Static Analysis for Node.js

Benjamin Barslev Nielsen

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Static Analysis for Node.js

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

by
Benjamin Barslev Nielsen
December 2, 2020

Abstract

JavaScript is everywhere. With the introduction of Node.js in 2009, JavaScript can be used for implementing web servers and desktop applications, and it has become one of the most widely used languages. The dynamic nature of JavaScript is a challenge for static analysis. When statically analyzing Node.js programs, we additionally face challenges due to the fact that Node.js applications typically depend on many third-party packages. Unfortunately, previous sound static analyses for JavaScript fail to analyze some of the most popular packages in Node.js.

Security vulnerabilities are often reported for Node.js packages. This is critical, since a Node.js application has direct access to the underlying file-system and operating system resources. This means that if an attacker can exploit an injection vulnerability, then the attacker might gain full control of the underlying machine. Static analysis can be used to detect such injection vulnerabilities, but unfortunately no existing sound static analyses scales to large Node.js applications. A typical Node.js application is quite large, since it depends on many third-party packages. Package-level security scanners exist that report if an application uses a dependency that has a vulnerability. These security scanners have many false positives, since they only look at the dependency tree, without considering the application code. To fix a security vulnerability in an application, the solution sometimes is to update to a newer version of the vulnerable dependency. However, performing such updates are not trivial. Updating a dependency might result in the application no longer working, due to breaking changes in the dependency update. Currently, understanding whether an update contains breaking changes, whether the application is affected by them, and if affected, then how to patch the application, is a completely manual, time-consuming, and error-prone task.

This thesis presents new sound static analysis techniques that can analyze the most popular packages in Node.js, and a technique for scaling a static analysis that detects injection vulnerabilities in Node.js applications. For aiding developers in updating libraries with breaking changes, we present a tool that combines static analysis and user interaction for automatically patching the code to become compatible with the new version. To address the problem with too many false positives in package-level security scanners, we develop a modular call graph analysis, and a security scanner that only reports a potential vulnerability if the vulnerable part of the library API is reachable according to the call graph.

Resumé

JavaScript er overalt. Med introduktionen af Node.js i 2009 kan web servere og desktop programmer blive programmeret i JavaScript, og i dag er JavaScript et af de mest brugte programmeringssprog. JavaScript er svært at analysere statisk pga. dets dynamiske features. Statisk analyse af Node.js programmer har endnu flere udfordringer, da Node.js programmer som regel afhænger af mange third-party pakker. Eksisterende sunde statiske analyser for JavaScript fejler desværre for nogle af de mest populære pakker i Node.js.

Der bliver ofte rapporteret sikkerhedssårbarheder i Node.js pakker. Dette er kritisk, da et Node.js program har direkte adgang til det underliggende fil-system og operativ systemets resurser. Det betyder at, hvis en hacker kan udnytte en injektionssårbarhed, så kan hackeren muligvis få fuld kontrol over den eksekverende maskine. Statisk analyse kan opdage sådanne sikkerhedssårbarheder, men desværre er der ingen sunde statiske analyser som skalere til store Node.js programmer. Et almindeligt Node.js program er temmelig stor, da den afhænger af mange third-party pakker. Pakke-level security scanners eksisterer, og de rapporterer, hvis et program afhænger af en pakke som har en kendt sårbarhed. Disse security scanners har mange falske rapporter, da de kigger på pakkerne, som et program afhænger af, uden at kigge på hvordan de bliver brugt. En sikkerhedssårbarhed kan nogle gange fikses ved at opdatere en af de pakker den bruger. At opdatere en sådan pakke er dog ikke trivielt. Opdatering af en pakke kan få programmet til ikke længere at virke pga. breaking changes i opdateringen. En opdatering består i at udvikleren skal undersøge om opdateringen har breaking changes, om programmet er påvirket af dem, og hvordan programmet skal fikses, hvis den er påvirket af breaking changes. I dag er dette en manuel, tidskrævende og fejlbarlig opgave.

Denne afhandling præsenterer nye sunde statisk analyse teknikker som kan analysere de mest populære pakker i Node.js, samt en teknik der kan skalere en statisk analyse til at finde injektionssårbarheder i Node.js programmer. For at hjælpe udviklere med at opdatere pakker med breaking changes, præsenterer vi et værktøj som kombinerer statisk analyse og brugerinteraktion for automatisk at fikse et program til at være kompatibel med den nye version af en pakke. Vi adresserer problemet med mange falske rapporter i pakke-level security scanners ved at udvikle en modulær call graph analyse, og en security scanner som kun rapporterer en sårbarhed, hvis den sårbare funktion i en pakke kan nås gennem den genererede call graph.

Acknowledgments

I would like to express my thanks to the following people for their support and guidance throughout my PhD.

Anders Møller, my advisor, for introducing me to static analysis, and for showing me that static analysis for JavaScript is a fun and difficult challenge. I'm very grateful for our discussions as well as the opportunities and experiences I have had during my PhD. Esben Andreasen, for providing guidance with respect to both technical challenges and life as a PhD student in the first years of my PhD. You made the transition into a PhD student much easier. My colleagues at Oracle in Brisbane, and especially my advisor Behnaz Hassanshahi, for making the six months internship a memory for a lifetime. Benno Stein and Bor-Yuh Evan Chang for the collaboration on one of my first research projects. Martin Toldam Torp, for the close collaboration in the last year of my PhD. Working from home during the initial corona times would have been much harder without your collaboration. I would also thank my colleagues in the Programming Languages and Logics and Semantics groups for providing a good work environment. Finally, I would like to thank Freja and my parents for their love and support both before and during my PhD studies.

Benjamin Barslev Nielsen,
Aarhus, December 2, 2020.

Contents

Abstract **i**

Resumé **iii**

Acknowledgments **v**

Contents **vii**

I Overview **1**

1 Introduction **3**
1.1 Challenges 6
1.2 Methodology 7
1.3 Contributions 8
1.4 Outline 11

2 Background **13**
2.1 JavaScript 13
2.1.1 Runtime Types 14
2.1.2 Dynamic Object Structure 15
2.1.3 Prototype-Based Inheritance 16
2.1.4 Higher-Order Functions and Closures 18
2.1.5 Implicit Calls through Type Coercions and Getters/Setters . 20
2.2 The Node.js Ecosystem 21
2.3 Sound Static Program Analysis 24
2.3.1 Lattices 25
2.3.2 Monotone Frameworks 27

3 Sound Static Analysis Techniques for JavaScript and Node.js **31**
3.1 Existing Work 31
3.2 TAJS 34
3.2.1 Program Representation 34
3.2.2 Analysis Lattice 34

3.2.3 Transfer Functions . . . 38
3.2.4 Context Sensitivity . . . 42
3.3 Contributions . . . 43
3.3.1 Understanding Analysis Limitations . . . 44
3.3.2 Increasing Analysis Precision for FCT Patterns . . . 47
3.3.3 Improving Scalability of Taint Analysis for Node.js Applications 51

4 Lightweight Static Analysis Techniques for JavaScript and Node.js 55
4.1 Existing Work . . . 55
4.2 Updating Dependencies with Breaking Changes . . . 58
4.3 Function-Level Security Scanning . . . 61

5 Conclusion 69

II Publications 73

6 Systematic Approaches for Increasing Soundness and Precision of Static Analyzers 75
6.1 Introduction . . . 75
6.2 Basic Techniques . . . 77
6.2.1 Delta Debugging . . . 77
6.2.2 Soundness Testing . . . 79
6.2.3 Blended Analysis . . . 81
6.3 Combining the Basic Techniques . . . 82
6.3.1 Soundness Testing and Delta Debugging . . . 83
6.3.2 Soundness Testing and Blended Analysis . . . 83
6.3.3 Delta Debugging and Blended Analysis . . . 84
6.3.4 Combining All Three Techniques . . . 85
6.4 Conclusion . . . 86

7 Static Analysis with Demand-Driven Value Refinement 87
7.1 Introduction . . . 88
7.2 Motivating Example . . . 89
7.3 A Simple Dynamic Language and Dataflow Analysis . . . 92
7.3.1 A Simple Dynamic Language . . . 93
7.3.2 Dataflow Analysis . . . 94
7.4 Demand-Driven Value Refinement . . . 96
7.4.1 Value Refinement . . . 96
7.4.2 Using Value Refinement in Dataflow Analysis . . . 97
7.4.3 Using the Base Analysis During Refinements . . . 98
7.5 Backwards Abstract Interpretation for Value Refinement . . . 100
7.5.1 Abstract Domain . . . 101
7.5.2 Backwards Abstract Interpretation . . . 102

7.5.3 Integration of Base Analysis State 104
7.6 Instantiation for JavaScript 105
7.6.1 A Value Refiner for JavaScript 105
7.6.2 Functions with Free Variables 106
7.6.3 Performance Improvements 107
7.7 Evaluation 107
7.7.1 Comparison with State-of-the-Art Analyzers 107
7.7.2 Understanding the Effectiveness of the Value Refiner 111
7.8 Related Work 113
7.9 Conclusion 115
7.10 Soundness of Demand-Driven Value Refinement 116
7.11 Soundness of the Value Refiner $R^{\leftarrow}$ 117
7.12 Supplementary Experimental Data 117

8 Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript 121
8.1 Introduction 121
8.2 Motivating Example and Overview 124
8.3 Background: The TAJS Analyzer 128
8.4 Value Partitioning 130
8.5 Three Instantiations of Value Partitioning 132
8.5.1 Property Name Partitioning 132
8.5.2 Free Variable Partitioning 135
8.5.3 Type Partitioning 140
8.6 Evaluation 143
8.6.1 RQ1: Comparison with State-Of-The-Art Analyses 143
8.6.2 RQ2: Effects of the Three Instantiations 147
8.7 Related Work 148
8.8 Conclusion 150

9 NODEST: Feedback-Driven Static Analysis of Node.js Applications 151
9.1 Introduction 152
9.2 Motivating Example 153
9.3 Background: static analysis in TAJS 154
9.4 Feedback-driven Analysis 155
9.4.1 Extending the Static Analysis in TAJS 157
9.4.2 Post-processing Analysis Results 158
9.4.3 Optimized Worklist Algorithm 161
9.5 Static Taint Analysis 162
9.5.1 Incorporating Taint Analysis 162
9.5.2 Taint Analysis Configuration 163
9.5.3 Revisiting the Motivating Example 163
9.6 Evaluation 163
9.6.1 RQ1 - Precision 164

9.6.2 RQ2 - Scalability and Accuracy 164
9.6.3 RQ3 - Optimized Worklist Algorithm 168
9.7 Case Studies 169
9.7.1 Mqtt-growl 169
9.7.2 Codem-transcode 170
9.8 Related Work 171
9.9 Conclusion 172

10 Detecting Locations in JavaScript Programs Affected by Breaking Library Changes 175
10.1 Introduction 175
10.2 Motivating Example 179
10.3 Preliminary study 181
10.4 The TAPIR Approach 183
10.5 A Pattern Language for Describing API Access Points 184
10.6 A Static Analysis for Finding Uses of API Access Points in Client Code 188
10.6.1 Phase 1: Alias Analysis 188
10.6.2 Phase 2: Access Path Inference 190
10.6.3 Phase 3: Pattern Matching 193
10.6.4 Reporting Analysis Confidence 194
10.7 Evaluation 196
10.7.1 RQ1 (Expressiveness) 197
10.7.2 RQ2 (Recall, Precision, Confidence, and Performance) . . . 200
10.8 Related Work 202
10.9 Conclusion 204

11 Semantic Patches for Adaptation of JavaScript Programs to Evolving Libraries 205
11.1 Introduction 206
11.2 Motivating Example 207
11.3 Overview 209
11.4 A Semantic Patch Language 210
11.5 Interactive Phase 215
11.6 Evaluation 217
11.6.1 RQ1 (Expressiveness of transformations) 218
11.6.2 RQ2 (Correctness and quality of transformations) 220
11.6.3 RQ3 (Questions asked by JSFIX) 223
11.7 Related Work 224
11.8 Conclusion 225
11.9 Additional Examples 226

12 Modular Call Graph Construction for Security Scanning of Node.js Applications 231
12.1 Introduction 231

12.2 Motivating Example 233
12.3 Key Challenges 235
12.4 Module Summary Construction 236
12.5 Call Graph Construction 240
12.6 Security Scanning using Call Graphs 245
12.7 Evaluation 247
12.8 Related work 250
12.9 Conclusion 251

Bibliography **253**

Part I

Overview

Chapter 1

Introduction

In the digital world of today, software is everywhere and the demand for software developers is increasing. To meet this higher demand it is important to increase the productivity of the developers. Developers typically write software using IDEs (Integrated Development Environments) that provide tool support for helping the developer write code, for instance, by reporting potential bugs and supporting code-navigation, auto-completion, and automatic refactoring. The tool support directly affects how productive a developer is. This means that we can improve developer productivity by improving the tool support.

Developers can write software in many different programming languages, and the tool support varies significantly between programming languages. For statically typed languages, i.e., languages where a program needs to type-check before it can run, tool support is typically quite good, since many details of the code are easy to reason about automatically without running the program. On the other hand, for dynamically typed languages, i.e., languages where any syntactically valid program is executable, the tool support is limited, since it is hard to automatically reason about the program without running it. This may lead to low productivity for developers of dynamically typed languages, because they do not have proper code-navigation and code-completion, and they need to run their programs to find even simple type-related bugs. These bugs would have been caught immediately in a statically typed language. Despite this shortcoming of dynamically typed languages, they are widely used due to other limitations of statically typed languages. These limitations include having to write verbose type definitions and supply hints that are needed for the type checker to suppress a warning about an error, where the developer knows that an error cannot occur.

There has also been a lot of work on optional typing that tries to combine the benefits from statically typed languages and dynamically typed languages, by letting the developer provide static types for parts of an application. The typed parts will be type-checked, while untyped parts are unchecked as long as the types cannot be automatically inferred. Optionally typed languages therefore give the developer an opportunity to find a sweet spot between the advantages and disadvantages of statically and dynamically typed languages. However, much code is still written in purely dynamically typed languages.

This thesis focuses on dynamically typed languages and develops techniques for automatically reasoning about the widely used dynamic language JavaScript. In the future, these techniques might be used as tool support to increase the productivity of JavaScript developers. More specifically, the work focuses on Node.js, which is a JavaScript runtime that allows web servers and stand-alone applications to be written in JavaScript. Node.js has a package manager, called npm, with more than 1.4 million packages[1], which is significantly more packages than any other package manager for any other language. On average, a Node.js application depends on more than 50 direct and transitive packages, and 90% of the application code comes from dependencies. This means that even applications with a small code-base might be quite large when considering dependencies. Automatic reasoning techniques therefore have to scale well to be able to reason about Node.js applications. Due to the heavy use of dependencies, Node.js developers also face the challenge of keeping their dependencies up-to-date. The dependencies are projects on their own that are continuously updated, and sometimes the updates include breaking changes. If an update has breaking changes, it means that an application that was working correctly with the prior version of the dependency might not work with the newer version. Currently, this work of updating a package is a completely manual process consisting of three steps. The first step is to understand an informal changelog that describes the breaking changes in a library update. The second step is to find the locations in the application code that are affected by the breaking changes. The last step is to manually patch the code to become compatible with the new version of the library. Each of these three steps can be time-consuming and error-prone, so tools for automating this approach are desired.

Another challenge when developing Node.js applications is how to make sure that the application is secure. This can be difficult, since vulnerabilities might exist in some of the dependencies and not the application code itself. For this reason, there are public databases with known vulnerabilities, for instance, the npm advisory.[2] The npm audit security scanner uses the npm advisory to warn developers when their application depends on a package that has a known security vulnerability. However, such a security scanner gives many false positives, since the vulnerability might be in a part of the dependency that the application does not actually use. For this reason, developers can easily be overwhelmed by the warnings from the security scanner and miss actual vulnerabilities, because the developers assume the reports are all false positives.

We have now presented some of the challenges Node.js developers face. In this thesis, we will address these challenges by developing tools for automatic reasoning about Node.js programs.

Automatic reasoning about programs can be carried out by program analyses,

[1]http://www.modulecounts.com/

[2]https://www.npmjs.com/advisories

which themselves are programs that take other programs as input, and output some information about the input program. For instance, a program analysis can try to find errors that exist in the program or other useful facts that can guide developers. Program analyses are typically either static or dynamic. Static analysis reasons about the program without executing it, and dynamic analysis executes the program and typically works by either instrumenting the executed program or by modifying the runtime environment. The focus of this thesis is on static analysis, since static analysis in principle can overapproximate the behaviors of all possible executions of a program and thereby provide guarantees that, for instance, a program is free of a certain class of bugs. In comparison, dynamic analysis can only reason about the concrete executions it has observed, which means that it cannot provide such guarantees.

A static analysis is *sound* when it overapproximates all possible executions. Unfortunately, due to Rice's theorem it is not possible to be both sound and complete, which means that a sound analysis sometimes have to report bugs that are not present in the program. We refer to such reports as false positives. A challenge when designing a static analysis therefore is to avoid too many false positives, because it is counterproductive if developers waste a lot of time on spurious bug reports. To avoid false positives, some static analyses for dynamic languages have been designed to not overapproximate all the behaviors of the program. Instead they aim at reporting few false positives while still producing useful results. In this thesis, we will refer to these analyses as *practical* static analyses, since they aim for practicality rather than providing firm guarantees. This thesis presents new sound static analysis techniques for an existing analysis and also designs some new practical static analyses.

Security vulnerabilities have always been an issue in client-side JavaScript, but on the server-side with Node.js, security bugs are much more severe. This is due to JavaScript programs in browsers typically are being executed in a sandbox, whereas in Node.js the application has direct access to the underlying filesystem and operating system resources. Static analysis can be used for detecting such security vulnerabilities through taint analysis that checks whether unsafe input (a source) can end up in a security critical method (a sink). An example of a security critical method is the `eval` function in JavaScript, which allows arbitrary code execution, meaning that if there is a taint flow to that sink, then an attacker can inject arbitrary code to be executed.

Another challenge when designing a static analysis is that it has to be efficient to analyze real-world applications. Typically, static analyses can be designed as a trade-off between the amount of false positives and the analysis time, however, for JavaScript it is different. The imprecision in JavaScript analysis that results in false positives often also results in the analysis wasting a lot of time on analyzing spurious dataflow, i.e., analyzing executions that are infeasible in practice. In the design of a JavaScript analysis, we have to carefully design the abstractions to make the analysis both precise and efficient.

