\textit{Example 7} \ The \texttt{joi} package is a schema validation library, which can be used to validate that objects and strings have a certain structure. Specifically, \texttt{joi} has a method \texttt{uri} that returns a schema object for validating that strings are valid RFC3986 URIs. The \texttt{uri} method takes a configuration object argument, specifying for example that only URIs of certain schemes are allowed:

\begin{verbatim}
50 var v = joi.string().uri({ scheme : 'http' })
51 v.validate('http://foo.bar').error // => null
52 v.validate('https://foo.bar').error // => ValidationError
\end{verbatim}

In version 13.5.0, a new optional property \texttt{allowQuerySquareBrackets} was introduced. Setting this property to \texttt{true} configures the schema object such that URIs with square brackets in query variables are allowed. \texttt{N}o\texttt{R}egrets\texttt{+} reports a type regressions for this change, because \texttt{joi} reads the path

\texttt{require(joi).string()\rightarrow ARG_0.allowQuerySquareBrackets}

in version 13.5.0, although the model states that no read should occur on it. However, the developers of \texttt{joi} were careful enough to introduce this change such that no existing

\footnote{https://caolan.github.io/async/}
8.7. RELATED WORK

clients were impacted. If a client does not supply the `allowQuerySquareBrackets` property, then `joi` will automatically assume it is `false` to preserve the old behavior for existing clients. This means that although the library API has changed in a way that could in principle break clients, the type regression is most likely benign.

Even for type regressions that are benign as in Example 7, the library developers may benefit from the warnings provided by `NOREGRETS`+. The warnings point the library developers to parts of the API where extra care must be taken to ensure backward compatibility and communicate to the client developers that the newly added properties like `allowQuerySquareBrackets` may conflict with existing properties in the client objects.

8.7 Related Work

Our approach builds on the recent work by Mezzetti et al. [96], but the challenge of detecting breaking changes in libraries also appears with other programming languages, and there are also connections to other testing techniques, in particular model-based testing.

Studies of breaking changes in library updates Breaking changes are common across languages and ecosystems [96, 119, 141]. According to Mezzetti et al. [96], at least 5% of JavaScript packages they studied have experienced a breaking change in a non-major update, and that the majority of the breaking changes are due to type-related issues. Brito et al. [16] conducted a study on why and how Java developers intentionally break APIs, concluding that the primary reasons are to add new features (32%), simplify the API (29%), and improve maintainability (24%). Zerouali et al. [143] showed that using strict version number constraint results in slow adoption of security critical updates. Many developers want to adopt semantic versioning, but do not trust that their dependencies adhere to the guidelines [14].

The study by Gyori et al. [54] used client test suites to detect breaking changes in library updates, similar to the `dont-break` methodology mentioned in section 8.1 but for Java. They note that it is common practice in industry to use this form of testing, but also that applying certain test case selection criteria could yield a considerable speed-up while preserving coverage, similar to how `NOREGRETS`+ avoids running all the client test suites at every library update.

Tools for JavaScript To our knowledge, only the two tools `NOREGRETS` and `dont-break` exist for detecting breaking changes in JavaScript libraries; the relations between `NOREGRETS`+ and those tools are explained in detail in the preceding sections.

Like `NOREGRETS`+, `NOREGRETS` also looks for type regressions in Node.js library updates, but it instead generates models for both the pre-update and the post-update version of the library, and then compares the models to identify type regressions [96]. Because `NOREGRETS` needs to compute a model twice to check an update, it is important that the clients’ dependency constraints on the library is within the same major number as the pre-update version of the library. As an example of why this is important,
consider the case where a library \( l \) is at version 2.0.0 and the library developer wants to check some changes for type regressions before releasing 2.1.0. If \( \text{NoRegrets} \) now picks a client \( c \) that depends specifically on version 1.0.0 of \( l \), then \( c \) expects \( l \) to expose an API that might be considerably different from the API in 2.0.0. If \( l \) deprecated a function \( g \) in version 2.0.0 and the developer now plans to remove it entirely in version 2.1.0, then if the client uses this function, it will crash with version 2.1.0, which will result in a quite different model and therefore also many type regressions. One of these type regressions will correctly state that \( g \) went from function to undefined, but the rest of them are false positives caused by the premature termination of the client. In contrast, \( \text{NoRegrets}^+ \) is able to continue testing of the library even if a type regression that would have crashed the client is detected. This difference allows \( \text{NoRegrets}^+ \) to use a larger set of clients and thereby produce better API models.

**Tools for other programming languages** For other languages than JavaScript, there are numerous tools that help library developers detect breaking changes. Common to all these tools is that they work for statically typed languages and rely on explicitly typed library APIs, which make it much easier to detect type-related breaking changes than for dynamically typed languages. For Java there is APIDiff [17], Clirr [14], japicmp [15], SigTest [16] and Revapi [17]. The Elm package manager (elm-package) promises to automatically enforce semantic versioning, although it is also limited to detecting type-related breaking changes.

For a dynamically typed language like JavaScript, the public API of a library is not easily identifiable statically, which is why we resort to the use of dynamic analysis for the model generation phase. JavaScript library developers can choose to write TypeScript declarations that define the public APIs of their libraries. However, declaration files are often full of errors and rarely kept up-to-date making them unsuitable for breaking change detection [79].

A problem related to breaking change detection is how to update clients when their libraries evolve, also called collateral evolution. As an example, the Coccinelle tool [110] has been designed to support Linux developers in this respect, but it does not help the developers determine if and where breaking changes are introduced.

**Model-based testing and related techniques** Our approach can be seen as a form of model-based testing [136]. In \( \text{NoRegrets}^+ \), the models are inferred automatically based on dynamic analysis of client usage.

The SCARPE tool by Joshi and Orso [69] uses a capture phase that generates a model of a software component based on live executions, and a replay phase that can produce regression tests from the model. This is reminiscent of how Krikava and Vitek [78] produce tests for CRAN packages written in R, based on executing the small snippets of executable example code that is often included in the documentation.
of such packages. In comparison with those techniques, we use the test suites of the
client packages to obtain realistic executions of the library.

The techniques by McCamant and Ernst [94, 95] construct logical models of
software components written in C by dynamically inferring likely invariants. Incom-
patibilities at component upgrades can then be detected by comparing the models
using an automatic theorem prover.

The idea of extracting new tests from existing tests also appears in the test carving
technique by Elbaum et al. [36], which aims to generate effective differential unit tests
from existing system tests. In comparison, NoRegrets+ exploits the fact that the test
suite of a client of a library often indirectly functions as a system test of the library,
which makes it possible to generate useful regression tests from existing client test
suites.

8.8 Conclusion

Breaking changes in libraries are a major concern for JavaScript developers. For Java,
the static type checker helps detecting such issues when building an application with a
new version of a library, but due to the dynamic nature of JavaScript, breaking changes
are rarely discovered before failures appear at run-time. Type regressions are a kind
of breaking changes that manifest as incompatible changes in the types of the method
parameters, return values, and object properties that constitute the API of a library.
Previous work has shown that type regressions account for many breaking changes
in widely used JavaScript libraries. The NoRegrets tool introduced the concept of
type regression testing for detecting such issues automatically, but it is inefficient and
inadequate for libraries with relatively few clients.

By taking a model-based testing approach, our tool NoRegrets+ creates tests from
a model of the library API, all fully automatically. As shown in our experimental
evaluation, this new approach is significantly more efficient and capable of finding
many more breaking changes, especially in libraries with fewer available clients.

With such tool support, it is our hope that JavaScript library developers can
make more informed decisions when releasing updates and using semantic versioning.
The experiments have also demonstrated that there is room for improvement of the
technique, especially concerning the use of proxies in the model generation phase,
which we plan to pursue in future work.
Chapter 9

Detecting Locations in JavaScript Programs Affected by Breaking Library Changes

By Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. Published in Proc. of the ACM on Programming Languages (OOPSLA), November 2020.

Abstract

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

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

9.1 Introduction

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

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

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

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

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

[^2]: http://semver.org/
[^3]: https://github.com/jquery/jquery-migrate/
[^4]: https://www.npmjs.com/package/lodash-migrate
[^5]: https://www.npmjs.com/browse/depended
[^6]: https://www.npmjs.com/package/rxjs-compat
9.1. INTRODUCTION

Techniques and tools used for other programming languages are difficult to adapt to JavaScript because of the dynamic nature of the language. As an example, for Java, simply recompiling the client code often reveals which program locations require attention, in the form of static type errors. Specialized tools, such as gofix\(^7\) for Go and Coccinelle\(^70\)[110] for C and Java, exploit the existing static type systems of those languages. In comparison, JavaScript does not have a static type system, and static type analysis for JavaScript is notoriously difficult\(^80\)[133].

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

Although the breaking change descriptions must be written manually (for now), they are usually quite short and easy to write (as we demonstrate in Section 9.7), and each library typically has many clients, which makes this modest amount of manual work acceptable. Importantly, most of the breaking changes documented informally in changelogs are expressible in our pattern language. (An example of a breaking change that cannot be captured as a pattern is removing support for outdated JavaScript engines, which generally affects the entire library and typically does not require changes to client code.) In situations where changelogs are unavailable or incomplete, existing tools, such as noRegrets\(^+\)[100], can be used for detecting the breaking changes in the libraries.

We similarly leave performing the actual changes of the client code to the developer; in this paper we focus on how to automate the pattern matching process. The common case is that only a small fraction of the breaking changes in a library update are relevant for a given client, so if not having any tool support, most of the manual effort involved in adapting client code is typically spent on finding the affected pieces of code, not on performing the needed changes. In fact, quite often when a client developer wants to upgrade to a new major version of a library to get access to new functionality, none of the breaking changes affect the client code (see Section 9.7). In that situation, it may take a long time for the client developer to realize that no changes in the client code are needed.

Our key insight is that it is possible to express breaking change patterns in a way that permits accurate and efficient pattern matching based on lightweight static analysis. Ideally, the pattern matching should have neither false positives (reporting locations that are in fact not related to the breaking changes) nor false negatives.
(missing locations that are related to breaking changes). Achieving that is of course impossible, not only theoretically by Rice’s theorem, but also practically due to the known difficulties involved in performing accurate and efficient static analysis for JavaScript, as mentioned above. Some false positives are tolerable, as long as there is not an overwhelming number and they are easy to dismiss manually. It is more important to avoid false negatives; a single false negative may cause a required change to the client code to be overlooked. Existing library-specific migration tools require high-coverage test suites to avoid false negatives, and not many programs have such extensive tests. Our static analysis is efficient and has no false negatives in our experiments (Section 9.7). Furthermore, the static analysis is designed such that it can report its confidence, which makes it easier to identify false positives. With these properties, the approach is a promising alternative to the current fully manual practice.

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

• We present a preliminary study of breaking changes in real-world JavaScript packages (Section 9.3), which has guided the design of our approach.

• We propose a simple pattern language for describing the API access points that are involved in breaking changes, and we provide an accompanying program analysis tool, named TAPIR for locating which parts (if any) of the client code may be affected by breaking changes (Sections 9.4–9.6).

• An experimental evaluation showing that the pattern language is sufficiently expressive in practice, and that the static analysis is accurate and efficient: 187 breaking changes from 15 package updates can be expressed using a total of 283 patterns, and running TAPIR on 265 clients of these packages takes less than a second per client and has a recall of 100% with only 1 in 7 alarms being false positives, and with all high confidence alarms being true positives (Section 9.7).

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

9.2 Motivating Example

Let us consider the npm package postal, an in-memory message bus library, which is currently downloaded more than 20000 times weekly. Based on its git commit
2: Removed category names from module paths

4: Removed this.**arg** params from most methods because they were largely unused, complicated implementations, & can be tackled with **_.bind**, **Function#bind**, or arrow functions

47: Dropped boolean options param support in **_.debounce**, **_.mixin**, & **_.throttle**

51: Removed 17 aliases

  - **_.all** in favor of **_.every**
  - **_.any** in favor of **_.some**

Figure 9.1: The subset of the breaking changes in **lodash** 4.0.0 that affected **postal**. The descriptions are as they appear in **lodash**’s changelog but with examples elided.

history we can reconstruct a typical update scenario. On April 30, 2016 the maintainer of **postal** decided to update the **lodash** dependency from version 3.10.1 to 4.11.1, which was the newest version at the time. The **postal** maintainer was aware of a breaking change in **lodash**’s **debounce** method. He therefore located the places in **postal**’s source code where **debounce** was used and updated the code accordingly. He then pushed a patch update of **postal** to the npm registry, probably assuming that no other breaking changes in **lodash** were affecting **postal**.

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

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

Consequently, the newest version of **postal** in the npm registry for more than a month was not properly adapted to work with **lodash** in the version 4 major range, which could lead to crashes and misbehavior of clients depending on **postal**. This example clearly illustrates the difficulties in adopting major updates of dependencies. In essence, the client maintainer must first go through the list of all documented breaking changes in the update, then determine which breaking changes are relevant for the client, and then adapt the client code to the breaking changes. As the **postal** example illustrates, relying on test suites will not always catch all the client code locations that are affected by breaking changes. The test suites of the clients are designed to test the client code, not to check for failures in dependencies, so even
CHAPTER 9. DETECTING LOCATIONS IN JAVASCRIPT PROGRAMS
AFFECTED BY BREAKING LIBRARY CHANGES

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

Figure 9.2: The output of TAIPIR after analysis of the source code of postal using the breaking change detection patterns for lodash 4.0.0.

high-quality test suites can be inadequate for this purpose. The client developer usually has no other option than reading through the changelog of the dependency and then manually determining which breaking changes are relevant for the client’s usage of the dependency. This is a time consuming and error-prone task since many clients use only a small subset of the APIs of their dependencies, so often only a few or none of the breaking changes are actually relevant for each client. For example, postal is only affected by 4 of the 54 breaking changes introduced in lodash version 4.0.0. The relevant items from lodash’s changelog are shown in Figure 9.1 (see the full list on https://brics.dk/taipir/). Evidently, finding these relevant items among all the 54 entries in the changelog is a major effort if done manually, even for someone deeply familiar with the postal source code. Interestingly, the second one (about removed thisArg params) is relevant for postal despite being described as “largely unused” in the changelog.

Using TAIPIR, the push of a single button will list all of the 9 places in postal’s source that had to be changed when updating the lodash dependency, due to breaking changes in the library API. As we explain in Section 9.7, the breaking changes in lodash version 4.0.0 can be concisely captured by a collection of patterns that describe the affected API access points, making it substantially easier to update all the many thousands of clients of lodash. The actual output of the tool is shown in Figure 9.2. A manual inspection reveals that 2 of the 11 matches reported are false positives (meaning that those locations in the program are actually not affected by any of the breaking changes). As part of its output, TAIPIR shows its confidence, and those two cases are indeed matches with low confidence. All the other locations are true positives that require small changes to adapt to the new version of the library. Conversely, all the changes made manually by the postal developer to adapt to lodash 4.0.0 are correctly detected by TAIPIR (disregarding a couple of places where postal accesses internal functionality of lodash that is not part of its public API and therefore not mentioned in the changelog).

In conclusion, if TAIPIR had been available to the postal maintainer, it would likely have substantially reduced the manual workload, and it would likely also have
9.3 Preliminary Study

To understand what kinds of breaking changes package maintainers typically introduce in major updates, we have conducted a manual study of the changelogs from 10 major updates of some of the most widely used npm packages. As a methodology for selecting packages, we picked the top 10 packages with the highest number of direct dependents in the npm registry, disregarding a package if its newest version was less than 1.0.0, or if the changelog was unavailable or did not mention the latest major version. As we focus on the Node.js platform, we also chose to disregard packages that are used only for front-end web development or in build systems (e.g., react and webpack).

Each of the changelogs contains a bullet list of changes. We disregarded changes that do not break backward compatibility, including addition of new features and bug fixes. (In theory, clients may apply workarounds for known bugs, which may cause the client code to break when the bug is fixed, but we ignore that here.) Each changelog bullet is counted as one change, even though one single bullet sometimes covers many library functions. (For example, the changelog of lodash 4.0.0 contains a single bullet describing the removal of 17 aliases; see breaking change number 51 in Figure 9.1.) Furthermore, we disregarded changes explicitly marked as deprecations. (For such changes, the old behavior or feature is still present, but new clients are discouraged from using it, and in many cases the deprecated features are scheduled to be removed in some future major update, in which case they will then be treated as actual breaking changes.)

Interestingly, the changelogs do not always clearly specify which parts of the API are involved in a change. (An example from lodash is breaking change number 4 shown in Figure 9.1: a closer inspection reveals that this one affects 64 different functions.) This means that even for manual use by the client developers, the existing informal changelogs do not provide enough information to be able to safely adapt the client code.
Table 9.1: Breaking changes in npm packages.

<table>
<thead>
<tr>
<th>Library</th>
<th>Module</th>
<th>Property</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Removal</td>
<td>Move</td>
<td>Removal</td>
</tr>
<tr>
<td>lodash 4.0.0</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>async 3.0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>express 4.0.0</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>chalk 2.0.0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>bluebird 3.0.0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>uuid 3.0.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>commander 3.0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rxjs 6.0.0</td>
<td>0</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>core-js 3.0.0</td>
<td>3</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>yargs 14.0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3</strong></td>
<td><strong>24</strong></td>
<td><strong>32</strong></td>
</tr>
</tbody>
</table>
9.3. PRELIMINARY STUDY

To learn about the nature of the collected real-world breaking changes, we grouped them into a number of categories. The collection of packages and the number of changes belonging to each category are listed in Table 9.1.

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

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

This study has several interesting findings, which we leverage in the design of our solution in Section 9.4.

Most importantly, we see from Table 9.1 that the majority of breaking changes, 130 out of 153, belong to the categories concerning specific points in the package APIs (Module, Property, and Function).

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

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

### 9.4 The TAPIR Approach

We have developed the tool **TAPIR** for finding the source locations in client code that may be affected by the breaking changes in a major update of a dependency. In Section 9.5 we introduce a simple pattern language for specifying API access points where breaking changes occur and what conditions must be met for the breaking changes to be relevant.

We envision that the library developer writes patterns in this language while developing a major update, to accompany the changelogs that are traditionally written. However, as we demonstrate in Section 9.7, a collection of patterns for a library update is typically relatively small and quite easy to write, even without expert knowledge of the library, so it is also possible that, for example, a client developer performs this task. This resembles the practice for TypeScript declaration files[^9] which are also often contributed to the community by other programmers than the JavaScript

[^9]: https://github.com/DefinitelyTyped/DefinitelyTyped
library developers themselves. In situations where comprehensive changelogs are not available, existing tools can be used for detecting the breaking changes in the libraries [100].

Once a collection of patterns has been written for a library update, it can be reused for all the clients that need to be adapted. Especially for widely used libraries with thousands or millions of clients, this justifies the effort required to write the patterns. The client developers can use the TAPIR tool, which applies a lightweight static analysis to the client code as explained in Section 9.6, to automatically find the places in the client code that are affected by the specified breaking changes.

### 9.5 A Pattern Language for Describing API Access Points

Figure 9.3 shows the syntax of our pattern language for describing API access points of interest. There are four kinds of patterns.

First, an import pattern, written `import q` where `q` is a Unix path-like pattern (also known as a glob pattern), matches client code that imports a module with a name described by `q`. Such patterns are intended for describing the Module breaking changes from Section 9.3. A glob consists of a file system path that, when matched against a concrete file system, results in a set of matched files or directories. A glob can also contain wildcards, such as the * that matches all files, ** that matches all
directories recursively, and \{i_1, ..., i_n\} that matches the union of the globs \(i_1\) to \(i_n\). The specialized import pattern \texttt{importD} is described at the end of this section.

\textit{Example 1.} The pattern

\begin{verbatim}
import core-js/client/**
\end{verbatim}

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

A \textit{read} pattern, written \texttt{read p}, matches property read operations in the client code that match the property path pattern \(p\). Most of the \textbf{Property} changes from Section \[9.3\] can be described by read patterns. A \textit{write} pattern, \texttt{write p}, similarly matches property write operations. We introduce a notion of access path patterns to be able to characterize the relevant dataflows:

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

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

- \(p.f\) matches all property read and write operations \(E.f\) in the client code (for a JavaScript expression \(E\) and a property name \(f\)) where \(p\) recursively matches \(E\). (In principle this kind of pattern also matches dynamic property accesses where the property name is dynamically computed and may evaluate to \(f\), but see Section \[9.6\])
- \(p.{f_1, \ldots, f_n}\) is similar to the preceding kind but matches any of the given property names.

In addition to these rules, if a pattern \(p\) matches an expression \(E_1\) in the client code and the run-time value of \(E_1\) may flow (via assignments, function calls, etc.) to another expression \(E_2\), then \(p\) also matches \(E_2\). In Section \[9.6\] we show how to approximate this statically using a simple alias analysis.

\textit{Example 2.} The following read pattern describes a breaking change in the \texttt{lodash} version 4.0.0 update.

\begin{verbatim}
read <lodash>.any
\end{verbatim}
9.5. A PATTERN LANGUAGE FOR DESCRIBING API ACCESS POINTS

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

**Example 3.** The following write pattern describes the breaking change in the _async_ version 3.0.0 update that was mentioned in Section 9.3.

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

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

A call pattern, call \(p, f_1, \ldots, f_n\), matches function/method/constructor call operations that match \(p\) and also satisfy each of the filters \(f_1, \ldots, f_n\). The call patterns are designed to handle the **Function** changes from Section 9.3. We use a notion of filters to describe the changes that are conditional on the number of arguments or the argument types.

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

The types we support for filtering include the usual JavaScript types (string, number, boolean, undefined, object, array, function), but we also allow singleton types (expressed as **Literal**) and functions with specific numbers of parameters (**function[Int]**), which are useful for describing callbacks. Another task of the static analysis presented in Section 9.6 is to approximate this filtering information statically for a given client and pattern. The specialized call pattern callR is described at the end of this section.

**Example 4.** The following call pattern describes the `thisArg`-related breaking change in the _lodash_ version 4.0.0 update mentioned in Section 9.3.

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

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

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

```plaintext
call <commander>**.parse
```
Example 6. The following pattern recognizes the removal of the `merge` observable creator function from observable objects.

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

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

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

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

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

Example 8. The `rxjs` library introduces a breaking change in the update to version 6, where default imports from the `rxjs` module are no longer supported. Prior to version 6, clients could write `import rx from 'rxjs'` to load what is known as the default exported object into the `rx` variable. However, the developers of `rxjs` wanted to encourage clients to only load the functions from `rxjs` that are used in the code. For example, clients should instead write `import {merge, interval} from 'rxjs'` if the `merge` and `interval` methods are used. Therefore, the ability to use a default import was removed from the module. To catch this breaking change we use the pattern

```
importD rxjs
```

which matches only those imports from `rxjs` that import the default exported object. Thereby, clients who already load only the required methods from the module will not get any warnings.
9.6 A Static Analysis for Finding Uses of API Access Points in Client Code

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

The static analysis of TAPIR is separated into three phases that are run in isolation on each file of the client code. The first phase is an alias analysis that finds expressions that may alias a given variable or object property. The second phase infers access paths for all expressions, to identify the connections between the client code and the imported modules. The third phase performs pattern matching, to find the expressions that have an access path that matches one of the API access point patterns of interest. For each match, the user is notified with the source location of the expression, as shown in Figure 9.2.

Phase 1: Alias Analysis

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

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

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

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

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

The analysis completely ignores dynamic property write assignments, the flow of objects at function calls and returns, exceptions, etc. The analysis design is inspired by Feldthaus et al. [43] who found field-based analysis to be highly scalable and surprisingly accurate; in particular, ignoring dynamic property accesses in their analysis loses only little soundness in practice. Unlike their analysis, we additionally ignore function calls and returns, which of course makes the analysis even more unsound in theory, but we find that typical JavaScript code rarely passes module objects through function calls or returns. (We discuss in Section 9.8 why we are not using the call graph construction of Feldthaus et al.)

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

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

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

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

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

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

```
α_ : [{_ := require('lodash')},
         libAsync := {require('async')},
         ...
]
9.6. A STATIC ANALYSIS FOR FINDING USES OF API ACCESS POINTS IN
CLIENT CODE

AP_S(E) := \[
\begin{cases}
\{ <m> \} & \text{if } E = \text{require}(m) \\
\{ \text{ap} . f | \text{ap} \in \text{AP}_S(E') \} \cup \text{lookup}_S(f) & \text{if } E = E'.f \\
\{ \text{ap}(\cdot) | \text{ap} \in \text{AP}_S(E') \} & \text{if } E = E'(. \cdot) \text{ or } E = \text{new } E'(\ldots) \\
\text{lookup}_S(x) & \text{if } E = x \text{ where } x \text{ is a variable that is not a parameter} \\
\{ \text{U} \} & \text{otherwise}
\end{cases}
\]

lookup_S(z) := \[
\begin{cases}
\bigcup_{E \in \alpha_s(z)} \text{AP}_S, (z)(E) & \text{if } z \notin S \\
\emptyset & \text{otherwise}
\end{cases}
\]

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

Figure 9.4: Access path inference.

Phase 2: Access Path Inference

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

\[
\text{AccessPath} ::= < \text{ImportPath} > \\
| \text{AccessPath} . \text{Property} \\
| \text{AccessPath}(\cdot) \\
| \text{U}
\]

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

The function AP for computing access paths for a given expression E is defined in Figure 9.4

- If E is a module load operation, such as require(’lodash’), AP returns the corresponding import path.

- For a property read E'.f, the access paths are computed by recursively computing the access paths of E' and adding .f, and by recursively calling AP on all aliased expressions using the lookup function, which uses the alias map \(\alpha_s\) from phase 1. Notice that lookup calls AP recursively and thereby takes care of transitive dataflow as discussed in Example 9; see also Example 12 below.

- For a function call expression E'(\ldots) (with or without new), AP similarly constructs the access paths by recursively computing the access paths for E' and then appending (\cdot).

• When $E$ is a variable read (excluding function parameters), the access paths are obtained using the `lookup` function.

• Otherwise, $AP$ returns $\{U\}$ to indicate that we do not have any knowledge about the access paths for the expression. This case covers, for example, function parameters, arithmetic operators, and literals.

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

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

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

```javascript
15 function x() {
16   x.f = require('lib').fun;
17   x.f(42);
18 }
```

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

**Example 13.** For the following code,

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

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

\[
AP_0(x.f) = \{ap.f \mid ap \in AP_0(x)\} \cup \text{lookup}_0(f)
\]

\[
= \{ap.f \mid ap \in \text{lookup}_0(x)\} \cup \bigcup_{E \in \alpha_s(f)} AP_{\{f\}}(E)
\]

\[
= \{ap.f \mid ap \in \bigcup_{E \in \alpha_s(x)} \text{AP}_{\{x\}}(E)\}
\]

\[
= \{<\text{lib}.f\}
\]

since

\[
AP_{\{x\}}(\text{require('lib'))} = \{<\text{lib}>\}.
\]
9.6. A STATIC ANALYSIS FOR FINDING USES OF API ACCESS POINTS IN
CLIENT CODE

and

\[ AP_{\{x\}}(x.x) = \{ ap.x | ap \in AP_{\{x\}}(x) \} \cup lookup_{\{x\}}(x) \]
\[ = \{ ap.x | ap \in lookup_{\{x\}}(x) \} \cup \bigcup_{E \in \alpha(\mathcal{E})} AP_{\{f\}}(E) \]
\[ = \emptyset. \]

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

**Example 14.** The API access path pattern

\[
\text{write } <\text{async}>.\text{queue()}?.\{\text{drain, saturated}\}
\]

matches both the access paths

\[
<\text{async}>.\text{queue()} . \text{drain}
\]

and

\[
U . \text{saturated}
\]

but not

\[
<\text{lodash}>.\text{queue()} . \text{drain}
\]

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

Based on our experience, it is generally quite easy to decide whether to use \( p \) or \( p? \). One can always choose the \( p? \) variant if in doubt since it overapproximates the other one, but it may result in more false positives in the pattern matching in phase 3. However, as we show in the evaluation, the number of false positives is not as problematic as one might expect, since variable and property names tend not to overlap much across different libraries. Overall, this design choice of introducing the relaxed form of detection patterns is a pragmatic compromise between simplicity of the pattern language and scalability and accuracy of the analysis; we return to this discussion in Section 9.8.

Phase 3: Pattern Matching

The third phase of the TAIPR analysis matches the API access point patterns against the client code. The matching is defined as a relation \( > \) between access path patterns
### CHAPTER 9. DETECTING LOCATIONS IN JAVASCRIPT PROGRAMS AFFECTED BY BREAKING LIBRARY CHANGES

**Figure 9.5: Pattern matching relation** $p \succ t$ where $p$ is an access path pattern and $t$ is an access path.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IMPORT</strong></td>
<td>$m$ matches $q$</td>
</tr>
<tr>
<td></td>
<td>$\langle q \rangle \succ m$</td>
</tr>
<tr>
<td><strong>UNCERTAIN-1</strong></td>
<td>$p \succ t$</td>
</tr>
<tr>
<td></td>
<td>$p? \succ t$</td>
</tr>
<tr>
<td><strong>UCERTAIN-2</strong></td>
<td>$p? \succ U$</td>
</tr>
<tr>
<td><strong>DISJUNCTION</strong></td>
<td>$p_i \succ t$ $i \in {1, \ldots, n}$</td>
</tr>
<tr>
<td></td>
<td>${p_1, p_2, \ldots, p_n} \succ t$</td>
</tr>
<tr>
<td><strong>EXCLUSION</strong></td>
<td>$p \succ t$ $p \not\succ t$</td>
</tr>
<tr>
<td></td>
<td>$p \not\succ U$</td>
</tr>
<tr>
<td><strong>STAR-1</strong></td>
<td>$p^{**} \succ t$</td>
</tr>
<tr>
<td></td>
<td>$p^{**} \succ t.f$</td>
</tr>
<tr>
<td><strong>STAR-2</strong></td>
<td>$p^{**} \succ t$</td>
</tr>
<tr>
<td></td>
<td>$p^{**} \succ t.f$</td>
</tr>
<tr>
<td><strong>STAR-3</strong></td>
<td>$p^{**} \succ t$</td>
</tr>
<tr>
<td></td>
<td>$p^{**} \succ t.()$</td>
</tr>
<tr>
<td><strong>PROP-READ-1</strong></td>
<td>$p \succ t$</td>
</tr>
<tr>
<td><strong>PROP-READ-2</strong></td>
<td>$p \succ t.f$</td>
</tr>
<tr>
<td></td>
<td>$p.{f_1, \ldots, f_n} \succ t.f$</td>
</tr>
<tr>
<td><strong>CALL</strong></td>
<td>$p \succ t$</td>
</tr>
<tr>
<td></td>
<td>$p.() \succ t.()$</td>
</tr>
</tbody>
</table>

The inference rules directly reflect the descriptions of the access path patterns from Section 9.5 (the only exception being the two **UNCERTAIN** rules for handling $p?$ as discussed in Section 9.6).

For an **import** pattern, performing the matching is straightforward. The analysis matches the access paths computed for every module load AST node in the previous phase against the import pattern using the **IMPORT** rule (as shown in Example 1).

For an **import**D pattern, it must furthermore be the case that the default loading mechanism is used, which is always trivially decidable from the syntax of the module load.

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

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

For a **call** pattern, TAPIR similarly looks at every call node in the client code. First it matches the access paths of the function (computed in the previous phase) against the access path pattern using $\succ$ exactly as with **call** and **write** patterns. For all calls where an access path matches, TAPIR matches the arguments of the call against the filters of the pattern. The number of arguments to the call can be extracted.

---

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

directly from the AST. For type filters, TAPIR also attempts to extract the argument type directly from the argument AST node: If the argument is a literal, then the type is simply the type of that literal; otherwise, TAPIR conservatively assumes that the type filter matches (but with low confidence; see Section 9.6). For a callR pattern to match, the result of the call must not be discarded.

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

```javascript
var _ = require('lodash');
...
_.each(_.keys(this.cache), function (cacheKey) {
...
});
```

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

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

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

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

Reporting Analysis Confidence

As a convenient extra feature of the static analysis, it can classify each match as either high confidence or low confidence, as shown in the example output in Figure 9.2. The intention is that this extra information can be useful for the client developer when deciding how to adapt the code to the breaking changes in the library. The client developer can trust the high confidence matches and only needs to manually review the low confidence matches, thereby further reducing the manual work. (Of course, this requires that high confidence matches are not false positives in practice; we demonstrate experimentally in Section 9.7 that the confidence classification has this property.)

As explained in Section 9.6, the analysis is designed such that it is unlikely to have false negatives, but it may have false positives meaning that it can report spurious matches. Despite the simplicity of the analysis, there are only three sources of likely

---

13 For calls made using `Function.prototype.apply` or using the spread operator, TAPIR conservatively assumes that the `[n,m]` filter always matches.
false positives: the relaxed form of patterns $p$? (see Section 9.6), the filters at call patterns (see Example 17), and inference of multiple access paths for an expression. This gives us a simple confidence classification mechanism. A match between a pattern $p$ and an expression $E$ is classified as low confidence if

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

and otherwise it is a high confidence match.

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

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

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

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

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

```javascript
31 function f(x) {
32   ...
33   return _.dropWhile(arr, pred, x)
34   ...
35 }
```

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

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

```javascript
36 var program = require('commander');
37 var UglifyJS = require('..tools/node');
38 ...
39 options.parse = program.parse;
40 ...
41 UglifyJS.parse(...)
```
Our simple field-based alias analysis cannot distinguish between the `parse` property from `commander` and the `parse` property from the `uglify-js` module. The set of access paths inferred for the expression `uglifyJS.parse` is therefore `{<../tools/node>.parse,<commander>.parse}`, which gives a low confidence match with the pattern call `<commander>**.parse`, resulting in a false positive.

### 9.7 Evaluation

We have implemented TAPIR in TypeScript using acorn\(^4\) and recast\(^5\) for producing and traversing ASTs. Our implementation of the static analysis is less than 1000 LoC, including the alias analysis, the access path inference, and the pattern matcher. We evaluate TAPIR by answering the following research questions:

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

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

**Benchmark selection** To answer the research questions, we base our experiments on the 10 major updates of library packages from the npm registry that we also considered in our preliminary study in Section 9.3. Furthermore, we include 5 additional major updates of other libraries to reduce bias towards the types of breaking changes observed in the preliminary study. The 15 libraries and a summary of the experimental results are shown in Tables 9.2–9.5.

To address RQ2, we extracted client packages from the npm registry that depend on one or more of the libraries in the major version prior to the one for which the patterns are written for. For example, for the lodash version 4.0.0 update, we collected all packages in the npm registry that depend on any version of lodash between version 3.0.0 and 3.10.1 (3.10.1 is the last version before 4.0.0). We then retrieved the GitHub repository for each package and found the git commit matching the required version of the client (the newest version where the benchmark update has not been applied). We discarded a package if a GitHub repository was not available, we could not identify the right commit, the repository contained no test suite, or running the test suite did not succeed.

For measuring recall we are interested in client packages whose test suite fails after switching to the new version of the library (without making any changes to the client code). For each of the libraries, we attempted to extract at least 10 such clients.

---

\(^4\)https://www.npmjs.com/package/acorn
\(^5\)https://www.npmjs.com/package/recast
CHAPTER 9. DETECTING LOCATIONS IN JAVASCRIPT PROGRAMS

130

AFFECTED BY BREAKING LIBRARY CHANGES

Table 9.2: Overview of detection patterns.

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

(a few more for lodash since that update contains many breaking changes). For some libraries, we could not find that many clients, typically because the latest major update is so old that there are few packages that ever depended upon versions before it, or because the breaking changes are relatively benign and therefore unlikely to cause any test failures. As result, we obtained 115 such clients (Table 9.4). For measuring precision and performance, we collected 150 additional clients (Table 9.5) whose test suites are unaffected by switching to the new version of the library.

RQ1 (Expressiveness)

For each of the breaking changes in the 15 package updates, we created corresponding API access patterns (apart from one case, which we explain later). In some cases, multiple patterns were required to describe a single breaking change. Consider, for example, the breaking change concerning the removal of the thisArg argument from many lodash methods, which (among other patterns) requires both the patterns from Example 4 and Example 17.16 This particular breaking change requires 9 patterns since the thisArg argument can either be the second, third, or fourth argument of a method, some methods are chainable, and some can be imported from different modules. However, the breaking change affects 64 different methods, so being able to express it with just 9 patterns is acceptable.

When performing the experiments for RQ2, we discovered 5 breaking changes in rxjs, 2 in express, 2 in winston, 1 in bluebird and 1 in node-fetch that were not mentioned in the changelogs. We have also written patterns for those changes. The

---

16Both examples are shortened versions of the actual patterns since more methods are affected, which is expressed with \{f_1, \ldots, f_n\} patterns.
### Table 9.3: Detection pattern statistics.

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

The total number of breaking changes for each library is shown in the #BC column in Table 9.2.

In total, we have written 283 patterns, where 15 are for the 11 breaking changes we found and the remaining 268 are for the 176 breaking changes mentioned in the changelogs, so on average each breaking change amounts to 1.5 patterns. The number of patterns written for each benchmark varies heavily (see the #Patterns column in Table 9.2), from 1 for uuid to 113 for lodash, but this is well-aligned with the differences in the number of breaking changes per update as observed in Section 9.3.

The columns #Import, #Read, #Write, and #Call show the number of patterns of the different kinds.

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

As explained in Section 9.5, the patterns are generally short and simple. Table 9.3 shows the mean length of the patterns and the mean number of occurrences of each kind of pattern construct per pattern. More specifically, we count the number of AccessPathPattern, PropertyPathPattern, and Filter derivations in the parse tree of each pattern according to the grammar of Figure 9.3. For example, the path in Example 1 has length 1, the path in Example 3 has length 4, and the path in Example 4 has length 3. The mean length of each pattern is only 3.25, which shows that patterns are typically quite small.

With this length metric, we choose to count a property path pattern with multiple
Chapter 9. Detecting Locations in JavaScript Programs

Affected by Breaking Library Changes

Table 9.4: Experimental results for clients with test suites that fail after the update.

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

Property names as one (and similarly for type filters containing multiple types, and for globs at import patterns), because those do not contribute much to the pattern complexity. For example, the thisArg breaking change of lodash affects 64 different methods, which leads to a property path pattern with 64 property names, but finding that list of names was trivial, and the pattern did not take more time to write than other patterns.

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

As discussed in Section 9.6, deciding whether to use p or p? depends on the typical usage pattern of the API. In cases where an access path pattern identifies an API on an object where that object serves as a form of namespace, we avoid using the latter since TAPIR can construct access paths for these usages with high precision. For more dynamic cases where objects are constructed and commonly passed around in the client code, we use the more conservative variant p?. For all the constructed patterns, we found it easy to determine which of these categories the API usage pertained to. We ended up using the p? variant in 61 of the 283 patterns.

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

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

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

RQ2 (Recall, Precision, Confidence, and Performance)

Recall  For TAPIR to be useful in practice, the recall must be high, as discussed in Section 9.1: it is important that it does not miss places in the client code that require changes when updating to the new version of the library. To measure the recall, we need a collection of clients that are known to be affected by breaking changes. For this purpose we use the 115 client packages whose test suites fail when switching to the new version of the library without adapting the client code.

We manually investigated the cause of each of the failing tests, and then modified the client code to adapt to the new version of the library. We then reran the tests and confirmed that the tests succeeded. Finally, we compared the set of all the source code locations that we had to modify in this process with the set of source locations reported by TAPIR. As result, all the source code locations that we had to manually modify were found by TAPIR, indicated by the 100% recall in Table 9.4.

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

Precision  To measure the precision, we ran TAPIR on all 265 clients (without the code changes made for the recall experiment). For each alarm that did not match any of the manually patched source locations from the recall experiment, we manually investigated the alarm and recorded whether it was a true positive (i.e., the source location actually involves the API access point) or a false positive (shown as TP and FP, respectively, in Tables 9.4 and 9.5).

For some of the breaking changes (primarily belonging to the behavioral category from Table 9.1), TAPIR cannot decide with certainty whether a call is affected by the
### Table 9.5: Experimental results for clients with test suites that are unaffected by the update.

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

Breaking change, either because the pattern language is too coarse-grained, or because the client application is itself a library, which makes it impossible to determine if some client uses that library in a way that involves the breaking change. We conservatively mark alarms of this form as true positives as long as TAPIR points to the correct function and it satisfies the filters, but we report them separately as TP_{B} in Tables 9.4 and 9.5 since we acknowledge that they may not actually be true positives in all cases.

As shown in Table 9.4 for the group of clients whose test suites detect breaking changes in the libraries, 86% of all alarms are true positives (Precision), showing that the precision of TAPIR is high in practice. Only a few of the alarms (18%) belong to the more uncertain TP_{¬}B category, which means that the remaining 82% of alarms (TP_{¬}B) point to places in the client source code that with certainty have to be patched to update the benchmark.

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

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

The average precision of TAPIR for the second group of clients is 86%. For winston the precision is only 39%, but that is mostly due to one breaking change affecting the info, warn, and error methods. This breaking change only occurs when the second argument is a function, but for most calls to these methods, TAPIR is unable to determine the type of that argument and therefore reports a low confidence alarm. For 9 of the 15 libraries, the precision is 100%.
The false positives that TAPIR reports are mostly cases where the relaxed pattern variant $p?$ is used together with a breaking change that affects a relatively common property name such as `map` or `catch`. Using the strict pattern $p$ instead would reduce the recall, and the false positives are easily identified as false positives, so using the relaxed form is the best trade-off. The remaining false positives are due to imprecise checking of `call` filters or objects being mixed together due to imprecision in the field-based analysis.

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

**Confidence** 54% of the alarms in the affected test suite clients and 57% of the alarms in the unaffected test suite clients are marked as high confidence by the analysis mechanism described in Section 9.6. We have not seen any high confidence alarms in the false positive category, which shows that for around half of the reported alarms, in practice the user does not even have to consider the possibility of false positives. Notice that low confidence alarms do not coincide with the uncertainty alarms ($TP_B$). A low confidence alarm occurs because the analysis is too imprecise to either prove or disprove that a source location is affected by a breaking change. An uncertain alarm occurs whenever TAPIR lacks information on how the potentially affected API is used, or the detection pattern language is too coarse-grained to capture the constraints for the breaking change.

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

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

### 9.8 Related Work

**Software evolution studies** Several papers have investigated the evolution of software libraries, for example how and why breaking changes are introduced in library updates [16, 34, 76, 96, 99]. Dig and Johnson [34] find that more than 80% of all breaking changes in Java seem like refactorings from the library’s perspective (renames, method split-ups, moves, etc.). This is similar to our findings for JavaScript from Section 9.3.

Multiple studies have investigated why clients do not keep their dependencies up-to-date [33, 145]. Concerns about breaking changes are for the majority of developers a strong reason for not updating dependencies, which motivates the creation of tools like TAPIR.

**Breaking change detection and patching tools** Automatically detecting locations in client code affected by breaking changes has been pursued by previous work [23, 28].
Several techniques also include mechanisms for patching the affected client code \[23, 28, 70, 103, 108\] and even how to automatically infer broken APIs from the diff between library versions \[28, 39, 87, 103\]. Common amongst these papers is that they all target statically typed languages such as C or Java, which makes identifying API usages and extracting type information much simpler. To the best of our knowledge, the only exception is the PyCompat tool for Python \[146\]. While Python is also dynamically typed, it is still significantly different from JavaScript. The API usage in Python is sufficiently static for PyCompat to be able to identify the relevant call sites in the client code directly from the AST. For JavaScript, a more sophisticated mechanism, such as the access path analysis of \textsc{TAPIR}, is required to achieve good precision.

**Lightweight static analysis**  Creating sound context-sensitive dataflow analyses for JavaScript is notoriously difficult due to the highly dynamic language features \[80, 133\]. For that reason, using lightweight, unsound analysis techniques has been explored by previous work, for example to help with refactoring \[41\] and for constructing call graphs \[43\]. Common for these approaches is that they are unsound but work well in practice. As shown in Section 9.7, the static analysis of \textsc{TAPIR} belongs to this family of analyses.

A disadvantage of our pattern language is that pattern writers must consider whether to use the relaxed variant of patterns (i.e., $p\ ?$). As we argue in Section 9.7, this choice is usually simple, but ideally the pattern writer should not have to make it. For future work, we plan to explore whether a more powerful analysis can remove the need for the relaxed variant, however, it is not easy to accomplish that while preserving the essential properties of our current approach: (1) it avoids false negatives, (2) it has an acceptable number of false positives, and (3) it is fast. In particular, using the analysis technique of Feldthaus et al. \[43\] would not allow us to drop the relaxed variant of patterns. Although using that technique could perhaps improve precision, it is too unsound, meaning that we would lose the excellent recall of our current approach.

### 9.9 Conclusion

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

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

Once the affected locations have been found in the client code, the next task for the developer is to update those parts of the code (if any), to adapt to the breaking changes in the library. That task can possibly also be partly automated, which we will explore in future work. We also plan to investigate how to automatically generate API access point patterns for breaking changes, possibly using NOREGRETS\textsuperscript{+} [100] as a starting point.
Chapter 10

Semantic Patches for Adaptation of JavaScript Programs to Evolving Libraries


Abstract

JavaScript libraries are often updated and sometimes breaking changes are introduced in the process, resulting in the client developers having to adapt their code to the changes. In addition to locating the affected parts of their code, the client developers must apply suitable patches, which is a tedious, error-prone, and entirely manual process.

To reduce the manual effort, we present JSFIX. Given a collection of semantic patches, which are formalized descriptions of the breaking changes, the tool detects the locations affected by breaking changes and then transforms those parts of the code to become compatible with the new library version. JSFIX relies on an existing static analysis to approximate the set of affected locations, and an interactive process where the user answers questions about the client code to filter away false positives.

An evaluation involving 12 popular JavaScript libraries and 203 clients shows that our notion of semantic patches can accurately express most of the breaking changes that occur in practice, and that JSFIX can successfully adapt most of the clients to the changes. In particular, 31 clients have accepted pull requests made by JSFIX, indicating that the code quality is good enough for practical usage. It takes JSFIX only a few seconds to patch, on average, 3.8 source locations affected by breaking changes in each client, with only 2.7 questions to the user, which suggests that the approach can significantly reduce the manual effort required when adapting JavaScript programs to evolving libraries.
CHAPTER 10. SEMANTIC PATCHES FOR ADAPTATION OF
JAVASCRIPT PROGRAMS TO EVOLVING LIBRARIES

10.1 Introduction

The JavaScript-based npm ecosystem consists of more than a million packages, most of them libraries used by many JavaScript applications. Libraries constantly evolve, and client developers want to use the latest versions to get new features and bug fixes. However, not all library updates are backwards compatible, so the client developers may be discouraged from switching to newer versions of the libraries because of changes that break the client code. Currently, to update a client to a new major version of a library, the client developer needs to examine the changelog to discover which parts of the client code are affected by breaking changes and find out how to adapt the code accordingly – a process which is entirely manual, error-prone, and time-consuming. Existing tools, such as GitHub’s Dependabot[1], only warn about outdated dependencies and provide no other assistance in this process.

A detection pattern language and an accompanying analysis, TAPIR, have recently been proposed for finding locations in client code affected by breaking changes [101]. Inspired by the Coccinelle tool [108, 109], in this paper we build on top of TAPIR and introduce a notion of code templates for expressing how to also patch the affected locations. Paired with a detection pattern, a code template forms a semantic patch. Provided with a collection of semantic patches specifying where breaking changes occur (the detection patterns) and how to adapt the affected code (the code templates) for a library update, our tool JSFIX can semi-automatically adapt clients to become compatible with the new version of the library. A run of JSFIX goes through three phases: an analysis phase (based on the TAPIR analysis) for over-approximating the set of affected locations, an interactive phase for filtering away false positives from that set, and a transformation phase for adapting the client code using the code templates. In the interactive phase, JSFIX asks the user a set of yes/no questions about the behavior of the client code, to remedy inherent limitations in the TAPIR analysis; all those questions concern properties the client developer would have to consider anyway if performing the patching manually.

We envision that semantic patches can be written either by the library developer or by someone familiar with the library code, along with the customary changelogs. With JSFIX, all the clients of the library then benefit from the mostly automatic adaptation of their code (and the client developers do not need to understand the notation for semantic patches).

To summarize, the contributions of this paper are:

- We propose a notion of code templates to formalize the transformations required to adapt client code to typical breaking changes. A TAPIR detection pattern together with a code template form a semantic patch. We present the JSFIX tool, which, based on a collection of semantic patches, semi-automatically adapts client code to breaking changes in a library update (Sections 10.3 and 10.4).
### 10.2 Motivating Example

The `rxjs` library[^1], with more than 17 million weekly downloads, is a popular library for writing reactive JavaScript applications. In April 2018, `rxjs` was updated to version 6.0.0, a massive major update introducing many new features, bug fixes, and performance improvements, but unfortunately also many breaking changes. Our manual investigation of the `rxjs` changelog shows that it contains at least 38 separate breaking changes, many of which involve multiple functions and modules. The developers of `rxjs`, who were probably well aware that these breaking changes would

[^1]: [https://www.npmjs.com/package/rxjs](https://www.npmjs.com/package/rxjs)

---

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>import { Observable } from 'rxjs/Observable';</td>
</tr>
<tr>
<td>2</td>
<td>import { Subject } from 'rxjs/Subject';</td>
</tr>
<tr>
<td>3</td>
<td>import 'rxjs/add/observable/timer';</td>
</tr>
<tr>
<td>4</td>
<td>import 'rxjs/add/operator/takeUntil';</td>
</tr>
<tr>
<td>5</td>
<td>+ import { defaultIfEmpty, take, takeUntil, tap }</td>
</tr>
<tr>
<td>6</td>
<td>+ from 'rxjs/operators';</td>
</tr>
<tr>
<td>7</td>
<td>+ import { timer, Subject } from 'rxjs';</td>
</tr>
<tr>
<td>8</td>
<td>- const c$ = (new Subject()).take(1);</td>
</tr>
<tr>
<td>9</td>
<td>- Observable.timer(warnTimeout)</td>
</tr>
<tr>
<td>10</td>
<td>- .takeUntil(c$.defaultIfEmpty(true))</td>
</tr>
<tr>
<td>11</td>
<td>- .do() =&gt; {...}</td>
</tr>
<tr>
<td>12</td>
<td>- .subscribe();</td>
</tr>
<tr>
<td>13</td>
<td>+ const c$ = (new Subject()).pipe(take(1));</td>
</tr>
<tr>
<td>14</td>
<td>+ timer(warnTimeout).pipe(</td>
</tr>
<tr>
<td>15</td>
<td>+ takeUntil(c$.pipe(defaultIfEmpty(true))),</td>
</tr>
<tr>
<td>16</td>
<td>+ tap() =&gt; {...}</td>
</tr>
<tr>
<td>17</td>
<td>+ ).subscribe();</td>
</tr>
</tbody>
</table>

Figure 10.1: Excerpts from `redux-logic` before (marked with red background and ‘-’) and after (green and ‘+’) adapting to the breaking changes in the library `rxjs` 6.0.0.

- We propose an interactive mechanism for filtering away false positives from the detection pattern matches reported by TAPIR (Section 10.5).
- We present the results of an evaluation based on 12 major updates of popular npm packages and 203 clients, showing that most breaking changes can easily be expressed as semantic patches, and that JSFIX in most cases succeeds in making the clients compatible with the new versions of the libraries. Furthermore, the evaluation demonstrates the practicality of JSFIX. It takes only a few seconds to patch, on average, 3.8 affected locations per client, with only 2.7 questions to the user, and 31 pull requests based on the output from JSFIX have been accepted, which shows that the quality of the patches is good enough for practical use (Section 10.6).
discourage many client developers from upgrading, decided to create both an auxiliary compatibility package that introduces temporary workarounds and a migration guide detailing how clients should adapt to all the breaking changes in the new version of rxjs. While the migration guide is quite helpful (and also not something provided with most major updates of other libraries), it may still take a significant amount of work for a client developer to upgrade to rxjs 6.0.0. Consider, for example, the redux-logic package that depends on rxjs. In September 2018, redux-logic was updated to depend on rxjs 6.0.0. This update required 784 additions and 308 deletions to 21 files over 3 commits by no means a small task.

Figure 10.1 shows two excerpts of the update of redux-logic to rxjs 6.0.0 (modulo some newlines and insignificant differences in variable naming). The first change is that Observable is no longer imported from 'rxjs/Observable' as in line 1, in fact it is not imported at all. This is because the timer function (Observable.timer in line 9) in rxjs 6.0.0 should be accessed directly from 'rxjs' as can be seen by the import in line 7. Therefore line 14 is also updated to use timer directly. The second change is that Subject should be imported from 'rxjs' instead of 'rxjs/Subject', which is why the import in line 2 is replaced with the import of Subject in line 7. The imports in lines 3–4 add properties to Observable and rxjs observables (through Observable.prototype), but have been removed in the new version. Instead of using those properties, the functions should now be imported from either 'rxjs' or 'rxjs/operators' as can be seen in the imports on lines 5 and 7. Line 8 used one of these properties, take, which in the new version should be replaced with a call to .pipe, where the operator function (take) is then provided as an argument, as shown in line 13 in the patched code. For the same reason, lines 9–12 have been updated to use .pipe in lines 14–17. Evidently, adapting client code to breaking changes in a library can be difficult and time-consuming, so tool support is desirable.

While the redux-logic example is one of the more extreme cases, it clearly demonstrates that updating dependencies is no minor undertaking. Considering that the average npm package already in 2015 had an average of 5 direct dependencies and that number has been growing over time [140], keeping everything up-to-date becomes insurmountable.

Using JSFIX it would have been possible for the redux-logic developer to adapt the client code almost automatically. Given a collection of semantic patches that describe the breaking changes in the library, JSFIX is designed to both find the locations in the client code that are affected by the breaking changes, and to adapt those parts of the client code to the new version of the library. The analysis that finds the affected locations is designed such that it leans towards over-approximating, meaning that it may flag too many source code locations as potentially requiring changes, but rarely too few. When it cannot establish with complete certainty whether some source location is affected by the breaking changes, it asks the client developer for advice.

---


10.3. **OVERVIEW**

In this specific case, the *redux-logic* developer would only have to answer 14 simple yes/no questions, which all concern only *redux-logic* (not *rxjs*).

For transforming the excerpts shown in Figure 10.1, *JSFIX* does not ask any questions. However, suppose the analysis were too imprecise to determine that $c$ on line 10 is an *rxjs* observable, then *JSFIX* would have asked this question:

```
src/createLogicAction$.js, 10:14:10:36:
Is the receiver an *rxjs* observable?
```

All the 14 questions are of this kind but with different source code locations, and they all originate from such analysis imprecision. With the help from the *redux-logic* developer, the uncertainty can be resolved, and the patches produced by *JSFIX* for the affected locations successfully adapt the *redux-logic* source code to the new version of the *rxjs* library.

Comparing the *JSFIX* autogenerated transformations with the patches made manually by the developer of *redux-logic* shows that the transformations are identical (ignoring white-space and the order of property names in imports).

### 10.3 Overview

The tool *JSFIX* is designed to adapt client code to breaking changes in libraries. An execution of *JSFIX* is divided into three phases: (1) an analysis phase, (2) an interactive phase, and (3) a code transformation phase. As input it takes a client that depends on an old version of a library, together with a collection of semantic patches that describe the breaking changes in the library. Each semantic patch contains a detection pattern that describes where a breaking change occurs in the library API and a code template that describes how to adapt the client code. We explain the notion of semantic patches in more detail in Section 10.4. As output, *JSFIX* produces a transformed version of the client that, under certain assumptions described in Section 10.5, preserves the semantics of the old client code but now uses the new version of the library.

The analysis phase uses the *TAPIR* [101] light-weight static analysis to detect locations in client code that may be affected by breaking changes in the library. We treat *TAPIR* as a black-box component as it is only loosely coupled with the other phases of *JSFIX*. The input to *TAPIR* consists of the client code and the detection patterns coming from the semantic patches, and as output it produces a set of locations in the client code that match the detection patterns, meaning that they may be affected by the breaking changes in the library.

Being fully automatic, *TAPIR* cannot always find the exact set of affected locations, but the analysis is designed such that it leans towards over-approximating, meaning that it sometimes reports too many locations but rarely too few. Moreover, it is capable of classifying each match being reported as either a high or a low confidence match. In practice, all false positives appear among the low confidence matches, meaning that only those need to be manually validated. In *JSFIX*, we take advantage of that confidence information. In the second phase of *JSFIX*, it asks the user for help at each
low confidence match, such that the false positives from TAPIR can be eliminated. The text for the questions to the user comes from the semantic patches. The questions all take yes/no answers, and they concern only the client code, not the library code. We describe the interactive phase in more detail in Section 10.5 where we also give additional representative examples of questions presented to the user.

Next, JSFIX runs a transformation phase where the client code is patched to adapt to the changes in the library. The transformations are specified using a form of code templates that specify how each affected location should be transformed to become compatible with the new version of the library. The transformation process is explained together with the notation for semantic patches in the next section.

### 10.4 A Semantic Patch Language

To adapt client code to breaking changes in a library, the parts of the client code that use the affected parts of the library API must be transformed accordingly.

#### Example 1

Lines 18–19 in the following program use the `max` function from the `lodash` library.