We have now introduced some of the challenges that developers of JavaScript and Node.js face, and hinted that static analysis might be able to help developers facing these challenges. In the following, we will summarize these challenges, and explain how this thesis contributes to solving the challenges through static analysis.

1.1 Challenges

The overall statement of this thesis is: *Despite the highly dynamic features of JavaScript and Node.js, it is possible to develop static analysis techniques to accurately infer useful facts for Node.js applications. The useful facts are type-related information for detecting type errors, taint information for detecting injection vulnerabilities, call-graph information for security scanning and code-navigation, and library API usages for aiding in updating dependencies with breaking changes.* To support the thesis statement, we consider the following five challenges:

C1 **How to design and implement sound static analyzers?** Designing analyses for JavaScript requires finding a delicate balance in the abstractions, since too imprecise abstractions lead to an abundance of spurious dataflow making the analysis results useless, while too fine-grained abstractions make the analysis too costly. As analysis designer we therefore need to know what precision improvements are necessary to analyze a program precisely, such that we can design new techniques that only provide the precision necessary without increasing the analysis cost unnecessarily. Manually finding what precision improvements are needed is a time-consuming task, so how can we automatically aid the analysis designer in understanding what precision improvements are needed? Due to the complexity of the JavaScript language, the complexity of the analysis implementation is also high, and subtle soundness bugs are likely to be present in the implementation. How do we automatically detect and understand such soundness bugs, such that we can gain more confidence in the soundness of the analysis results?

C2 **How to precisely analyze correlated read/write pairs?** JavaScript supports dynamic property reads (`o[p1]`) and dynamic property writes (`o[p2] = v`), where the property names `p1` and `p2` are dynamically computed strings. These instructions can cause precision loss if the analysis has imprecise information for the property name. The precision loss is especially critical for correlated read-/write pairs, i.e., where the value to be written comes from a dynamic property read, and the property name of the write depends on the property name of the read. Such a precision loss basically results in the analysis mixing together all properties of one object in all properties of the other object. How do we make an efficient analysis that can precisely reason about such code when the property name is inevitably imprecise and the correlated read/write pairs can span multiple functions?

C3 **How to scale taint analysis to Node.js applications?** Node.js applications typically depend on many third-party libraries making even simple applications contain much code. Current state-of-the-art static analyzers for JavaScript/Node.js do not scale to applications of that size and therefore they cannot easily be extended to perform static taint analysis for Node.js applications. How can the scalability of such static analyzers be improved to perform static taint analysis for Node.js applications?

C4 **How to aid Node.js developers in updating dependencies with breaking changes?** Node.js developers have to update dependencies frequently, and these might include breaking changes, so blindly updating a dependency might break the application. This means that developers have to investigate what has been changed, which is done by reading a changelog provided with the update of the dependency. This process is very time consuming since developers first need to comprehend the changelog, then figure out whether the application is actually affected by any of the breaking changes, and if so, what locations are affected? How can we design an analysis that checks whether an application is affected by breaking changes, and how can we find the affected locations in the source code? Next, the developer has to understand how to patch the code at the affected locations, which might not be an easy task. Is it possible to automatically patch the affected locations?

C5 **How to develop a security scanner with high precision?** Currently, the best tools for informing developers about potential security vulnerabilities are package-level security scanners. They use a database of vulnerabilities, and report a vulnerability for an application, if the application depends on a package in a version that has a known vulnerability. This results in a lot of false positives, since applications often use only a small part of a dependency's API, and vulnerabilities are reported, even though the application does not use the vulnerable part of the API. How can we create a more fine-grained security scanner that aims at only reporting the vulnerabilities when the vulnerable part of the API is actually used?

1.2 Methodology

For each of the challenges discussed above, we have applied the same overall research methodology. The scope was narrowed down to a concrete problem, for which we could develop a tool, and evaluate whether our proposed solution solves the problem for real-world Node.js code. The overall methodology in designing that solution has been to understand the challenges that our solution should handle, then to develop an initial idea, and convince ourselves that it works conceptually. When the idea works conceptually, the corresponding analysis is implemented and tested on suitable real-world Node.js benchmarks. We evaluate our technique on the benchmarks, and based on the results, we iteratively refine our technique until it provides satisfying results. Each of the listed challenges has thereby resulted in a prototype tool and an empirical evaluation.

1.3 Contributions

The contributions of this thesis are improvements on state of the art solutions for the five challenges mentioned above. The following lists the research papers that present our solutions, followed by a summary of the main contributions that they present for each of the five challenges. The papers that have been published are included in Part II of this thesis. The papers are identical to the published ones, except for minor adjustments to the layout.

P1 *Systematic Approaches for Increasing Soundness and Precision of Static Analyzers*
Esben Sparre Andreasen (Aarhus University, Denmark), Anders Møller (Aarhus University, Denmark), and Benjamin Barslev Nielsen (Aarhus University, Denmark). Published in the Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP), June 2017. Included in Chapter 6.

P2 *Static Analysis with Demand-Driven Value Refinement*
Benno Stein (University of Colorado Boulder, USA), Benjamin Barslev Nielsen (Aarhus University, Denmark), Bor-Yuh Evan Chang (University of Colorado Boulder, USA), and Anders Møller (Aarhus University, Denmark). Published in the Proceedings of the ACM on Programming Languages, Volume 3, Issue OOPSLA, October 2019. Included in Chapter 7.

P3 *Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript*
Benjamin Barslev Nielsen (Aarhus University, Denmark) and Anders Møller (Aarhus University, Denmark). Published in the Proceedings of the 34th European Conference on Object-Oriented Programming, ECOOP 2020, November 2020. Included in Chapter 8.

P4 NODEST*: Feedback-Driven Static Analysis of Node.js Applications*
Benjamin Barslev Nielsen (Oracle Labs, Australia and Aarhus University, Denmark), Behnaz Hassanshahi (Oracle Labs, Australia), and François Gauthier (Oracle Labs, Australia). Published in the Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), August 2019. Included in Chapter 9.

P5 *Detecting Locations in JavaScript Programs Affected by Breaking Library Changes*
Anders Møller (Aarhus University, Denmark), Benjamin Barslev Nielsen (Aarhus University, Denmark), and Martin Toldam Torp (Aarhus University, Denmark). Published in the Proceedings of the ACM on Programming Languages, Volume 4, Issue OOPSLA, November 2020. Included in Chapter 10.

Furthermore, this thesis also includes the following two papers, which are under review at the time of writing:

P6 *Semantic Patches for Adaptation of JavaScript Programs to Evolving Libraries*
Benjamin Barslev Nielsen (Aarhus University, Denmark), Martin Toldam Torp (Aarhus University, Denmark), and Anders Møller (Aarhus University, Denmark). Included in Chapter 11.

P7 *Modular Call Graph Construction for Security Scanning of Node.js Applications*
Benjamin Barslev Nielsen (Aarhus University, Denmark), Martin Toldam Torp (Aarhus University, Denmark), and Anders Møller (Aarhus University, Denmark). Included in Chapter 12.

The above papers improve state of the art for the five challenges mentioned above in the following ways:

C1 **How to design and implement sound static analyzers?** We present systematic approaches for testing an analysis for soundness bugs, and for finding root causes of precision loss by leveraging delta debugging, dynamic analysis and blended analysis (P1, Chapter 6, [11]). Using these approaches we find the root causes of precision losses that prohibit a current static analyzer in analyzing the most popular npm libraries *lodash* and *underscore*.

C2 **How to precisely analyze correlated read/write pairs?** We present two sound analysis techniques that can precisely analyze correlated read/write pairs. The first technique is *demand-driven value refinement* that extends an ordinary analysis with a backwards value refinement mechanism. When critical precision loss is about to be introduced at a dynamic property write, the property name to write to is refined relatively to the value being written. At the imprecise dynamic property write, the ordinary analysis issues queries to a goal-directed backwards analysis. If the queries are answered precisely, the ordinary analysis can avoid critical precision loss at the dynamic property write. Our evaluation shows that using this technique, we can analyze the most popular npm libraries, *lodash* and *underscore*, with useful results (P2, Chapter 7, [137]). The second technique, *value partitioning*, is also sound but far more practical as it does not require a separate backwards analysis. The technique can efficiently and precisely reason about relational program properties. Our evaluation shows that this simpler approach outperforms the demand-driven analysis technique when analyzing, for instance, *lodash* and *underscore* (P3, Chapter 8, [108]).

C3 **How to scale taint analysis to Node.js applications?** We propose a static analysis technique for Node.js applications that can make a taint analysis scale to larger Node.js applications. The technique scales through only analyzing the packages that are relevant for detecting taint flows. It heuristically detects which dependencies should be analyzed to avoid missing taint flows, while attempting to overapproximate the remaining dependencies without analyzing them. The evaluation shows that the analysis can scale to Node.js applications with many dependencies, and that it detects almost all known taint flows in the benchmarks with no false positives (P4, Chapter 9, [109]).

C4 **How to aid Node.js developers in updating dependencies with breaking changes?** We first present a practical static analysis that leans toward over-approximating usages of specific parts of a library API, which are described using detection patterns. Our evaluation has shown that we can write detection patterns for the affected parts of the APIs for almost all breaking changes in 15 of the most popular npm packages. Using the written patterns on 265 clients of the popular npm packages, our analysis finds all known API usages, with a false positive rate on only 14% (P5, Chapter 10, [106]). Furthermore, we have shown that the analysis results can be used to semi-automatically patch those locations. Our tool JSFIX uses general descriptions of how to patch the affected locations, and then patches the locations that uses the affected API. JSFIX incorporates an interactive phase that asks the user when the analysis is uncertain about whether a location is a false positive match. Given these answers, our tool automatically patches the affected locations, such that they use the new version of the API correctly. Our evaluation shows that most breaking changes can be formalized in our language, and only few questions are asked per client. The evaluation also indicates that the transformations are correct, since they can repair almost all applications used in the evaluation. Furthermore, 31 pull requests created using the patches have been accepted by client developers, which indicates that the transformations are of high quality (P6, Chapter 11).

C5 **How to develop a security scanner with high precision?** We present a security scanner that only reports vulnerabilities if the vulnerable part of the API of a dependency is reachable. To create such a security scanner, we develop a modular call graph analysis that can construct a call graph for a Node.js application in a few seconds by reusing call graphs for its dependencies. We have used the call graph for security scanning, and the evaluation on 12 Node.js applications showed that we do not miss any true positive security alarms, while decreasing the number of false positives by 81% (P7, Chapter 12).

The author of this thesis has been a major contributor throughout the entire process from idea to implementation to paper writing for all the papers. However, for the paper Static Analysis With Demand-Driven Value Refinement, the author was mainly responsible for the implementation extending TAJS, while co-author Benno Stein was mainly responsible for designing and implementing the backwards analysis. For the last three research papers, P5–P7, the author has been working closely together with Martin Toldam Torp. We have both contributed almost equally to all parts of the projects, however, the author had a slightly higher focus on the experiments and evaluations, while Martin had a slightly higher focus on writing the technical part of the papers.

1.4 Outline

This thesis is separated into two parts. Part I provides an overview of the work carried out by the author, while Part II contains the research papers. We will now present the outline for Part I. We start by this introduction, and next provide some additional background material to help the reader understand some of the key challenges when analyzing JavaScript (Section 2.1) and Node.js (Section 2.2). This is followed by some underlying theory about program analysis that provides a better understanding of the analyses presented in the research papers (Section 2.3). We split the research papers into the two categories *sound static analysis* and *practical analysis*. Chapter 3 describes the sound static analysis category by presenting existing work, background for the TAJS analysis, which is a state-of-the-art sound static analyzer for JavaScript, and an overview of the techniques in the first four research papers that build upon TAJS. The second category is described in Chapter 4 by giving an overview of existing work and the new practical analyses for Node.js applications we present in the last three research papers. Chapter 5 concludes the thesis and explains how the work supports the thesis statement.

Chapter 2

Background

2.1 JavaScript

The first version of JavaScript was developed in 1995 at Netscape with the aim of being a scripting language for the browser. The initial prototype was developed by Brendan Eich in just 10 days [145]. JavaScript was designed to be a "little language" meaning a small and easy to learn language that is "specialized to a particular problem domain and does not include many features found in conventional languages" [15, 145]. The initial domain for JavaScript was scripting for web pages, so JavaScript programs are single-threaded and event-based, such that events can be registered and triggered by user actions. Since the language should be easy to learn, languages such as Java were too complex, since programmers would have to learn the concepts of a class and a main method as well as parameter and return types just for writing a simple program. The core design choices from the first prototype are still part of the language, but it has grown to be a language with many of the features found in conventional languages.

The JavaScript language is specified by the ECMAScript language specification[1]. This section will explain some of the most important ECMAScript features that pose challenges in analyzing JavaScript programs. The features we will cover are:

- Runtime types
- Dynamic object structure
- Inheritance
- Higher-order functions and closures
- Implicit calls through type coercions and getters/setters

We focus on these five features, since the way that a static analysis reasons about them is essential for its soundness, precision, and performance on real-world JavaScript code.

[1]https://www.ecma-international.org/ecma-262/6.0/index.html

2.1.1 Runtime Types

JavaScript does not have any static types, but JavaScript values do have types at runtime. The ECMAScript standard defines nine types. Six of the types are the primitive data types `undefined`, `boolean`, `number`, `string`, `bigInt` and `symbol`. The remaining three are `null`, `object`, and `function`. The type `undefined` describes only the value `undefined` that is used to represent that a value does not exist, for example, for variables that have been declared but not assigned, and for reading a non-existing property of an object. The type `boolean` describes the traditional `true` and `false` values, and the `number` type represents double-precision 64-bit floating point numbers. Values of type `string` are sequences of characters, and `bigInt` is a datatype for representing numbers of arbitrary precision format. The type `symbol` represent unique tokens, meaning that if a symbol is used as a property name, then it cannot be accessed by others by mistake. For instance, symbols have the property that they are unique even though they represent the same name, so `Symbol("foo") === Symbol("foo")` is `false`. The type `null` describes the value `null` that is a special primitive pointer pointing to a non-existent object. The `object` data type is basically a data structure consisting of key-value pairs, where all keys are strings or symbols, and the values can be any JavaScript values. The type `function` is naturally the type given to all functions.

Standard library The ECMAScript standard also specifies a standard library that, for instance, include other data structures, such as, `Array`, `Map` and `Set`. Furthermore, the standard library includes wrappers for the primitive types `Number`, `String`, `Boolean`, `BigInt` and `Symbol`. Surprisingly, instances of these builtin objects in JavaScript all have the type `object` instead of specific types for each data structure.

Dynamic type check JavaScript has the `typeof` operator that given an expression returns a string representation of its type. As one would expect, the string value is the name of one of the above mentioned types, except that `typeof null` returns `"object"`. Surprisingly, the code `typeof [1, 2, 3]` and `typeof (new Boolean(false))` return the string `"object"`. For this reason, many JavaScript developers have written their own versions of type predicate functions to, for instance, check whether an object is actually an array. The research paper in Chapter 8 shows a technique to precisely analyze code that uses type predicate functions.

Type coercion For flexibility, JavaScript also supports type coercions, i.e., implicit type conversions. For instance, for the binary operator `+`, JavaScript will make the necessary coercions for converting each of the operands into types for which `+` can be applied. For the code `1 + "1"`, JavaScript will coerce the first operand to a string, so the computation would be `"1" + "1"`, which results in the string `"11"`. More details about how this works is described in Section 2.1.5.

Truthy and falsy values All JavaScript values can be used in conditionals, for instance, `if (v) { ... } else { ... }` where `v` is either a string or number is also

```
1 var o1 = {foo: 1, bar: 2};
2 var o2 = {};
3 o2.foo = 1;
4 o2.bar = 2;
5 var o3 = {};
6 Object.keys(o1).forEach(k => o3[k] = o1[k]);
7 // o3 is {foo: 1, bar: 2}
```

Figure 2.1: Small example showing object instantiations and updates.

valid JavaScript. In JavaScript the conditional branch is not determined by whether the value is `true` or `false` but instead whether the value is truthy or falsy. The falsy values are `false`, `0`,`-0` , `0n` (bigInt `0`), `""`, `null`, `undefined`, and `NaN`. All other values are truthy values. This might lead to unexpected results. For example, when executing the code `if (new Boolean(false)) { ... } else { ... }`, the true-branch is visited since the wrapper boolean is technically an object, which is a truthy value.

2.1.2 Dynamic Object Structure

The dynamic object structure in JavaScript poses a great challenge for precise analysis of JavaScript, since a lot of spurious dataflow can arise from properties that are mixed together. This typically happens when analyzing JavaScript libraries. The research papers in Chapters 7 and 8 present techniques for precisely analyzing the dynamic object structure in JavaScript. We will now explain how the dynamic object structure work and afterwards why this feature is challenging for static analysis.

We refer to a key-value pair in an object as a property, and the key as the name. Properties can be added and deleted at runtime. Fig. 2.1 shows how to create objects and dynamically add properties. Line 1 creates a new object with two properties, a `foo` property, i.e., a property with name `foo`, with the value `1` and a `bar` property with the value `2`. The `foo` property can later be accessed by either reading `o1.foo`, `o1["foo"]` or `o1[x]`, where `x` evaluates to the string `"foo"` at runtime. If `x` is not a `string` or `symbol`, then it is coerced into a `string`. Line 2 creates another new object, and lines 3 and 4 dynamically add a `foo` and `bar` property to `o2` with the values `1` and `2`, respectively. Lines 5 and 6 show another way to create a new object and assign the same properties. The call `Object.keys(o1)` returns an array of all property names in `o1`, i.e., the array `['foo', 'bar']`. The `forEach` function then invokes the callback function `k => o3[k] = o1[k]` for each element in the receiver array, resulting in a call for each of the strings `"foo"` and `"bar"`. The code `o3[k] = o1[k]` first reads the `k` property of `o1` and then writes the value read to the `k` property of `o3`. Since `k` starts as `foo` and later becomes `bar`, it means that both the `foo` and `bar` properties are copied. The value of `o3` after line 6 is therefore `{foo: 1, bar: 2}`.