```javascript
18 var _ = require('lodash');
19 - _.max(coll, iteratee, thisArg);
20 + _.max(coll, iteratee.bind(thisArg));
```

The optional third argument on line 19, `thisArg`, lets the client specify a custom receiver of the second argument, `iteratee`. In version 4.0.0 of lodash, the support for the third argument was removed from 64 functions, including the `max` function. To restore the old behavior, clients using `max` or one of the other 63 functions would have to explicitly bind `thisArg` to `iteratee`. For example, for the program above, line 19 has to be transformed by inserting a call to `bind` as shown on line 20.

#### Example 2

Lines 21–23 in the program below use the `async` library’s `queue` data structure, which holds a queue of tasks (asynchronous functions) to be processed.

```javascript
21 var async = require('async');
22 var q = async.queue(...);
23 - q.drain = () => console.log('Done');
24 + q.drain(() => console.log('Done'));
```

On line 23 a function is written to the `drain` property of the queue. In version 2 of async, this function is called when all tasks in the queue have been processed. However, in version 3 of async, `drain` is no longer a property the client should write, but instead a function the client should call. The function to be called once the queue has been processed is then passed as an argument to `drain`. Hence, the call to `drain` must be transformed as shown on line 24.

#### Example 3

Lines 25–26 below import the `find` and `map` functions from the `rxjs` library.

```javascript
25 - import find from 'rxjs/operator/find'
26 - import map from 'rxjs/operator/map'
27 + import {find, map} from 'rxjs/operators'
```
10.4. A SEMANTIC PATCH LANGUAGE

\[ \alpha \in \text{Expression} ::= \ldots \]
\[ \quad | \; \$\; \text{RefElement} (\;:\;\text{RefElement})^* \]
\[ \quad | \; \langle \text{ModuleElement} \rangle^+ \]
\[ \quad | \; \# i \; \text{Replacer}^? \]

\[ \text{RefElement} ::= \text{prop} \; \text{Replacer}^? \mid \text{value} \mid \text{base} \mid \text{callee} \mid i \mid \text{args} \; \text{Selector}^? \]

\[ \text{Replacer} ::= [s_1 \Rightarrow s'_1, \ldots, s_n \Rightarrow s'_n] \]

\[ \text{Selector} ::= [j, k] \mid [j] \]

\[ \text{ModuleElement} ::= s \mid / \mid \# i \; \text{Replacer}^? \]

Figure 10.2: Grammar for code templates. The ‘...’ in the first production refers to the ordinary constructs of JavaScript expressions. The notation \( X^*, X^?, \) and \( X^+ \) mean zero-or-more, zero-or-one, and one-or-more occurrences of \( X \), respectively. The meta-variable \( s \) ranges over strings, \( i \) ranges over positive integers, and \( j \) and \( k \) range over integers.

In version 6 of \texttt{rxjs}, these import paths \texttt{'rxjs/operator/find'} and \texttt{'rxjs/operator/map'} are no longer available. Instead, clients must import the functions from \texttt{'rxjs/operators'} as demonstrated by the transformed import on line 27.

To automate the transformation of the client code, we define a suitable notion of semantic patches. A \textit{semantic patch}, \( \rho \sim \alpha \), models a breaking change and consists of a detection pattern \( \rho \) that identifies the affected part of the library API (the affected location), and a \textit{code template} \( \alpha \) that describes how client code that uses that part of the API can be transformed to adapt to the new version of the library. A semantic patch can also contain question text for the interactive phase, which we describe in Section 10.5. The detection patterns are identical to the patterns used by the API access point detection tool \texttt{TAPIR}, so we omit a detailed description of the pattern language in this paper. Although we treat the accompanying algorithm that performs the matching between the patterns and the client code as a black box, as mentioned in Section 10.3 we provide intuitive explanations of the meaning of the concrete detection patterns that appear in examples in the remainder of this paper.

To adapt a client that uses some library with breaking changes, for example, to perform the transformations in Examples 1–3, we need a mechanism for specifying the required transformations. For this purpose, we introduce the notion of a \textit{code template}, which is an incomplete JavaScript expression that has one or more missing pieces (or holes) that must be instantiated with other JavaScript expressions for the template to become a syntactically valid JavaScript expression.

We can view a code template as a form of meta-program that takes one or more expressions as input and then interpolates these expressions into the holes of the template to form a valid JavaScript expression. The key idea behind the templating mechanism is that the holes of the template are instantiated with code from the vicinity of the location in the client code that is matched by the detection pattern \( \rho \). A detection pattern can either match a call, a property read, a property write, or a module import.
In a transformation, parts of the subtree of the matched AST node have to be replaced (as in Example 1) or, in some cases, the kind of the matched AST node has to change (as in Example 2 where a property write operation is changed into a method call). In either case, the variable names and literals used in the original client code typically also have to appear in the transformed version of code for the transformation to be correct. For example, it is essential that the function written to the `drain` property on line 23 is the same function passed to the `drain` function on line 24.

To facilitate these kinds of transformations, we introduce an AST reference notation that is used to specify both the holes of the templates and how expressions should be retrieved for these holes. The idea is that one can use this notation to interpolate expressions into the template, where these expressions are retrieved relative to the AST node matched by $\rho$. For example, if $\rho$ matches a call node, then an AST reference can be used to obtain, for example, the receiver or the arguments of that call. While AST references technically reference AST nodes, it is often more convenient to think of them as references to expressions since all the allowed AST references always point to nodes whose subtrees form expressions.

Figure 10.2 shows the grammar for code templates. A code template is a JavaScript expression (Expression) that can contain some special constructs explained in the following.

**AST references** An AST reference consists of a ‘$’ symbol followed by a list of ‘:’-separated elements specifying which node is referenced relative to the node matched by $\rho$. Six kinds of AST reference elements (RefElement) are available:

- prop refers to the property name $p$ in a property access $e.p$ of a method call, property read, or property write expression.
- value refers to the right-hand-side expression $e_2$ of a property write expression $e_1.p = e_2$.
- base refers to the receiver value $e$ in a property access $e.p$ of a method call, property read, or property write expression.
- callee refers to the function value of a function, method, or constructor call.
- $i$ refers the $i$'th argument of a call.
- args refers to all the arguments of a call.

**Example 4** In Figure 10.3, we show examples of which nodes various AST references refer to, relative to a call node (left) and a property write node (right). Notice how ‘:’ is used to combine references: $\$base:base$ refers to the receiver of the receiver, $\$base:callee$ refers to the function that is called to compute the receiver of the `sum` method call, and $\$base:args$ refers to the arguments passed to this function.

Not all the different kinds of AST references make sense for every kind of detection pattern, for example, $\$value$ is only meaningful if transforming a property write. If a user writes a template that is invalid for a node matched by the detection pattern, for example, if $\$value$ is used when transforming a method call, or if $3$ is used when
10.4. A SEMANTIC PATCH LANGUAGE

transforming a call with fewer than three arguments, then JSFIX is unable to perform the transformation and instead reports an error.

The syntax $\text{args}[j, k]$ denotes a slice of the arguments, from the $j$'th until (and including) the $k$'th argument. (The notation $j$ can thus be seen as an abbreviation of $\text{args}[j, j]$.) Negative numbers count from the right, for example $-1$ denotes the last argument. The variant $\text{args}[j,]$ refers to every argument from $j$ and onwards.

Example 5  Consider the call expression on the left of Figure 10.3. We can obtain a reference to the individual arguments of $\text{sum}$ using the argument index reference, for example, $\text{2}$ refers to $\text{ys}$. We can similarly select slices of the arguments. For example, $\text{args}[1, 2]$ results in the slice $\text{x}_s, \text{ys}$, and $\text{args}[-2, -2]$ results in the slice $\text{x}_s$.

Being able to select arguments counting from right to left is sometimes needed for selecting the last argument in a variadic function as we demonstrate in Example 17.

Example 6  Continuing Example 1, the calls to $\text{max}$ that have to be transformed are exactly those calls where $\text{max}$ is called with three arguments and the second argument is a function. Hence, the following semantic patch will automate the update of the $\text{max}$ calls:

```
call <lodash>.max [3, 3] 2:function $\text{callee}(\text{1}, \text{2}.\text{bind}(\text{3}))
```

The detection pattern matches the calls to the $\text{max}$ function on the $\text{lodash}$ module object, where $\text{max}$ is called with exactly three arguments and the second argument has type $\text{function}$. The code template specifies that those calls must be transformed such that the method call (where $\text{callee}$ refers to the function value) has the same first argument ($\text{1}$), but where the second argument is the result of calling $\text{bind}$ on the old second argument ($\text{2}$) passing the old third argument ($\text{3}$) as an argument to $\text{bind}$. Hence, the transformation that is applied is exactly the one shown on line 20 in Example 1.

Example 7  Continuing Example 2, the following semantic patch expresses the required transformation:

```
write <async>.queue().\text{drain} \rightarrow \text{base}.\text{drain}(\text{value})
```

The detection pattern matches writes of the $\text{drain}$ property on objects returned by calls to the $\text{queue}$ function on the $\text{async}$ module object. These writes are transformed such

---

$^5$The detection pattern of the semantic patch can easily be extended to also match many other functions affected by the thisArg breaking change.

$^6$The $[j, k]$ call filter restricts the pattern to only match calls with between $j$ and $k$ arguments, i.e. calls to $\text{max}$ with exactly 3 arguments in this case.
that drain is instead invoked on the queue object (referenced by \$base), such that the value previously written to drain (\$value) is passed as an argument to drain.

The notation \[s_1 \Rightarrow s'_1, \ldots, s_n \Rightarrow s'_n\] allows us to express identifier replacements, as shown in the following example.

**Example 8** Among the breaking changes in lodash 4, the function any is renamed to some, and all is renamed to every. We can capture both using a single, concise semantic patch (where \{any, all\} matches either any or all):

```
read <lodash>.{any, all} \~ \$base.$prop[any \Rightarrow some, all \Rightarrow every]
```

**Module imports** Many breaking changes in libraries involve the structure of their modules. With the npm module system, modules are files that are loaded using import (as in Example 3) or using require (as in Example 1). We provide the notation `<ModuleElement+>` for transforming and adding module loading. As an example, the code template `<rxjs/operators>.find` will generate code that ensures that the 'rxjs/operators' module is loaded and then access its find property. Using this special notation instead of simply using calls to require in code templates to load modules has several advantages: (1) it will move all module loads to the outer-most scope, which is the more idiomatic way to load modules in JavaScript; (2) if loading a module from a package that the client currently does not depend upon, JSFIX will add that package as a dependency to the client’s package.json file; (3) it will ensure that the same module is not loaded multiple times, which could happen if using require in a code template and that code template is used for multiple transformations in the same file; and (4) it will use the same style of module loading as the client already uses, i.e., require or import.

The detection patterns supported by TAPIR for specifying module names can contain wildcards (e.g. *) and sets of filenames (e.g. {find, map}). To allow the code templates to refer to the corresponding parts of the module names, we provide the notation \#i to refer to the \(i\)’th non-constant component of the detection pattern. (This mechanism is inspired by the use of capturing groups and backreferences in traditional regular expression notation.)

**Example 9** Continuing Example 3, we can express the required transformation with the following semantic patch:

```
import rxjs/operator/{find, map} \~ <rxjs/operators>#1
```

The detection pattern matches imports from the modules 'rxjs/operator/find' and 'rxjs/operator/map' (the wildcard \{find, map\} matches both 'find' and 'map'), and the code template says that these imports must be transformed to reads of, respectively, the find or map property from the 'rxjs/operators' module. Notice we use the \#1 syntax to refer to the string matched by \{find, map\} in the detection pattern.

**Example 10** In some cases, adapting code to a breaking change is only possible if the client adds a new dependency. For example, in version 3.0.0 of the uuid library, the parse and unparse methods are removed, and clients must use the uuid-parse package
if they want to keep using the removed methods. When using the \(<M>\) syntax, JSFIX will automatically add the dependency that contains the module \(M\), and therefore the transformation can be expressed using this semantic patch:

\[
\text{call } \langle \text{uuid}.\{\text{parse, unpars}\} \rangle \Rightarrow \langle \text{uuid-parse}.\$.prop($args)\rangle
\]

Semantic patches can also be useful for post-processing transformed code to improve its readability and performance as demonstrated by Example 18.

## 10.5 Interactive Phase

The TAPIR analysis that we use for the analysis phase is designed to over-approximate the AST nodes, which means that we can generally trust that no locations are missed (see the discussion about correctness assumptions at the end of this section), but some mechanism is needed for removing spurious matches. JSFIX does not require the user to manually inspect all the potential matches found in the analysis phase, to eliminate the false positives before the transformation phase. Instead, it generates a set of questions to the user about the client code, and then performs the filtering based on the answers to these questions. The questions only concern the library usage at the potentially affected locations, and the user generally does not need to be aware of the specifics of the breaking changes. In practice, JSFIX asks 0.7 questions per code transformation on average (see Section 10.6).

There are two main sources of precision loss in the TAPIR static analysis that cause JSFIX to ask questions to the user. The first kind of precision loss (here named OBJ) occurs when the static analysis is unsure if the object, on which the potentially broken operation takes place, is the right type of object. To make it possible for client developers to use JSFIX without requiring them to understand the semantic patch language, we let the authors of the semantic patches manually write the questions in a client-understandable style.

**Example 11** Consider the code

```javascript
28\ const f = (x, y) => x.map(y);
```

and the following semantic patch for rxjs version 6:

\[
\text{call } <\text{rxjs}>?**.map \Rightarrow \text{Base.pipe(<rxjs/operators}.map($arg$))}
\]

The detection pattern matches reads of the map property on any chain of operations on the rxjs module. The detection pattern also contains a ‘?’, which means that, in particular, the expression `x.map(y)` will match even when TAPIR is too imprecise to detect whether the \(x\) object comes from rxjs. TAPIR marks such matches as *low confidence*, which causes JSFIX to ask the user for validation. If \(x\) happens to be, for instance, an ordinary JavaScript array rather than an object from rxjs, then applying the transformation would break the code. For this example, JSFIX will ask the question “Is the receiver an rxjs observable?” (together with the source code location of the match), and only apply the patch if the answer is “yes”.
The second kind of precision loss (here named \texttt{CALL}) occurs when \texttt{TAPIR} is unable to determine if constraints on arguments (so-called call filters) are satisfied.

\textbf{Example 12} Consider the following code:

\begin{verbatim}
29 var _ = require('lodash');
30 const f = (x, y) => _.pick(x, y);
\end{verbatim}

The \texttt{pick} function from \texttt{lodash} is called with two arguments, an object and a property selector, and it returns an object with all the properties that satisfy the selector. The selector can either be a predicate function, or a list of strings such that all property names that appear in that list are selected. For the former case, clients should replace calls to \texttt{pick} with calls to \texttt{pickBy} when upgrading to \texttt{lodash} version 4. This transformation is captured by the following semantic patch:

\begin{verbatim}
call <lodash>.pick [2, 2] 2:function
$base.pickBy($args)
\end{verbatim}

For this example, \texttt{JSFIX} will automatically generate the question “\texttt{Is the argument y of type function in line 30}”, and only perform the transformation if the answer is “yes”. Notice how answering the question does not require any knowledge about the breaking change, but only some basic understanding of the client source code. Therefore, the client developer who is familiar with the code will likely find the question easier to answer compared to manually trying to understand the breaking change and performing the transformation. In particular, the developer would have to consider anyway whether argument \texttt{y} is of type function in line \texttt{30}.

While the primary purpose of the interactive phase is to filter away spurious AST nodes to avoid redundant or wrong transformations, \texttt{JSFIX} also has support for two other categories of questions used when: (1) the detection patterns are too coarse-grained to determine if a breaking change can occur (see Example 19), and (2) a breaking change has relatively minor implications that some clients may prefer not to fix (see Example 13). We refer to these two categories as \texttt{EXTRA} and \texttt{MINOR}, respectively.

\textbf{Example 13} The \texttt{node-fetch} library is a polyfill for the browser HTTP API \texttt{window.fetch}. In the update of \texttt{node-fetch} to version 2, the \texttt{json} method on response objects is modified such that it throws an error instead of returning an empty object when the HTTP response code is 204. A semantic patch for this breaking change can be expressed as follows.

\begin{verbatim}
call <node-fetch>?**.json 2:function ~> $base.json($args)
\end{verbatim}

The transformed code calls \texttt{json} if the response is not equal to 204 and otherwise results in an empty object.

While this transformation is semantically correct, it is unlikely to be the desired solution for the client developer. A more idiomatic solution would be to catch the error, and then handle it accordingly. For that reason, we allow semantic patches to be marked as “low priority”. For such patches, instead of always asking for each potential match, the user now also gets the options to select “yes to all” or “no to all”.

\begin{verbatim}
call <node-fetch>?**.json ~> ($base.status === 204 ? {} : $base.json($args))
\end{verbatim}

The transformed code calls \texttt{json} if the response is not equal to 204 and otherwise results in an empty object.
10.6 Evaluation

We have implemented JSFIX in only 1200 lines of TypeScript. Apart from the detection analysis and transformation phase as presented in Section 10.4, JSFIX also performs some auxiliary tasks to improve the transformed code. For example, imports that are unused after the transformations and multiple imports of the same module are automatically removed. We evaluate JSFIX by answering the following research questions:

**RQ1** For how many of the breaking changes in major updates of widely used npm packages can the patch be expressed as a code template, and how complex are the code templates?

**RQ2** For clients that are affected by breaking changes in a library, do the applied transformations make them compatible with the new version of the library? Are the transformations of sufficient quality that client developers are willing to accept them as pull requests?

**RQ3** How many questions and which types of questions does JSFIX typically ask the user in the interactive phase?

The part of JSFIX concerning detection patterns is evaluated previously [101] and is therefore not considered in this evaluation.

**Experiments** To address RQ1, we first performed an experiment where we attempted to write semantic patches for as many as possible of the breaking changes appearing in major updates of widely used npm packages. The full list of semantic patches is shown in the supplementary material on the paper webpage. We were able to write these in only a few days without expert knowledge of any of the libraries. For RQ2 and RQ3, we performed a second experiment to test if JSFIX can, based on [https://cs.au.dk/~amoeller/papers/jsfix/supplementary-material.pdf](https://cs.au.dk/~amoeller/papers/jsfix/supplementary-material.pdf)
the semantic patches written in the first experiment, patch clients whose test suites fail when switching to the new library versions. Also for RQ2 and RQ3, we finally performed a third experiment, where we created pull requests with the transformations generated by JSFIX for a number of clients, to determine if the quality of the transformations is acceptable to client developers. The two latter experiments are a best-effort attempt to establish some confidence that the transformations created by JSFIX are correct; in the second experiment by showing that the transformations are correct enough to fix the broken test suites and not introduce new test failures, and in the third experiment by showing that the reviewers of the pull requests trust the transformations enough to accept them.

**Benchmark selection** Our experiments for answering the research questions are based on 12 major updates of top npm packages (selected from the TAPIR benchmark suite [101]), shown in the first column of Table 10.1.

To address RQ2 and RQ3, we selected the 89 clients from the evaluation of TAPIR that are known to be affected by one of the 12 major updates. Since the test suites of the clients succeed prior to the update but fail afterwards, we know that these clients are affected by some of the breaking changes. Therefore, these clients require patches to become compatible with the new major version of the library.

For the third experiment, we used only those 41 of the 89 clients where no version of the client existed that already depended on the new major version of the library with breaking changes. (Comparing the patches that have been applied to the already updated clients with patches generated by JSFIX remains an interesting opportunity for future work.) In addition, we randomly selected 10 clients for each benchmark that, at the time of the selection, depended upon the benchmark in the major-range below the one of the benchmark, e.g., for the lodash 4.0.0 benchmark, we selected 10 clients that depend on some version of lodash below 4.0.0 but at least 3.0.0. We also required that these clients were updated within 180 days, since maintainers of recently updated libraries are presumably more likely to react to pull requests. For express, we could only find such 4 clients, so in total we consider 155 clients of the 12 libraries for the pull request experiment.

**RQ1 (Expressiveness of transformations)**

A total of 326 detection patterns were required for describing the breaking changes in the 12 libraries. Most of these detection patterns are identical to the patterns from the TAPIR paper [101], but in some cases we had to split a pattern if, for example, different code templates were required depending on the number of function arguments at a call pattern.

---

8We omitted three of the TAPIR benchmarks because they contain only a few breaking changes that are all unlikely to require any form of patching (for example, changes to formatting of a help message in the commander library) and are therefore not interesting for JSFIX.

9In the evaluation of TAPIR, 115 clients were considered, but some of them are using the omitted libraries mentioned in footnote 8 and some of them are written in languages that compile to JavaScript, which means that JSFIX cannot patch the source code of those clients.
A summary of the results for each library update is shown in Table 10.1, where “BC” is the number of breaking changes in the update, “Patterns” is the number of detection patterns written, “Temp” is the number of code templates written, “U” (Unexpressible) is the number of detection patterns for which the required transformation cannot be expressed in our code template language, “NGP” (No general patch) is the number of detection patterns where, according to our knowledge of the breaking change, no single fix exists that applies to every location affected by that breaking change, “?” (Unknown) is the number of breaking changes for which the changelog and associated resources like GitHub issues were too incomplete for us to understand the breaking changes well enough to write a correct semantic patch. Representative examples of breaking changes in categories “Unexpressible” and “No general patch” are shown in Examples 14 and 15.

**Example 14** As an example of a breaking change in the “Unexpressible” category, the changelog of the web framework express for version 4 contains this item: “app.router - is removed.” While it is easy for JSFIX to detect where app.router is used, upon closer inspection it turns out that the breaking change has larger implications. The whole semantics around routing has changed such that the order in which routes (HTTP end-points) and middleware (plugins run as intermediate steps in the request/response cycle) are registered on the express application object needs to change. Consider the following excerpt of an application that uses express version 3:

```javascript
31 var app = require('express')();
32 app.use(app.router);
33 // middleware
34 app.use(function(req, res, next) { ... });
35 ...
36 // routes
37 app.get('/' ...);
38 app.post(...);
```

In express version 3, the code `app.use(app.router)` makes express handle the registered routes before the middleware registered in later calls to `app.use` in the request/response cycle. In version 4 of express, the call to `app.use` on line 32 must be removed. However, due to the change in the ordering semantics, the call to `app.use` on line 34 must move below the calls to `app.get` and `app.post` on lines 37 and 38. Such a change is not currently expressible in the transformation language, and probably also out of scope for what an automated technique can realistically be expected to handle. To perform this change, a total ordering of how middleware and routes are registered to the express app is required, which is not easily obtained. Notice that JSFIX can still detect reads of `app.router`, so the user is being notified about this breaking change, but the transformation must be applied manually.

**Example 15** An example of a breaking change in the “No general patch” category appears in the node-fetch HTTP library. In version 1 of node-fetch, clients could use the `getAll(name)` method on header objects to get an array of all header values for the `name` header. In version 2 of node-fetch this method is removed, so clients must now resort to using the `get(name)` method that instead returns a comma-separated string value of the `name` header values. It might seem that this breaking change is easily fixed
CHAPTER 10. SEMANTIC PATCHES FOR ADAPTATION OF JAVASCRIPT PROGRAMS TO EVOLVING LIBRARIES

Table 10.1: Experimental results for RQ1.

<table>
<thead>
<tr>
<th>Library</th>
<th>BC</th>
<th>Patterns</th>
<th>Temp</th>
<th>U</th>
<th>NGP</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>lodash 4.0.0</td>
<td>51</td>
<td>123</td>
<td>123</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>async 3.0.0</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>express 4.0.0</td>
<td>18</td>
<td>24</td>
<td>23</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>chalk 2.0.0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bluebird 3.0.0</td>
<td>8</td>
<td>12</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>uuid 3.0.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rxjs 6.0.0</td>
<td>26</td>
<td>55</td>
<td>53</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>core-js 3.0.0</td>
<td>26</td>
<td>41</td>
<td>35</td>
<td>0</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>node-fetch 2.0.0</td>
<td>9</td>
<td>9</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>winston 3.0.0</td>
<td>23</td>
<td>30</td>
<td>26</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>redux 4.0.0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>mongoose 5.0.0</td>
<td>14</td>
<td>19</td>
<td>13</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>186</td>
<td>326</td>
<td>298</td>
<td>7</td>
<td>19</td>
<td>2</td>
</tr>
</tbody>
</table>

by replacing `getAll` with `get` and then splitting the resulting string at all commas:

```javascript
call <node-fetch>?**.getAll
$base.get($args).split(',')
```

However, since commas may also appear inside each of the header values, the resulting array may not be correct. Since there are no other value separators in the string, this breaking change does not have a general patch.

The number of breaking changes is smaller than the number of detection patterns, because we count each bullet in the changelogs as one breaking change (as in [101]). Some bullets may only concern a single method or property, while others concern tens of methods or properties. Hence the number of breaking changes should only be viewed as a weak indicator of how extensive the impact of each major update is.

We were able to write code templates for 298 of the 326 detection patterns. The remaining 28 patterns fall into three different categories: (1) For 7 patterns, the code template language is not expressive enough to describe the required transformation (see Example 14); (2) for 19 patterns, according to our knowledge of the breaking change, no general patch exists that will work for all clients (see Example 15); (3) for 2 patterns, the breaking change was not documented sufficiently well for us to write a correct template.

For the 298 cases where we successfully managed to write a code template, the biggest challenge was to understand how to address the breaking changes. Not all changelogs specify in detail how clients should migrate, so we sometimes had to rely on, for example, observations of how existing clients have upgraded. For example, the update of `uuid` to version 3 removes the `parse` and `unparse` methods, but does not specify what clients should use as alternatives. While searching for solutions, we came across a package named `uuid-parse`, which contains exactly the two methods removed from `uuid` in version 3. Writing the required semantic patch that replaces the `parse` and `unparse` method calls from `uuid` with calls to the methods from `uuid-parse` was then a simple matter (see Example [10]). This again demonstrates one of the strengths
of our approach: Instead of requiring every client developer to understand the details of the breaking changes, once a semantic patch has been written, it can be reused for many clients.

**Example 16** Some of the breaking changes required more sophisticated semantic patches. Consider the `whilst(test, iteratee, callback)` function of the `async` library, which implements an asynchronous while loop where `iteratee` (the loop body) is called as long as `test` (the loop condition) is succeeding, and `callback` is called on an error or when the iteration ends. In version 2 of `async`, `test` is expected to be synchronous, that is, it should provide its return value through a normal return statement. However, in version 3 of `async`, `test` is expected to be asynchronous, and must therefore instead provide its return value by calling a callback. The modifications required are expressed by the following semantic patch:

```javascript
const cb = arguments[arguments.length - 1];
const args = Array.prototype.slice.call(arguments, 0, arguments.length - 1);
try {
  cb(null, $1.apply(this, args));
} catch (e) {
  cb(e, null);
}
```

The code template wraps the test function in a new function (lines 10.2–10.11), which extracts the new callback (line 10.3), calls the old test function, and passes the result to the callback (line 10.7). Using a wrapper function like this is unlikely to be the preferred choice if a developer were to perform the update manually. In that case, a simpler and more idiomatic solution would be to modify the test function itself by adding the callback to its argument list, and replacing all of its return statements with a call to this callback. However, that solution will only work if the definition of the test function is available and never throws an error, which may not be the case if, for example, the test function is imported from a library, which is why we resort to using the more general wrapper function.

It is not always possible to express a transformation that preserves the semantics (see Example 15). However, the experiments show that such situations are rare, so this does not pose a major threat to the applicability of the technique.

In conclusion, we have found that writing the templates is relatively simple for most breaking changes, but that some code templates have to be quite general and non-idiomatic to preserve the semantics in every case. While writing a template is sometimes more difficult than transforming the client code manually, the fact that the template is reusable across all clients of the library, makes the investment worthwhile.

**RQ2 (Correctness and quality of transformations)**
Client test suites experiment To test if JSFIX can repair broken clients, we ran JSFIX on 89 clients whose test suite succeeded before the update and failed when switching to the new version of the library. We used JSFIX to patch the client code, and then we checked if the test suite of the patched client passed. This is of course not a guarantee that the patches are correct. However, if none of the test suites fail due to missed or wrong transformations, it is a strong indication that JSFIX can successfully patch the client code. To increase confidence further, we also consider feedback from client developers (see the pull request experiment below).

The results of this experiment are shown in the first 11 columns of Table 10.2: “C” is the number of clients, “Tr” is the number of transformations done by JSFIX, “✓” (resp. “✗”) is the number of client test suites succeeding (resp. failing) after the patching, “−CT” is the number of clients that are affected by a breaking change for which it is impossible to write a correct template in our semantic patch language, and “Time” is the average time (in seconds) used for the detection and patching phases per client, excluding time spent on parsing the client code. The last four columns are described in Section 10.6.

The patches produced by JSFIX are successful in making 82 of the 89 clients pass their test suites. Of the remaining 7 clients, 5 are affected by breaking changes for which it is not possible to write a correct template. For example, the three express clients in this category are affected by the breaking change presented in Example 14. The remaining 2 clients do not fail due to unhandled breaking changes, but they contain testing code that is indirectly affected by changes to the library’s API. For example, the rxjs client whose test suite fails asserts that some specific properties exist on rxjs.Observable.prototype, however, these properties have been removed and JSFIX has removed all usages of those properties, so the assertions can safely be removed. Similarly, a test in a redux client fails due to a bug fix in redux such that a message is no longer written to console.error. As such a bug fix is not considered a breaking change, it is not the responsibility of JSFIX to address this problem.

For the 84 clients where all code templates were expressible, JSFIX made a total of 451 changes to the client code. The number of changes differs substantially between the benchmarks, ranging from 1 per client for node-fetch, redux, and mongoose to 33 per client for rxjs. This discrepancy is expected since the number of breaking changes varies considerably between major updates as shown in Table 10.1. The likelihood of a client using a broken API also fluctuates across the benchmarks. For example, rxjs has many changes in commonly used APIs, and therefore updating clients of rxjs is a cumbersome and time-consuming task, which may explain why developers released rxjs-compat and a migration guide along with the changelog. However, the rxjs breaking changes are also easily expressible as semantic patches, which means that JSFIX could patch all of the breaking changes in the 6 clients.
Table 10.2: Experimental results for RQ2 and RQ3.

<table>
<thead>
<tr>
<th>Library</th>
<th>C</th>
<th>Tr</th>
<th>✓</th>
<th>X</th>
<th>-CT</th>
<th>Time</th>
<th>OBJ</th>
<th>CALL</th>
<th>EXTRA</th>
<th>MINOR</th>
<th>PR</th>
<th>Acc</th>
<th>Rej</th>
<th>Tr</th>
<th>Time</th>
<th>OBJ</th>
<th>CALL</th>
<th>EXTRA</th>
<th>MINOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>lodash</td>
<td>13</td>
<td>70</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>1.76</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>25</td>
<td>2.01</td>
<td>5</td>
<td>10</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>async</td>
<td>8</td>
<td>10</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>2.29</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0.63</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>express</td>
<td>10</td>
<td>18</td>
<td>7</td>
<td>0</td>
<td>3</td>
<td>1.19</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>17</td>
<td>5.89</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>chalk</td>
<td>10</td>
<td>54</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0.21</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0.78</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bluebird</td>
<td>9</td>
<td>18</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0.28</td>
<td>3</td>
<td>0</td>
<td>18</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1.21</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>uuid</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0.95</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rxjs</td>
<td>6</td>
<td>200</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>2.20</td>
<td>40</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>112</td>
<td>1.18</td>
<td>106</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>core-js</td>
<td>8</td>
<td>28</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>6.25</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>140</td>
<td>17.46</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>node-fetch</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>0.46</td>
<td>7</td>
<td>1</td>
<td>6</td>
<td>17</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>0.24</td>
<td>11</td>
<td>0</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>winston</td>
<td>8</td>
<td>36</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>0.80</td>
<td>16</td>
<td>14</td>
<td>7</td>
<td>7</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>19</td>
<td>1.12</td>
<td>34</td>
<td>50</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>redux</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0.33</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3.59</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>mongoose</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0.29</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1.81</td>
<td>16</td>
<td>6</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>89</td>
<td>451</td>
<td>82</td>
<td>2</td>
<td>5</td>
<td>1.53</td>
<td>92</td>
<td>17</td>
<td>39</td>
<td>37</td>
<td>114</td>
<td>27</td>
<td>4</td>
<td>325</td>
<td>2.92</td>
<td>190</td>
<td>66</td>
<td>79</td>
<td>33</td>
</tr>
</tbody>
</table>
The detection and patching took on average 1.53s per client (excluding parsing time), so JSFIX is clearly efficient enough to be practically useful. Most of the time is spent by the analysis (TAPIR), whereas the patching took on average only 0.11s.

We thereby conclude that JSFIX is almost always successful in producing code transformations that cause client test suites to succeed, and that it does so using relatively little time.

**Pull request experiment** We also investigated the quality of the transformations by creating pull requests of the updates produced by JSFIX to see if the transformations created by JSFIX are acceptable to the client developers. We conducted this experiment by first forking the client, then running JSFIX on the forked client, and eventually, manually performing some styling fixes to satisfy the linter of the client if necessary. The styling fixes had to be done manually since the code style convention varies from client to client. We then created a pull request based on these changes.

We first created pull requests for 41 clients from the previous experiment that had not already updated the benchmark libraries. So far, 4 of these pull requests have been accepted and 2 have been rejected. The rejections were not due to specific issues within the pull requests, but because the client developer was not willing to risk breaking the application by updating the dependency. Many of those 41 clients are no longer maintained, which probably explains why the maintainers reacted to only 6 of the pull requests. We therefore extended the experiment by adding an additional 114 clients (10 for each library, except for express as mentioned earlier) that had been updated within 6 months of the experiment. The results for these pull requests are shown in the last 9 columns of Table 10.2. For each library, we show the number of pull requests (“PR”), the number of accepted pull requests (“Acc”), the number of closed (i.e., rejected) pull requests (“Rej”), the number of transformations (“Tr”), and the average time (in seconds) used for detection and patching per client, excluding parsing time (“Time”). The remaining columns are described in Section 10.6.

Of the 114 pull requests created, 43 involved one or more transformations. For the remaining 71 pull requests, JSFIX did not find any source locations in the client code affected by breaking changes. For these clients, the pull request messages state that the code was not affected by any breaking changes, and the only file change was updating the version of the relevant dependency in the package.json file. So far, 27 of the pull requests have been accepted (in addition to the 4 mentioned above). For 16 of these 27 pull requests, JSFIX did not find any pattern matches in the client code. Only 4 have been rejected, and only 1 of these was affected by breaking changes. The maintainer of the async client who rejected a pull request did end up updating the code manually (at the same source locations as transformed by JSFIX), but using more idiomatic transformations. The transformations made by JSFIX were similar to the transformation shown in Example 16, so, as explained in that example, a more idiomatic transformation was possible. For the other rejected pull requests, the client developers did not report any issues with the pull request. For the 11 accepted pull requests that required modifications, manual styling fixes were only applied to two of them. Example 20 describes some of the accepted pull requests.
In total, JSFIX made 325 transformations across the 114 clients. Most of the transformations were applied to clients of core-js and rxjs. Since the experiments indicate that 43 out of 114 clients required transformations as part of the updates, we can conclude that breaking changes impact a significant proportion of clients, motivating the need for tools like JSFIX.

Overall, the average time is less than 3 seconds. Again, almost all the time is spent by the analysis phase, whereas patching takes on average only 0.06s.

In summary, based on the client test suites experiment, we conclude that transformations produced by JSFIX are generally trustworthy. Based on the pull request experiment, we can furthermore conclude that the transformations are generally of a high enough quality that developers are willing to use them in their code.

**RQ3 (Questions asked by JSFIX)**

The interactive phase of JSFIX is evaluated by looking at the questions asked during the two experiments described in Section 10.6. The number of questions in each of the four categories OBJ, CALL, EXTRA, and MINOR from Section 10.5 is shown in the last four columns of the “Client test suite experiment” and “Pull request experiment” sections of Table 10.2. All the questions have been answered by the authors of JSFIX. The questions in the first three categories are not subjective, as they all concern well-defined properties of the possible program behaviors. For the MINOR category, the answers are sometimes more subjective, which means that our answers may diverge from what the client maintainer would have chosen. However, since the questions in that category concern breaking changes that have only minor implications, this divergence is unlikely to affect the client behavior in a significant way. Our answers to these questions were based on a thorough investigation of the affected client code to determine the importance of the breaking change for each client.

For the 198 clients in Table 10.2 that JSFIX transformed successfully, JSFIX asked a total of 553 questions (2.8 per client). Of these questions, 365 questions (1.8 per client) are related to imprecision in the analysis, with 282 questions (1.4 per client) being related to imprecise matching of objects and 83 questions (0.4 per client) being related to imprecise reasoning of call filters.

A total of 118 questions (0.6 per client) concern the detection patterns being too coarse-grained to accurately describe the API usage that triggers the breaking change. For the last category, breaking changes with minor implications, there are 70 questions (0.4 per client), where the majority are asked when transforming node-fetch clients (see Example 13). For most of these questions, it was relatively simple to determine if the transformation should be applied by considering the client code surrounding the affected location. Unlike the other question types, a client developer needs to understand the implications of a breaking change to accurately answer these questions.

Since JavaScript is dynamically typed, it is possible that the correct answer to a question like “Is the receiver an rxjs observable?” is “sometimes”, however, we have not observed any occurrences of this situation in our experiments.
questions, but since fewer than 1 question of this type is asked per client, we do not consider them a concern for the practicality of JSFIX. It is also worth noticing that without JSFIX, the client developer would instead be forced to understand the implications of every breaking change on the client code.

Only 2.8 questions are asked per client and 0.7 questions per transformation on average. In our experience, it is easy and fast to answer the questions, even without expert knowledge of the benchmarks. To conclude, JSFIX only needs to ask a modest number of questions during the interactive phase, which makes it useful and time-saving when adapting client code to breaking changes in libraries.

**Threats to Validity**

One potential concern about the validity of the experimental results is whether we, as designers of JSFIX, are at an advantage when it comes to answering the questions in the interactive phase. As explained in Section 10.5, the questions only concern the client code, not the internals of JSFIX or the library code. We therefore believe that the intended user of JSFIX, who is familiar with the client code, will find these questions easier to answer than we did.

Another threat to validity of our conclusion about the expressiveness is that the semantic patches used in the experiments were written by the creators of JSFIX. Semantic patches written for one library can benefit numerous clients, however, it is of course important that other people can use the semantic patch language. We therefore plan to conduct user studies with library developers and other potential authors of semantic patches.

10.7 Related Work

The idea of automatically transforming clients to become compatible with new library versions has been considered in previous work. The term collateral evolution, describing exactly the process of patching client code based on some formalization of the required transformation, was coined by the authors of Coccinelle [108, 109], which is designed to adapt Linux drivers to breaking changes in the kernel. It has also been adapted to Java [70]. While Coccinelle and JSFIX share many traits, the large differences between C (or Java) and JavaScript make the internals of the tools quite different. Most importantly, Coccinelle relies on the fact that C and Java are statically typed, which makes it easier to connect calls with the relevant function and method definitions than in JavaScript. The same applies to the gofix tool for Go [11].

Others have also looked at ways to automatically compute differences between library versions, and based on these differences either suggest changes for adapting clients to breaking changes in the APIs or directly transform the client code as with JSFIX [28, 39, 83, 87, 103, 146]. Most of these approaches are for Java, where each member of a class has a fully qualified name, which makes it tractable to

[1] https://golang.org/cmd/fix/
compute differences on a type-level between two versions of a Java API, and thereby automatically identify many breaking changes. The only approach for a dynamically typed language is the PyCompat tool for Python [146]. While it is fully automatic, it is limited to a set of 11 different kinds of breaking changes, which all concern removals, moves, and additions of fields, parameters, and classes. In particular, these approaches generally do not handle behavioral changes (sometimes called semantic changes) where the behavior of a function changes without affecting its signature. The same limitation does not apply to JSFIX, which, as the evaluation shows, is able to patch almost all breaking changes appearing in library updates. The public interface of a JavaScript library can change dynamically and has no access modifiers, which makes it difficult to statically compute changes in the interface, and therefore the existing work for other languages will not directly work for JavaScript.

The idea of using a templating mechanism for specifying program transformations has been explored in other settings. With the Spoon framework [113], Java program transformations can be specified in Java code that directly manipulates the AST, or as class templates, which are classes with holes that must be instantiated with program elements to form valid Java classes. The class templates of Spoon resemble the code templates of semantic patches. Because Java is statically typed, the holes in the class templates also have a type, and Spoon can check that the program elements used in the substitution adhere to types of the holes. The lack of static type checking in JavaScript makes that difficult in our setting, however, it may be interesting in future work to look at opportunities for validating semantic patches, for example by attempting to check that any transformation resulting from a match of a semantic patch is syntactically valid JavaScript code.

JSFIX can be viewed as a specialized program repair tool, although it is designed to prevent rather than fix bugs. Unlike, for example, template based program repair techniques [90], the patches produced by JSFIX are almost always successful, as demonstrated by the experimental evaluation, and it does not require test suites.

The concept of breaking changes has been explored extensively in previous work. Several papers have shown that breaking changes are common in both patch and minor updates that are supposed to be backward compatible [16, 33, 68, 96, 119]. A few tools have also been designed for automatically detecting breaking changes in library updates [17, 96, 109]. The JSFIX approach is designed on the premise that the semantic pattern designer is already aware of where and how breaking changes appear. However, by combining our approach with existing tools, such as NoRegrets [100], it might be possible to derive the detection pattern part of the semantic patches automatically, which may reduce the overhead of writing semantic patches.

10.8 Conclusion

We have presented an approach to automate much of the work involved in adapting JavaScript programs to evolving libraries. It is based on a notion of semantic patches that combines the pattern matching technique from TAPIR [101] with a specialized
notation for code templates, thereby making it possible to express how to find and patch locations in the client code that are affected by breaking changes in the libraries. An extensive experimental evaluation on real-world libraries and clients using our implementation JSFIX demonstrates that the code template language is sufficiently expressive to precisely capture most breaking changes, and that most semantic patches are relatively simple. As an alternative to the current practice, the manual effort required by the client developers is reduced to answering a few questions about the program behavior. On average, only 2.7 questions are asked while patching 3.8 locations per client. The tool is fast and produces useful patches for the broken clients. In particular, 31 pull requests (many involving substantial changes to the client code) produced directly from the output of JSFIX have already been accepted by client developers.

10.9 Additional Examples

Example 17  Consider the breaking change affecting the applyEach function of the async library when it is updated to version 3.0.0. Prior to version 3, applyEach took as arguments a collection of functions followed by a varying number of general arguments and at last a callback function. It would then apply each of the functions in the collection with all the general arguments. Upon success or failure the callback would be called. In version 3, applyEach is modified to become a curried function that no longer takes the callback argument, but instead returns a function that is called with the callback. Since applyEach is variadic, the transformation must use negative indexing to reference all but the last argument ([0, -2]) and to reference just the last argument ($-1) as demonstrated by the transformation:

```javascript
call <async>.applyEach [2, ] $callee($args[0, -2])(-$1)
```

The detection pattern part of the semantic patch matches calls to the applyEach method on the async module where applyEach is called with at least two arguments.

Example 18  Semantic patches can also be useful for post-processing transformed code, to improve its readability and performance. In version 6 of rxjs, all operator functions called on rxjs observable objects must be modified to use the pipe function. For example, assuming x is an observable the following transformation is needed for the calls to map and filter.

```javascript
39 - x.map(...).filter(...)
40 + import {map, filter} from 'rxjs/operators';
41 + x.pipe(map(...)).pipe(filter(...))
```

That transformation is performed by applying the following semantic patch twice:

```javascript
call <rxjs>*.pipe(filter) $base.pipe(<rxjs/operators>$prop($args))
```

The ‘***’ part of the detection pattern matches any (potentially empty) sequence of operations between the load of the rxjs module and the read of map or filter. For example, the detection pattern would match both require('rxjs').map and require('rxjs').foo.bar.map.
For this particular breaking change, a more desirable result is obtained by combining the operators into a single `pipe` call, as shown by the following transformation.

```
42 - x.pipe(map(...)).pipe(filter(...))
43 + x.pipe(map(...), filter(...))
```

The resulting code is more idiomatic and also more efficient. This post-processing transformation can be described using an extra semantic patch that accompanies the one shown above:

```
call <rxjs>**.pipe().pipe
$base:callee($base:args, $args)
```

It takes two sequential calls to `pipe` and merges the arguments.

In principle, such semantic patches could be applied independently of the semantic patches used for adapting client code to breaking changes in libraries. For example, the transformation in line 43 is also sensible for `pipe` calls that were not inserted by NOREGRETS. Nevertheless, NOREGRETS only applies such post-processing patches to clean up code it has inserted, to avoid unnecessary transformations that are unrelated to breaking changes.

**Example 19** The promise library bluebird contains a function `promisify` that takes as argument a normal callback-based event style function, and then returns a promise wrapper around this function, where the promise is resolved whenever the callback of the wrapped function is called. Prior to the update of bluebird to version 3, the promise would resolve to a single value when the callback is called with a single success value and resolve to an array of values when the callback is called with multiple success values. In version 3, the promise will always resolve to the value corresponding to the first argument of the callback, unless an object with a `multiArgs` field set to `true` is passed to `promisify` in which case the promise always resolves to an array.

To preserve the semantics of code affected by this breaking change, the `multiArgs` field must be set to `true` for exactly those calls to `promisify` where, in version 2 of bluebird, the calls would result in a promise that resolves to an array. Because it is impossible to express this constraint in the detection pattern language, NOREGRETS will ask the user **“Is the first argument a function that calls its callback with more than two arguments?”** Answering “yes” to this question will result in the following semantic patch being applied:

```
call <bluebird>.promisify.asCallback
$callee($1, {multiArgs: true})
```

and answering “no” will result in no transformation.

**Example 20** The largest accepted pull request was to the core-js client `prebid.js` with 97 transformations performed, all for fixing locations affected by the same breaking change.

---

12The first argument of a standard callback is used to represent errors and is always set to `null` when no error occurs. The remaining arguments are called success values and contain what is typically viewed as the return value of the function to which the callback is supplied.

13Deciding such properties fully automatically is beyond the capabilities of any existing static analysis for JavaScript.
change. Of the accepted pull requests, the two most complicated ones were for two 
rxjs clients, where one required 12 transformations for 3 different breaking changes, 
and the other required 16 transformations for 4 different breaking changes. The 
first of these clients is the contentful-cli command line interface npm package for 
the contentful content manager. The contentful-cli package has more than 13000 
weekly downloads. Below are three excerpts of the transformation of contentful-cli made by NoREgrets.

```javascript
const { Observable, Subject } = require('rxjs/Rx')
const { map, filter } = require('rxjs/operators')
const { merge, Subject } = require('rxjs')

const loggingData$ = scopedEvents$.map(({ payload }) => {
  ...
}).filter(message => {
  ...
})
 const loggingData$ = scopedEvents$.pipe(
  map(({ payload }) => {
    ...
  }),
  filter(message => {
    ...
  })
)

const ls$ = Observable.merge(...lss)
 const ls$ = merge(...lss)
```

The imports are transformed, to adapt to a breaking change where imports from 
'rxjs/Rx' should be changed to imports from 'rxjs' instead. Notice also how the 
import of the Observable property has been replaced with an import of the merge 
property, due to a breaking change that results in merge no longer being a property 
on rxjs.Observable but instead a function that must be imported directly from 'rxjs'. 
As a result of this change, merge (line 55) now is used instead of observable.merge 
(line 54). The last breaking change affecting contentful-cli concerns the move of op-
erators, such as map and filter, from methods on observable objects into independent 
functions that must be called through pipe, as previously demonstrated in Example 18. 
The two operators, map and filter, are therefore imported on line 45 and lines 47,49 
have been transformed into lines 50–53 resulting in the operators being used inside 
pipe.

---

14https://www.contentful.com/
Chapter 11

Extracting Taint Specifications for JavaScript Libraries


Abstract

Modern JavaScript applications extensively depend on third-party libraries. Especially for the Node.js platform, vulnerabilities can have severe consequences to the security of applications, resulting in, e.g., cross-site scripting and command injection attacks. Existing static analysis tools that have been developed to automatically detect such issues are either too coarse-grained, looking only at package dependency structure while ignoring dataflow, or rely on manually written taint specifications for the most popular libraries to ensure analysis scalability.

In this work, we propose a technique for automatically extracting taint specifications for JavaScript libraries, based on a dynamic analysis that leverages the existing test suites of the libraries and their available clients in the npm repository. Due to the dynamic nature of JavaScript, mapping observations from dynamic analysis to taint specifications that fit into a static analysis is non-trivial. Our main insight is that this challenge can be addressed by a combination of an access path mechanism that identifies entry and exit points, and the use of membranes around the libraries of interest.

We show that our approach is effective at inferring useful taint specifications at scale. Our prototype tool automatically extracts 146 additional taint sinks and 7840 propagation summaries spanning 1393 npm modules. By integrating the extracted specifications into a commercial, state-of-the-art static analysis, 136 new alerts are produced, many of which correspond to likely security vulnerabilities. Moreover, many important specifications that were originally manually written are among the ones that our tool can now extract automatically.
CHAPTER 11. EXTRACTING TAINT SPECIFICATIONS FOR JAVASCRIPT LIBRARIES

11.1 Introduction

JavaScript is powering a wide variety of web applications, both client-side and server-side. Many of these applications are security-critical, such as PayPal, Netflix, or Uber, which handle massive amounts of privacy-sensitive user data and other assets. An important characteristic of modern JavaScript-based applications is the extensive use of third-party libraries. On the npm platform more than 1 million packages (mostly libraries) are available[^1] and only a few of them have been screened intensively for security vulnerabilities. A challenge when analyzing the security of npm packages is that they are often not self-contained, but they in turn depend on other npm packages for providing lower-level functionality. Recent work shows that, on average, every npm package depends on 79 other packages and on code published by 39 maintainers[^147]. To correctly understand an application that uses npm packages, one needs to consider all these dependencies.

Two main directions are being pursued for automatically securing npm packages. First, there are tools that aggregate known security vulnerabilities in specific versions of individual libraries and report them to the developer directly. For example, npm audit analyzes all the dependencies of a Node.js application and warns the developer about any known vulnerabilities in the dependent-upon code. GitHub, Snyk, and other companies offer similar services, and related work[^84] advertises such security controls. The main limitation of this approach is the high number of false positives. Often the critical part of the library is not used by the application, or it is used in a way that is completely harmless. For example, an application may use an npm module vulnerable to command injection attacks, but it passes only string constants provided by the developer as input to this module. We believe it is important to make the distinction between merely relying on a library that contains a potential known vulnerability and using that library in an insecure way. Another problem with these tools is that libraries that use insecure features of the JavaScript language or of the Node.js framework are often not registered as having “known vulnerabilities” if their documentation indicates that these features are being used internally. An example of such a library is the package jsonfile that provides functionality for easily accessing JSON files. Even though such a library is not considered vulnerable by itself, it may be used in an insecure manner, e.g., by propagating attacker-controlled data into file system paths.

A more precise approach for securing JavaScript applications, pursued both by academia and by industry practitioners, is static program analysis. In taint analysis, which is a kind of program analysis that can in principle detect most common forms of security issues, security properties are expressed as direct information flows from sources to sinks: either from untrusted sources to sensitive sinks (integrity) or conversely from sensitive sources to untrusted sinks (confidentiality). We focus on integrity because it covers the vast majority of security vulnerabilities reported by the

11.1. INTRODUCTION

and we ignore indirect flows, also called implicit flows, because they have been shown to appear seldom in real-world npm vulnerabilities [131].

Modularity is the key to scalable static analysis. For example, GitHub’s LGTM platform includes a state-of-the-art taint analysis for JavaScript (and other languages), which achieves high scalability by analyzing modularly. When analyzing one module of an application, other modules are either ignored or treated according to manually written specifications that describe essential taint flows where available. Ignoring modules leads to inaccurate analysis results, while manually constructing specifications is a demanding and error-prone task, so only a limited number of npm modules are considered. An important question hence is how to obtain specifications of modules in an automated way.

Inspired by Modelgen for Android [25], we present a technique that dynamically infers explicit taint flow summaries for npm modules, to be utilized in a static analysis, such as LGTM. Besides being designed for JavaScript, our technique is more general than Modelgen, allowing for complex summaries to be extracted. For example, we are the first to support summaries involving callback arguments and instantiated exported classes. Moreover, our technique considers the large amount of transitive dependencies in npm and thus allows the extraction of summaries for multiple npm packages in the same execution.

Another source of inspiration is the NoRegrets tool [96] that leverages the vast number of open source packages available in the npm repository to obtain information about how the most important libraries are being used. Many of those packages have test suites, and by running the test suite of a package we can gain information about the taint flows in all the packages it depends on, both directly and transitively.

A central technical challenge for adapting the Modelgen idea to our setting is that JavaScript is a highly dynamic language, which makes it non-trivial to map observations from a dynamic analysis to taint specifications that fit into a static analysis. To this end, we adopt the notion of dynamic access paths from NoRegrets, allowing us to identify entry and exit points of taint flow in the libraries. Our dynamic analysis uses a variant of membranes [27, 51, 72, 97] for tracking the taint flow between libraries and clients. It identifies flows between entry and exit points (propagations), between entry points and existing sinks (additional sinks) and between existing sources and exit points (additional sources). Finally, we propose deploying one membrane per npm module and hence extracting summaries for multiple modules at once.

We show that our approach is highly scalable by successfully running our dynamic analysis on 15892 clients of 751 packages. The dynamic analysis is efficient, spending, on average, only 112 seconds per successfully analyzed client or 302 seconds per inferred specification. In total, it extracts 146 additional taint sinks and 7840 propagation summaries spanning 1393 modules. 35% of the summaries contain complex taint flows, such as between an argument of an exported method and a parameter passed by the library to a callback. The evaluation also shows that the extracted summaries can

\footnotesize{\url{https://www.npmjs.com/advisories}}

\footnotesize{\url{https://lgtm.com}}
let userInput = {
  tempDir: "./path/to/dir",
  cacheDir: "./path/to/cache"
}

const _ = require("lodash");
const rimraf = require("rimraf")

let obj = _.forIn(userInput, function(value) {
  rimraf(value, function(err) {
    if (err)
      console.log(err)
  });
});

Figure 11.1: A typical example of JavaScript code that uses npm modules. With dotted/blue we mark exit points from the client code and with solid/orange the entry points from the library code.

improve static analyses by enabling it to reveal otherwise missed vulnerabilities: 136 new alerts are produced, many of which correspond to likely vulnerabilities.

In summary, our contributions are:

- We present a novel, highly-scalable specification extraction technique for JavaScript libraries that builds on a dynamic taint analysis and leverages existing test suites.

- We report our results from an extensive experimental evaluation of the approach. The results show that the dynamic analysis is able to infer non-trivial and accurate taint flow models in widely used npm modules.

- We demonstrate that the inferred taint specifications can be integrated into an existing static analysis tool, thereby enabling discovery of previously unknown security vulnerabilities.

11.2 Motivating Example

Let us consider the example in Figure 11.1. This code fragment uses two of the most popular npm packages: lodash, a general-purpose utility library, and rimraf, a simple library for recursively deleting directories on the disk. In the presented example, the forIn method from lodash is used to iterate through the values of each property on the user input object. Each of these values is then passed to the rimraf module.

A human or an automated tool that aims at analyzing the code fragment in Figure 11.1 must first understand the essential semantics of the two modules. For example, one needs to understand that if some user input is passed to rimraf without sanitization, then it exposes a directory traversal vulnerability. However, this style of code can hinder understandability, both for unexperienced users and for static analysis tools. Specifically, it may not be clear that by invoking the forIn method with two
parameters – an object to be traversed and a callback function – the second parameter will be invoked with the property values of the first parameter as arguments.

One way to address this problem is to analyze the library code together with the client code using a whole-program dataflow analyzer. However, that approach suffers from serious scalability issues. For example, the implementation of the apparently trivial `forIn` method spans across 32 files. In Figure 11.2 we show a subset of the code that needs to be analyzed. Statically analyzing such a large amount of highly dynamic code is extremely expensive and tends to give prohibitively imprecise results [4].

When trying to analyze the source code of the `rimraf` module, one is faced with even greater challenges, as illustrated by Figure 11.3. To reveal the directory traversal problem discussed earlier, one needs to show that there is an unsanitized flow from the first parameter of the `rimraf` function to one of the file system access methods, e.g., `fs.rmdir`. However, as shown by the example, the call to this method is dispatched using dynamically attached methods on the `options` object. Once again, to the best of our knowledge, existing static analysis tools for JavaScript are unable to successfully analyze such highly-dynamic code at scale and with a precision that is practically useful.

Modular analysis, as exemplified by LGTM, addresses this challenge by analyzing each package in isolation. If a package depends on other packages, those are either ignored or modeled using manually-written specifications that capture the essential dataflows. Relying on simple generic specifications, e.g., saying that whenever a parameter is tainted then so is the return value, would be too imprecise for this example and lead to the static analysis missing important flows. Since creating useful specifications manually is difficult and not scalable, efficient automated alternatives are needed.

Our approach leverages the information in the npm repository about packages and their dependencies, together with the package source code available on GitHub. For this specific example, both `lodash` and `rimraf` have numerous open-source clients, many with test suites. By dynamically analyzing the executions of those test suites, we can automatically learn useful specifications.

### 11.3 Taint Specifications for Modules

The specifications we are interested in summarize the taint-relevant information for entry and exit points of JavaScript libraries. For example, one can specify that the information from entry point $A$ may flow into exit point $B$ or that values passed to an entry point eventually reach a potentially dangerous operation.

The careful reader may have observed that there is a duality between the exit points of the client code, e.g., in Figure 11.1 and the entry points of the library, e.g., in Figure 11.2. For example, the `userInput` argument in line 8 corresponds to the `object` parameter in line 18. We will refer to both an entry point and its corresponding exit point by using the term **contact point**. We also introduce an access path mechanism to uniquely identify each contact point.
Figure 11.2: The implementation of lodash’s forIn method. For space reasons, only two of the 31 dependent files are shown. With dotted/blue we mark entry points to the library code and with solid/orange the exit points from the library code.

The specifications described in the remainder of this section can in principle be produced in multiple ways: either manually or by using a static or dynamic analysis. Section 11.4 presents an automatic inference process based on dynamic analysis.

Specifying Contact Points

Inspired by previous work to detect breaking changes in npm package updates [96, 100], we propose using an access path mechanism for specifying contact points. An access path, or short ap, can be described as an S-expression, which is read from the innermost expression outwards. Each type of symbol corresponds to an operation in
11.3. TAINT SPECIFICATIONS FOR MODULES

```javascript
var fs = require("fs")

function defaults (options) {
  var methods = [
    'unlink', 'chmod', 'stat', 'lstat', 'rmdir', 'readdir'
  ]
  methods.forEach(function(m) {
    options[m] = options[m] || fs[m]
    m = m + 'Sync'
    options[m] = options[m] || fs[m]
  })
}

function rmdir (p, options, originalEr, cb) {
  defaults(options)
  options.rmdir(p, function (er) {
    cb(er)
  })
}

module.exports = function rimraf(p, options, cb) {
  options.lstat(p, function (er, st) {
    return rmdir(p, options, er, cb)
  });
};
```

Figure 11.3: Simplified source code for the rimraf module.

the JavaScript language.

\[
ap ::= (\text{root} \ <\text{uri}>)
| (\text{member} \ <\text{name}> \ <\text{ap}>)
| (\text{parameter} \ <i> \ <\text{ap}>)
| (\text{return} \ <\text{ap}>)
| (\text{instance} \ <\text{ap}>)
\]

The innermost subexpression of a path always contains a root symbol, which holds an URI that refers to the module. For space reasons, we use package names instead of package URIs. For example, (root dotenv) refers to the module that is loaded when calling require('dotenv'). The other symbols are member to refer to properties of objects, parameter that refers to the i-th parameter of a function, return that refers to the return value of a call, and instance that refers to constructed values. For example, the path (parameter 0 (member forIn (root lodash))) represents both the exit point in line 8 of Figure 11.1 and the first entry point in line 18 of Figure 11.2. In the remainder of this section we show how access paths can express different kinds of taint specifications.

We assume that a collection of so-called known sources and sinks is provided. For example, values obtained from network communication via the Node.js standard library are commonly treated as sources, and arguments to exec and eval are sinks.

We are interested in three kinds of specifications: additional sinks when we observe a flow from an entry point to a known sink, additional sources when there is flow from a known source to an exit point, and propagation summaries when there is
a flow from an entry point to an exit point. We will now proceed to describe each of them in detail.

**Propagation Summaries**

The propagation summaries, or propagations for short, specify how taint may flow in and out of a library’s functions. For example, a propagation summary can specify that if a tainted value enters the library as a specific argument to a function, then specific exit points of the library, e.g., properties on the return value, should also be considered tainted. Having such information available allows program analyses to reason about the potential taint flows without needing to reanalyze the source code of the library for every client.

The most basic form of flow is from an argument of a function to its return value, either because the argument is returned directly, or because the argument is used in the computation of the return value. Other more complicated forms of flow may also occur. For example, if an argument is written to some internal state of the library, and this state is then returned from another function, then we have a taint flow from the argument of one function to the return value of another function, which can also be captured as a propagation summary.

A propagation summary consists of two access paths: one that represents the point in the library API where the tainted value enters, and one that represents the point where the tainted value exits. Consider Example 1 where a function $f$ has a parameter $x$ and returns an object that has a property $p$ with a value obtained from the $p$ property of $x$.

**Example 1**

```javascript
69 //module m
70 function f(x) {
71   return { p: x.a }
72 }
73 module.exports.f = f;
```

The interesting taint flow for this code is modeled by the taint specification shown next to the example, which indicates that taint flows from $x.a$ to the $p$ property of $f$’s return value.

**Taint Specification**

```
(member a (parameter 0 (member f (root m))))
↓
(member p (return (member f (root m))))
```

This way of expressing taint flows is sometimes inconvenient. For example, a common JavaScript pattern is to iterate through all the properties of an object, which means that the accessed property names differ from client to client. An example of this reflective pattern is seen in Example 2. With the current notion of propagation summaries, we can only express flows involving specific properties, but in this case the relevant property names depend on the clients. For this purpose we introduce a wildcard notation for referring to every property of an object: $(member * <ap>)$. For example, one may refer to all the properties of the obj parameter in the program example with $(member * (parameter 0 (root sum)))$ as shown in the taint specification of Example 2.
11.3. TAINT SPECIFICATIONS FOR MODULES

Example 2

```javascript
//module sum
function f(obj) {
  let sum = 0;
  for (prop in obj) {
    sum += obj[prop];
  }
  return sum;
}
module.exports.f = f;
```

```
Taint Specification
(member * (parameter 0 (member f (root sum))))
↓
(return (member f (root sum))

As mentioned earlier, callbacks are common contact points in npm modules. Our specifications refer to callbacks by treating a parameter as a function. The following specification summarizes the part of the lodash library presented in Figure 11.2, using a callback parameter exit point:

```
(member * (parameter 0 (member forIn (root lodash))))
↓
(parameter 0 (parameter 1 (member forIn (root lodash))))
```

This propagation says that the value of every property of the object passed as the first argument of the forIn function may flow into the first parameter of the callback passed as the second argument.

A final propagation pattern worth discussing is one that involves contact points with return values. In Example 3 the padder module exports a single anonymous function in line 84. However, this function in turn creates an object with an lpad property pointing to an internal anonymous function. This case corresponds to the factory method design pattern from object-oriented literature. After invoking the main exported method of the module, a client obtains a reference to the internal object declared in line 85, which in turn allows the client to invoke the internal anonymous function from line 86. Thus, there are two exit points of the padder library in the presented example: one that returns an object with an lpad method, in line 89, and one corresponding to that method itself, in line 87. The latter depends on the former, because an object with the lpad method is only exposed to the client through the first exit point, which in turn creates more entry and exit points for the lpad method.

Example 3

```javascript
//module padder
module.exports = function() {
  let res = {};
  res.lpad = function(s) {
    return " " + s;
  }
  return res;
}
```

```
Taint Specification
(parameter 0 (member lpad (return (root padder))))
↓
(return (member lpad (return (root padder))))
```

```
(member * (parameter 0 (member forIn (root lodash))))
↓
(parameter 0 (parameter 1 (member forIn (root lodash))))
```

```
(member * (parameter 0 (member forIn (root lodash))))
```

```
(return (member f (root sum)))
```

```
(return (member f (root sum)))
```

```
(return (member lpad (return (root padder))))
```

```
(return (member lpad (return (root padder))))
```

This propagation says that the value of every property of the object passed as the first argument of the forIn function may flow into the first parameter of the callback passed as the second argument.
CHAPTER 11. EXTRACTING TAINT SPECIFICATIONS FOR JAVASCRIPT LIBRARIES

Additional Sinks and Sources

If a value passed into a library reaches a known sink, we say that the entry point through which the value entered is an additional sink. Intuitively, passing the value to that contact point or to the sink itself has the same security implications for the client of the library, hence a program analysis can treat them the same way.

Revisiting the source code of the rimraf library in Figure [11.3], we can observe that the value passed as first argument to the main library function ends up in `fs.rmdir()`, which is a known sink for directory traversal vulnerabilities. This method allows recursively removing any folder on the disk, hence if an attacker can control the value passed into it, she can cause serious harm on the system. Therefore, it makes sense to specify the contact point (parameter 0 (root rimraf)) as an additional sink.

Conversely, if inside the library a tainted value is created which then escapes into the client code through an exit point, we say that the exit point is an additional source. Example 4 shows a simple module that performs a TCP request and invokes a callback whenever data is received from the target server. This data should be considered tainted since it comes from untrusted third-party computers, so it is reasonable to specify the contact point (parameter 0 (parameter 2 (root my-tcp))) as an additional source.

```javascript
91 //module my-tcp
92 module.exports = function (host, port, cb) {
93 const net = require('net');
94 const client = new net.Socket();
95 client.connect(port, host, function () {});
96 client.on('data', function (data) {
97 cb(data);
98 });
99 }
```

Even though our dynamic analysis presented in Section [11.4] can in theory extract all the three kinds of specifications presented so far, our prototype implementation introduced in Section [11.6] only supports the extraction of propagations and additional sinks. The main reason for omitting extraction of additional sources is that existing security vulnerability reports for npm packages often involve additional sinks, for example CVE-2017-1000219 or CVE-2018-3772, but vulnerabilities caused by additional sources are less common.

11.4 Inferring Taint Specifications via Dynamic Analysis

We now present a technique for dynamically inferring taint specifications, i.e., propagation summaries and additional sinks, of the form described in Section [11.3]. The
11.4. INFERRING TAINT SPECIFICATIONS VIA DYNAMIC ANALYSIS

Goal is to find relations between entry points and exit points, between entry points and existing sinks, and between existing sources and exit points.

Figure 11.4 illustrates how our technique works for a single npm module. The arrows represent information flow, possibly spanning multiple methods and modules. When the test suites are executed, values are intercepted at entry points and tainted with a unique identifier per entry point. The taint inside the module is then propagated using a dynamic taint analysis. Whenever a tainted value reaches a sink or an exit point, an additional sink or a propagation summary, respectively, is generated. Similarly, if a value that is tainted by an internal source is observed at an exit point, an additional source is generated for that exit point. All taints are removed at exit points, so we only infer specifications for the library code and not for the client code.

Previous work [25] considers arguments of methods in the public API as entry points and return values as exit points, but as the motivating example shows, this is insufficient for many npm modules. JavaScript libraries interact with their clients in complex ways, e.g., through callbacks like the ones in Figure 11.1, or by allowing plugins to be configured inside the library. Therefore, it is non-trivial to determine where the library code starts and where the client code ends. One way to refer to this point of contact between components and thus to generalize the idea of entry and exit points is by using the concept of membranes [27].
Table 11.1: Creation of contact points inside the membrane and the corresponding taint operations executed before and after the proxied operation. The direction indicates whether the proxy corresponds to an entry or an exit point. The taint($v, ap$) action associates a taint corresponding to the access path $ap$ to runtime value $v$. The checkTaint($v, ap$) action recursively searches for tainted values in $v$, where the taint has the same root package as the access path $ap$. Finally, untaint($v, ap$) recursively declassifies all the values in $v$ that have a taint with the same root as the access path $ap$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Existing contact point</th>
<th>New contact point(s)</th>
<th>Pre action</th>
<th>Post action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Access path</td>
<td>Direction</td>
<td>Access path</td>
<td>Direction</td>
</tr>
<tr>
<td>require(&quot;foo&quot;);</td>
<td>-</td>
<td>-</td>
<td>ap = (root foo)</td>
<td>ENTRY</td>
</tr>
<tr>
<td>x.prop</td>
<td>$ap_x$</td>
<td>ENTRY</td>
<td>$ap = (member prop \ ap_x)$</td>
<td>ENTRY</td>
</tr>
<tr>
<td></td>
<td>$ap_x$</td>
<td>EXIT</td>
<td>$ap = (member prop \ ap_x)$</td>
<td>EXIT</td>
</tr>
<tr>
<td>res = x(arg)</td>
<td>$ap_x$</td>
<td>ENTRY</td>
<td>$ap_{par} = (parameter &lt;i&gt; \ ap_x)$</td>
<td>EXIT</td>
</tr>
<tr>
<td></td>
<td>$ap_x$</td>
<td>EXIT</td>
<td>$ap_{ret} = (return \ ap_x)$</td>
<td>ENTRY</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$ap_{par} = (parameter &lt;i&gt; \ ap_x)$</td>
<td>ENTRY</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$ap_{ret} = (instance \ ap_x)$</td>
<td>ENTRY</td>
</tr>
<tr>
<td>res = new x(arg)</td>
<td>$ap_x$</td>
<td>ENTRY</td>
<td>$ap_{par} = (parameter &lt;i&gt; \ ap_x)$</td>
<td>EXIT</td>
</tr>
<tr>
<td></td>
<td>$ap_x$</td>
<td>EXIT</td>
<td>$ap_{ret} = (instance \ ap_x)$</td>
<td>ENTRY</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$ap_{par} = (parameter &lt;i&gt; \ ap_x)$</td>
<td>ENTRY</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$ap_{ret} = (instance \ ap_x)$</td>
<td>EXIT</td>
</tr>
</tbody>
</table>
11.4. INFERRING TAINT SPECIFICATIONS VIA DYNAMIC ANALYSIS

Membrane-Based Analysis

The main idea of a membrane is to interpose analysis behavior on every interaction between the client and the library. Moreover, every reference that passes through the membrane becomes part of it. Existing work describes how to implement membranes and how to use them for implementing generic policies such as “the library should never use the native module fs”. However, to be useful in our setting, we need a way to distinguish between entry and exit points of the library and to uniquely refer to every such point in the membrane.

To rigorously define membranes, we first introduce a way of intercepting operations on a given value. To this end, we rely on proxies, a concept introduced in ECMAScript 6. A proxy \( P(v) \) for a value \( v \) is a wrapper object that attaches traps to the wrapped value. Every operation applied to the proxy results in an invocation on the corresponding trap. For example, property reads, property writes, function applications, and constructor applications all result in their corresponding traps firing. The traps can modify the behavior of the operation or just perform observing operations, such as logging.

A proxy can therefore observe operations applied to the wrapped value, and even decide to modify these operations. Our analysis uses proxies to perform taint-relevant operations before and after the proxied operation is executed. We also need a way to associate a unique address, i.e., an access path, to each proxy and to specify whether the proxy corresponds to an exit or an entry point:

**Definition 1.** A contact point, denoted \( \langle v, ap, d \rangle \), is a tuple consisting of a proxy \( P(v) \) around a value \( v \), an access path \( ap \) that uniquely identifies the contact point, and a direction flag \( d \) that specifies whether the contact point is an entry or an exit point.

For simplicity, we abuse the notation for a contact point \( \langle v, ap, d \rangle \) by using \( \langle v \rangle \) whenever the access path and the direction are not relevant for the description. One can specify entry and exit points for a library by introducing proxies around exported API methods in the library source code. The challenge lies in automatically identifying all the values that need to be proxied for intercepting all the interactions between two npm modules. Membranes provide an elegant solution to this problem:

**Definition 2.** A membrane \( M \) is a set of contact points interposed between a library \( \ell \) and its clients. \( M \) is initialized with \{ \langle \text{root } \ell, \text{ENTRY} \rangle \}, i.e., the contact point that wraps the main value \( v_\ell \) exported by the library. For every value \( v \) that is passed into or returned by an existing contact point in \( M \), a new contact point \( \langle v, ap_v, d' \rangle \) is added to the membrane, i.e., \( M := M \cup \{ \langle v, ap_v, d' \rangle \} \).

The new access point \( ap_v \) is derived from the existing \( ap \) by picking the grammar rule from Section 11.3 that corresponds to the JavaScript operation \( v \) passing through the exit point, e.g., a property access or a parameter to a function call. Similarly, the direction of the new contact point \( d' \) is derived from the direction of the original contact point \( d \) by using the following observation: the direction changes for all the values that are passed as arguments to a method in the membrane. Let us consider a
function object that is passed into an entry point of a library as an argument. Once it reaches the other side of the membrane, i.e., in the library code, it should be considered as an exit point for the library. In Table 11.1 we summarize all the possible operations on a contact point and how to derive the access paths and direction flags for the new contact points. We also show the auxiliary operations necessary for tracking tainted values in the pre and post action columns. Note that both arguments and return values can be entry or exit points, depending on the direction flag.

To illustrate how contact points are created, consider the membrane between lodash and its client in Figure 11.1. The first contact point of the membrane is created when the library is required in line 5, i.e., $M := \{\langle \_ \rangle\}$. When the `forIn` property is accessed in line 8 a new contact point is added to the membrane, $M := M \cup \{\langle \_.forIn \rangle\}$. When the accessed property is invoked in the same line, three contact points are created, i.e., $M := M \cup \{\langle \text{userInput} \rangle, \langle \text{function} \ldots \rangle, \langle \text{obj} \rangle\}$. Finally, when the callback is invoked, three more contact points are created, one for each parameter. The access paths for each of these contact points are shown in Figure 11.5; they correspond to a derivation tree of the grammar in Section 11.3. To obtain the access path of a given contact point, one should traverse the tree from the root and replace all the $\diamond$ symbols with the access path of the parent node. For example, the access path of $\langle \text{first} \rangle$ is:

$$\langle \text{parameter 0 (parameter 1 (member forIn (root lodash))))}$$

The dynamic taint analysis we use for propagating taint inside analyzed modules is fairly standard, with few idiosyncrasies. As noted earlier, we implement the taint-relevant operations described in the last column of Table 11.1 inside each module’s membrane. These operations are in fact additional sources and sinks from the taint analysis’ perspective since they either attach taint or check/remove taint. Once a property $p$ is accessed on a value having a taint $t$, instead of directly propagating the
taint, we create a new tainted value (\texttt{member p t}). If the property \( p \) itself is also tainted then we propagate the taint (\texttt{member * t}). The intuition is that the tainted property comes from outside the module or from iterating through a tainted object, hence it should be considered as a generic access.

**Multi-Module Analysis**

Since an npm module can in turn use other npm modules, we propose deploying a membrane around each module to maximize the number of extracted specifications. We present this setup in Figure 11.6 in which module \( M \) interacts with two other modules: a direct dependency \( L \) and a plugin \( P \). For now let us consider the relation between module \( M \) and its dependency \( L \). Every entry point attaches a taint that uniquely identifies that entry point to each value that passes through it, e.g., entry point \( M_1 \) sets taint \( m_1 \). When a value passes through an exit point of a module, the analysis removes all the taints corresponding to that particular module. As a result, tainted values for module \( M \) can only live inside \( M \) or inside \( M \)'s transitive dependencies, such as \( L \). This behavior can be observed when following the information flow between entry point \( M_1 \) and exit point \( M_3 \). The taint \( m_1 \) is carried by the value all the way through module \( L \) until the exit point \( M_3 \). Our analysis infers two propagation specifications: \( M_1 \rightarrow M_3 \) and \( L_1 \rightarrow L_3 \).

Similarly, the value that enters through \( M_2 \) inside module \( M \) gets attached the taint \( m_2 \), and further enters through \( L_2 \) inside module \( L \), where it gets attached taint \( l_2 \). Thus, when the value finally reaches the sink inside module \( L \), it has two taints, \( m_2 \) and \( l_2 \). The analysis generates two additional sinks, for \( M_2 \) and for \( L_2 \), since from the

Figure 11.6: Inferring specifications for multiple modules at once: every entry point adds a unique taint and every corresponding exit point declassifies it. The semantics of shapes and colors are the same as in Figure 11.4. The dotted arrows depict equivalences between contact points.
client’s perspective a value may flow from one of these entry points into an existing sink.