These object copy patterns are challenging for static analysis, since an analyzer will inevitable have an imprecise value for `k` in some cases, which results in all properties of `o1` being mixed together in `o3`. When analyzing JavaScript libraries, we often encounter such code patterns, and if not analyzed precisely, the analysis will mix

```
 8 var shared = {foo: 1, bar: 2};
 9 var obj1 = {};
10 obj1.__proto__ = shared;
11 obj1.foo; // has value 1
12 obj1.bar; // has value 2
13
14 var obj2 = {};
15 Object.setPrototypeOf(obj2, shared);
16 obj2.foo; // has value 1
17 obj2.bar; // has value 2
18
19 obj1.foo = 3;
20 obj1.foo; // has value 3
21 obj2.foo; // has value 1
22
23 shared.foo = 4;
24 obj1.foo; // has value 3
25 obj2.foo; // has value 4
```

Figure 2.2: Examples of inheritance in JavaScript.

100s of properties together, resulting in useless analysis results. The research papers presented in Chapters 7 and 8 present techniques for avoiding mixing properties together.

2.1.3 Prototype-Based Inheritance

JavaScript uses prototype-based inheritance, meaning that objects can inherit properties from other objects through a prototype chain. Prototype-based inheritance poses challenges for static analysis, since the prototype chain can be changed dynamically. For this reason, an analysis needs to be precise in reasoning about the prototype chain to avoid mixing a lot of objects together.

Internal prototype link Fig. 2.2 shows an example of how prototype-based inheritance works. Line 8 creates an object and writes it to the variable `shared`. This object will be the object that other objects inherit from in this example. On line 9 a new object is created, and the code `obj1.__proto__ = shared` in line 10 sets the internal prototype link to the object stored in `shared`. If an `x` property is looked up in `obj1`, and `obj1` does not have an `x` property, then the property is looked up in the `shared` object. This is illustrated in lines 11 and 12, where reading `obj1.foo` and `obj1.bar` yield the values `1` and `2` respectively, since these are the values inherited from `shared`. Lines 14–17 show another way to dynamically change the internal prototype link. It is changed through a call to `Object.setPrototypeOf` instead of directly writing to the internal prototype link `__proto__` as before. Line 19 writes the value `3` to `obj1.foo`, and the following two lines read the `foo` property of `obj1` and `obj2`, respectively. We see that `obj1.foo` is `3`, since it was just written. The read `obj2.foo` is still the value `1`, since the new value was written to the `foo` property in `obj1` instead of overwriting the `foo` property in `shared`. Line 23 updates the `foo` property of `shared`, so now when

```
26 function F(x) {
27   this.x = x;
28 }
29 F.prototype.foo = function () {
30   return this.x;
31 };
32 var obj = new F(2);
33 obj.foo(); // returns 2
34
35 F.prototype.foo = function () {
36   return this.x + 2;
37 }
38 obj.foo(); // returns 4
```

Figure 2.3: Constructor example.

reading `obj2.foo` the value is `4`, but `obj1.foo` remains the value `3`. We have shown how the internal prototype link can be dynamically modified, and how the internal prototype chain is used for resolving values of properties.

Mimicking classes We will proceed by showing how the internal prototype chain can be used for mimicking classes and subclassing as known from other object-oriented languages that use class-based inheritance, such as Java. Fig. 2.3 shows a short example on how classes can be mimicked[2]. Lines 26-28 creates a function named `F`. In JavaScript, all functions can be used as constructors, so new instances can be created using the `new` keyword. A convention is to start function names of functions that are intended to be used as constructors with a capital letter. Functions are also objects with properties, and a newly created function automatically has an object written to the `prototype` property of the function object. This object is shared between all objects created using this constructor. This means that the function written to `F.prototype.foo` in line 29 is accessible to all instantiations of `F`. When calling `new F(2)` in line 32, an empty object is constructed and used as the `this` value in the constructor, and the internal prototype link (the `__proto__` property) points to `F.prototype`. When reading `obj.foo()` in line 33, the `foo` property is read from `foo.__proto__`, i.e., `F.prototype` since the `foo` property does not exist in `obj`. The call returns the value `2`, since the `this` keyword refers to the receiver of the call, which is `obj` and the value `2` was written during the constructor call. Since the function is found using the prototype chain, it also means that the function can be changed by writing a new function to the property as done in lines 35 to 37. Since this newly written function is invoked by the call in line 38, the value `4` is returned.

Mimicking subclassing In practice, the `prototype` property does not always point to the object that was created together with a function, since it might be changed dynamically. This can happen when mimicking class-based inheritance with sub and super classes as shown in Fig. 2.4. Lines 39 to 44 creates a constructor named

[2]ES6 introduced the class construct, but that is just syntactic sugar for mimicking the class behavior.

```
39 function SuperClass(x) {
40   this.x = x;
41 }
42 SuperClass.prototype.foo = function () {
43   return this.x;
44 };
45 function SubClass(x) {
46   SuperClass.call(this, x);
47 }
48 SubClass.prototype = new SuperClass()
49 var subClassObject = new SubClass(2);
50 subClassObject.foo(); // returns 2
```

Figure 2.4: Subclassing example.

`SuperClass` and writes a function to `SuperClass.prototype.foo` similar to the previous example. Lines 45 to 47 creates a new constructor called `SubClass`. The constructor contains the line `SuperClass.call(this, x)` that invokes the `SuperClass` constructor with the same **`this`** and argument value as `SubClass` is called with. Line 48 sets up the prototype chain to make `SubClass` objects inherit from `SuperClass`, by creating a `SuperClass` object and write it to `SubClass.prototype`. Line 49 now creates a `SubClass` object and line 50 invokes the `foo` method that is implemented in the super class. The `SuperClass.prototype.foo` function is found by the following line of computation. Since `subClassObject` does not have a `foo` property, we lookup `subClassObject.__proto__.foo`. The value for `subClassObject.__proto__` is the result of **`new`** `SuperClass()` that was written to `SubClass.prototype` in line 48, and since that object itself does not have a `foo` property, `subClassObject.__proto__.__proto__` is looked up. This results in the `SuperClass.prototype` object, meaning that the `foo` property is the function created in lines 42 to 44, and the resulting value is `2`.

We will now discuss how this code can be analyzed, such that `obj.foo` soundly is inferred to be the function defined in lines 29 to 31. The sound static analyses in Chapter 3 solves this problem by precisely modelling each object and its prototype chain. These analyses reason similarly to what was explained throughout the example. A less precise, but more practical approach is used in Chapter 4, where all objects are mixed together, such that `obj.foo` refers to all values that are written to a `foo` property in the program.

2.1.4 Higher-Order Functions and Closures

JavaScript also has traits from functional programming in the form of higher-order functions and closures. The support for higher-order functions means that functions are ordinary values. Functions can therefore be written at dynamic property writes, passed as arguments to functions, and used as return values. Higher-order functions is one of the main reasons that imprecision in JavaScript analysis might result in avalanches of spurious dataflow and thereby render the analysis results useless.

JavaScript also supports closures that provide access to free variables through

```
51 function curry2 (f) {
52   return function g(a1) {
53     return function h(a2) {
54       return f(a1, a2);
55     }
56   }
57 }
58 function plus(a, b) {
59   return a + b;
60 }
61 var curriedPlus = curry2(plus);
62 var partialPlus = curriedPlus(2);
63 partialPlus(5); // returns 7
```

Figure 2.5: Higher order functions and closure example.

lexical scoping.[3] Variables are function-scoped, not block-scoped as in Java, so variables in different blocks are shared across a function.[4] The variables are stored as properties of special activation objects which are created on function invocations.

Fig. 2.5 shows an example of how to design a curry function using higher-order functions and closures. Lines 51 to 57 create the `curry2` function that takes a function as argument that itself takes two arguments. The function `curry2` returns a curried version of the function argument, i.e., a function that takes the two arguments as two single argument calls instead. The details are explained later. Lines 58 to 60 define a `plus` function that takes two numeric arguments and adds them together. The function `curry2` is invoked in line 61 with the `plus` function as argument. The invocation now returns a new function that takes a single argument, and that function is written to the variable `curriedPlus`. Line 62 invokes the previously returned function with the argument `2` and this invocation returns another function taking a single argument. This function is stored in `partialPlus` and is invoked at line 63 with the value `5`. This call returns the value `7`, since `f` is the `plus` function, `a1` is `2`, and `a2` is `5`. The resolution of `f` follows the scope chain, by first checking whether the activation object for the `h` function contains `f`. Since this is not the case, the activation object for the invocation of `g` is tested, and since it does not contain `f` either, the activation object for the invocation of the `curry2` function is checked. This activation object contains `f`, which is the `plus` function due to the invocation on line 61. The values for `a1` and `a2` are resolved similarly from the activation objects of `g` and `h`, respectively.

Reasoning about higher-order functions and closures is a challenge to static analysis. An analysis has to track the function values, and for precise analysis of JavaScript libraries, it is important not to mix different closures together. The research papers in Chapters 7 and 8 present efficient techniques for avoiding mixing different closures together. The research paper in Chapter 12 presents a practical analysis for

[3]JavaScript uses lexical scoping, except when using the `with` statement that dynamically adds an object to the scope chain.

[4]ES6 introduces the `let` statement which can be used to declare block-scoped variables, but we focus on ES5, where all variables are function-scoped.

```
64 var counter = {
65   _count: 0,
66   get current() {
67     return this._count;
68   },
69   get count() {
70     return ++this._count;
71   },
72   set count(newCount) {
73     this._count = newCount;
74   }
75 }
76 counter._count; // has value 0
77 counter.current; // has value 0
78 counter.count; // has value 1
79 counter.count; // has value 2
80 counter.current; // has value 2
81 counter.count = 5;
82 counter.current; // has value 5
83 counter.count; // has value 6
```

Figure 2.6: An example JavaScript program using getters and setters.

reasoning about higher-order functions when building call graphs.

2.1.5 Implicit Calls through Type Coercions and Getters/Setters

Another challenge for sound static analysis of JavaScript is implicit calls, i.e., calls that are not obvious from the syntax.

Type coercions As mentioned earlier, JavaScript supports type coercions, and we will now explain how they work through implicit calls. JavaScript makes these coercions by introducing wrapper objects and calling either `toString` or `valueOf` depending on whether the coercion should be to a string or to a number. This means that the code `1 + "1"` has the same behavior as `(new Number(1)).toString() + "1"`, where a `Number` object is created and the `toString` function is called on that object. Depending on the types, different coercions occur. For instance, evaluating `1 + false` results in the value `1` by evaluating `1 + new Boolean(false).valueOf()`, since `valueOf` the wrapper object for `false` is `0`. These coercions happen for all binary operators, and the specific type to coerce to depends on the operator.

To soundly analyze these implicit calls, an analysis needs to track all data-flow. The analyses presented in Chapter 3 tracks all data flow and can therefore reason about type coercions soundly. However, tracking all data flow is expensive, so in the practical analysis we present in Chapter 4, we ignore implicit calls due to type coercions.

Getters and setters ECMAScript 5 introduced other implicit calls through getters and setters. Fig. 2.6 shows an example using getters and setters. Line 64 to 75 create a new counter object that have the property `_count` that represents the current value of the counter. The object also contains two getters (`current` and `count`) and a setter (`count`) that are explained later. Line 76 simply reads the current value of the `_count` property, which is `0`. The code `counter.current` in line 77 implicitly calls the function `current` defined in lines 66 to 68. This function simply returns the value of the `_count` property, so in this case it is also the value `0`. Line 78 invokes the `count` getter defined in lines 69 to 71 that first increments the `_count` property and then returns the incremented value, resulting in the value `1` being returned. The next line works similarly and returns the value `2`. Next, line 79 again reads the current value, but now that value is `2`. Line 81 invokes the setter `count` defined in lines 72 to 74. The argument `newCount` gets the value that is being written, so in this case the value `5`. Note that the getter and setter can have the same name, and which one is invoked depends on the context, i.e., whether it is a property read or a property write. The `count` setter simply writes the written value to the `_count` property. Therefore the next invocation of the `current` getter in line 82 returns `5`, and the following invocation of the `count` getter returns `6`.

The sound analyses we present in Chapter 3 tracks getters and setters similar to how other values are tracked. For the practical analyses in Chapter 4, we track getters and setters per property name, such that getters with the same name are mixed together and similarly for setters.

We have now presented some of the JavaScript features that are important to consider when designing a static analysis for JavaScript, and we will now proceed with explaining about the Node.js.

2.2 The Node.js Ecosystem

From the first prototype of JavaScript in 1995 and until 2009, JavaScript was mostly used for client-side web applications, but this changed with the introduction of Node.js[5]. Node.js provides a runtime environment that allows web servers to be implemented in JavaScript. Node.js relies on the V8[6] JavaScript engine and combines it with an event loop and a low-level I/O API written in C++. This design choice allows for interaction with the underlying machine, its file-system, operating system resources and databases, whereas JavaScript in browsers are executed in a sandbox.

Node.js has since its introduction in 2009 increased significantly in popularity, and in the Stack Overflow survey from 2019, Node.js was the most commonly used framework, library or tool, with around 50% of the 58 543 responders using Node.js.[7] An important part of the Node.js ecosystem is the large reuse of packages, which is facilitated by the npm package manager. Meta-information for an application, such as what dependencies the application has, is provided through a package.json file.

[5]https://nodejs.org/

[6]https://v8.dev/

[7]https://insights.stackoverflow.com/survey/2019#technology-_-other-frameworks-libraries-and-tools

```
{
  "name": "simple-express-server",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

Figure 2.7: An example of a package.json file.

Dependencies and package.json All Node.js applications contain a package.json file that includes a list of all the packages and package-versions that the application depends on. Fig. 2.7 shows an example of a small package.json file. It contains the name of the application and its version number. The `"main"` entry specifies the main file of the package, and `"scripts"` is an object containing command names as keys and commands as values. In the example there is only a single command, `"test"`, which just writes back the error message `"Error: no test specified"`. It is very common for packages to have a test script for running the test suite of the package by running `npm test`. We will use this in the research papers presented in Chapters 10 and 11. The `"dependencies"` entry consists of an object where the keys are names of packages, and the corresponding values are the acceptable version range for the package. In this example, the package only depends on *express*, which is a framework for writing web servers. The express entry has automatically been added to the package.json file by running `npm install express`. The version numbering scheme used by npm is semantic versioning that uses version numbers of the form `x.y.z`, where `x` is the major version, `y` is the minor version and `z` is the patch version. Generally, patch updates should only contain bug fixes, while minor versions are allowed to add functionality without introducing breaking changes, and major version updates are allowed to include breaking changes, i.e., changes that require the application developers to update their code to be compatible with the new version. However, there are no guarantees that package developers actually follow these guidelines, which means that there can also be breaking changes in minor and patch updates. The version number `^4.17.1` used for *express* in Fig. 2.7 means that it depends on version 4.17.1 or any newer minor and patch update, so by running `npm install`, it would be possible that express v. 4.23.2 is installed if it existed. With more than 1.4 million packages[8], npm is by far the package manager that contains most packages.

An express application An example of a hello world express app is shown in Fig. 2.8.[9] Once the command `npm install express` has been executed, the hello

[8]`http://www.modulecounts.com/`

[9]The code is taken from: `https://expressjs.com/en/starter/hello-world.html`

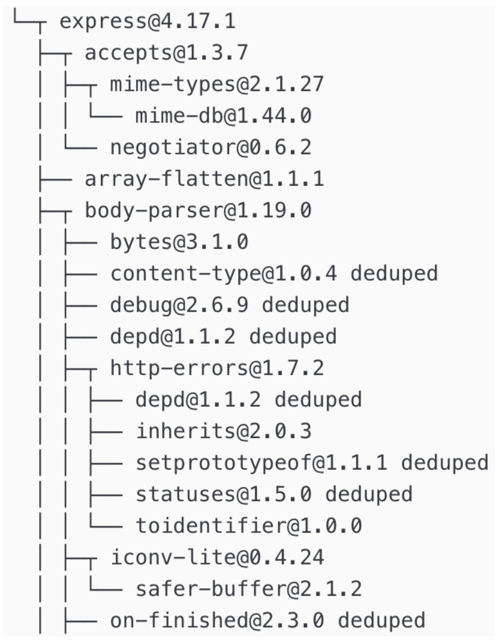
```
 95 const express = require('express')
 96 const app = express()
 97 const port = 3000
 98
 99 app.get('/', (req, res) => {
100   res.send('Hello World!')
101 })
102
103 app.listen(port, () => {
104   console.log('Example app listening at http://localhost:' + port)
105 })
```

Figure 2.8: A hello world express app.

world express app is ready to run. Line 95 imports the express package through the require call, and line 96 creates the express app. Lines 103 to 105 set the express server to listen at port 3000, and when the connection is setup, the message `"Example app listening at http://localhost:3000"` is written to the console. Lines 99 to 101 register a get handler that is executed when `http://localhost:3000` is visited. The get handler simply returns a page with the text "Hello World!". As can be seen from the example it is very easy to create a simple web app using Node.js.

Node.js applications heavily rely on third-party packages. An average application depends on more than 50 packages, either directly or transitively [81, 152], and the application code excluding the dependencies only constitutes 10% of the application [87]. In the example express app in Fig. 2.8, the code of the dependencies constitutes a very large fraction of the actual code. Even though the application only contains 9 lines of code, the application including the dependencies are quite large due to *express* being large and due to other dependencies used by *express*. The command `npm ls` gives a textual representation of the applications dependency tree. Fig. 2.9 shows the first 20 lines of running `npm ls` on the hello world express app. The full output is 85 lines. The output shows, for example, that the application depends on *express*@4.17.1, which itself depends on *accepts*@1.3.7, *array-flatten*@1.1.1, and *body-parser*@1.19.0. The lines that end with `deduped` mean that the package can be reused from another path in the dependency tree, such that the same package is not installed multiple times. The application transitively depends on 50 different third-party packages, just by directly depending on *express*.

Security Security has been deemed an important issue for Node.js since Node.js applications have access to the underlying file-system and operating system resources, meaning that injection vulnerabilities might give an attacker access to the underlying system. There are primarily two functions that are critical for injection attacks, `exec` and `eval`. The `exec` function is from the builtin Node.js module `child_process`, and allows arbitrary shell commands to be executed. The `eval` function is from JavaScripts standard library, and allows arbitrary JavaScript code execution. The heavy use of third-party packages can introduce these kinds of security vulnerabilities, since some of the dependencies might be vulnerable. In order to mitigate these