Handling Plugins

One may wonder whether or not a module’s dependencies should be considered as part of the module’s code as described in the previous section. We propose distinguishing between two types of dependencies: direct dependencies and plugins. A direct dependency is one that is required verbatim by the developer in the source code of the module, and a plugin is a dependency that is injected by the client code. For registering a plugin, a client needs to pass a reference to the plugin through the membrane. An example of this pattern can be observed in Example 5 that shows the popular express framework instantiated with the plugin body-parser. The client code loads the body-parser plugin and passes it to the express module through the use method that is part of express’s membrane.

Example 5

```javascript
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
app.use(bodyParser.json());
```

Treating plugins differently when extracting specifications is extremely important because we do not want to infer specifications that only apply when a certain plugin is loaded. Instead, we want the specifications to be as widely applicable as possible. Therefore, direct dependencies that are loaded inside the module are considered part of the code base of the module, while plugins are not.

Consider the relation between module M and its plugin P in Figure 11.6. When the value carrying the taint $m_4$ reaches the membrane that separates M from P the taint is removed; we say that the value is declassified. Overall, for the flow between $M_4$ and $M_7$ that passes through plugin P, our analysis infers three specifications: $M_4 \rightarrow M_6$, $M_5 \rightarrow M_7$, and $P_1 \rightarrow P_2$.

11.5 Using Taint Specifications

The main use case of the extracted taint specifications is for improving existing program analyses. Most importantly, taint specifications can be consumed by static analyses. The benefits of hybrid analyses, i.e., static plus dynamic, are thoroughly explored in the literature. Typically, a static analysis uses the results from a dynamic analysis, either to get a more precise result or to get coverage of code that is otherwise difficult to analyze statically. As mentioned in the introduction, industrial static analyses sometimes do not even try to analyze Node.js modules, but instead rely on manually written taint specifications or coarse-grained assumptions about taint flow in modules. However, such specifications are both error-prone and hard to maintain. In contrast, our specification generation analysis is fully automatic, and can therefore
11.6. EVALUATION

Easily be re-run whenever modules are updated. Moreover, we show that there is a significant overlap between the specifications our analysis generates and existing manually-written models used by the commercial LGTM taint analysis, demonstrating that our analysis can infer precise module specifications that resemble and improve upon hand-written specifications.

Another use case for the extracted specifications is to serve as a form of documentation for the module they were extracted from. Effectively, they can act as a contract between module developers and module users that specifies, for example, who is responsible for sanitizing end user input. We observe that many security vulnerabilities reported by the community or by researchers [130] are actually additional sinks. For example, a typical vulnerability occurs when a user-supplied value is involved in constructing some string that is then executed by the eval function. In some unfortunate situations attackers can compose the user-supplied value in ways that enables executing malicious code. To warn users of modules about potential vulnerabilities, inferred specifications could be shown to developers. For example, an additional sink could inform the client that an argument passed to a specific method should be sanitized to prevent malicious code injection attacks.

Finally, we propose using the generated taint specifications for regression analysis. When a previously unobserved taint specification is suddenly generated for a new version of a library, e.g., a new additional sink appears, both the developer of the library and its clients should be alerted. Essentially, a change in a taint specification should be treated as a change to the API. Automatically inferred specifications could help automate this kind of regression analysis.

11.6 Evaluation

Implementation We implement our specification extraction technique in a tool called TASER. The dynamic analysis component is built on top of NodeProf [134], an instrumentation framework for Node.js. As a starting point for finding additional sinks, we mark 40 methods of the built-in JavaScript APIs as sinks. These methods cover five well-known security issues: command injection, code injection, directory traversal, regular expression injection, and NoSQL injection. We implement limited support for sanitizers by declassifying any information flow that passes through a function and an npm module whose name or dynamic access path contains specific strings, e.g., “escape” or “sanitize”.

Benchmarks We apply TASER to 751 npm packages, all from the top-1000 most depended upon packages. Because some packages contain multiple modules and because TASER performs a multi-module analysis we analyze a total of 2300 modules. For each analyzed npm package, we consider the 200 highest rated clients, according to npm stars, and execute their test suites to analyze the execution with TASER. We stop a test suite after a timeout of 10 minutes. If available, we also use the test suite of

\[\text{It is an abbreviation of the longer Taint Spec Extractor.}\]
the npm package itself for driving the dynamic analysis. Ignoring some clients that we currently cannot analyze, e.g., due to test frameworks TASER does not support, or due to limitations of our implementation, the evaluation covers 15,892 clients, out of which 5,707 clients trigger at least one creation of a tainted value.

Research questions Our evaluation focuses on the following research questions:

RQ1 How many taint specifications does TASER extract?

RQ2 How efficient is the analysis?

RQ3 Are the extracted specifications useful for statically analyzing the security of npm modules?

RQ4 How do the extracted specifications compare to manually created models of npm modules?

The implementation of TASER and experimental data are available at [http://brics.dk/taser/](http://brics.dk/taser/).

RQ1: Extracted Taint Specifications

For the 2,300 analyzed modules, TASER extracts 7,840 propagation summaries and 146 additional sinks. For 457 packages, the tool extracts at least one propagation summary, and for 118 packages, it extracts at least one additional sink. The overall amount of specifications shows that manually writing taint specifications for thousands of packages is highly impractical. Instead, TASER enables extracting specifications automatically and updating them regularly with little effort.

We also check whether the specifications TASER extracts contain advanced language constructs not supported by previous work [8, 25]. To that end, we count every propagation summary that involves (i) instantiated objects, i.e., an instance symbol in one of its access paths, (ii) callbacks, i.e., two or more parameter symbols in one of its access paths, or (iii) nested API calls, i.e., two or more return symbols in one of its access paths. We find 595 propagation summaries with instantiated objects, 1,467 with callbacks and 1,578 with nested API calls. In total, at least 2,838 specifications, i.e., 35% of the total, could not have been extracted by those previous approaches (even if re-implemented for JavaScript).

RQ2: Efficiency of the Dynamic Analysis

Generating specifications is not something that should be done often, so having a relatively large one-time cost is acceptable in practice. However, over time new libraries are created and existing libraries are updated, so new taint specifications naturally have to be generated for those libraries. Therefore it is interesting to consider the computational cost of generating taint specifications. On average, it takes 112 seconds to run the test suite of one client with the dynamic analysis enabled. This
number depends on many factors, such as the size of the test suite, and how many
JavaScript statements are being executed. Regenerating specifications for updated
libraries and generating specifications for new libraries can be done in only a few
hours per library, which we consider acceptable for specifications that can be reused
repeatedly by a static analysis and other applications.

RQ3: Usefulness for Static Analysis

We evaluate the usefulness of TASER-extracted taint specifications by integrating
them into LGTM, a state-of-the-art, industrial static analysis platform. A free instance
hosted at https://lgtm.com continuously checks more than 130000 open-source
projects (including thousands of npm modules) for security problems. Without the
specifications, LGTM reasons about third-party npm modules based on a limited
number of manually created taint specifications. We add the extracted specifications
into the static analysis and measure how many additional security alerts the analysis
reports.

Figure 11.7 shows the improvements gained from enhancing LGTM’s standard
security analysis suite with the additional sinks and propagation summaries extracted
by TASER. The first column lists the LGTM rule ID; for instance, js/path-injection
flags potential directory-traversal vulnerabilities. The second column shows the
number of new alerts found by incorporating our additional sinks and propagation
summaries. In total, TASER enables LGTM to find 136 otherwise missed potential
security problems.

To better understand the quality of the added alerts, we randomly sample 30 of the
new alerts (five for rules with five or more results, and all results for the other rules).
We find that 24 of them are true positives in the sense that they exhibit flow from a
source to a sink. Of the six false positives, five are due to imprecision of the static

5How many of these new results correspond to exploitable security vulnerabilities is a different
question, which we do not consider here.
Figure 11.8: Example of a new alert found based on a TASER-inferred specification.

analysis (and hence unrelated to TASER), and one is due to a spurious additional sink extracted by TASER.

Figure 11.8 shows a simple example of a newly identified alert for the js/path-injection rule, which originates from the FineUploader/server-examples project from GitHub. The req argument contains an HTTP request object, so the LGTM security analysis considers req.params.uuid to be untrusted data since it might originate from a malicious attacker. After being concatenated with another string, it is passed to the rimraf function, which (recursively) deletes the file system path denoted by this string if it exists. The value of req.params.uuid is not checked, so in particular it could contain “..” components, allowing an attacker to delete arbitrary files on the file system.

Even though the flow from source to sink is very simple, LGTM does not flag this out-of-the-box, since it does not have a model of the rimraf package, and its implementation is too complicated for the static analysis to model as explained above. Our additional sinks, however, identify the first parameter of rimraf as a taint sink for js/path-injection, allowing LGTM to flag this code.

As an example of the use of propagation summaries, we notice that four of the five new alerts for js/remote-property-injection we examined make use of the propagation summaries for _.forEach, a lodash function similar in style to _.forIn. These propagation summaries describe flow through a callback parameter, underscoring the importance of supporting such summaries.

RQ4: Comparison with Manually Created Specifications

The standard LGTM security analysis suite already includes manually written models of many popular npm packages, including sinks and taint propagation rules. By examining our automatically extracted taint specifications for overlap with these manually written models, we find that 12 of our additional sinks and 40 of our propagation summaries correspond to existing models. On the one hand, this confirms that the specifications we extract are practically relevant. On the other hand, it also shows that the vast majority of the TASER-extracted specifications are not yet covered by manual models.

As one example, our dynamic analysis correctly identifies the first parameter of the
single function exported by the cross-spawn package as a sink for js/command-line-injection. LGTM includes a manual model for this. Additionally, TASER also identifies an analogous sink for the win-spawn package, a by now deprecated predecessor of cross-spawn. LGTM does not include a model for this, presumably because win-spawn is less popular than cross-spawn, and the LGTM analysis authors focused on popular packages in writing their models. Our automated approach is not limited by such considerations and can hence provide a much broader coverage.

11.7 Discussion

In this section we present limitations of our work, and we discuss how automatically inferred taint specification can improve the current security practices in the JavaScript community.

Limitations

TASER is affected by the well-known limitations of dynamic analysis, i.e., one can analyze only code that is executed. Therefore, adequate test coverage is essential for effectively extracting taint specifications. Even though in our evaluation we do not directly measure or aim to increase coverage for the used test suites, by analyzing several clients of a given library, we increase the chance of observing multiple realistic use cases of the library. Our hypothesis is that these inputs are representative for most of the library usages in the wild. Related work employs similar assumptions [96, 100].

In the current work, we do not consider implicit flows which were shown to have limited value for detecting integrity issues in server-side JavaScript [131]. However, future work should evaluate whether this assumption also holds for extracting taint specifications.

In our evaluation, we judge the usefulness of the extracted summaries by showing that they improve an existing static analysis. Similarly to the work of Clapp et al. [25], future work should perform a more extensive set of experiments in which the quality of the extracted specifications is directly evaluated, e.g., by extensively comparing with manually written specifications.

Comparison with Coarse-Grained Warnings

The current security practice in the npm community, as implemented, e.g., in the npm audit tool, is to warn users whenever they are relying on a module with a known vulnerability. This approach suffers from two limitations. First, it is limited to previously known and reported vulnerabilities. Second, it often causes spurious warnings, as a warning is issued for every package that depends on a vulnerable module, independently of whether the first module’s use of the second module is affected by the vulnerability.

We show that our approach can help address both these limitations. First, one can use TASER to automatically find vulnerabilities, i.e., unsanitized, undocumented
Figure 11.9: Benign input for the vulnerability described in npm advisory number 27.

additional sinks. To evaluate the effectiveness of this approach, we run TASER using benign inputs for 24 vulnerable packages aggregated by related work [130]. Our approach finds additional sinks in 11 of the 24 packages. Limitations of the existing policy, i.e., missing sources, and insufficient modeling of arrays are the reasons why TASER does not find the remaining sinks.

Second, TASER-extracted specifications can help identify the problematic entry point of a vulnerable library. This can reduce the false positive rate of the npm audit solution by only reporting an alarm when user-controlled values can reach that entry point. While implementing a more precise replacement for npm audit based on TASER-extracted specifications is out of the scope of this work, we illustrate its potential effectiveness with the vulnerability in Figure 11.9. The example shows benign inputs passed to a module that suffers from a known vulnerability. TASER infers the following additional sink for the vulnerable module:

\[(\text{member printer (parameter 0 (member printDirect (root printer)))})\]

Instead of alerting all users of the printer module, as npm audit would do, the extracted specification could help raise an alarm only for users that call the vulnerable entry point with a non-constant string value. Similarly to Synode [130], an improvement over the current npm audit tool could check whether the value passed at the entry point is statically computable, and raise an alarm only if that is not the case. As illustrated by this example, TASER can help reduce the false positives of the existing technique by only alerting developers when necessary. In addition to an npm audit-like tool, IDEs could also alert developers that specific entry points should be treated as sinks.

https://www.npmjs.com/advisories/27
11.8  Related Work

Specifications of libraries and frameworks  The idea of using pre-generated specifications to aid static analysis of library and framework code has been pursued previously [8, 11, 25, 61, 112]. The only other work that uses a dynamic analysis to infer taint specifications is the technique by Clapp et al. [25], which infers specifications for the Android SDK. Our work differs in multiple ways. First, we introduce the idea of membrane-based, multi-module analysis, allowing TASER to infer specifications for all modules used directly or indirectly by a client. In contrast, Clapp et al. [25] infer specifications from a client’s usage of a single framework. Second, we use a fine-grained specification mechanism that can track flows at the level of individual properties and can express flows via callbacks, while their specifications are coarse-grained, i.e., only tracking flows between parameters and return values. Finally, our analysis accounts for the dynamic nature of JavaScript, e.g., using the star expression (*) as described in Section 11.3.

Taint analysis  Taint analysis [26] has been used for checking security properties [9, 52, 102, 135] and other analysis problems [48, 62]. In particular, there are both static [105] and dynamic [71, 86] taint analyses for JavaScript. Taint specifications inferred with a TASER-like approach could in principle be plugged into any static taint analysis that involves third-party modules. To the best of our knowledge, we are the first to present such an approach for static taint analysis for JavaScript. To facilitate the use of taint analysis for checking security properties, some work proposes to infer which functions to consider as sources, sinks, and sanitizers [21, 120]. In contrast, TASER infers specifications that summarize flows through entire third-party modules.

JavaScript security  Previous work has shown that there is a wide range of vulnerabilities in JavaScript software in general and for the Node.js platform in particular, e.g., injection vulnerabilities [130], regular expression-based denial of service vulnerabilities [29, 129], and implementation issues in Node.js [19]. Many existing mitigation techniques rely on some form of dynamic enforcement [6, 30, 51, 130, 137]. Since even a small runtime overhead is often unacceptable, especially for server-side applications, our work instead aims at improving the static detection of vulnerabilities, e.g., via the LGTM analysis tool used in our evaluation. Zimmermann et al. [147] have shown that npm modules depend on many (79, on average) other npm modules, which become part of a module’s attack surface. The TASER-inferred taint specifications enable a static analysis to consider such third-party modules without relying on manually created specifications or whole-program analysis.

Membranes  The membrane pattern, introduced by Miller [97], has been applied in several settings [27, 51, 72, 96, 97, 100]. The idea is to separate two object graphs, such that operations taking place on the boundary between the graphs can be captured and potentially modified. TASER uses membranes at the boundary between a module and a client, and between different modules, to capture taint flows between them.
Coarse-grained alerts  Some tools, most prominently npm audit[7] and Snyk[8], warn developers about known vulnerabilities in any of their dependencies. As discussed by Lauinger et al. [84], an important limitation is that such tools do not analyze how dependencies are used, and will warn even about vulnerabilities in code that a client does not use, or not use in a vulnerable way. A more precise analysis, e.g., based on specifications inferred by TASER, avoids the inevitable false positives caused by coarse-grained alerts.

JavaScript program analysis  The dynamic and reflective nature of JavaScript makes it difficult to construct sound, whole-program static analyses that scale to large real-world applications [4, 43, 66, 80, 105, 128, 129]. For that reason, much research has been devoted to constructing more pragmatic bug-detection tools [6, 24, 49, 56, 57, 71, 116, 124, 125, 131]. Some frameworks facilitate the implementation of dynamic JavaScript analyses [126, 134], including NodeProf [134] that TASER builds upon.

11.9 Conclusion

The massive use of third-party libraries in modern JavaScript web development calls for new techniques to discover security vulnerabilities. Modular static taint analysis is a powerful approach, as demonstrated by the successful commercial tool LGTM, but it critically relies on taint specifications of the libraries being used. Writing such specifications manually is demanding and error-prone, so automated solutions are needed. This work presents such a solution. It combines and adapts a number of ideas from previous work, in particular the idea of inferring information flow specifications using dynamic analysis [25], the membrane mechanism [27, 97], the use of test suites of open-source library clients, and the notion of dynamic access paths [96].

Our implementation and experiments demonstrate that this design is able to automatically detect non-trivial and accurate taint flow specifications in widely used Node.js modules, which enables an existing static analyzer, LGTM, to discover many previously unknown security vulnerabilities. We believe this approach is a promising alternative to the current coarse-grained security tools like npm audit that only consider the package dependency structure but completely ignore the dataflow. Our next step is to extend the implementation with support for more testing frameworks, and then apply the approach in production. Thereby we can gain experience with its use in practice and possibly refine the expressiveness of the taint specifications to further increase the ability to detect vulnerabilities in real-world JavaScript applications.

[7] https://docs.npmjs.com/cli/audit
[8] https://snyk.io/
Chapter 12

Modular Call Graph Construction for Security Scanning of Node.js Applications

Abstract

Most of the code in typical Node.js applications comes from third-party libraries that consist of a large number of interdependent modules. Because of the dynamic features of JavaScript, it is difficult to obtain detailed information about the module dependencies, which is vital for reasoning about the potential consequences of security vulnerabilities in libraries, and for many other software development tasks. The underlying challenge is how to construct precise call graphs that capture the connectivity between functions in the modules.

In this work we present a novel approach to call graph construction for Node.js applications that is modular, taking into account the modular structure of Node.js applications, and sufficiently accurate and efficient to be practically useful. We demonstrate experimentally that the constructed call graphs are useful for security scanning, reducing the number of false positives by 81% compared to npm audit and with zero false negatives. Compared to js-callgraph, the call graph construction is significantly more accurate and efficient. The experiments also show that the analysis time is reduced substantially when reusing modular call graphs.

12.1 Introduction

The npm package repository is the largest software repository in the world with more than one million JavaScript packages. These packages tend to depend heavily on each other: on average each package depends on more than 50 other packages when considering both direct and transitive dependencies [74][47]. Packages are comprised of modules, which correspond to JavaScript files that are loaded individually by the module system. A typical Node.js application thus consists of hundreds or
thousands of JavaScript files, with more than 90% of the code coming from third-party libraries [77].

As security vulnerabilities in libraries are frequently discovered [32, 130, 132, 142, 144, 147], to ensure maximal security of the applications it is important for the application developers to know the structure of dependencies within the applications. One of OWASP’s top 10 categories of web application security risks is “Using Components with Known Vulnerabilities”[1]. A study has shown that up to 40% of all npm packages depend on code with at least one publicly known vulnerability [147]. Another study has found that 12% of the available packages have a release that directly relies on a version of a package that contains a vulnerability listed in Snyk’s security reports [32], and if taking transitive dependencies and more security reports into account the percentage is likely much higher. (A related study [84] shows similar numbers for JavaScript on web pages, but we here focus on the Node.js ecosystem.) This situation has motivated the development of security scanners, which are tools that warn developers if their programs either directly or transitively depend on a library with a known security vulnerability. Existing security scanners, such as Dependabot[2] npm audit[3], and Snyk[4], only consider the package dependency structure that is specified in the package.json files, without looking at the program code. This means that they cannot tell whether the client actually uses the vulnerable part of the library, and consequently client developers are often overwhelmed with false-positive warnings. In a study of npm projects where such security scanners reported high-priority security warnings, 73% of the projects did not actually use the vulnerable parts of the libraries [142]. That study also concludes that mapping the usage of library code in client projects is difficult and that better automatic approaches are needed.

In this paper, we present an analysis that constructs call graphs for Node.js applications. A call graph has a node for each function in the application and an edge from a node $F$ to a node $G$ if $F$ may call $G$ [122]. It is well known that call graphs have many applications for a variety of development tools [43]. We demonstrate that it is possible to considerably improve the precision and usefulness of security scanning by using call graphs. For this purpose, the call graph analyzer ideally needs to be sound, precise, and efficient when applied to real-world applications. We do not require theoretical soundness guarantees, but if the constructed call graph misses many call edges that are possible in concrete executions then security issues may be overlooked. High precision is important, because if the call graph has too many edges then the technique is no better than the existing security scanners that only look at the package dependency structure. Efficiency is necessary such that the tool can be integrated into existing development processes.

Besides security scanning, other possible applications of the call graphs include change impact analysis [2, 55], which may be useful for finding out how breaking

---

[3] https://docs.npmjs.com/cli/audit
changes in library updates affect client code \[31\]. Furthermore, precise knowledge of function-level dependencies across packages can also be useful for library developers to learn how the library features are being used, and in IDEs for code navigation, completion, and refactoring tools \[43\].

Multiple approaches for constructing call graphs for JavaScript programs already exist (see Section \[12.3\]), but none of them take advantage of the module structure of Node.js applications. The salient feature of the call graph analysis we present is its *modularity*. The analysis has two stages: First, each module is analyzed separately, resulting in a module summary. Second, the module summaries are composed for producing call graphs for collections of modules. This modular approach is an ideal match with the massive reuse of packages in the Node.js ecosystem. As a variant of the example above, assume both packages \(A_1\) and \(A_2\) depend on \(B\), which in turn depends on \(C\). Call graphs can then be built bottom-up in the package dependency graph. After creating module summaries for \(B\) and \(C\), we can build a call graph \(G_{BC}\) for the collection \(\{B, C\}\). Then later we can build call graphs for both \(\{A_1, B, C\}\) and \(\{A_2, B, C\}\) by reusing \(G_{BC}\) and only adding information from the module summaries for \(A_1\) and \(A_2\), respectively, thereby avoiding redundant work.

In summary, the main contributions of this paper are:

- We propose an analysis, JAM\(^5\), that constructs call graphs for JavaScript programs modularly, by first creating module summaries (Section \[12.4\]) and then composing the summaries and building call graphs for collections of modules (Section \[12.5\]).

- We present a proof-of-concept tool that leverages call graph construction for security scanning (Section \[12.6\]).

- We demonstrate experimentally (Section \[12.7\]) that on 12 Node.js applications, the call-graph-based security scanner finds the same 8 vulnerabilities as npm audit while reducing the number of false positives by 81\% (from 26 to 5), and that the analysis time is reduced substantially when reusing modular call graphs. Moreover, compared to the state-of-the-art call graph construction tool *js-callgraph*\(^6\), which is a further development of the tool by Feldthaus et al. \[43\], JAM achieves substantially better precision, accuracy, and analysis time.

### 12.2 Motivating Example

Consider the npm application *writex*\(^7\) for converting markdown files into latex. For version 1.0.4 of *writex* (the most recent version as of August 2020), the npm audit security scanner reports that *writex* may be affected by up to 10 known vulnerabilities. They originate from 5 different security advisories, but npm audit reports an alarm

---

\(^5\) [JavaScript module analyzer](https://github.com/Persper/js-callgraph)

\(^6\) [https://github.com/Persper/js-callgraph](https://github.com/Persper/js-callgraph)

\(^7\) [https://www.npmjs.com/package/writex](https://www.npmjs.com/package/writex)
for every occurrence of a vulnerable dependency, and some appear through several
dependency chains. For example, a prototype pollution vulnerability affecting lodash
prior to version 4.17.19 is reported twice, because a vulnerable version of lodash is
required through both writex → lodash-template-stream → lodash and writex → gaze →
globule → lodash.

By manually examining the source code of writex, we find that only 1 of the 5
different advisories is a true positive: a regular expression vulnerability affecting
the minimatch(path, pattern) function of the minimatch library for matching strings
against glob patterns. We classify an alarm as a true positive if the vulnerable library
function is used by the application, disregarding whether an actual exploit is feasible.
For the remaining 4 vulnerabilities (spanning 8 different alarms), the vulnerable
function is not reachable from the writex application, and those alarms can therefore
safely be ignored.

Using JAM to run a call-graph-based security scan of writex, only the true positive
minimatch vulnerability is reported. Furthermore, the JAM call graph shows through
which chain of function calls the vulnerable function is reachable, making it easier to
determine whether the vulnerability is exploitable compared to the alarms reported by
npm audit. For the true positive alarm in the writex client, the following fragment of
a stack trace shows how the vulnerable function on line 114 of minimatch.js may be
reached via the globule package.

writex/node_modules/minimatch/minimatch.js:114:0
writex/node_modules/minimatch/minimatch.js:74:9
writex/node_modules/globule/lib/globule.js:35:30
...

Two other functions in the minimatch API, filter and match, use the vulnerable
minimatch function internally. This means that a client using those functions may also
be vulnerable, however, this fact is unclear from the advisory description, so the client
developer might be inclined to regard the alarm from npm audit as a false positive. A
user of JAM is unlikely to make a similar mistake, because the call graph generated by
JAM records the internal calls to minimatch.

The writex application transitively depends on 53 different packages consisting
of a total of 187 JavaScript files (modules). The call graph generated by JAM shows
that only 90 of the modules (spanning 42 packages) are reachable from the writex
application. These numbers illustrate why the npm audit security scanner produces
so much noise; if half of the files are dead code, it is unsurprising that most of the
security scanner alarms are false positives.

JAM builds the call graph for writex and all its dependencies in 2.6 seconds, and it
infers a unique caller to the vulnerable function in minimatch.js. In comparison,
the existing tool js-callgraph takes 24 minutes to analyze that application, and the

\[https://www.npmjs.com/advisories/1523\]
\[https://www.npmjs.com/advisories/118\]
resulting call graph contains 1379 call sites to the vulnerable function, so it cannot be used for providing a stack trace as the one shown above.

Furthermore, the modular analysis approach of JAM makes it possible to reuse the module summaries. For example, if we have already produced modular call graphs for `writex`'s direct dependencies (which are all used also by many other applications), then the analysis time for `writex` is reduced from 2.6s to 0.2s.

### 12.3 Key Challenges

To understand some of the challenges with computing call graphs for JavaScript applications, we describe two examples.