```
└─┬ express@4.17.1
  ├─┬ accepts@1.3.7
  │ ├─┬ mime-types@2.1.27
  │ │ └── mime-db@1.44.0
  │ └── negotiator@0.6.2
  ├── array-flatten@1.1.1
  ├─┬ body-parser@1.19.0
  │ ├── bytes@3.1.0
  │ ├── content-type@1.0.4 deduped
  │ ├── debug@2.6.9 deduped
  │ ├── depd@1.1.2 deduped
  │ ├─┬ http-errors@1.7.2
  │ │ ├── depd@1.1.2 deduped
  │ │ ├── inherits@2.0.3
  │ │ ├── setprototypeof@1.1.1 deduped
  │ │ ├── statuses@1.5.0 deduped
  │ │ └── toidentifier@1.0.0
  │ ├─┬ iconv-lite@0.4.24
  │ │ └── safer-buffer@2.1.2
  │ ├── on-finished@2.3.0 deduped
```

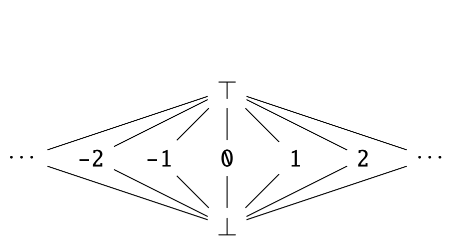
Figure 2.9: The top 20 lines output from running `npm ls` on the hello world express app.

issues, vulnerability databases exist, for instance, the npm advisory[10] and the snyk.io vulnerability database[11]. Currently the best way of detecting such vulnerabilities is through package-level security scanning, where the vulnerability databases are checked to see whether an application depends on any packages in a version that has a vulnerability. If so, it is reported as a security vulnerability, no matter how the dependency is actually used by the application, so this might produce many false positives. The research paper presented in Chapter 12 addresses this problem.

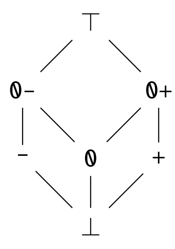
2.3 Sound Static Program Analysis

Reasoning about correctness and other properties of programs is typically done through program analysis. A program analysis takes as input a program and outputs facts about that program. It is not possible to design an analysis that outputs the exact facts about the input programs. Therefore, the analysis have to do some approximation. In this thesis, the focus is on overapproximating program behavior, because we do not want our analyses to miss potential problems in a program. In the first part of the thesis we design analyses that are sound, i.e., overapproximates the program behavior for any program, and in the second part the analyses are not sound, but aim at producing sound results for real-world programs. The theory presented in

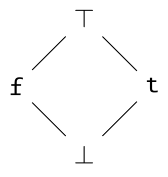
[10] `https://www.npmjs.com/advisories`
[11] `https://snyk.io/vuln?type=npm`



(a) Constant-propagation *Num* lattice



(b) *Sign* lattice



(c) *Boolean* lattice

Figure 2.10: Examples of numeric and boolean lattices.

this section is not new, but explains some of the ideas proposed for sound static analysis back in the 1970s [31, 76, 82]. The three research papers presented in Chapters 7–9 are all sound static analyses for JavaScript that builds upon this theory. More specifically, the analyses are defined as monotone frameworks [76], which are presented in Section 2.3.2. Now, we briefly introduce lattices, since they are a key component in a monotone framework.

2.3.1 Lattices

A lattice is a partial order with an additional constraint, which we will present when we have introduced what a partial order is. A partial order (S, $\sqsubseteq$) consists of a set S and an ordering $\sqsubseteq$, that must be reflexive ($\forall x \in S : x \sqsubseteq x$), transitive ($\forall x,y,z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$), and anti-symmetric ($\forall x,y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$). For $x,y \in S$, y overapproximates x when $x \sqsubseteq y$, we also say that y is less precise than x. Partial orders also have the two operators least-upper bound $\sqcup$ and greatest lower-bound $\sqcap$. For $x,y \in S$, $x \sqcup y$ gives the most precise element z that satisfies $x \sqsubseteq z$ and $y \sqsubseteq z$, while $x \sqcap y$ gives the least precise element z that satisfies $z \sqsubseteq x$ and $z \sqsubseteq y$. Now that we have described what a partial order is, we can properly define what a lattice is.

A lattice is a partial order, where for all $x,y \in S$, $x \sqcup y$ and $x \sqcap y$ are defined. The lattices we consider in this thesis have a finite height, meaning that the chain $a_1 \sqsubseteq a_2 \sqsubseteq a_3 \sqsubseteq \cdots$ must be finite for $a_i \in S$ and $a_i \neq a_j$ for $i \neq j$. Using finite height lattices in an analysis ensures its termination. We will refer to $x \sqcup y$ as joining x and y. Fig. 2.10 shows two lattices representing numeric abstract domains and a lattice for a boolean abstract domain. The first lattice is a constant-propagation lattice (Fig. 2.10a) and the second is a *Sign* lattice (Fig. 2.10b). The constant-propagation lattice is precise for all values where a unique value can be determined, i.e., the numeric elements such as -1, 0, and 1 have zero loss of abstraction, meaning that they only abstract a single concrete value. However, when joining two values, for instance, 1 $\sqcup$ 2, we get the abstract value $\top$, since $\top$ is the least element z that satisfies 1 $\sqsubseteq$ z and 2 $\sqsubseteq$ z. We define the height of the lattice to be the longest chain from $\bot$ to $\top$. For lattices with height 2 such as the constant propagation lattice, the result of using least-upper bound on two distinct values (excluding $\bot$) is always $\top$. For higher lattices such as the *Sign* lattice (height 3), an analysis might be slower to reach its fixpoint. In the *Sign* lattice,

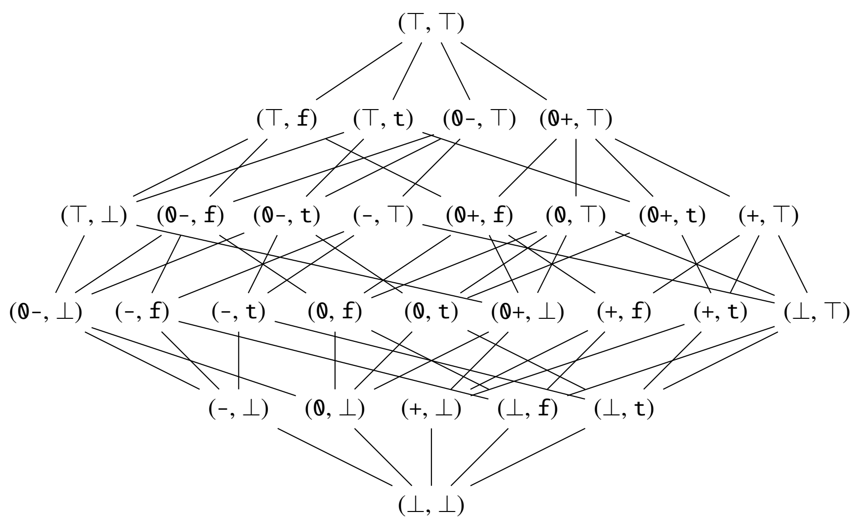

(⊤, ⊤)
(⊤, f) (⊤, t) (0-, ⊤) (0+, ⊤)
(⊤, ⊥) (0-, f) (0-, t) (-, ⊤) (0+, f) (0, ⊤) (0+, t) (+, ⊤)
(0-, ⊥) (-, f) (-, t) (0, f) (0, t) (0+, ⊥) (+, f) (+, t) (⊥, ⊤)
(-, ⊥) (0, ⊥) (+, ⊥) (⊥, f) (⊥, t)
(⊥, ⊥)

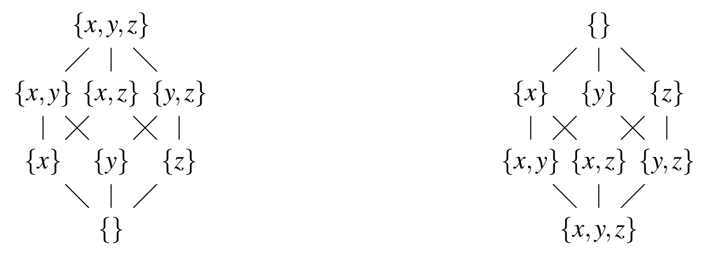
Figure 2.11: Product lattice: Sign $\times$ Boolean.

there is a loss of abstraction for unique numbers since they are abstracted to their sign, i.e., whether they are a negative number (`-`), `0`, or a positive number (`+`). For example `-` $\sqcup$ `0` yields `0-`, `+` $\sqcup$ `0` yields `0+` and `-` $\sqcup$ `+` yields $\top$. The *Boolean* lattice simply contains elements for false (`f`) and true (`t`), and the $\top$ element indicating a value that might be both false and true. The following paragraphs explain some lattice constructions that will be used in the abstract domains later in this thesis.

Product lattice The lattices used as abstract domains later in this thesis consist of composing simple lattices as the ones presented above. Lattices can be composed as product lattices, i.e., $L = L_1 \times L_2$. Fig. 2.11 shows the product lattice *Sign* $\times$ *Boolean*. The product lattice is constructed by having lattice elements contain a component for each of the lattices in the product, and each step upwards the lattice corresponds to a step up in one of the lattices. The height of the lattice is the sum of the heights of the lattices used in the product, which means that the height of the product lattice in Fig. 2.11 is five, since the *Sign* lattice has height three and the *Boolean* lattice has height two.

Powerset lattice Another common lattice type is the powerset lattice, written $\mathscr{P}(S)$ where S is a set. The lattice elements of the lattice $\mathscr{P}(S)$ are all subsets of the set S. The order of the elements then follows either $\subseteq$ or $\supseteq$. Fig. 2.12 illustrates the powerset lattice $\mathscr{P}(\{x, y, z\})$ with both orderings. The height of a powerset lattice $\mathscr{P}(S)$ is the size of the set S.

Map lattice The analysis domains we will see in this thesis describes properties for each program point. Map lattices can provide this kind of reasoning: $N \to L'$, where N describes all program points, i.e., all nodes in the program, and L' describes the



(a) With ordering $\subseteq$ (b) With ordering $\supseteq$

Figure 2.12: Powerset lattice $\mathscr{P}(\{x,y,z\})$.

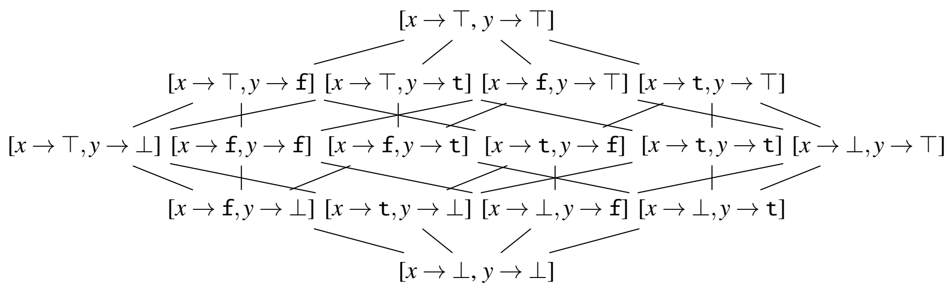

[x → ⊤, y → ⊤]
[x → ⊤, y → f] [x → ⊤, y → t] [x → f, y → ⊤] [x → t, y → ⊤]
[x → ⊤, y → ⊥] [x → f, y → f] [x → f, y → t] [x → t, y → f] [x → t, y → t] [x → ⊥, y → ⊤]
[x → f, y → ⊥] [x → t, y → ⊥] [x → ⊥, y → f] [x → ⊥, y → t]
[x → ⊥, y → ⊥]

Figure 2.13: Map lattice: $Vars \rightarrow$ Boolean, where $Vars = \{x,y\}$.

information we want to infer for each program point. More generally, given a set S, and a lattice L, we can construct the map lattice $S \rightarrow L$ that intuitively provides an element of the lattice L for each element in S. The map lattice consists of the set of all functions from S to L ordered pointwise as shown in Fig. 2.13 for the map lattice mapping the set of variables $Vars = \{x,y\}$ to the boolean domain: $Vars \rightarrow Boolean$. In each step up in the lattice, one of the variables goes up one step in the *Boolean* lattice.

2.3.2 Monotone Frameworks

A monotone framework [76] defines a flow-sensitive static analysis, i.e., an analysis where the order of statements is taken into account. A monotone framework is the combination of a lattice describing the program properties that should be inferred for each program point, and transfer functions for each program point. We use control flow graphs (CFGs) for representing programs, meaning that a program point is a CFG node. Each transfer function f has to be monotone, i.e., $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$. A monotone framework is sound if all the transfer functions are sound, i.e., they overapproximate the concrete behavior of the program points that they abstract. Since a monotone framework is used to infer something about each program point, we use the map lattice $N \rightarrow L'$, where N is the set of flow graph nodes, and L' is a lattice describing what we would infer for each node. We will refer to an element $\sigma_n \in L'$ as an abstract state for node n.

Intuitively, an abstract state for a program point is computed by joining the abstract states from its predecessors, and then applying the transfer function for the program point. These computations are then performed until a fixpoint has been reached. More formally we can define the *JOIN* function as

$$JOIN(v) = \bigsqcup_{n \in pred(v)} \sigma_n$$

where *pred* takes a node as argument and returns all the predecessor nodes. A solution to the analysis problem is then found by reaching a fixpoint in the following equation system:

$$\begin{aligned}
\sigma_{n_1} &= f_{n_1}(JOIN(n_1)) \\
\sigma_{n_2} &= f_{n_2}(JOIN(n_2)) \\
&\vdots \\
\sigma_{n_m} &= f_{n_m}(JOIN(n_m))
\end{aligned}$$

where f_{n_i} is the transfer function for node n_i.

Solving the equation system As seen by the definition of the functions f_{n_i}, the abstract state at n_i only depends on the abstract states of the predecessors to n_i in the CFG. We can exploit this dependency when solving the equation system by having a worklist of the abstract states that need to be recomputed, and a worklist algorithm [31, 82] for deciding the order to recompute abstract states in. The worklist algorithm works by using a dependency map $dep : N \to \mathscr{P}(N)$ that for each node returns the set of nodes that depends on its abstract state. For ordinary forward analysis, this dependency map is just defined as a map from a node to all of its successors. The analysis we present in Chapter 7 needs to add more dependencies to this map in order to remain sound. A solver can simply be implemented as shown in Algorithm 1. The entry node of the program is initially added to the worklist in line 1, and then the while loop in lines 2 to 11 keeps iterating until the worklist is empty. In the loop body, a node n_i is taken out of the worklist in line 3. Line 4 stores the current state, and line 5 then updates the current state by running the transfer function for node n_i. If the state has been updated, i.e., the condition in line 6 is true, all dependents in the dependency map are added to the worklist in lines 7 to 9.

Context sensitive analysis A standard approach to increase precision of a static analysis is to use context sensitivity [129]. Recall when using monotone frameworks we use a lattice of the form $L = N \to L'$, where N is the set of program locations, i.e., the set of nodes in the CFG. For context sensitive analysis, we augment this lattice with contexts such that we have the lattice $L = (C \times N) \to L'$, where C contains identifiers for different contexts. In this new lattice, we can have multiple abstract states for each CFG node. Contexts can, for instance, be distinguished by the call stack, such that different calls are analyzed separately and the arguments for different calls are not

Algorithm 1 Standard worklist algorithm

```
1:  add the initial node $n_1$ to the worklist
2:  while worklist not empty do
3:      remove node $n_i$ from worklist
4:      $preState = \sigma_{n_i}$
5:      $\sigma_{n_i} = f_{n_i}(JOIN(n_i))$
6:      if $preState \neq \sigma_{n_i}$ then
7:          for $n_j \in dep(n_i)$ do
8:              add $n_j$ to worklist
9:          end for
10:     end if
11: end while
```

mixed together. Chapter 3 goes more into detail about the context sensitivity used in the TAJS analysis.

Pointer analysis In JavaScript, objects are represented by pointers, so pointer analysis is important in static analysis for JavaScript. Basically, pointer analysis is about abstracting which objects a variable may point to. In the literature there exists a wide variety of analyses [9, 134] that specifically compute points-to information, but such analyses cannot be used directly when analyzing JavaScript. The dynamic nature of the language means that to get precise points-to information, it is natural to integrate the points-to analysis into a monotone framework. This is done through another map lattice $Loc \to Obj$, where Loc is the set of all object addresses and Obj is a lattice describing abstract objects. There is a huge design space in abstracting the set of object addresses Loc, and Chapter 3 and Chapter 4 use two quite different abstractions. In Chapter 3 we will see how Obj is designed in TAJS.

Chapter 3

Sound Static Analysis Techniques for JavaScript and Node.js

3.1 Existing Work

Static analysis for JavaScript has been an active research area in the past decade. Four static analysis frameworks for JavaScript have been proposed, TAJS [70], SAFE [91], JSAI [80], and WALA [133]. These works have found that imprecision in the static analysis often results in avalanches of spurious data-flow that render the analysis results useless. The designs of SAFE and JSAI are in many ways similar to that of TAJS. JSAI has been used to experiment with variations of classical context sensitivity strategies, such as call-string and object sensitivity. The results showed that some applications need very high degrees of context sensitivity to provide useful results, while for other applications where the high degree is unnecessary, the analysis does not scale when applying the high degrees of context sensitivity. This show that in JavaScript analysis, we do not have the same precision/performance trade-offs that is typical to analysis of other languages, since we cannot expect a faster analysis when using a more imprecise analysis. The context sensitivity strategies therefore need to be more specialized as in the following works.

Andreasen and Møller [10] extend TAJS and show that high degrees of specialized context sensitivity is needed for analyzing *jQuery*. This finding is also supported by the technique SAFE_{LSA} [115] that uses 10-CFA and loop unrolling for nested loops up to level 30 with up to 1000 unrollings per loop as the context sensitivity strategy.

Another line of work have tried to improve JavaScript analysis by having more expressive abstract domains. Especially, the string domain [8, 96, 116] has been investigated to avoid imprecise dynamic property reads and writes. Even though an analyzer has high degrees of context sensitivity and an elaborate abstract domain, imprecision is inevitably introduced. The techniques discussed so far often performs poorly once such imprecision has been introduced.

To summarize, JavaScript analyzers need to strike a very delicate balance for what precision is needed. Blindly increasing context sensitivity has a negative impact on

```
...
var t = obj[p];
...
obj2[p] = t;
```

Figure 3.1: Example of a correlated read/write pair.

performance, while being too imprecise makes the analysis results useless due to avalanches of spurious dataflow. Therefore, one of the challenges when designing static analysis for JavaScript is to understand the limitations of an analyzer, i.e., to understand why the analyzer does not perform well on a JavaScript program. The first contribution we will present in this thesis is an automated approach that can find source locations in the analyzed programs where an analysis has introduced critical precision loss (Chapter 6, Section 3.3.1). These source locations can then be used to understand the limitations of the analysis.

The existing analyzers previously mentioned all focus on client-side JavaScript programs, not Node.js applications, which are the focus of this thesis. Our initial approach was to investigate whether TAJS can analyze the most popular Node.js libraries *underscore* and *lodash*, which unfortunately was not the case. To investigate this, we applied the automatic technique from before for finding analysis bottlenecks on these libraries. We found that the critical code that hindered TAJS in analyzing *underscore* and *lodash* was correlated read/write pairs, i.e., code of the form shown in Fig. 3.1. The code copies the property name that `p` refers to from `obj` to the same property of `obj2`. If a static analysis has an imprecise value for `p`, then all properties of `obj` are mixed together in all properties of `obj2`. This means that if `obj` is the object `{foo: function foo() {...}, bar: function bar() {...}}`, then `obj2` becomes an object where all properties of `obj2` points to both the `foo` and `bar` function. This means that a call like `obj2.bar()` will spuriously add call edges to the `foo` function as well.

Multiple attempts of analyzing such code patterns precisely have been presented in previous work. The first work that identified this pattern as critical was the correlation tracking work from 2012 [133] that syntactically identified these code patterns where the property name at the dynamic property read and dynamic property write are the same. In these cases, context sensitivity is increased to avoid mixing multiple values of `p` together. The technique, however, only works if `p` is not already imprecise. Furthermore, the patterns are widely used in JavaScript libraries and in many varieties, resulting in it being too fragile to only target the syntactic patterns where the property name is the same in the read and the write.

After the correlation tracking paper, analyzers tried to increase context sensitivity to avoid the imprecision of `p`, for instance the static determinacy technique in TAJS [10], and the loop-sensitive technique in SAFE$_{\mathrm{LSA}}$ [115]. However, even with the high degrees of context sensitivity, the property name in such patterns will inevitably be imprecise in some cases, and this is specifically the case for the *underscore* and *lodash* libraries. This means that to analyze some of the most popular Node.js packages such as *underscore* and *lodash*, an analysis has to reason precisely about correlated read/write pairs as the one in Fig. 3.1 where the value for the property name is imprecise. We will now explain the only technique that tries to address this problem.

Prior to our work, the CompAbs analyzer [84] is the only analyzer that tries to analyze correlated read/write pairs precisely even though the property name is imprecise. CompAbs is built on top of SAFE. We will refer to the technique implemented in the CompAbs analyzer as *state partitioning* throughout this thesis. Ko et al. [84] generalizes the correlated read/write pattern to *field copy or transformation* (FCT) patterns. A correlated read/write pair is an FCT pattern. Furthermore, FCT patterns also allow arbitrary transformations of the property name and the value read between the dynamic property read and dynamic property write. The technique relies on a syntactic pre-analysis for identifying the patterns. This restricts the technique to only work for intraprocedural FCT patterns, where no transformations are performed on the property name `v`. For a simple pattern as the one shown in Fig. 3.1, the pre-analysis can detect the FCT pattern, and therefore CompAbs applies state partitioning when analyzing the dynamic property read at line 107. The state partitioning technique partitions the current state where `p` is imprecise, to a precise abstract state for each property in `obj` and a state for all properties that are not in `obj`. Assuming that `obj` is the object `{foo: function foo() {...}, bar: function bar() {...}}`, three new states will be introduced, one where `p` is `"foo"`, one where `p` is `"bar"`, and one for all other properties. The write then happens precisely in each state, and the states are joined together afterwards. State partitioning thereby avoids mixing together the properties of `obj` as desired. However, the state partitioning technique is also fragile, since it only works for intraprocedural FCT patterns. The *lodash* library, for instance, contains interprocedural FCT patterns, meaning that the read is in one function and the write is in another.

Even in the hypothetical situation that the state partitioning technique works for interprocedural FCT patterns, it would still be too expensive to be practically useful. An experiment described in the paper presented in Chapter 7 shows that if state partitioning is applied at the necessary locations in *lodash*, then the analysis will take more than 200 seconds to analyze the initialization code of *lodash*. New techniques are therefore required to make sound static analysis for Node.js applications. The research papers presented in Chapters 7 and 8 show such new techniques for handling FCT patterns and other code patterns that are tricky for static analysis for JavaScript. Section 3.3.2 shows how the techniques solve the problem about FCT patterns.

Security analysis for Node.js applications has also received some attention in the past years. As mentioned in Section 2.2, Node.js applications have direct access to the underlying file system and operating system resources. Injection vulnerabilities through the `exec` and `eval` functions can therefore give an attacker control over the machine running the Node.js application. A study by Staicu et al. [135] shows that 20% of all modules either directly use or depend on a module that uses either `eval` or `exec`. Staicu et al. [135] also developed a tool called SYNODE that uses an intraprocedural analysis trying to determine whether calls to the `eval` and `exec` functions are safe or not. If the analysis finds that a call might be vulnerable, then it is dynamically enforced that only safe values are passed to the `eval` and `exec` functions. If the analysis finds that an unsafe value is passed to one of the functions, then the command is not executed. Since the analysis used by SYNODE is intraprocedural, it is too imprecise and therefore leads to situations where safe usages of `eval` and `exec` are being blocked at runtime. Other papers have tried to detect injection vulnerabilities using dynamic analysis [52, 78].

Prior to the work presented in Chapter 9, no static analysis existed that aimed at detecting injection vulnerabilities for Node.js applications through an interprocedural static analysis.

3.2 TAJS

The research papers presented in Chapters 6–9 extend the static analyzer TAJS. This section does not present any contributions of this thesis, but explains the fundamentals of TAJS that are useful for understanding some of the contributions of this thesis. The TAJS analysis is defined using monotone frameworks (see **??**), and this section describes how JavaScript programs are represented in the analysis, and how the analysis lattice is designed to handle the JavaScript features explained in Section 2.1. TAJS was originally presented by Jensen et al. [70] and extended in subsequent work [10, 71–73]. To gain a sufficient understanding of TAJS we summarize the fundamentals of the analysis from the original paper and the context sensitivity strategies [10] that enable analysis of the initialization code of the *jQuery* library.

3.2.1 Program Representation

TAJS uses control flow graphs (CFGs) for representing JavaScript programs, where nodes represent instructions and edges represent control-flow. The nodes in the flow graph uses temporary registers for breaking composite expressions and statements into instructions. Some of the flow graph node types are shown in Fig. 3.2. Flow graph nodes related to exceptional dataflow, `for-in` and `with` statements are not included, since they are not important for understanding the contributions of this thesis. Fig. 3.3 shows the flow graph for the example program in Fig. 3.1. The top four nodes represent the dynamic property read `t = obj[p]`, and the last four nodes represent the dynamic property write `obj2[p] = t`.

3.2.2 Analysis Lattice

The analysis lattice will be described in its original form [10, 70], i.e., without the extensions proposed in the research papers included in Chapters 7–9. The analysis lattice has a collection of abstract states for each program point:

$$\mathit{AnalysisLattice} = C \times N \to \mathit{State}$$

Since TAJS is context sensitive, we use the component C to represent identifiers for contexts. We describe the context sensitivity used by TAJS in Section 3.2.4. The `declare-variable[`x`]`: Declares a variable x with the value `undefined`.

`read-variable[`x, r`]`: Reads the value of variable x and stores it in the temporary register r.

`write-variable[`r,x`]`: Writes the value of r to the variable x.

`constant[`c, r`]`: Writes the constant value c to the temporary register r.

`read-property[`r_1, r_2, r_3`]`: Reads a property where the property name is in r_2, the base value is in r_1, and the result is stored in r_3.

`write-property[`r_1, r_2, r_3`]`: Writes the value in r_3 to the property name in r_2 of the value in r_1.

`if[`r`]`: Represents conditional flow, i.e., jumps to another instruction based on the result stored in r. Used for, e.g., `if` and `while` statements.

`call[`$r_f, r_{res}, r_{this}, r_1, \ldots, r_n$`]`, `construct[`$r_f, r_{res}, r_{this}, r_1, \ldots, r_n$`]`: Calls are represented by a `call` node. The register r_f contains the function being called, register r_{this} contains the this value, and registers r_1 to r_n contain the arguments for the call. The return flow from the function f is returned to the successor of the call, where r_{res} contains the return value. The `construct` node is similar to the `call` node but used for constructor calls, i.e., calls using the **`new`** keyword.

`return[`r`]`: Return node that returns the value in r.

$\langle$ `op` $\rangle$`[`r_1, r_2`]`: Unary operator, where the result of applying `op` to the value in r_1 is stored in r_2.

$\langle$ `op` $\rangle$`[`r_1, r_2, r_3`]`: Binary operator, where the result of applying `op` to the value in r_1 and r_2 is stored in r_3.

Figure 3.2: Flow graph nodes in TAJS.

lattice of abstracts states in TAJS is defined as follows:

$$State = (Loc \to Obj) \times Stack \times \mathscr{P}(Loc) \times \mathscr{P}(Loc) \times MustEquals$$

where *Loc* is the set of object addresses. Since TAJS uses heap specialization [110], allocation-site abstraction [24], and recency abstraction [12], we have:

$$Loc = C' \times N \times \{*, @\}$$

where C' is a set of context identifiers used for heap specialization. C' will be described in more detail when discussing context sensitivity strategies in Section 3.2.4. The allocation-site abstraction is incorporated by the N component that refers to the nodes of the program. Allocation-site abstraction means that objects defined at different nodes are represented by different abstract objects. The last component $\{*, @\}$ is used for recency abstraction. Recency abstraction abstracts an object in a most recent object (a singleton) represented by @ and a summary object represented by $*$. When

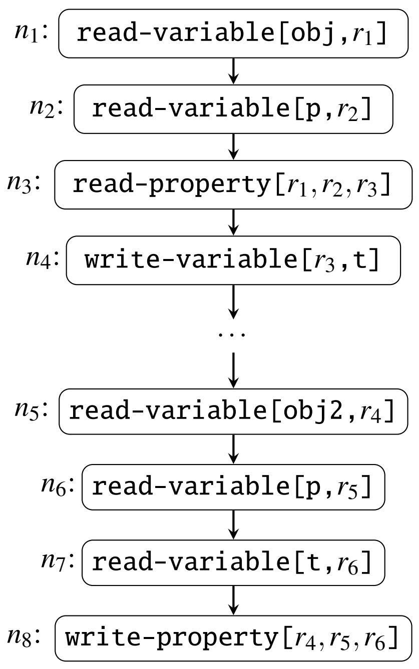

n_1: read-variable[obj,r_1]
n_2: read-variable[p,r_2]
n_3: read-property[r_1,r_2,r_3]
n_4: write-variable[r_3,t]
...
n_5: read-variable[obj2,r_4]
n_6: read-variable[p,r_5]
n_7: read-variable[t,r_6]
n_8: write-property[r_4,r_5,r_6]

Figure 3.3: Control flow graph for the program in Fig. 3.1.

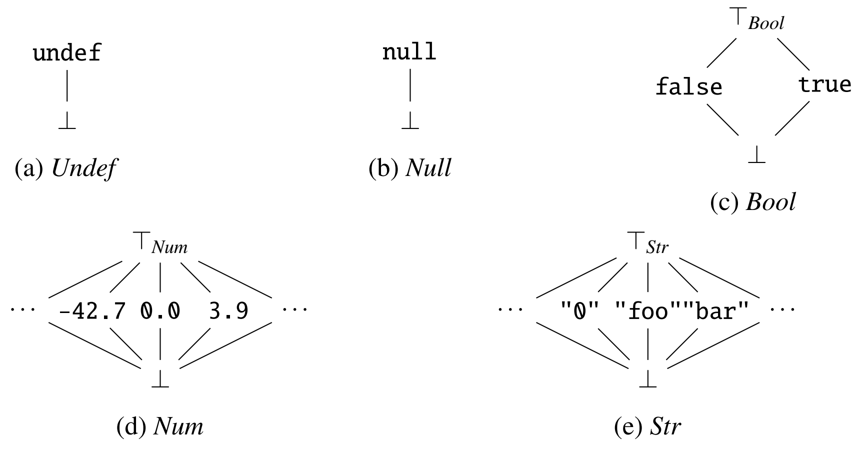


Figure 3.4: Lattices for primitive types in TAJS.

using recency abstraction we know that a singleton object, i.e., an @ object represent only a single object, and therefore TAJS can apply strong updates on such objects. Without recency abstraction, all objects might represent multiple objects and therefore all updates would have to be weak updates.

The lattice of abstract objects in TAJS is defined as:

$$Obj = ((P \times Loc) \hookrightarrow Value \times Absent \times Attributes \times Modified) \times ScopeChain$$

Since functions are a special kind of objects in JavaScript, TAJS represents them as abstract objects. P is the infinite set of all possible property names including four special property names $[[\mathit{Prototype}]]$, $[[\mathit{Value}]]$, *default_num* and *default_other*. $[[\mathit{Prototype}]]$ and $[[\mathit{Value}]]$ correspond to the internal properties defined in the ECMAScript standard. $[[\mathit{Prototype}]]$ is the internal prototype link, i.e., the same as the `__proto__` property presented in Section 2.1.3, and $[[\mathit{Value}]]$ is the internal value that is used by wrapper objects to refer to the primitive value that they wrap. The *default_num* property represents all numeric properties except those in the map, and the *default_other* property represents all non-numeric properties except those in the map. These properties are used when writing to an unknown property name. The *Loc* component is used for supporting symbols as property names. The *Value* lattice describes abstract values and is defined as the product of lattices for the primitive types and the powerset lattice of object addresses:

$$\mathit{Value} = \mathit{Undef} \times \mathit{Null} \times \mathit{Bool} \times \mathit{Num} \times \mathit{String} \times \mathscr{P}(\mathit{Loc})$$

The primitive lattices are similar to the ones shown in Fig. 3.4[1]. The primitive lattices in the *Value* domain correspond to the primitive types defined in Section 2.1.1 with the exception of the `BigInt` type that TAJS does not support. The $\mathscr{P}(\mathit{Loc})$ component describes the object addresses that a value can be and corresponds to the `symbol`, `function` and `object` types.

We refer to the original TAJS paper [70] for descriptions of the *Absent*, *Attributes* and *Modified* components in the abstract object, since they are orthogonal to the JavaScript features presented in Section 2.1 and the extensions to TAJS that will be presented later in this section. The *ScopeChain* component describes the scope chain for a function object at the point of creation. The *ScopeChain* component will be explained in more details after we explain how TAJS models the execution context at calls.

In the *State* lattice, the *Stack* component is defined as:

$$\begin{aligned} \mathit{Stack} &= (R \to \mathit{Value}) \times \mathscr{P}(\mathit{ExecutionContext}) \\ \mathit{ExecutionContext} &= \mathit{ScopeChain} \times \mathit{Loc} \times \mathit{Loc} \\ \mathit{ScopeChain} &= \mathscr{P}(\mathit{Loc})^* \end{aligned}$$

The first component provides abstract values for the temporary registers, and the second component defines the execution context, which is the ECMAScript term for a stack frame. The execution context consists of a scope chain that is a sequence of sets containing object addresses. The scope chain is used for looking up variables as explained in Section 2.1.4. The first *Loc* component specifies the `this` object, and the second *Loc* component describes the variable object for the execution context. In most cases, the last element of the scope chain is the same object as the variable object, however, it is not always the case when analyzing programs using the `with`-statement, since it can dynamically update the scope chain.

[1]The *Num* and *Str* lattices are more expressive in TAJS, but to understand the contributions of this thesis, it suffices to think of the lattices as constant propagation lattices.

The two $\mathscr{P}(Loc)$ components in the *State* lattice are needed due to the use of recency abstraction in interprocedural analysis. This will be explained further when explaining how TAJS handles interprocedural analysis. A simplified version of the *MustEquals* component is defined as:

$$MustEquals = R \to \mathscr{P}(Loc \times P)$$

It describes for each register $r \in R$ what object properties must have the same value as r. This will be exploited in the value partitioning technique presented in Chapter 8.

The primitive lattices in the *Value* lattice are ordered as shown in Fig. 3.4, the product and map lattices are ordered pointwise as explained in Section 2.3.1, and powerset lattices are ordered by subset inclusion, except for the last $\mathscr{P}(Loc)$ in *State* and the powerset lattice used in *MustEquals* that are ordered by $\supseteq$ instead of $\subseteq$.

We have now explained the core parts of the TAJS lattice to ease the understanding of the contributions presented in the research papers in Chapters 6 to 9.

3.2.3 Transfer Functions

A monotone transfer function is defined for each of the flow graph nodes (recall Fig. 8.2) and for the functions defined in the ECMAScript standard library. In this section, we will explain the transfer functions for dynamic property read and dynamic property write for understanding how TAJS analyzes the code in Fig. 3.1. Furthermore, we will explain the transfer function for a call node and explain how interprocedural analysis is handled by TAJS.

Read property nodes We start by presenting the transfer function for a read property node `read-property[`r_1, r_2, r_3`]`, where r_1, r_2 have the values v_{obj}, v_{prop} that are the values of the object to read from and the property name to read, respectively. The resulting value should be stored in r_3. If v_{obj} is not an object, then it is coerced into one. If the value might be either `null` or `undefined`, then a warning that a `TypeError` might occur is produced. The value v_{prop} is coerced into a string as explained in Section 2.1.5. The read follows the prototype chain on all the objects represented by v_{obj}. The property is first read from the property map at the represented objects, and if the property is not definitely there, then the read traverses the prototype chain. Furthermore, if the property is not in the property map, then the value of the *default_num* or *default_other* properties are read as well depending on whether the property name is numeric or not. If any of the properties read is a getter, then an implicit call is made to the getter function, and the value returned from the getter will be treated as the value being read. If v_{prop} is an imprecise string, then the special properties *default_num* and *default_other* are read together with all properties in the property map.[2] The result becomes the join of the values read for all objects in the traversed prototype chain that might have the property. Furthermore, if the analysis is not sure that the property exists, the value `undef` is joined to the result as well. The result is stored in r_3.

[2]The string domain in TAJS can distinguish imprecise numeric strings from other strings, so often TAJS only has to read either *default_num* or *default_other* when the string value is imprecise.

Write property nodes We now explain the transfer function for a write property node `write-property[`r_1, r_2, r_3`]`, where r_1, r_2, r_3 have the values $v_{obj}, v_{prop}, v_{value}$. These values are the values of the object to write to, the property name to write to, and the value to write, respectively. If v_{obj} is not an object, then it is coerced into one, and if the object might be either `null` or `undefined`, then a warning about a potential `TypeError` is raised. If v_{prop} is not a string, then it is coerced into one. Next, for all matching object properties, if the object property is a setter, then we implicitly call the setter function with v_{value} as argument, and otherwise we write v_{value} to the object property. If v_{prop} is imprecise, then the write happens to all properties for all the objects described by v_1 including the *default_num* and *default_other* properties. For this reason, the analysis does not have to read the special properties at property reads with a precise property name that exists in the object that is being read from. Note that TAJS can make strong updates when v_{obj} represents a single singleton object and v_{prop} represents a single string value.

Interprocedural analysis Now, we explain how TAJS handles interprocedural analysis, by explaining the transfer function for a call node `call[`$r_f, r_{res}, r_{this}, r_1, \ldots, r_n$`]` where the registers except r_{res} have the corresponding values v_f, v_{this}, v_1, …, v_n. The value v_f is the function value, v_{this} is the value for **`this`**, and v_1 to v_n are the argument values. First, all function objects are extracted from v_f. For each function object, a call edge is added to the entry of the function. A new activation object is then created, and the parameter passing is modeled by writing the arguments to the corresponding properties of the newly created activation object. The transfer function also models the creation of a new execution context used for analyzing the function body. The scope chain for the execution context is the result of pushing the new activation object onto the *ScopeChain* component from the abstract function object. The value for **`this`** in the execution context is v_{this}, and the variable object is the newly created activation object. At returns the return value is propagated to the r_{res} register in the successor of the `call` node being analyzed.

The two $\mathscr{P}(Loc)$ components in the *State* lattice describe maybe-summarized and definitely-summarized object addresses. These are needed to propagate the switching between singleton and summary objects that might have happened when analyzing the function body of the function currently being analyzed. For example, if an object is summarized inside the function, the summarization should also happen in the caller-state when the return-state is propagated to the caller-state. TAJS does a few more things to increase precision as explained in the original TAJS paper [70], but those techniques are not important for the contributions of this thesis.

Analyzing a correlated read/write pair We will now show how TAJS analyzes the control flow graph presented in Fig. 3.3. In this example, we will assume that the analysis state before n_1 is the one shown in Fig. 3.5. For this example, we only show the $P \times Loc \hookrightarrow Value$ part of abstract objects and ignore the two $\mathscr{P}(Loc)$ components in *State*. Furthermore, when representing abstract values, we omit the $\perp$ components,

$$\begin{aligned}
\mathit{state} &= (\mathit{store}, \mathit{stack}, \mathit{mustEquals}) \\
\mathit{store} &= [\\
&\qquad \ell_{\text{GLOBAL}} \to [\\
&\qquad\qquad \texttt{obj} \to \{\ell_{obj}\}, \\
&\qquad\qquad \texttt{obj2} \to \{\ell_{obj2}\}, \\
&\qquad\qquad \texttt{p} \to \top_{Str}, \\
&\qquad\qquad \texttt{t} \to \texttt{undef} \\
&\qquad], \\
&\qquad \ell_{obj} \to [\\
&\qquad\qquad \texttt{foo} \to \{\ell_{f1}\}, \\
&\qquad\qquad \texttt{bar} \to \{\ell_{f2}\} \\
&\qquad], \\
&\qquad \ell_{obj2} \to [], \\
&\qquad \ell_{f1} \to \ldots, \\
&\qquad \ell_{f2} \to \ldots \\
&], \\
\mathit{stack} &= (\mathit{registers}, \mathit{executionContext}) \\
\mathit{registers} &= [] \\
\mathit{executionContext} &= \{[\{\ell_{\text{GLOBAL}}\}], \ell_{\text{GLOBAL}}, \ell_{\text{GLOBAL}}\} \\
\mathit{mustEquals} &= []
\end{aligned}$$

Figure 3.5: Initial state for the correlated read/write pair example.

so $(\texttt{undef}, \bot, \bot, \bot, \bot, \{\})$ will be written as `undef` and $(\bot, \bot, \bot, \bot, \bot, \{\ell\})$ will be written as $\{\ell\}$. The initial state contains the object addresses $\ell_{\text{GLOBAL}}, \ell_{obj}, \ell_{obj2}, \ell_{f1}$ and ℓ_{f2}. The object address ℓ_{GLOBAL} represents the global object that has the variables `obj`, `obj2`, `p`, and `t` as properties, since the code is in the global scope. The initial state specifies that `obj` is of the form `{foo: f1, bar: f2}`, where `f1` and `f2` are the object addresses ℓ_{f1} and ℓ_{f2}. We do not specify the abstract objects for ℓ_{f1} and ℓ_{f2}, since their specifics are unrelated to analyzing the correlated read/write pair in the example. The object at ℓ_{obj2} is an object without any properties. The variable `p` is the $\top$-element in the *Str* lattice. The *executionContext* shows that the code is run in the global context, since the *ScopeChain* component is $[\{\ell_{\text{GLOBAL}}\}]$. The **`this`** keyword also refers to ℓ_{GLOBAL}, and ℓ_{GLOBAL} is also the variable object.

The transfer function for `read-variable[obj,`r_1`]` in Fig. 3.3 reads the variable `obj` by reading the `obj` property on objects in the *ScopeChain* component in *executionContext*. The analysis therefore reads the `obj` property of ℓ_{GLOBAL} and writes

$$
\begin{array}{rl}
\mathit{store} = [& \\
 & \ell_{\text{GLOBAL}} \rightarrow [\\
 & \quad \ldots \\
 & \quad \mathtt{t} \rightarrow (\mathtt{undef}, \{\ell_{f1}, \ell_{f2}\}) \\
 &], \\
 & \ldots \\
], & \\
\mathit{registers} = [& \\
 & r_1 \rightarrow \{\ell_{obj}\}, \\
 & r_2 \rightarrow \top_{Str}, \\
 & r_3 \rightarrow (\mathtt{undef}, \{\ell_{f1}, \ell_{f2}\}) \\
], & \\
\mathit{mustEquals} = [& \\
 & r_1 \rightarrow \{(\ell_{\text{GLOBAL}}, \texttt{"obj"})\}, \\
 & r_2 \rightarrow \{(\ell_{\text{GLOBAL}}, \texttt{"p"})\} \\
] &
\end{array}
$$

Figure 3.6: Abstract state after line 107 in Fig. 3.1. "..." means that the rest of the mappings from Fig. 3.5 remain unchanged.

it to the register r_1. After the transfer function, the abstract state is updated, so the *registers* component becomes $[r_1 \rightarrow \{\ell_{obj}\}]$, and the *mustEquals* component becomes $[r_1 \rightarrow \{(\ell_{\text{GLOBAL}}, \texttt{"obj"})\}]$. The read `read-variable[p,`$r_2$`]` similarly adds $r_2 \rightarrow \top_{Str}$ to the *registers* component and adds $r_2 \rightarrow \{(\ell_{\text{GLOBAL}}, \texttt{"p"})\}$ to *mustEquals*. The `read-property[`r_1, r_2, r_3`]` node then reads the property name stored in r_2 from the object in r_1 to r_3. Since r_2 is an imprecise string, all properties from ℓ_{obj} is read, meaning that the *registers* component is extended by $r_3 \rightarrow (\mathtt{undef}, \{\ell_{f1}, \ell_{f2}\})$[3]. The value `undef` is also read, since the read might be from a property that does not exist in `obj`. At the `write-variable[`r_3`,t]` node, the value of r_3 is then written to the variable `t`. Since the variable object (the third component in the execution context) is ℓ_{GLOBAL}, the `t` mapping for ℓ_{GLOBAL} in the *store* is updated to $(\mathtt{undef}, \{\ell_{f1}, \ell_{f2}\})$. A summary of the changes to the abstract state corresponding to the code `var t = obj[p]` can be seen in Fig. 3.6.

After analyzing the three read variable nodes, `read-variable[obj2,`r_4`]`, `read-variable[p,`r_5`]`, and `read-variable[t,`r_6`]`, the *registers* component is ex-

[3]Technically, the read should also use the prototype chain, so the value read can also be the standard library functions from `Object.prototype`, but that is omitted in this example, since it is not important for the example.

$$
\begin{aligned}
\mathit{store} = [\ & \\
& \ell_{\mathit{obj2}} \to [\\
& \quad \mathit{default\_num} \to (\texttt{undef}, \{\ell_{\mathit{f1}}, \ell_{\mathit{f2}}\}), \\
& \quad \mathit{default\_other} \to (\texttt{undef}, \{\ell_{\mathit{f1}}, \ell_{\mathit{f2}}\}) \\
&], \\
& \ldots \\
], & \\
\mathit{registers} = [\ & \\
& \ldots \\
& r_4 \to \{\ell_{\mathit{obj2}}\}, \\
& r_5 \to \top_{\mathit{Str}}, \\
& r_6 \to (\texttt{undef}, \{\ell_{\mathit{f1}}, \ell_{\mathit{f2}}\}), \\
], & \\
\mathit{mustEquals} = [\ & \\
& \ldots \\
& r_4 \to \{(\ell_{\text{GLOBAL}}, \texttt{"obj2"})\}, \\
& r_5 \to \{(\ell_{\text{GLOBAL}}, \texttt{"p"})\}, \\
& r_6 \to \{(\ell_{\text{GLOBAL}}, \texttt{"t"})\} \\
] &
\end{aligned}
$$

Figure 3.7: Abstract state after line 109 in Fig. 3.1. "..." means that the rest of the mappings from Fig. 3.6 remain unchanged.

tended by $r_4 \to \{\ell_{\mathit{obj2}}\}$, $r_5 \to \top_{\mathit{Str}}$, and $r_6 \to (\texttt{undef}, \{\ell_{\mathit{f1}}, \ell_{\mathit{f2}}\})$. The *mustEquals* component is similarly extended with $r_4 \to \{(\ell_{\text{GLOBAL}}, \texttt{"obj2"})\}$, $r_5 \to \{(\ell_{\text{GLOBAL}}, \texttt{"p"})\}$, and $r_6 \to \{(\ell_{\text{GLOBAL}}, \texttt{"t"})\}$. Next, we analyze the node `write-property[`r_4, r_5, r_6`]`. Since r_5 is an imprecise string, the value in r_6 is written to all properties of the object in r_4, which in TAJS is modeled as writes to the two special properties *default_num* and *default_other*. The summary of the state changes for analyzing the code `obj2[p] = t` is shown in Fig. 3.7. The abstract state now has an imprecise value for `obj2`, since all the properties of `obj1` have been mixed together in `obj2`. Section 3.3.2 shows how the techniques presented Chapters 7 and 8 can analyze this example precisely by modifying the transfer functions for property reads and writes.

3.2.4 Context Sensitivity

In this section, We will define the *C* component of the *AnalysisLattice* that controls the degree of context sensitivity. The context sensitivity applied by TAJS in the original paper was based on object sensitivity [102]. In object sensitivity states are distinguished based on different values for `this`, meaning that

$$C = \mathscr{P}(Loc)$$

The context sensitivity that this thesis builds upon is the object sensitivity combined with the context sensitivity strategies presented with the static determinacy technique [10]. We will here provide a very brief description of the resulting context sensitivity strategy and refer to Andreasen and Møller [10] for motivation and more details. The context sensitivity component of the *AnalysisLattice* then becomes:

$$C = \mathscr{P}(Loc) \times (A \hookrightarrow Value) \times (B \hookrightarrow Value) \times (R \hookrightarrow Value)$$

where A is the set of all function parameters, and it encodes parameter sensitivity for selected arguments. Parameter sensitivity is applied for all calls with a constant value or a single object address as argument. B is the set of all local variables appearing in the program, and it is used to encode loop unrolling. Loop unrolling is used at a loop if a local variable with a concrete numeric value according to the abstract state appears syntactically in the condition, the loop is not a nested loop, and the variable is involved in a dynamic property read. TAJS performs up to 50 loop unrollings for such loops. The R component is the TAJS registers, and this last component is used to incorporate for-in specialization. JavaScript has the for-in statement `for (x in obj) { ... }` that iterates over each property in `obj`. The variable `x` represents the property name of the property being iterated over. For-in specialization makes sure that all the iterations are analyzed in different contexts.

The static determinacy paper [10] also extended TAJS with heap specialization, and we will show how it works by defining the C' component from *Loc*:

$$C' = (A \hookrightarrow Value) \times (B \hookrightarrow Value) \times (R \hookrightarrow Value)$$

There are four heuristics for applying heap specialization, and when applied, the heap specialization inherits the components from the context in which the objects are created. Heap specialization is applied when creating one of the following objects: 1) function objects, 2) new objects on a function invocation such as the activation record, 3) wrapper objects in type coercions, and 4) object literals in a for-in loop or object literals that syntactically contains a function parameter that is present in the current parameter sensitivity component.

We have now described the core parts of TAJS that are relevant to get a better understanding of the contributions of this thesis.

3.3 Contributions

This section explains the techniques and contributions of the research papers presented in Chapters 6–9 that improves state-of-the-art for sound static analysis of JavaScript and Node.js. Section 3.3.1 summarizes the paper in Chapter 6 that presents a technique

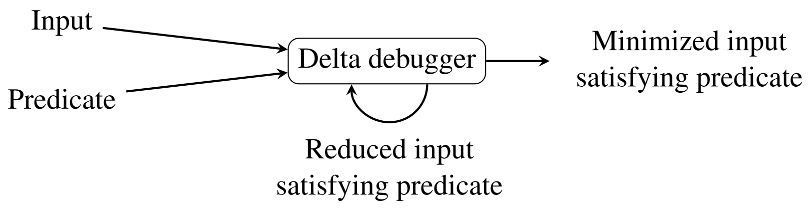

Input
Predicate
Delta debugger
Minimized input satisfying predicate
Reduced input satisfying predicate

Figure 3.8: Delta debugging.

for understanding limitations of a static analysis. Section 3.3.2 summarizes the papers in Chapters 7 and 8 that present techniques for analyzing FCT patterns, while Section 3.3.3 summarizes the research paper in Chapter 9 that presents a technique for improving scalability for a static taint analysis.

3.3.1 Understanding Analysis Limitations

This subsection summarizes the contributions from the paper presented in Chapter 6. The paper suggests automatic techniques for detecting and understanding soundness bugs, and it also provides an automatic technique for finding root causes of precision loss in a static analysis.

Delta debugging If we have a large JavaScript program and the static analysis times out when analyzing the program, it can be difficult to know why the analysis does not terminate within a reasonable amount of time. For large programs, it is not feasible to manually understand all steps in the analysis. We therefore propose to use delta debugging [148] to understand the analysis limitations. Fig. 3.8 illustrates how delta debugging works. It takes some input and a predicate where the predicate should hold for the input. The delta debugger then tries to reduce the input and checks whether the predicate still holds for the reduced input. If it holds, the delta debugger attempts to reduce the reduced program further. If the predicate does not hold, another reduction is attempted on the program where the current reduction has not been applied. This process keeps running in this manner until the input cannot be reduced further, while it is still satisfying the predicate. The output of the delta debugger is the reduced input that cannot be reduced further. As an example, the input can be a JavaScript program, and the predicate can be that TAJS should time out when analyzing the program. The output is then a small program that satisfies the predicate, i.e., a small program where the analysis still times out. One limitation with delta debugging is that since the reduced program does not preserve the semantics of the original program, the reduced program might contain problems that are not present in the original code. Later in this subsection, we therefore also suggest another technique for understanding precision issues.

Soundness testing Another challenge when designing a sound static analysis is how to ensure that the implementation is sound. We frequently update the analysis, so we want a fully automatic way to detect soundness bugs, such that the approach does not have to be updated every time the analysis is updated. For this purpose, we propose to perform *soundness testing*. Soundness testing uses a dynamic analysis to record dataflow facts observed in concrete executions, and check that the analysis results overapproximate the concrete executions. More specifically, the dynamic analysis produces a *value log* that contains all the dataflow facts related to variable reads and writes, property reads and writes, and call expressions that have been observed during the dynamic analysis. The analysis result is then soundness tested by checking that it overapproximates the dataflow facts that were recorded by the dynamic analysis. For example, for a function call we check that the static analysis has a call to the function that was called concretely, and that the arguments in the static analysis also overapproximate the arguments of the concrete call. Any dataflow fact that is not overapproximated is a soundness failure.

A single soundness bug might result in many soundness failures making it difficult to understand the soundness bug. To better understand the soundness bug, we propose to combine soundness testing and delta debugging. The predicate for delta debugging is then that the soundness tests should fail. The result is a minimal program that fails soundness testing. From the minimized program it is typically easy to understand the soundness bug. It might be the case that the minimized program is unsound for another reason than the original program, but such a soundness error should be fixed anyway. After the soundness bug has been fixed, the technique could be rerun, and the actual soundness bug will likely be present in the new minimized program.

Blended analysis We will now present a technique that can perform soundness testing on applications that an analysis is too imprecise to analyze. We can artificially increase precision of an analysis by using *blended analysis* [44]. In blended analysis, the analysis state is refined based on the facts observed concretely. We can use the facts from the value log to refine the analysis state. For example, at a dynamic property write, the analysis simply performs the same write as was observed concretely. Thereby, the analysis avoids introducing precision loss if, for instance, the property name is imprecise. Using blended analysis is obviously not sound, but it allows analysis of some JavaScript programs that TAJS cannot analyze without, and when combined with soundness testing, we can do soundness testing for JavaScript programs that TAJS cannot analyze yet. Since blended analysis and the soundness tester use the same value log, we do not get any spurious soundness failures due to the use of blended analysis.

Find root causes of imprecision We can also combine blended analysis and delta debugging to detect the locations that an analysis needs to handle precisely to analyze the program. This works by only allowing blended analysis at certain locations, and then use delta debugging to minimize those locations with the predicate that the analysis should succeed. The initial input is the set of all locations in the JavaScript program, and the output is then a minimal set of locations where blended analysis

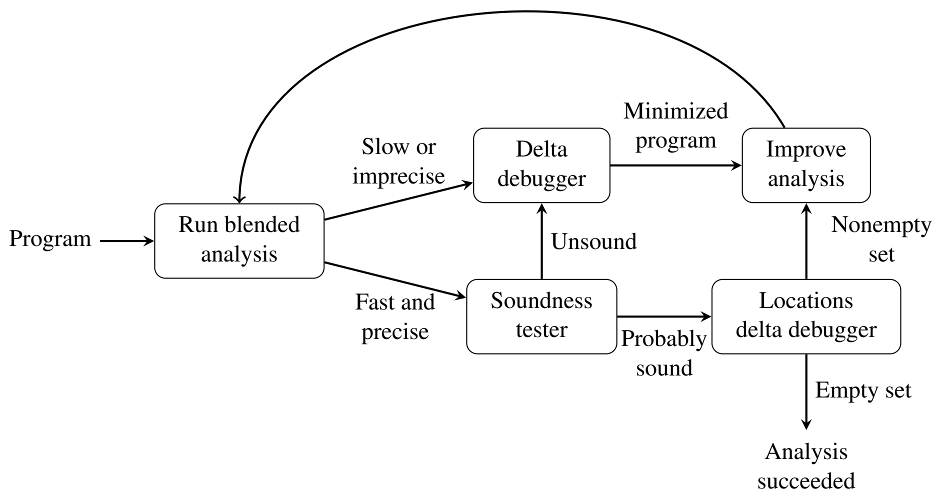


Figure 3.9: Diagram showing recommended workflow for developing a static analysis.

is needed to analyze the program. This resulting set tells us two things, first that we need to handle these locations precisely, and second, if we can handle these locations precisely, then we can analyze the program. Using this technique, we can therefore get an overview of the analysis bottlenecks that prevent analysis of a program.

Combined workflow Based on the techniques presented, we propose a workflow as shown in Fig. 3.9 when implementing a static analysis. The input is the program to analyze. First we run blended analysis on the program, and if the analysis is slow or imprecise, then we use delta debugging to find a minimal program that cannot be analyzed using blended analysis. Based on this minimized program, we can find the analysis bottlenecks and improve the analysis. Then we can rerun the improved analysis using blended analysis. If the blended analysis succeeds, then we do a soundness test of the analysis results. If the analysis is unsound, then we use delta debugging to find a minimized program with the soundness bug and then fix the soundness bug. Then the pipeline starts over from the beginning again. When the analysis results pass the soundness tests, we use delta debugging to find the minimal set of locations where blended analysis is needed to analyze the program. If the set is non-empty, then we use those locations to understand precision bottlenecks and improve the analysis, and start over again. If the result is an empty set, then the analysis can soundly analyze the program without the use of blended analysis. Using this workflow the analysis designer gets a good starting point for understanding how to improve the analysis, with respect to both soundness and precision.

3.3.2 Increasing Analysis Precision for FCT Patterns

The papers in Chapters 7 and 8 present techniques for precisely analyzing FCT patterns, free variables used inside FCT patterns, and type predicate functions. This subsection explains on a high-level how the techniques presented in the two papers can analyze FCT patterns precisely. We explain the techniques on the example code from Fig. 3.1 using the initial state presented in Fig. 3.5.

Demand-driven value refinement The first paper presents the technique *demand-driven value refinement*. At imprecise `write-property[`r_1, r_2, r_3`]` nodes, as the one in the example program, the technique tries to make the write precise, even though both the property name in r_2 and the value to write in r_3 are imprecise. Consider the abstract state from the previous example before the node `write-property[`r_4, r_5, r_6`]` in Fig. 3.3. The state is shown in Fig. 3.6 extended with the *registers* and *mustEquals* component from Fig. 3.7. The new transfer function for property writes checks if both r_5 and r_6 are imprecise, meaning that r_5 is not a constant string and r_6 can point to multiple objects or functions. Since r_5 has the value $\top_{Str}$ and r_6 has the value $($`undef`$, \{\ell_{f1}, \ell_{f2}\})$, both r_5 and r_6 are imprecise. The technique splits the value in r_6 into precise values containing only a single type and only a single object address. This means that the value is split into the three values, `undef`, $\{\ell_{f1}\}$, and $\{\ell_{f2}\}$. For each of these values, v_i, we query a goal-directed backwards abstract interpreter based on separation logic to provide a sound value for r_5, with the condition that the value of r_6 is v_i. One of the queries is: "What is the value of r_5 when the value of r_6 is $\{\ell_{f1}\}$?".

Fig. 3.10 shows in a simplified manner how the backwards analysis reasons to answer the query. It starts at the dynamic property write with the constraint $r_5 = \text{RES} \wedge r_6 = \{\ell_{f1}\}$, where RES represents the value that the backwards analysis should infer. When analyzing a read-variable node, the register storing the result is simply replaced with the variable that is being read, so when analyzing `read-variable[t,`r_6`]`, r_6 is replaced with the variable `t`, and similarly for the other read-variable nodes. When traversing over the node `write-variable[`r_3`,t]`, the variable `t` is replaced by r_3. At the node `read-property[`r_1, r_2, r_3`]`, r_3 is again replaced by $r_1[r_2]$, indicating that the value is the result of reading the property name in r_2 from the object in r_1. Now, when analyzing the node `read-variable[p,`r_2`]`, r_2 is replaced by `p`, and when analyzing the node `read-variable[obj,`r_1`]`, r_1 is replaced by `obj`. The constraint to be solved is now $\texttt{p} = \text{RES} \wedge r_1[\texttt{p}] = \{\ell_{f1}\}$. The backwards analysis can answer the query using the analysis state from before analyzing the `read-variable[obj,`r_1`]` node, which is the state shown in Fig. 3.5. Since `obj` has the value $\{\ell_{obj}\}$ and ℓ_{obj} points to the abstract object, $[\texttt{foo} \rightarrow \{\ell_{f1}\}, \texttt{bar} \rightarrow \{\ell_{f2}\}]$, the backwards analysis infers that `p` can only be the string `"foo"`, since that is the only property with the value $\{\ell_{f1}\}$. Similarly, the backwards analysis infers that the value $\{\ell_{f2}\}$ can only be written to the `bar` property, meaning that the values $\{\ell_{f1}\}$ and $\{\ell_{f2}\}$ are not mixed together in `obj2`.

Demand-driven value refinement also works across functions by using the call graph currently computed by TAJS. Since the backwards analysis relies on dataflow facts from other nodes than just the predecessors of the node where the query is issued,

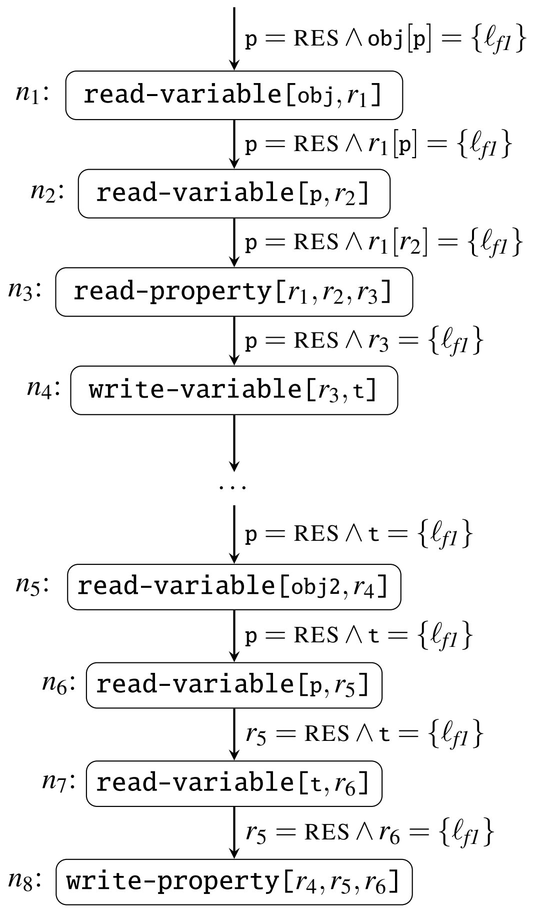

$\mathtt{p} = \mathrm{RES} \land \mathtt{obj}[\mathtt{p}] = \{\ell_{f1}\}$
n_1: read-variable[obj, r_1]
$\mathtt{p} = \mathrm{RES} \land r_1[\mathtt{p}] = \{\ell_{f1}\}$
n_2: read-variable[p, r_2]
$\mathtt{p} = \mathrm{RES} \land r_1[r_2] = \{\ell_{f1}\}$
n_3: read-property[r_1, r_2, r_3]
$\mathtt{p} = \mathrm{RES} \land r_3 = \{\ell_{f1}\}$
n_4: write-variable[r_3, t]
...
$\mathtt{p} = \mathrm{RES} \land \mathtt{t} = \{\ell_{f1}\}$
n_5: read-variable[obj2, r_4]
$\mathtt{p} = \mathrm{RES} \land \mathtt{t} = \{\ell_{f1}\}$
n_6: read-variable[p, r_5]
$r_5 = \mathrm{RES} \land \mathtt{t} = \{\ell_{f1}\}$
n_7: read-variable[t, r_6]
$r_5 = \mathrm{RES} \land r_6 = \{\ell_{f1}\}$
n_8: write-property[r_4, r_5, r_6]

Figure 3.10: Backwards analysis solving the query: "What is the value of r_5 when the value of r_6 is $\{\ell_{f1}\}$?". The query is issued at node `write-property`[r_4, r_5, r_6] in the example program.

we need to extend the dependency map introduced in Section 2.3.2. We add that n also depends on n' when the backwards analysis relies on a dataflow fact at node n', while solving a query issued at node n.

The evaluation in the paper shows that this technique enables precise analysis for the most popular JavaScript libraries *lodash* and *underscore*, which is an important first step towards sound static analysis for Node.js applications.

However, one drawback about this technique is that the backwards analysis is a completely separate analysis, so it needs to support the entire JavaScript language and the standard library on its own. This means that *two* analyses now have to be extended when adding support for new JavaScript features. It is therefore desirable to have a more lightweight approach capable of analyzing FCT patterns precisely. The technique *value partitioning* that we present in Chapter 8 is such a lightweight analysis that can analyze FCT patterns with high precision.

$$
\begin{aligned}
\mathit{store} = [& \\
& \ell_{\text{GLOBAL}} \to [\\
& \qquad \ldots \\
& \qquad \texttt{p} \to [t_1 \to \texttt{"foo"}, t_2 \to \texttt{"bar"}, t_3 \to \top_{Str}], \\
& \qquad \texttt{t} \to [t_1 \to \{\ell_{f1}\}, t_2 \to \{\ell_{f2}\}, t_3 \to \texttt{undef}] \\
&], \\
& \ldots \\
], & \\
\mathit{registers} = [& \\
& r_1 \to \{\ell_{obj}\}, \\
& r_2 \to [t_1 \to \texttt{"foo"}, t_2 \to \texttt{"bar"}, t_3 \to \top_{Str}], \\
& r_3 \to [t_1 \to \{\ell_{f1}\}, t_2 \to \{\ell_{f2}\}, t_3 \to \texttt{undef}] \\
], & \\
\mathit{mustEquals} = [& \\
& r_1 \to \{(\ell_{\text{GLOBAL}}, \texttt{"obj"})\}, \\
& r_2 \to \{(\ell_{\text{GLOBAL}}, \texttt{"p"})\} \\
] &
\end{aligned}
$$

Figure 3.11: Abstract state after line 107 in Fig. 3.1 using value partitioning. "..." means that the rest of the mappings from Fig. 3.5 remain unchanged.

Value partitioning Value partitioning is a lightweight approach that allows an analysis to precisely and efficiently reason about certain relations between different abstract values. We will explain the approach through the same example as before. The idea behind value partitioning is similar to that of state partitioning in the CompAbs analyzer. State partitioning extends the abstract domain of an analysis to have multiple abstract states per program location, similar to the effect of context sensitivity. Value partitioning instead allows to have multiple abstract values for the same variable, property or register inside the same abstract state. It is much cheaper to create partitions at the value level instead of at the state level, since the *Value* lattice is much simpler than the *State* lattice. Since partitioning at the value level is very cheap, we can apply it in many more locations and do not need a pre-analysis to find what locations to apply the partitioning technique in. Value partitioning therefore avoids the fragility of state partitioning, and at the same time value partitioning is much more efficient. Again, consider the flow graph in Fig. 3.3, and the initial state shown in Fig. 3.5. We apply value partitioning when analyzing the `read-property[`r_1, r_2, r_3`]` node, since the value in r_2 is an imprecise string. Value partitioning partitions the property name in r_2 to a partition for each property name in the object in r_1 and

$$
\begin{array}{rl}
\mathit{store} = [& \\
& \ell_{obj2} \rightarrow [\\
& \quad \mathit{default\_num} \rightarrow \texttt{undef}, \\
& \quad \mathit{default\_other} \rightarrow \texttt{undef}, \\
& \quad \texttt{foo} \rightarrow (\texttt{undef}, \{\ell_{f1}\}), \\
& \quad \texttt{bar} \rightarrow (\texttt{undef}, \{\ell_{f2}\}) \\
&], \\
& \ldots \\
], & \\
\mathit{registers} = [& \\
& \ldots \\
& r_4 \rightarrow \{\ell_{obj2}\}, \\
& r_5 \rightarrow [t_1 \rightarrow \texttt{"foo"}, t_2 \rightarrow \texttt{"bar"}, t_3 \rightarrow \top_{Str}], \\
& r_6 \rightarrow [t_1 \rightarrow \{\ell_{f1}\}, t_2 \rightarrow \{\ell_{f2}\}, t_3 \rightarrow \texttt{undef}] \\
], & \\
\mathit{mustEquals} = [& \\
& \ldots \\
& r_4 \rightarrow \{(\ell_{\text{GLOBAL}}, \texttt{"obj2"})\}, \\
& r_5 \rightarrow \{(\ell_{\text{GLOBAL}}, \texttt{"p"})\}, \\
& r_6 \rightarrow \{(\ell_{\text{GLOBAL}}, \texttt{"t"})\} \\
] &
\end{array}
$$

Figure 3.12: Abstract state after line 109 in Fig. 3.1 using value partitioning. "..." means that the rest of the mappings from Fig. 3.11 remain unchanged.

one partition for the properties not in r_1. This means that r_2 now becomes the value $[t_1 \rightarrow \texttt{"foo"}, t_2 \rightarrow \texttt{"bar"}, t_3 \rightarrow \top_{Str}]$, where t_1, t_2 and t_3 are identifiers for the different partitions. The partition identifiers are formally defined in the paper, but here we will only explain what is needed to understand the example. Each partition represents different executions, and t_1 represents the executions where the value of r_2 was `"foo"` at the read, t_2 represents executions where the value of r_2 was `"bar"`, and t_3 represents all other executions. We also exploit the must equals information and write the same value to `p`, since $r_2 \rightarrow \{(\ell_{\text{GLOBAL}}, \texttt{"p"})\}$ is in the *mustEquals* component. The property read now happens in each of the partitions, so the value for r_3 becomes $[t_1 \rightarrow \{\ell_{f1}\}, t_2 \rightarrow \{\ell_{f2}\}, t_3 \rightarrow \texttt{undef}]$. The t_3 partition for r_3 has the value `undef` since t_3 represents all the executions where r_2 is a property that is not in the object in r_1. The value in r_3 is then written to the variable t. The resulting state is shown in Fig. 3.11.

The `read-variable` nodes behave as in the ordinary analysis, and therefore we will not explain them again, but jump directly to explaining how value partitioning works at the node `write-property[`r_4, r_5, r_6`]`. At this point r_5 has the value $[t_1 \to$ `"foo"`$, t_2 \to$ `"bar"`$, t_3 \to \top_{Str}]$, and r_6 has the value $[t_1 \to \{\ell_{f1}\}, t_2 \to \{\ell_{f2}\}, t_3 \to$ `undef`$]$. Since both values are partitioned with the same partitions t_1, t_2, and t_3, we can exploit the partitions by only writing the value in the t_1 partition of r_6 to the property name in the t_1 partition of r_5, and similarly for the t_2 and t_3 partitions. The resulting abstract state is shown in Fig. 3.12. As it can be seen, ℓ_{f1} and ℓ_{f2} are copied to the `foo` and `bar` properties of `obj2`, respectively, without being mixed together.

The paper in Chapter 8 also shows how value partitioning can be used to precisely reason about free variables and type predicate functions. The evaluation shows that using value partitioning, TAJS can analyze all the benchmarks that can be analyzed using the *demand-driven value refinement* technique with comparable analysis times. Furthermore, using value partitioning, TAJS can also analyze some benchmarks that could not be analyzed using the demand-driven value refinement technique.

The techniques *demand-driven value refinement* and *value partitioning* are the only two techniques that can analyze FCT patterns precisely and efficiently when they span across functions. Reasoning precisely about such patterns is a prerequisite for analyzing the most popular Node.js libraries *lodash* and *underscore*, and is thereby also a prerequisite for analyzing many real-world Node.js applications.

3.3.3 Improving Scalability of Taint Analysis for Node.js Applications

The paper in Chapter 9 presents the first interprocedural static taint analysis for detecting injection vulnerabilities in Node.js applications. A dataflow analysis such as TAJS can easily be extended to support taint analysis by adding a taint component to the lattice for abstract values. However, existing dataflow analyses do not scale to Node.js applications, since they depend on many third-party packages.

The key insight for this work is that not all third-party packages are relevant for detecting taint flows in a Node.js application. The key challenge is to identify what third-party packages are relevant. Since the security critical sources and sinks might be in the third-party packages, ignoring all third-party packages will not work. We therefore have to come up with some heuristics for deciding which packages are relevant. This is challenging, since the only analyses that can provide the facts about the program that are useful for designing such heuristics are the dataflow analyses, which we are trying to make scalable. We therefore choose to use TAJS for analyzing the application by first ignoring all third-party packages, and then based on the analysis results, we apply some heuristics for refining the set of packages that should be analyzed. When the set of packages to analyze has been updated, the TAJS analysis starts over with the new set of packages to analyze. This process continues until a fixpoint for the packages to analyze has been reached.

We have implemented our solution in the tool NODEST, and Fig. 3.13 shows how it works. As usual, the input is the program to analyze. We will now briefly describe how the modified TAJS analysis works and afterwards how the post-processor works.

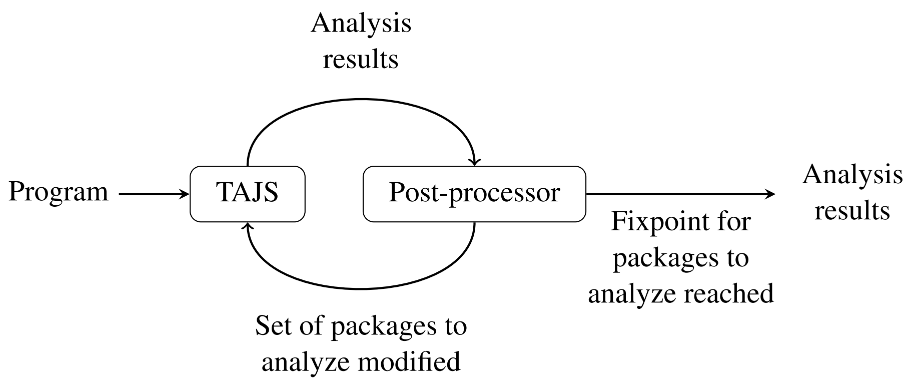

Analysis results
Program
TAJS
Post-processor
Analysis results
Fixpoint for packages to analyze reached
Set of packages to analyze modified

Figure 3.13: Diagram showing how NODEST works.

Modified TAJS analysis The modified version of TAJS only analyzes the packages that have been selected for analysis and additionally tracks taint together with other information that is used by the post-processor. The analysis aims to overapproximate the behavior of packages that are not analyzed, to avoid missing dataflow by not analyzing a package. For example, when analyzing code like `var _ = require('lodash')`, and *lodash* is not in the set of packages to analyze, then a very imprecise abstract value is written to the _ variable. The imprecise value also contains a tag that indicates that the value is overapproximated due to not analyzing the *lodash* package. When calling a function from an ignored package, we also aim to overapproximate the calls to callback functions and the return value. For example, when analyzing the code `_.forEach(x, function(y) { ... })` and _ is the overapproximated value from before, then the analysis adds a call edge to `function(y) { ... }` with a very imprecise value for the argument `y`. Furthermore, the return value from calling functions on packages that are not analyzed is also a very imprecise value with a tag specifying what package the function being called origins from. The analysis does not overapproximate side-effects in packages that are not analyzed, so dataflow that relies on side-effects can be missed. This way of overapproximating behavior from a package that is not analyzed can be seen as a kind of "reasonably most general package" as opposed to the notion of "reasonably most general client" proposed by Kristensen and Møller [88]. The analysis results are after the analysis passed to the post-processor.

Post-processor The post-processor takes as input the analysis results and based on those, it uses some heuristics for deciding the packages that should be analyzed by the modified version of TAJS in the next iteration. This process runs until a fixpoint is reached, where the output of NODEST is the analysis results for the last iteration. The challenge is how to design this post-processor such that we do not analyze too many packages, while analyzing the packages that are relevant for detecting taint flows.

We propose the following three criteria for selecting packages to be analyzed:

Precision: Trying to overapproximate the behavior of packages we do not analyze might result in critical precision losses. We have designed a heuristic that checks whether critical precision loss has been introduced by not analyzing a package, and if so, then we add the package to the list of packages that should be analyzed.

Side-effects: Since the analysis does not overapproximate side-effects in packages that are not analyzed, the analysis might miss dataflow that relies on side-effects. The analysis therefore tracks for each object what packages it has been passed to, since these packages could have modified the object through side-effects. Consider the code `x.foo()`, where `x` is an object that has been passed to the package *lib* where *lib* is not analyzed. If `x` does not have a `foo` property, then this code will throw a `TypeError` since the value `undefined` is used as a function. In such a case, our heuristic will include the package *lib* to be analyzed, since the `foo` property probably was added as a side-effect inside the package *lib*.

Taint: Another reason to include a package is if it might be part of a taint flow, i.e., either the source, the sink or a package that is on the path from a source to a sink. We use a syntactical pre-analysis to check what packages contain sources or sinks, and based on this information, we conservatively treat all functions from that package as sources or sinks, respectively. If this results in a taint warning, then we include the package in the analysis to find out, for instance, whether the sink is actually reachable with a tainted value.

The evaluation shows that NODEST can find the packages that are needed for detecting almost all taint flows in our 11 real-world Node.js benchmarks. More specifically, we find all known taint flows in 10 of the benchmarks without any false positives. Compared to using the ordinary TAJS analysis, NODEST finds 63 true positive taint flows, while the TAJS analysis only finds 4. We can thereby conclude that the approach used by NODEST can make a static dataflow analysis scale to some real-world Node.js applications.

Chapter 4

Lightweight Static Analysis Techniques for JavaScript and Node.js

Sound static analysis for Node.js is unfortunately still too expensive to be used as tool support to improve developer productivity. For tool support, the analysis should at most take a few seconds, while the sound static analysis techniques might take minutes or hours to analyze the Node.js applications that they currently can analyze. For this reason, this thesis also presents some more lightweight analysis tools in Chapters 10 to 12 that are sufficiently efficient for being used by developers. These analysis tools do not provide soundness guarantees, but lean towards overapproximation, which means that the analysis results are sound for most real-world Node.js applications.

This chapter starts by introducing some existing work in the following three areas: Practical static analysis, existing tools for updating dependencies with breaking changes, and security scanning. Section 4.2 presents the tools from the papers in Chapters 10 and 11 that provides semi-automatic support for updating dependencies when the update involves breaking changes. Section 4.3 presents the function-level security scanner presented in Chapter 12 that is much more precise than existing package-level security scanners.

4.1 Existing Work

Practical static analysis Feldthaus et al. [48] presented a practical static analysis for construction of approximate call graphs for JavaScript IDE services. The analysis is practical in the sense that it is fast enough for an IDE service, since the analysis runs in only a few seconds on their benchmark applications. Feldthaus et al. [48] show that a *field-based* analysis works well for JavaScript. A field-based analysis abstracts all objects as one, so objects are modelled by a single map $P \to \mathit{Value}$, where P is the set of all property names, and *Value* is the type of value stored at each property. For this reason, their analysis can ignore dynamic property reads and dynamic property writes