**Example 1** Consider the code below consisting of the two modules `lib1.js` and `client1.js`:

```javascript
lib1.js:
1 module.exports.filter = (iteratee) => {
2   return (arr) => {
3     const res = [];
4     for (var x of arr) {
5       if (iteratee(x))
6       res.push(x);
7     }
8     return res;
9   };
10 }

client1.js:
11 const filter = require('./lib1.js').filter;
12 console.log(filter(x => x % 2 == 0)([1, 2, 3]));
```

The `lib1.js` module implements a curried filter function that takes a function argument, `iteratee`, and returns another function. This function then takes an array argument, `arr`, and iterates over all the elements of the array, passing each element to the `iteratee` function, and eventually returns an array containing all of the elements for which `iteratee` returned a truthy value.

To analyze this code, the first challenge we must address is that the code is split into modules. The public interface of a module is constructed dynamically by writing properties to the special object `module.exports`. For example, the `filter` method is exported by `lib1.js` as illustrated on line 1. When a module is loaded, an object containing exactly the properties written to `module.exports` is returned. The module loading happens by calling the `require` function, as demonstrated on line 11. It is possible, and also quite common, to use dynamic property writes to create the `module.exports` object, and it is therefore in general difficult to statically compute the structure of a `module.exports` object. As we explain in Sections 12.4 and 12.5, we approximate the module structure using a light-weight field-based static analysis that tracks what functions are written to which fields (also called properties.

---

10^This module system is known as CommonJS. The standardized ES6 module system is also supported by JAM but is rarely used in practice.
in JavaScript) but without distinguishing individual objects. By combining the field-based observations with a heuristic for filtering irrelevant functions, we can statically compute the module structure with high precision.

The js-callgraph tool, which does not take a modular approach, loses precision in this example and confuses the filter function with Array.prototype.filter from JavaScript’s standard library.

The second challenge is that a higher-order function is used; the filter function on line 1 takes a function as argument and also returns a function. The analysis should be able to determine that the call to iteratee on line 5 is really a call to the arrow function on line 12, and that the call to the value returned from filter on line 12 (blue parentheses) is really a call to the function on lines 2–9. Our call graph analysis keeps track of all these functions and where they are being called.

**Example 2** Consider the following application consisting of lib2.js and client2.js:

```javascript
lib2.js:
13 function Arit () { ... }
14 Arit.prototype.sum = (x, y) => x + y;
15 Arit.prototype.mul = (x, y) => x * y;
16 ...
17 module.exports.Arit = Arit;

client2.js:
18 const lib = require('./lib2');
19 const arit = new lib.Arit();
20 ... arit.sum(a, b) ...
```

The lib2.js module exports a constructor, Arit, which is used to construct objects with a set of methods for performing basic arithmetic. The client2.js module imports lib2.js and then constructs an Arit object and stores it in the constant arit on line 19. On line 20, the sum method is called on arit, resulting in an invocation of the function defined on line 14.

For the call graph analysis to resolve the call on line 20, a natural approach would be a form of dataflow analysis or pointer analysis that keeps track of what objects each expression may evaluate to. However, such an approach is extremely challenging for JavaScript, and no existing analysis of that kind is capable of scaling to real-world programs [80, 105, 133]. As we explain in Section 12.4, the field-based approach ensures that our analysis both scales well and remains precise for real-world programs. In particular, since the method call on line 20 involves a property named sum, it is connected to the write to the property named sum on line 14.

While the field-based approach could easily result in spurious call edges added to functions stored in properties with the same name but in unrelated objects, previous work has demonstrated that it works well in practice for client-side JavaScript applications that build on jQuery and related libraries [43]. The js-callgraph tool extends the tool by Feldthaus et al. [43] with support for newer JavaScript features, including ES6/CommonJS/AMD module loading, but the analysis itself is not modular, i.e., it does not take advantage of the module structure of the applications. As we demonstrate in Section 12.7, js-callgraph is not suited for analyzing Node.js applications since they often contain many modules.
12.4 Module Summary Construction

The first phase of the analysis constructs a summary for each module, without considering the connections between the modules. Let $\text{Loc}$, $\text{Prop}$, $\text{Var}$, and $\text{Exp}$ denote the sets of all possible source code locations, property names, variable names (including parameters), and program expressions, respectively. Given a single JavaScript file $f$ as input, we compute a module summary $\beta_f$ consisting of three separate pieces of information:

- $\beta_f.\text{calls}$: $\text{Loc} \mapsto P(\text{AccessPath})$ is a function call summary, which for each function definition (represented by its source location) describes all the functions that are called within its body, using a special access path mechanism introduced below.

- $\beta_f.\text{returns}$: $\text{Loc} \mapsto P(\text{AccessPath})$ is a function return summary, which for each function definition similarly describes its possible return values.

- $\beta_f.\text{props}$: $\text{Prop} \mapsto P(\text{AccessPath})$ is an object property summary, which for each property name describes the values that may be assigned to object properties of that name.

We use a light-weight static analysis to compute the three components, as explained next.

**Access Paths** The static analysis uses an access path mechanism (inspired by Mezzetti et al. [96] and Møller et al. [101]) to describe values of expressions in the program. The language of access paths is defined by the grammar in Figure 12.1.

- $<m>$ denotes the loading of a module $m$, as in, e.g., `require('m')`.
- $\text{Fun}(f, l)$ denotes a function definition in file $f$ at line $l$.
- $\text{Fun}(f, l).\text{Param}[i]$ denotes the $i$'th parameter of the function definition in file $f$ at line $l$.
- $\text{ap}.P$ denotes accesses to properties named $P$ of objects denoted by the access path $\text{ap}$.
- $\text{ap}(S_1, \ldots, S_n)$ denotes calls to functions denoted by $\text{ap}$ where the $i$'th argument is denoted by an access path in $S_i$.
- $U$ is used for expressions where the static analysis is unable to assign any other access path, as explained later.

---

11Source locations consist of file name, begin line, and begin column, and are therefore always unique for every function definition. However, for brevity we omit the column information and only write $\langle$ file name, begin line $\rangle$.

12We use the terms module and file interchangeably since a module is always stored in a single file in Node.js.
AccessPath ::= < ImportPath >
| Fun(f, l)
| Fun(f, l).Param[i]
| AccessPath . Prop
| AccessPath (P(AccessPath), ...)
| U

Figure 12.1: Grammar for access paths.

APV(E) := |
{<m>}
if E = require(m) or import ... from m

{ap.P | ap ∈ APV(E')} ∪ lookupV(P)
if E = E'.P

{ap(APV(E1),...,APV(E_n)) | ap ∈ APV(E')}
if E = E'(E1,...,E_n) or E = new E'(E1,...,E_n)

lookupV(X)
if E = X where X is a non-parameter variable

{Fun(f,l).Param[n]} ∪ lookupV(X)
if E = X where X is the n’th parameter in a function created at line l in file f

{Fun(f,l)}
if E is a function definition at line l in file f

APV(E1) ∪ APV(E2)
if E = E' ? E1 : E2 or E = E1 || E2 or E = E1 & & E2

{U}
otherwise

lookupV(Z) := |
U E∈Alias(Z) APV∪Z(E)
if Z ∉ V

∅ otherwise

Figure 12.2: Access path computation.
As an example, the access path Fun(lib1,1).Param[0] describes the iteratee parameter of the filter function on line 1 in Example 1.

**Alias Analysis and Access Path Analysis** The module summary construction computes access paths for each expression in the analyzed file using an access path analysis. This analysis uses a simple field-based alias analysis to compute a map

\[ \text{Alias}_f: (\text{Var} \cup \text{Prop}) \rightarrow \mathcal{P}(\text{Exp}) \]

for the file \( f \), such that if the value of an expression \( E \) is written to a variable or property \( X \), then \( E \in \text{Alias}_f(X) \).\(^1\)

The alias analysis constructs the map through a single traversal of \( f \)'s AST. At each assignment \( X = E \) or \( E'.P = E \), the expression \( E \) is added to \( \text{Alias}_f(X) \) or \( \text{Alias}_f(P) \), respectively. Transitive dataflow is taken into account later when the alias information is being used.

Based on the alias analysis result, a map is computed that assigns a set of access paths to each expression in \( f \):

\[ \text{AccPaths}_f: \text{Exp} \rightarrow \mathcal{P}(\text{AccessPath}) \]

The map is computed by \( \text{AccPaths}_f(E) = AP_0(E) \) for each expression \( E \) in \( f \), where \( AP \) is defined in Figure 12.2. The subscript \( V \) in \( AP \) and in the lookup auxiliary function ensures termination for recurrences of expressions. For module loads, such as, require('lodash'), the access path corresponding to the module load string is returned. For a property read \( E.P \), \( AP \) computes the access paths both by recursively computing the access paths for the sub-expression \( E \) and appending \( .P \), and by using the lookup function to compute the access paths of the expressions that the alias analysis has determined to be aliased by \( .P \). For a call, \( AP \) computes the receiver and argument access paths recursively, and then creates a call access path for each receiver access path. For a read of a variable that is not a parameter, \( AP \) uses lookup to recursively compute the access paths for the expressions aliased by the variable. A parameter is treated similarly to a variable read but also adds a parameter access path. For a function definition expression, \( AP \) creates the corresponding Fun access path. For conditional and logical expressions, the access paths are computed as the union of the access paths for the sub-expressions. In any other case (e.g., a + operation), the access path \( U \) is assigned to the expression.

Notice that this analysis design combines field-based analysis \(^4\) and the use of access paths \(^{96,101}\), which enables the analysis to reason about individual modules.

---

\(^1\)In the implementation, we extend Prop to also include special pre- and postfix forms of property names. For example, for a write \( x["foo"] = y \), the analysis records that \( y \) is written to a property that has "foo" as a prefix: \( \text{Alias}_f(\text{foo}) = \{y\} \). When analyzing, for example, \( z.\text{fooBar()} \), the analysis will then predict that the function \( y \) is among the possible callees. This extension ensures that the analysis can handle some dynamic property reads, but in a way that does not lead to a major loss of precision.

\(^4\)The analysis ignores dynamic property reads that are not of the form described in footnote 13, but since it is field-based this has little effect on its recall (see Section 12.5).
CHAPTER 12. MODULAR CALL GRAPH CONSTRUCTION FOR SECURITY SCANNING OF NODE.JS APPLICATIONS

Summary Construction  The function call summary $\beta_f\text{.calls}$ is formed by grouping the access paths $\text{AccPaths}_f(E)$ for each expression $E$ according to the function definition containing $E$. (For an expression in a nested function, we here only consider the inner-most function.) Every Node.js module is wrapped in a function upon load of the module. We use the special access path $\text{Fun}(f, \text{Main})$ to refer to the function that wraps the analyzed file $f$. Similarly, $(f, \text{Main})$ denotes the location of that function.

The function return summary $\beta_f\text{.returns}$ is similarly computed by grouping the access paths assigned to the expressions of return statements in the function.

Finally, the object property summary $\beta_f\text{.props}$ is constructed as $\beta_f\text{.props}(P) = \bigcup_{E \in \text{Alias}_f(P)} \text{AccPaths}_f(E)$ for each property $P$.

Example 3  Continuing Example 1 we obtain the module summaries $\beta_{\text{client1}}$ and $\beta_{\text{lib1}}$. Since the `filter` function is called in the outermost scope of the `client1.js` file, the call of `filter` is recorded as follows.

$$\beta_{\text{client1}}\text{.calls}((\text{client1}, \text{Main})) = \{<\text{lib1}>.\text{filter}(...), \ldots\}$$

Furthermore, the return summary of the `filter` function records the access path of the function returned:

$$\beta_{\text{lib1}}\text{.returns}((\text{lib1}, 1)) = \{\text{Fun}(\text{lib1}, 2)\}$$

Example 4  Continuing Example 2, the function defined on line 14 is written to the property `sum`, and the function defined on line 15 is written to the property `mul`. Therefore the object property summary for the module `lib2` contains the following entries:

$$\beta_{\text{lib2}}\text{.props}(\text{sum}) = \{\text{Fun}(\text{lib2}, 14)\}$$
$$\beta_{\text{lib2}}\text{.props}(\text{mul}) = \{\text{Fun}(\text{lib2}, 15)\}$$

12.5 Call Graph Construction

Before constructing the call graph for a Node.js application, we combine the module summaries for all its modules. For example, $\beta\text{.calls}$ (omitting the module name) denotes the combined call summary and is computed by $\beta\text{.calls}(loc) = \bigcup_{f \in M} \beta_f\text{.calls}(loc)$ for all $loc \in \text{Loc}$ where $M$ is the set of modules, and similarly for the other components.

The call graph needs to span across multiple modules, so in the call graph construction phase, we combine the module summaries from each file into a call graph $G = (V, E, \beta, \alpha)$ with nodes $V \subseteq \text{Loc}$ corresponding to function definitions and edges $E \subseteq \text{Loc} \times \text{Loc} \times \text{AccessPath}$ that represent the call edges, $\beta$ is the combined module summary, and $\alpha$ is explained below. Each edge in $E$ is annotated with the access path of a function call between the two functions. We use these annotations when resolving calls to higher-order function parameters.
12.5. CALL GRAPH CONSTRUCTION

Computing the call graph amounts to solving the constraints generated by the rules of Figure 12.3. The constraints involve two relations: \( E \), which contains the call graph edges, and \( \alpha \subseteq \text{AccessPath} \times \text{AccessPath} \times \text{AccessPath} \), which is used for resolving function calls during the analysis. We say that an expression \( E \) in a file \( f \) is represented by an access path \( \text{ap} \) if \( \text{ap} \in \text{AccPaths}(E) \). The notation \( n \rightarrow_{\text{ap}} n' \) is a shorthand for \((n,n',ap) \in E\), meaning that the function at \( n \) may call the function at \( n' \) and \( \text{ap} \) is an access path that represents such a call. Similarly, \( \text{ap} \\sim \text{ap}' \) means \((\text{ap}, \text{ap}', \text{ap}'') \in \alpha \), which intuitively means that expressions represented by \( \text{ap} \) may obtain their function values from expressions represented by \( \text{ap}'' \), and \( \text{ap}' \) represents calls to such functions.

Example 5 The call to \( \text{filter} \) on line 12 marked with red parentheses in Example 1 gives rise to the following entries in \( \alpha \) and \( E \).

\[
\text{<lib1>.filter <lib1>.filter(...)} \sim \text{Fun<lib1>}
\]

\[
\langle \text{client1, Main} \rangle \sim \text{<lib1>.filter(...)} \rightarrow \langle \text{lib1} \rangle
\]

We explain in Examples 6 and 7 how these entries are produced.

The call graph computation works by iteratively extending \( E \) and \( \alpha \) according to the constraint rules until a fixed point is reached. Such a fixed point is guaranteed to exist since \( E \) or \( \alpha \) always increases in size and there are finitely many access paths in the module summaries.

The first two rules in Figure 12.3 only depend on the function call and object property summaries \((\beta \cdot \text{calls} \text{ and } \beta \cdot \text{props})\), and not on \( E \) or \( \alpha \), so they can be resolved in the first iteration of the algorithm.

- \textit{module-call} connects the access path \(<m>\ldots g\), representing the callee of a call access path that is in the function call summary, to an access path \( \text{ap}' \) if the module \( m \) resolves \( g \) to a file \( f' \) and the access path \( \text{ap}' \) is in the object property summary for a file \( f'' \) and property \( g \), where the package that contains \( f' \) (denoted \( \text{package}(f') \)) is the same as or depends directly or transitively on the package that contains \( f'' \). The reason for considering only the object property summaries for those files is explained by the following scenario: If a package \( A \) depends on a package \( B \), which in turn depends on package \( C \), then functions in \( A \) typically do not affect the interface of \( B \), whereas the functions of \( C \) may be re-exported through modules in \( B \). We here use the package dependency structure, because it can be extracted soundly, directly from the \text{package.json} files. If the call is to the module object directly, i.e., \( \text{ap} = <m>() \), then the default exported function is extracted from the object property summary using the property name exports.

\textsuperscript{15}The \texttt{resolve} function in this rule is similar to the \texttt{require.resolve} function from Node.js.

\textsuperscript{16}It is possible to construct a scenario where functions of \( A \) become part of \( B \)’s interface, but we have not observed this behavior in practice, which is why we resort to this heuristic even though it is theoretically unsound.
other-call connects the callee represented by an access path \( ap' \) to itself if \( ap'(\ldots) \) appears in the function call summary, provided that the module-call rule does not apply. The remaining analysis constraints will resolve \( ap' \) to the functions it represents.

Example 6 Based on the module summaries presented in Example 3, the analysis has recorded that \( \text{Fun}\langle \text{lib1},1 \rangle \) is written to the filter property. The rule module-call then applies to the call to filter on line 12, which results in the \( \alpha \) entry shown in Example 5. Since no other functions are written to a property named filter in lib1 or its dependencies, that entry is the only one added to \( \alpha \) for this call. We describe in Example 7 how the corresponding call edge is produced from \( \alpha \).

The next three constraint rules presented in Figure 12.3 model calls to functions returned by other functions (return-call), calls to function parameters (param-call), and calls to functions stored in object properties (prop-call). These rules are applied iteratively since, for example, for the expression \( f()() \), resolving the second call depends on the result of the first call.

• return-call ensures that if \( ap'' \) represents a call to a function value that is returned from a function call \( ap(\ldots) \), i.e. \( \cdots \sim ap'(\ldots) \), then \( ap(\ldots) \) may obtain its function values from the return values of \( ap \) functions. Those functions are retrieved using \( ap \sim \text{Fun}(f,l) \). In the special case where \( ap \) denotes a function definition \( ap = \text{Fun}(f,l) \), we use the return values of that function directly. We retrieve the access paths \( ap' \) representing return values using the function return summary \( (ap' \in \beta_{f}.\text{returns}(\langle f,l \rangle)) \).

• param-call ensures that if \( ap'' \) represents a call to a function value that comes from the \( n \)'th parameter of a function \( \text{Fun}(f,l) \), i.e. \( \cdots \sim ap'(\ldots) \), then the function values of that parameter may come from the corresponding arguments at call sites. The access paths \( ap \) of the call sites are retrieved from the annotations of the call edges that point to \( \langle f,l \rangle \), i.e. \( \cdots \sim ap \langle f,l \rangle \). We use \( \text{arg}(ap,n) \) to denote the set of access paths at the \( n \)'th argument of the call access path \( ap \).

• prop-call ensures that if \( ap'' \) represents a call to a function value that comes from an object property \( q \), i.e. \( \cdots \sim ap'(\ldots) \), then the possible function values \( ap' \) include those found in the object property summary for \( q \) \( (ap' \in \beta_{q}.\text{props}(q)) \).

The remaining two rules model transitivity and construction of call edges.

• transitive models the fact that, for calls represented by some access path \( ap'' \), if an expression \( E \) represented by \( ap \) may obtain its values from an expression \( E' \) represented by \( ap' \), and \( ap' \) may obtain its values from an expression \( E'' \) represented by \( ap'' \), then \( E \) may obtain its values from \( E'' \). We require the access path \( ap'' \) to be the same in the two premises to avoid mixing together calls from different call sites to the same function.
[module-call]
\[\begin{align*}
ap &= \langle m > \ldots g(\ldots) \in \beta \cdot \text{calls}((\_ , \_ ) ) \quad f' &= \text{resolve}(m) \\
ap' &\in \beta_f' \cdot \text{props}(g) \quad \text{package}(f') \text{ is or depends on package}(f'') \\
\langle m > \ldots g &\sim ap' \\
\end{align*}\]

[other-call]
\[\begin{align*}
ap &= ap'(\ldots) \in \beta \cdot \text{calls}((\_ , \_ ) ) \quad \text{module-call does not apply} \\
ap' &\sim ap' \\
\end{align*}\]

[return-call]
\[\begin{align*}
\ldots &\sim ap(\ldots) \\
ap &\sim \text{Fun}(f,l) \lor ap = \text{Fun}(f,l) \\
ap' &\in \beta_f \cdot \text{returns}((f,l)) \\
ap(\ldots) &\sim ap' \\
\end{align*}\]

[param-call]
\[\begin{align*}
\ldots &\sim \text{Fun}(f,l) \cdot \text{Param}[n] \\
\ldots &\sim (f,l) \\
ap' &\in \text{arg}(ap,n) \\
\text{Fun}(f,l) \cdot \text{Param}[n] &\sim ap' \\
\end{align*}\]

[prop-call]
\[\begin{align*}
\ldots &\sim ap.q \\
ap' &\in \beta \cdot \text{props}(q) \\
ap.q &\sim ap' \\
\end{align*}\]

[transitive]
\[\begin{align*}
ap'' &\sim ap' \\
ap' &\sim ap'' \\
ap &\sim ap'' \\
\end{align*}\]

[edge]
\[\begin{align*}
\ldots &\sim \text{Fun}(f',l') \\
ap &\in \beta \cdot \text{calls}((f,l)) \\
(f,l) &\sim (f',l') \\
\end{align*}\]

Figure 12.3: Analysis constraint rules.
CHAPTER 12. MODULAR CALL GRAPH CONSTRUCTION FOR SECURITY SCANNING OF NODE.JS APPLICATIONS

- **edge** ensures that a call edge is added from \( \langle f, l \rangle \) to \( \langle f', l' \rangle \) whenever there is an entry \( \cdots \sim \text{Fun}(f', l') \) in \( \alpha \) where the access path \( ap \) is in the function call summary of some function \( \langle f, l \rangle \). Intuitively, such an entry in \( \alpha \) tells us that \( \langle f', l' \rangle \) may be called from a call site represented by access path \( ap \), and if \( ap \in \beta \cdot \text{calls}(\langle f, l \rangle) \) then the call site is in the function \( \langle f, l \rangle \).

**Example 7** Continuing Examples 5 and 6, the call edge entry in \( E \) is created by the rule **edge** from the entry in \( \alpha \) since \( \langle \text{lib1}.filter(...) \rangle \in \beta \cdot \text{calls}(\langle \text{client1}, \text{Main} \rangle) \).

**Example 8** Continuing Example 1, let us now consider how the call in blue parentheses on line 12 to \text{filter}’s return value is resolved. From \( \beta_{\text{client1}} \cdot \text{calls} \) we get the access path of this call as \( \langle \text{lib1}.filter(...) \rangle \). Since **module-call** does not match, and the access path ends with \( (\ldots) \), the rule **other-call** applies:

\[
\langle \text{lib1}.filter(...) \rangle \sim \langle \text{lib1}.filter(...) \rangle
\]

Furthermore, we saw in Example 3 that the return summary contains the fact that \text{filter} returns the function \text{Fun}(\langle \text{lib1}, 2 \rangle). By the **module-call** rule, we also have

\[
\langle \text{lib1}.filter(...) \rangle \sim \langle \text{lib1}.filter(...) \rangle
\]

so by **return-call** we have

\[
\langle \text{lib1}.filter(...) \rangle \sim \langle \text{lib1}.filter(...) \rangle \sim \langle \text{lib1}.filter(...) \rangle
\]

which finally by **edge** ensures that an edge is added in \( E \) between the caller and the callee:

\[
\langle \text{client1}, \text{Main} \rangle \sim \langle \text{lib1}, 1 \rangle \sim \langle \text{lib1}, 1 \rangle
\]

**Example 9** Consider the call to \text{iteratee} colored brown on line 5 in Example 1. The access path is \text{Fun}(\langle \text{lib1}, 1 \rangle \cdot \text{Param}[0])(\ldots) for this call. Again, by **other-call**:

\[
\text{Fun}(\langle \text{lib1}, 1 \rangle \cdot \text{Param}[0])(\ldots) \sim \text{Fun}(\langle \text{lib1}, 1 \rangle \cdot \text{Param}[0])(\ldots)
\]

The **param-call** rule says that the parameter call access path is related to the access paths of the 0’th argument of calls flowing into \text{Fun}(\langle \text{lib}, 1 \rangle). The only such call is the call to \text{filter} on line 12 with access path \( \langle \text{lib1}.filter(\{\text{Fun}(\langle \text{client1}, 12 \rangle)\}) \rangle \) (we have omitted the arguments in the access path previously due to space constraints). So by the **param-call** rule:

\[
\text{Fun}(\langle \text{lib1}, 1 \rangle \cdot \text{Param}[0])(\ldots) \sim \text{Fun}(\langle \text{client1}, 12 \rangle)
\]

which by **edge** results in the call edge:

\[
\langle \text{lib1}, 1 \rangle \sim \langle \text{lib1}, 1 \rangle \sim \langle \text{client1}, 12 \rangle
\]
Example 10  Consider the call to sum on line 20 in Example 2. From $\beta_{\text{client2.calls}}$ we have that the access path of sum is $\langle \text{lib2}.\text{Arit().sum} \rangle$. Again, the rule other-call applies:

$$\langle \text{lib2}.\text{Arit().sum} \rangle \sim \langle \text{lib2}.\text{Arit().sum} \rangle$$

This triggers the prop-call rule, which says that $\langle \text{lib2}.\text{Arit().sum} \rangle$ is related to all functions in the object property summary for sum. From Example 4 we know that the only such function is the one defined on line 14.

$$\langle \text{lib2}.\text{Arit().sum} \rangle \sim \langle \text{lib2}.\text{Arit().sum} \rangle \sim \text{Fun} \langle \text{lib2}, 14 \rangle$$

By edge this results in the following call edge:

$$\langle \text{client2}, \text{Main} \rangle \langle \text{lib2}.\text{Arit().sum} \rangle \sim \langle \text{lib2}, 14 \rangle$$

Example 11  Let us consider an example program that requires multiple applications of return-call and an application of transitive, so that we can see how more complex calls are resolved.

```
lib3.js:
21 function e() {...}
22
23 function f() {
24 return e;
25 }

26 function g() {
27 return f;
28 }

29 module.exports.h = function() {
30 return g();
31 }
```

```
client3.js:
34 const lib = require('lib3');
35 const x = lib.h();
36 x();
```

The function h (defined at line 31) returns the return value of g (defined at line 27), which is the function f (defined at line 23), and f returns the function e (defined at line 21). Hence, the expression on line 35 results in calls to the functions f, g, and h, and the expression on line 36 results in a call to the function e.