```
  var _ = require('lodash');
- _.dropWhile([1, 2, 3], () => ..., {});
+ _.dropWhile([1, 2, 3], (() => ...).bind({}));
```

Figure 4.1: Example of code modifications when updating *lodash* to version 4.

without introducing too much unsoundness in practice when analyzing client-side applications that use at most one library. Field-based analysis is useful for JavaScript, since object copy patterns like the one presented in Fig. 3.1 can be soundly ignored, since if a property is written to one object, then abstractly it has been written to all objects. This avoids one of the big challenges presented in the previous section for sound static analysis. Furthermore, the work finds that analysis precision is also quite good when using this simple approach, since the same property names are rarely used in different objects. Ignoring dynamic property writes, however, means that the analysis is unsound for code like `o2["foo"+x] = ...`, where the property name is dynamically computed, and the value to write is not copied from the same property of another object. In practice, the analysis still has a recall on more than 85%, which means that their call graph misses at most 15% of the call edges for their benchmarks.

Feldthaus and Møller [46] also presented a practical semi-automatic static analysis supporting developers in rename refactoring for JavaScript. Their analysis groups occurrences of the identifier that is to be renamed, and then for each of the groups asks the developer whether the renaming should be applied or not. The analysis is fast enough to be used by developers, and the manual effort on a rename refactoring is reduced by 57% on average compared to doing the rename using search-and-replace.

Another practical analysis was proposed by Madsen et al. [97] that aims at detecting library APIs based on how the library is used. The analysis is shown to provide good support for auto-completion.

Tools for code modification As the previous work on practical static analysis illustrates, simple refactorings such as renaming are complicated in JavaScript. There are also a few tools that try to help developers refactor their code by using simple syntax or AST-based replacements. The tools we will discuss are recast[1], jscodeshift[2], codemod[3], comby[4], and semgrep[5].

Fig. 4.1 shows an example of a code modification that is required when updating *lodash* to version 4. The lines marked with +/- are added/removed when updating the code to become compatible with *lodash* version 4. The `dropWhile` function in *lodash* previously supported an optional third argument when the second argument was a function. The third argument is used as the `this` object when calling the callback. In

[1]https://www.npmjs.com/package/recast
[2]https://www.npmjs.com/package/jscodeshift
[3]https://github.com/facebook/codemod
[4]https://comby.dev
[5]https://semgrep.dev

lodash version 4, this feature is removed and the `bind` function should be called before passing the callback as shown in Fig. 4.1.

The tool recast is an AST-to-AST transformation tool that tries to preserve the original coding style as much as possible, i.e., it does not modify styling in parts of the code that should not be modified. The patch needed for Fig. 4.1 can be implemented through a simple AST transformation for this specific program. First, we should identify the call nodes where the base object of the callee is the variable `_`, the property name is dropWhile, and the call has three arguments. Next, we should filter out the calls where the second argument is not of type function. In the example the type of the second argument can be inferred from the call node, so the transformation can be expressed for this program. However, there are a couple of issues with this approach. Since the patch only works for a specific client, the client developer will have to write the patch, which means that the client developer will still have to do all the manual work in comprehending the changelog. The developer also needs to learn the AST representation and implement the transformation, which might take longer time than doing the update manually.

The tool jscodeshift is a wrapper around recast that provides a nicer API, but the patches are still client specific. The codemod and comby tools basically do search-replace with meta-variables, so they can search for the calls to `_.dropWhile` with three arguments and then replace as needed. The issue with this approach, however, is the same as with recast and jscodeshift, namely that the tools cannot infer any dataflow information, and therefore the patches have to be written for each specific client.

The tool semgrep is more expressive and can use meta-variables to encode some data-flow into the detections, so for instance, one can write a pattern `$A = require('lodash'); $A.dropWhile($X, $Y, $Z)` to detect calls to the dropWhile function of *lodash* with three arguments. The type constraint that the patch should only be applied if the second argument is of type function cannot be expressed. Furthermore semgrep does not support automatic patching of the code.

Since no existing tool supports client-independent descriptions of refactorings like the one in Fig. 4.1, a tool that can patch every client using a general patch description is desired. Section 4.2 explains how the tool JSFIX presented in Chapter 11 uses semantic patch descriptions that can be used across clients to automatically patch the code in Fig. 4.1. The semantic patches are very simple and can be written in seconds.

Security scanners Currently, package-level based security scanning is the best support developers have to know whether their application might be vulnerable due to vulnerable dependencies. Package-level security scanners issue a warning if there is a known vulnerability in any of the packages that the application depends on. An example is npm audit that reports all vulnerabilities from the npm vulnerability database[6] that are in packages that the application depends on. Another example is snyk that works similarly but uses snyk's own vulnerability database[7]. The drawback

[6]`https://www.npmjs.com/advisories`

[7]`https://snyk.io/vuln?type=npm`

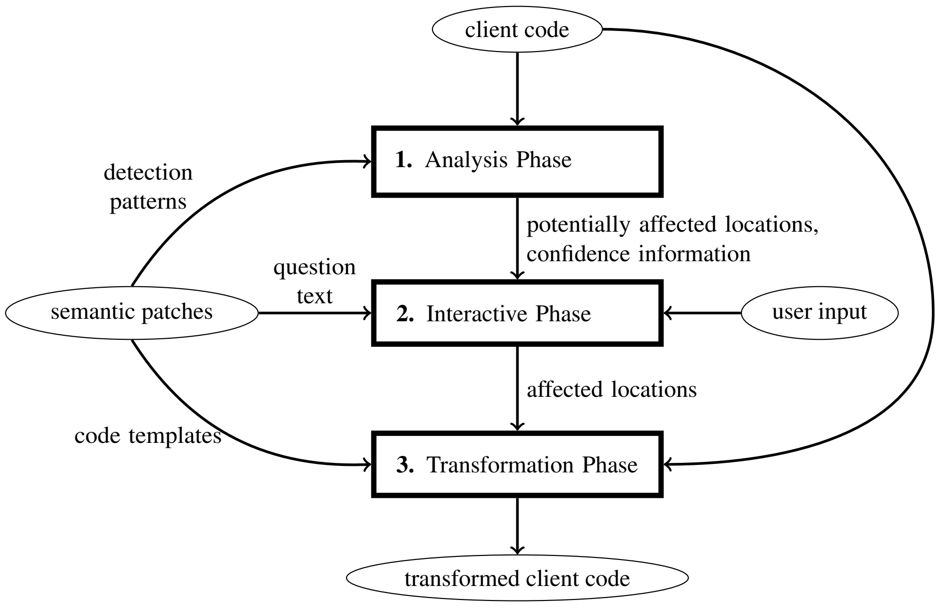


Figure 4.2: The phases of JSFIX.

of these tools is that they produce many false positives, since clients typically only use a small part of a dependency's API. The challenge for Node.js is that there does not exist any good static analysis that can provide more fine-grained security scanning. We address this problem in the paper in Chapter 12 by presenting a scalable and modular call graph construction algorithm for Node.js applications that can be used for security scanning. The call graph construction algorithm is summarized in Section 4.3.

4.2 Updating Dependencies with Breaking Changes

Dependencies are often updated, and patching a client to use the new version of a library is a cumbersome, time-consuming, and error-prone task when the update includes breaking changes. The research papers in Chapters 10 and 11 address this problem by providing a tool to semi-automatically patch clients to become compatible with the new version of a library.

JSFIX We present the tool JSFIX in Chapter 11. JSFIX can semi-automatically patch clients when updating a library from one major version to another. Fig. 4.2 shows the phases of JSFIX. Chapter 11 defines the notion of *semantic patches* that are formalized descriptions for how to handle breaking changes. A semantic patch consists of a *detection pattern* describing the API points of a library that are affected by a breaking change, and a *code template* that specifies how to patch the locations affected by the breaking change. The code template is basically a JavaScript expression with some meta-variables, where the meta-variables refer to expressions relative to the affected location. The detection pattern part is used in the first phase. The first phase is the static analysis presented in Chapter 10. The analysis outputs the locations that might be affected for each breaking change together with confidence information describing whether the analysis is certain that the location is affected, and if not, it provides a reason for the uncertainty.

The second phase asks the client developer yes/no questions about each of the affected locations where the analysis is uncertain. The questions are about the clients own source code, for instance, whether the type of the second argument at a call is a string, or the receiver of the call is an rxjs observable. These are all questions that the client developer also has to consider if the update was done manually. Based on the answers, all the false positives from the analysis phase are filtered out.

The third phase simply applies the code templates at all the affected locations. Applying a code template means interpolating the meta-variables and replacing the code at the affected location by the interpolated code. The result of the third phase is the transformed client code that is compatible with the new version of the library.

We will now explain in more detail how JSFIX works on the example program shown in Fig. 4.1. The semantic patch describing the breaking change is