Let us first consider the call to h (lib.h()). Since $\text{Fun} \langle \text{lib3}, 31 \rangle \in \beta_{\text{lib3.props(h)}}$, we have by the module-call rule:

$$\langle \text{lib3}.\text{h} \rangle \sim \text{Fun} \langle \text{lib3}, 31 \rangle$$

By the edge rule, we then have:

$$\langle \text{client3}, \text{Main} \rangle \langle \text{lib3}.\text{h} \rangle \sim \langle \text{lib3}, 31 \rangle$$
CHAPTER 12. MODULAR CALL GRAPH CONSTRUCTION FOR
SECURITY SCANNING OF NODE.JS APPLICATIONS

We now consider how the analysis resolves \texttt{lib.h}()() to \texttt{f}. By the \textit{other-call} rule we have:

\[
<\text{lib3}.h() <\text{lib3}.h()() \sim <\text{lib3}.h()
\]

From the resolution of the \texttt{lib.h()} call above, we have

\[
<\text{lib3}.h \sim \text{Fun}(\text{lib3}[\text{31}])
\]

and during the module summary construction we have recorded access paths representing return values of \texttt{h}:

\[
\text{Fun}(\text{lib3}[\text{27}])() \in \beta.\text{returns}(\langle \text{lib3}, \text{31} \rangle)
\]

The \textit{return-call} rule then applies:

\[
<\text{lib3}.h() <\text{lib3}.h()() \sim \text{Fun}(\text{lib3}[\text{27}])()
\]

and by a second application of the \textit{return-call} rule:

\[
\text{Fun}(\text{lib3}[\text{27}])() <\text{lib3}.h()() \sim \text{Fun}(\text{lib3}[\text{23}])
\]

Finally, by the \textit{edge} rule, an edge is added from the main function of \texttt{client3} to \texttt{f} in \texttt{lib3}:

\[
\langle \text{client3}, \text{Main} \rangle <\text{lib3}.h()() \sim \langle \text{lib3}, \text{23} \rangle
\]

The last call remaining is the call at line \texttt{36}. For resolving that call we first apply the \textit{other-call} rule:

\[
<\text{lib3}.h()() <\text{lib3}.h()() \sim <\text{lib3}.h()
\]

Next, we apply the \textit{transitive} rule, based on the entries added to \texttt{\alpha} by the two applications of the \textit{return-call} rule for the second call on line \texttt{35}

\[
<\text{lib3}.h()() <\text{lib3}.h()() \sim \text{Fun}(\text{lib3}[\text{23}])
\]

From the function return summary we have

\[
\text{Fun}(\text{lib3}[\text{21}]) \in \beta.\text{returns}(\langle \text{lib3}, \text{23} \rangle)
\]

and finally, by the \textit{return-call} rule:

\[
<\text{lib3}.h()() <\text{lib3}.h()() \sim \text{Fun}(\text{lib3}[\text{21}])
\]

The \textit{edge} rule then adds an edge from the main function of \texttt{client3} to \texttt{e} in \texttt{lib3}:

\[
\langle \text{client3}, \text{Main} \rangle <\text{lib3}.h()() \sim <\text{lib3}, \text{21} \rangle
\]
Modular Analysis  The call graph analysis is modular, as described in Section 12.1. Let us consider the example from the introduction. We assume a call graph \( G_{BC} \) has been built for the modules \( B \) and \( C \), and we want to create a call graph for the application \( A_1 \) that depends on \( B \) and \( C \). The analysis starts with the \( \alpha \) and \( E \) relations from \( G_{BC} \). The analysis then computes the module summaries for every module in \( A_1 \) and applies the first two rules of Figure 12.3 using these summaries. The remaining constraints are now solved using the combined module summaries from \( B \), \( C \), and \( A_1 \). Because \( G_{BC} \) represents a partial result of \( G_{A_1;BC} \), we can compute \( G_{A_1;BC} \) faster with \( G_{BC} \) precomputed, as demonstrated in Section 12.7. The resulting call graphs when using this bottom-up approach are the same as when combining all module summaries in a single step.

Analysis Extensions  The analysis described so far does not have support for built-in functions, getters, setters, and events. We now describe how the analysis is extended to handle these features. The source code of built-in functions is generally unavailable, so the analysis handles these by assuming that their function arguments are always called. We leverage the field-based analysis design, such that \( \text{ap} \sim \text{ap'} \cdot q \) from Figure 12.3 also adds \( \text{ap'} \cdot q \sim \text{ap''} \) for any access path \( \text{ap''} \) that represents callbacks to a built-in function \( q \). For example, if \( q \) is \( \text{map} \), the analysis adds the above entry to \( \alpha \) for each \( \text{ap''} \in x[0] \) since \( \text{Array.prototype.map} \) takes a callback function as the first argument.

JavaScript supports getter (and setter) properties that invoke a function when they are read (or written). For this reason, we extend the module summary with two maps from property names to sets of access paths describing the getter and setter functions similar to how we handle field-based information. We refer to these additional maps as \( \beta \cdot \text{getters} \) and \( \beta \cdot \text{setters} \). For each property \( q \) that is read or written, we add \( \text{Fun}(f, l) \sim \text{ap} \) to \( \alpha \) for each \( \text{ap} \in \beta \cdot \text{getters}(q) \) for getters and \( \text{ap} \in \beta \cdot \text{setters}(q) \) for setters.

We also added a special mechanism for handling the Node.js event system where events are registered using an \( \text{on} \) method and emitted using an \( \text{emit} \) method. Each module summary is augmented with an \textit{event summary} similar to the object property summary. An event summary is a map from event names to sets of access paths. At calls to a method named \( \text{on} \) with two arguments where the first argument is a string, the map is extended with the access paths of the second argument. At \( \text{emit} \) calls, the corresponding access paths for the emitted event is then looked up in the event summary, and call graph edges are added accordingly.

Restricting Object Properties to Adjacent Packages  The number of property writes in applications rises as the number of dependencies grow, so for large applications, it is possible that unrelated object properties from unrelated packages are mixed together. Since this blowup increases the risk of spurious edges added by the \textit{prop-call} rule, we have added a heuristic where the object property summary is only mixed between directly related packages. With this heuristic, the lookup \( \beta \cdot \text{props}(q) \) in \textit{prop-call} only considers the object property summary from the packages that are direct
dependencies or direct dependents to the package with the caller. While this heuristic theoretically makes the technique more unsound, it does not cause any vulnerabilities to be missed in our security scanning experiments (see Section 12.7).

Soundness Assumptions The call graph analysis is not theoretically sound. There are four potential sources of unsoundness [91]: (1) The analysis ignores dynamic property reads/writes unless they are of the special pre- or postfix form mentioned in footnote [13] but since the analysis is field-based, the analysis results are not affected much by this [43]. (2) The adjacent packages heuristic can result in missing edges if values flow between packages that are not directly linked in the package dependency graph. The design choice for the module-call rule may have a similar consequence as mentioned in footnote [16]. However, as such flows occur rarely in practice, the analysis result remains sound for practical purposes. (3) The analysis ignores dynamic module loads and dynamic code generation. We have not found many usages of dynamic module loads, and dynamic code generation is typically also only used sparsely in Node.js programs. (4) The analysis does not model all ECMAScript features such as iterators and implicit calls. However, the parser used by JAM is compatible with all existing ECMAScript versions, so the analysis will still produce results in the presence of these features.

12.6 Security Scanning

To use the call graph for security scanning, the analysis has to know which nodes represent vulnerable functions. We describe known security vulnerabilities from the npm vulnerability database [17] using a simple pattern language, and use the function findNodes (see Figure 12.5) to convert these patterns to source locations [18].

The grammar of the pattern language is shown in Figure 12.4. A vulnerability description (VulnDesc) consists of the advisory ID, the name of the package affected by the vulnerability, the range of affected versions, and an API-Pattern identifying the vulnerable parts of the library API. AdvisoryID is a number, and ModuleName and VersionRange are strings. An API-Pattern can express disjunctions ([API-Pattern, ..., API-Pattern]), values obtained from loading modules (<ImportPath>), values read from properties (API-Pattern . Prop), and return values of functions (API-Pattern ()). The language of API patterns resembles the language of access paths (Figure 12.1) but is designed for easily identifying API functions, whereas access paths are used only internally by the analysis.

[17] https://www.npmjs.com/advisories
[18] One might be tempted to simply describe the vulnerable functions as a set of source locations directly, but that would make the analysis sensitive to changes in source locations across different versions of the vulnerable dependency.
VulnDesc ::= (AdvisoryID, PackageName, VersionRange, API-Pattern)

API-Pattern ::= { API-Pattern, . . . , API-Pattern }
  | < ImportPath >
  | API-Pattern . Prop
  | API-Pattern ()

Figure 12.4: API patterns.

\[
\text{findNodes}(p) := \begin{cases} 
\bigcup_{p' \in \{p_1, \ldots, p_n\}} \text{findNodes}(p') & \text{if } p = \{p_1, \ldots, p_n\} \\
\{\langle f, l \rangle | f' = \text{resolve}(m) \}
\wedge \text{package}(f') \text{ is or depends on } \text{package}(f'') \\
\wedge ap \in \beta_{f''}.\text{props}(g) \\
\wedge ap \sim \text{Fun}(f, l) \} & \text{else if } p = <m>. . .g \\
\{\langle f, l \rangle | ap \in \beta_{f}.\text{props}(q) \wedge ap \sim \text{Fun}(f, l) \} & \text{else if } p = p'.q \\
\bigcup_{(f,l) \in \text{findNodes}(p')} \{\langle f', l' \rangle | ap \in \beta_{f}.\text{returns}(\langle f, l \rangle) \wedge ap \sim \text{Fun}(f', l') \} & \text{else if } p = p'() 
\end{cases}
\]

Figure 12.5: Algorithm for finding vulnerable functions from API patterns.
If the application depends on some version of the vulnerable package in the vulnerable version range, then the function \textit{findNodes} is used to find the source locations of the vulnerable functions. It uses the vulnerability descriptions and the module summaries to compute the source locations. The first case handles the disjunction pattern, \( p = \{p_1, \ldots, p_n\} \), as the union of the results of calling \textit{findNodes} for each subpattern. For property read sequences on a module object without any calls, \( p = \langle m \rangle \ldots g \), the function source locations are extracted similarly to the module-call rule of Figure 12.3 (this rule also applies when the module object is read directly, i.e., \( p = \langle m \rangle \), in which case the special \texttt{exports} property is used). For a property read, \( p = p'.q \), where \( p \) does not begin with a module load, we first extract the access paths of \( q \) from \( \beta \).\texttt{props} and then the concrete source locations of these access paths from \( \alpha \). For calls to returned values, \( p = p'() \), the source locations represented by \( p' \) are extracted by calling \textit{findNodes} recursively. For each function at these locations, the access paths representing the return values of that function are extracted by a lookup in the return summary of the function. Finally, the actual function definitions are extracted from \( \alpha \) (similar to the return-call rule).

The security scanner can then check whether these functions are reachable in the call graph from the entry node of the application.\footnote{While the call graph analysis works on both libraries and applications, the security scanner is limited to applications that have a single, well-defined entry point.} If any of the functions are reachable, the user is warned, and a link to the informal npm advisory description is presented together with the top of a stack trace leading to the vulnerable function as shown in Section 12.2. The stack trace is computed by traversing backwards in the call graph, from the vulnerable function.

12.7 Evaluation

We have implemented JAM (including the security scanner) in 3000 lines of TypeScript code, using \texttt{acorn}\footnote{https://www.npmjs.com/package/acorn} and \texttt{acorn-walk}\footnote{https://www.npmjs.com/package/acorn-walk} for parsing JavaScript files and traversing ASTs. We evaluate the approach by answering the following research questions.

\textbf{RQ1:} What are the precision and the recall of performing security scanning on Node.js applications based on call graphs constructed by JAM, compared to the \texttt{npm audit} approach that is based on package-level dependencies?

\textbf{RQ2:} What are the precision (measured by unique callees) and the recall (measured by comparing against dynamically created call graphs) for the call graphs constructed by JAM, and how do they compare to call graphs computed by the existing tool \texttt{js-callgraph}?

\textbf{RQ3:} How fast is the analysis? Is it faster if we, by taking advantage of the modularity of the application structure and the analysis, assume we have precomputed call graphs for the packages used by the applications?
Experimental Setup

To answer the research questions, we randomly selected 12 Node.js applications from the npm registry where *npm audit* reports one or more alarms (to get nontrivial data for RQ1). The benchmarks are listed in Table 12.1.

We run both *npm audit* and the JAM-based security scanner on each benchmark and manually classify the reported issues as true or false positives. Our security scanner is configured to use the same set of known library vulnerabilities as *npm audit*. As mentioned in Section 12.2, the security warnings generated by our approach provide reachability information at the level of functions, while the warnings from *npm audit* only contain coarse-grained information at the level of packages. For this experiment, we disregard this reachability information and only look at whether or not the given application is flagged as potentially affected by each of the known library vulnerabilities. We classify a security warning as a true positive if the vulnerable library function is reachable in some concrete execution of the application. (Note that reachability does not imply that the vulnerability is exploitable, which is a more subjective matter.) Since *npm audit* reports alarms for all the known library vulnerabilities in all transitive dependencies, a priori it has no false negatives, and the JAM-based security scanner by construction always reports the same or a subset of the issues reported by *npm audit*.

For RQ2, to measure recall relative to dynamically generated call graphs, we use NodeProf [134] and different inputs to the applications to cover a variety of combinations of their configuration options. Since it is difficult to obtain high dynamic coverage we measure precision of the call graphs independently of dynamic executions, as the percentage of call sites that have a unique callee according to the analysis (JAM or *js-callgraph*) as in previous work (e.g. [111, 133]). For these precision and recall measurements we disregard call edges that are not reachable from the entry, since only the reachable edges are relevant for security scanning. We run *js-callgraph* in “optimistic” mode (strategy DEMAND), which gives the best results.

To answer RQ3, for each application we first compute call graphs for each of its direct dependencies. From these call graphs, we compute an aggregated call graph of all the dependencies (see the paragraph on modular analysis in Section 12.5). Finally, we compute the call graph for the entire application using the aggregated call graph of the dependencies.

Our experiments have been run on a machine with a 2.9GHz Intel core i7 CPU with 10GB RAM for the analysis.

Results for RQ1 (Security Scanning Accuracy)

The results of the security scanning experiment are presented in Table 12.1, where “Functions” shows the total number of functions in the application and all its dependencies, and, in parentheses, the number of functions reachable from the application entry according to the call graph computed by JAM. “Modules” and “Packages” similarly show the numbers of modules and packages, the “npm audit” columns show
CHAPTER 12. MODULAR CALL GRAPH CONSTRUCTION FOR SECURITY SCANNING OF NODE.JS APPLICATIONS

Table 12.1: Experimental results for security scanning.

<table>
<thead>
<tr>
<th>Name</th>
<th>Functions</th>
<th>Modules</th>
<th>Packages</th>
<th>Time (modular)</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>npm audit</td>
<td>0.4.0</td>
<td>1.0.4</td>
<td>1.0.3</td>
<td>0.0.1</td>
<td>1.0.1</td>
<td>0.0.3</td>
</tr>
<tr>
<td>modules</td>
<td>4165(628)</td>
<td>1393(44)</td>
<td>13(13)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tools</td>
<td>6479(61)</td>
<td>1560(4)</td>
<td>25(1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>npm audit</td>
<td>8259(61)</td>
<td>783(3)</td>
<td>106(1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>modules</td>
<td>839(589)</td>
<td>85(79)</td>
<td>61(53)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>npm audit</td>
<td>898(357)</td>
<td>120(55)</td>
<td>41(35)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>modules</td>
<td>1557(143)</td>
<td>15(7)</td>
<td>4(4)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>modules</td>
<td>27703(3762)</td>
<td>1869(201)</td>
<td>93(55)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>modules</td>
<td>4334(1124)</td>
<td>261(124)</td>
<td>68(51)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>npm audit</td>
<td>1638(530)</td>
<td>206(31)</td>
<td>23(19)</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>modules</td>
<td>1228(732)</td>
<td>121(116)</td>
<td>64(56)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>modules</td>
<td>4777(1237)</td>
<td>187(90)</td>
<td>53(42)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>modules</td>
<td>6043(670)</td>
<td>1366(133)</td>
<td>69(37)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>69320(9894)</td>
<td>8026(887)</td>
<td>620(367)</td>
<td>8</td>
<td>25</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 12.2: Experimental results for call graph construction.

<table>
<thead>
<tr>
<th>Name</th>
<th>Precision</th>
<th>Recall</th>
<th>Time (full)</th>
<th>Time (modular)</th>
<th>JAM</th>
<th>js-callgraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>npm audit</td>
<td>0.4.0</td>
<td>1.0.4</td>
<td>1.0.3</td>
<td>0.0.1</td>
<td>0.0.1</td>
<td>0.0.3</td>
</tr>
<tr>
<td>modules</td>
<td>86.05%</td>
<td>100.00%</td>
<td>2.04s</td>
<td>0.07s</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>tools</td>
<td>92.08%</td>
<td>100.00%</td>
<td>0.69s</td>
<td>0.01s</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>npm audit</td>
<td>92.79%</td>
<td>100.00%</td>
<td>0.73s</td>
<td>0.01s</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>modules</td>
<td>87.42%</td>
<td>98.95%</td>
<td>1.39s</td>
<td>0.04s</td>
<td>80.00%</td>
<td>1.39s</td>
</tr>
<tr>
<td>npm audit</td>
<td>82.45%</td>
<td>94.78%</td>
<td>0.99s</td>
<td>0.04s</td>
<td>43.86%</td>
<td>85.32%</td>
</tr>
<tr>
<td>modules</td>
<td>70.65%</td>
<td>100.00%</td>
<td>0.83s</td>
<td>0.08s</td>
<td>–</td>
<td>0.00%</td>
</tr>
<tr>
<td>npm audit</td>
<td>71.43%</td>
<td>98.18%</td>
<td>23.01s</td>
<td>0.70s</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>modules</td>
<td>89.14%</td>
<td>99.41%</td>
<td>2.37s</td>
<td>0.27s</td>
<td>68.64%</td>
<td>62.24%</td>
</tr>
<tr>
<td>npm audit</td>
<td>97.42%</td>
<td>100.00%</td>
<td>1.80s</td>
<td>0.08s</td>
<td>59.81%</td>
<td>56.55%</td>
</tr>
<tr>
<td>modules</td>
<td>80.80%</td>
<td>96.51%</td>
<td>1.65s</td>
<td>0.09s</td>
<td>66.20%</td>
<td>63.53%</td>
</tr>
<tr>
<td>npm audit</td>
<td>86.07%</td>
<td>100.00%</td>
<td>2.56s</td>
<td>0.18s</td>
<td>52.14%</td>
<td>64.20%</td>
</tr>
<tr>
<td>modules</td>
<td>75.85%</td>
<td>95.59%</td>
<td>2.63s</td>
<td>0.11s</td>
<td>40.08%</td>
<td>52.04%</td>
</tr>
<tr>
<td>Average</td>
<td>84.35%</td>
<td>98.62%</td>
<td>3.39s</td>
<td>0.14a</td>
<td>58.64%</td>
<td>48.16%</td>
</tr>
</tbody>
</table>

the number of security alarms reported by npm audit security scanner, and the “JAM” columns show the alarms reported by JAM. The alarms are categorized into alarms about actual usage of a vulnerable library function (true positives, “TP”), alarms about a vulnerable library function that is never used by the application (false positives, “FP”), and usages of a vulnerable library function where no alarm was raised (false negatives, “FN”).

We have manually classified the alarms by npm audit into true and false positives. As can be seen in Table 12.1 npm audit reports 34 alarms for the 12 benchmarks, where only 8 are true positives and 26 are false positives, yielding a precision of only 24%.

The JAM security scanner found all 8 vulnerabilities, resulting in a perfect 100% recall of the security warnings. For all the 7 applications where npm audit reports false
positives, the call-graph-based security scanner reduces the number of false positives. For 5 of them, the call-graph-based security scanner even manages to remove all the false positives. In total, the call-graph-based security scanner reduced the number of false positives by 81% compared to *npm audit*, which means that the precision of the JAM security scanner is 61% compared to the 24% precision of *npm audit*.

The 5 false positives are caused by vulnerabilities in the *lodash* library. The reason for these false positives is not that the computed call graphs have too many edges, but that the vulnerable library function, which is not used by the applications, is mixed together with a function that is being used by the applications. This happens because those two functions are defined in the library via a higher-order function and originate from the same function definition, and they differ only because of their free variables. Since JAM uses the function definition source locations to identify the functions, it does not distinguish between the two functions. Improving this aspect is an interesting opportunity for future work.

Although JAM has no false negatives in the experiments, it is possible that it may miss some call edges, as discussed in Section 12.5. We have manually inspected the module connectivity in the call graphs for the three benchmarks with fewer than 10 reachable modules, and we find no inter-module edges missing. Also, we have checked for all the benchmarks that all modules that are being loaded in a concrete execution are reachable in the call graphs.

Naturally, any vulnerabilities that may exist in unreachable parts of the application code cannot affect the behavior of the applications. The applications altogether contain 69320 functions, 8026 modules, and 620 packages (including duplicates used by several applications). According to the computed call graphs, only 9894 (14%) of the functions, 887 (11%) of the modules, and 367 (59%) of the packages are reachable, which gives an indication of the overall potential of call-graph-based security scanning.

**Results for RQ2 (Call Graph Accuracy)**

The results of the call graph precision and recall measurements are shown in Table 12.2. JAM finds that on average 84.35% of the call sites have a unique callee, compared to only 58.64% for *js-callgraph*. Also, 98.62% of the call edges observed in the concrete executions are detected by JAM, while the corresponding result for *js-callgraph* is only 48.16%.

Moreover, *js-callgraph* fails on 5 of the applications, either crashing with out-of-memory or producing a call graph with no nodes reachable from the entry (both indicated by ‘–’). The few missing edges in the JAM results are triggered by some rare cases where the soundness assumptions do not hold (see Section 12.5).

These results suggest that the call graphs produced by JAM are substantially more accurate than those produced by *js-callgraph*. Even though the recall for JAM is not

---

22We have excluded the package *esprima* from the recall measurements because it has been bundled using webpack. The low recall for *nodetree* and *ragan-module* with *js-callgraph* is due to limitations in its parser and lack of support for getters.
perfect, a few false negatives is likely preferable to a large number of false positives or a significantly slower analysis.

**Results for RQ3 (Analysis Time and Modularity)**

The “**Time (full)**” columns in Table 12.2 shows the time it takes JAM (and js-callgraph) to compute the call graphs for the applications including all dependencies. The analysis time for JAM varies from less than one second for the toucht application to around 23 seconds for jwtnoneify, and js-callgraph is orders of magnitude slower.

The relatively large time for jwtnoneify is explained by a heavy usage of, for example, the forEach function from the lodash library. The forEach function is a higher-order function that takes a collection (typically an array) and some iterator function that is called with each element in the collection as an argument. Because the rule param-call from Figure 12.3 merges arguments from all call sites when a parameter is called, a massive amount of new entries are added to the α relation. This behavior is similar to what happens in a context-insensitive dataflow analysis. Perhaps surprisingly, despite the longer analysis time and the less precise call graph, the security scanner is still more precise than npm audit for this application. Nevertheless, investigating this outlier in more detail and improving its analysis time is an interesting challenge for future work.

The “**Time (modular)**” column in Table 12.1 shows the analysis time, when the call graphs for all direct dependencies of the application have been precomputed, which is a realistic situation in a scenario where many applications that share dependencies are being analyzed. The time includes aggregating the call graphs from the dependencies and computing the call graph for the entire application. The call graph construction is very efficient taking less than a second for all applications.

We conclude that the JAM full call graph analysis is highly efficient for most benchmarks. Furthermore, the modular approach ensures that all benchmarks are analyzed even faster, which is promising for, for example, IDE integration.

### 12.8 Related work

As discussed in the introduction, multiple studies show how JavaScript libraries are being used extensively, and how security vulnerabilities in such libraries cause serious problems for the applications [31, 32, 74, 84, 130, 140, 142, 144, 147]. In particular, Zapata et al. [142] conclude that security scanning based on package dependencies considerably overestimates the implications of security vulnerabilities in libraries, and they suggest that many false positives may be avoided by performing analysis at the function level, however, they do not present such an analysis.

The dynamic call graph generators by Herczeg et al. [60] and Hejderup et al. [59] have been developed for JavaScript security scanning and function-level dependency management, but unlike our approach they require high-coverage test suites to avoid missing security issues.
Präzi [58] is an approach to reason about package dependencies that resembles JAM by relying on statically computed call graphs, but it is developed for Rust, not JavaScript. Eclipse Steady [115] is a similar approach for Java. It has recently been adapted to JavaScript [22], however, that work uses a simple program analysis that ignores most JavaScript language constructs. Mininode [77] is a tool for reducing the attack surface of Node.js applications by removing unused code. It includes a form of call graph construction, but it is unclear how it works for the dynamic features of JavaScript.

Multiple static analyzers already exist for JavaScript [43, 63, 66, 85, 92, 93]. Although they can in principle produce call graphs, none of these analyzers have been designed for the modular structure and heavy reuse of libraries in Node.js applications. Moreover, the light-weight static analyzers (e.g., [43, 92, 93]) are fast but tend to miss many call edges, whereas abstract-interpretation-based analyzers (e.g., TAJS [66] and SAFE [85]) do not yet scale to real-world Node.js applications. The study by Antal et al. [7] compares different static call graph construction tools for JavaScript, with very limited success for Node.js applications.

As explained in Section 12.5, a key component of JAM is the field-based analysis inspired by Feldthaus et al. [43], extended with access paths [96, 101] to enable modular reasoning. The experimental results with JAM and js-callgraph demonstrate the advantages of the modular analysis. The modular approach of JAM is inspired by componential analysis [44], which also as a first step computes summaries for modules (Scheme program components) and then combines the summaries to obtain the analysis result for the full program. As discussed in Section 12.5, JAM is designed to reach a useful compromise between precision, recall, and efficiency [91]. Although it is theoretically unsound, no security issues are missed in the experiments described in Section 12.7.

Another approach to security scanning is taint analysis, which not only considers the call graph but also the dataflow, and can thereby in principle safely dismiss some security warnings as harmless. The Nodest analyzer [105] extends TAJS with taint analysis and circumvents the scalability problem by attempting to avoid analyzing irrelevant modules, but still is orders of magnitude slower than JAM. Staicu et al. [132] also discuss the problem with package-dependency-level security scanning and propose a dynamic analysis to infer taint summaries for libraries. Such taint summaries can be used with, for example, the static analyzer LGTM [23] which is designed to minimize the amount of false positives, at the cost of missing true positives, in contrast to JAM.

Change impact analysis is closely related to security scanning. Existing change impact analysis tools for JavaScript [2, 55] are designed for browser-based applications, not for reasoning about dependencies between modules in Node.js applications.

23https://lgtm.com/
12.9 Conclusion

We have presented JAM, a modular call graph construction analysis that scales for Node.js applications, and we have shown how the produced call graphs can be used for security scanning. Due to JAM’s modular design, call graphs can be computed for libraries and reused when computing call graphs for an application, and thereby scale for applications with complex dependencies.

JAM is designed to strike a balance between analysis precision (not producing an overwhelming amount of spurious call edges), recall (detecting almost all call edges that appear in concrete executions), and efficiency (analyzing real-world applications with thousands of functions within seconds).

We have shown experimentally on 12 Node.js applications that security scanning on the call graphs produced by JAM reports all true positive security warnings and reduces the number of false positives by 81% compared to the package-based security scanner *npm audit*. The analysis time for JAM using the modular approach is less than a second on average for our benchmarks, indicating that JAM is practically useful. Future work involves exploring more uses of the call graphs, for instance, change impact analysis as well as code navigation and refactoring in IDEs.
Bibliography