```
call <lodash>.dropWhile [3,3] 2:function ⇝ $callee($1, $2.bind($3))
```

which will be described in the following paragraphs.

Analysis phase The left-hand side of ⇝ is the detection pattern that is used in the analysis phase to find affected locations. The pattern matches calls to the `dropWhile` function of the *lodash* library object, where the call has three arguments, and the second argument is of type function. The analysis phase takes this description and the unmodified version of the code in Fig. 4.1 and finds the locations that match the detection pattern. The analysis is the TAPIR analysis presented in the paper in Chapter 10. It works in the three phases shown in Fig. 4.3. TAPIR is intramodular so it analyzes only one file at a time. For applications with multiple files the analysis is simply run for each file. The first phase is a simple AST-based alias analysis. The analysis is field-based and computed through a single scan of the AST. The result is a map $(Vars \times P) \to \mathscr{P}(Exp)$ from variables and property names to the expressions that have been assigned to them. For the example program in Fig. 4.1, the alias information is $[\_ \to$ `require('lodash')`$]$. The second phase uses the results from the alias analysis to compute access paths for each AST node. An access path describes how a value is accessed. For instance, the access path for the code `_.dropWhile` in Fig. 4.1 becomes `<lodash>.dropWhile`, where `<lodash>` refers to the *lodash* library object. The access path is inferred since the property `dropWhile` is read in the property read, and the alias information provides the information that _ is the *lodash* library object. The access path inference does not consider calls, so the analysis does not know the access paths for function parameters.

The third phase goes through all nodes and checks whether any of the inferred access paths for the node matches the detection pattern. The nodes that have an access path that matches a detection pattern are then reported as potentially affected

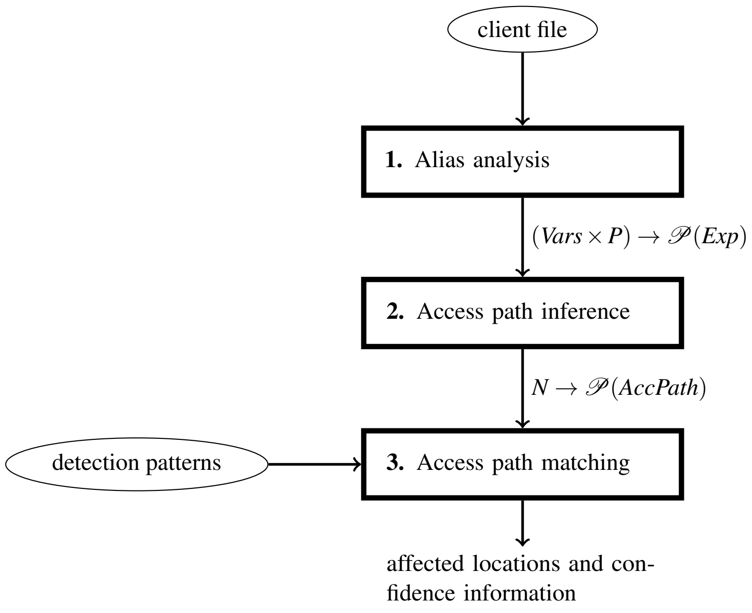

client file
1. Alias analysis
$(Vars \times P) \rightarrow \mathscr{P}(Exp)$
2. Access path inference
$N \rightarrow \mathscr{P}(AccPath)$
detection patterns
3. Access path matching
affected locations and confidence information

Figure 4.3: The phases of TAPIR.

```
var _ = require('lodash');
function f(x) {
-   _.dropWhile([1, 2, 3], x, {});
+   _.dropWhile([1, 2, 3], x.bind({}));
}
f(() => ...);
```

Figure 4.4: Example of code modifications when updating *lodash* to version 4.

locations. For each of the affected locations, the analysis also reports its confidence in whether the affected location is a true positive. For the example in Fig. 4.1, the analysis has high confidence, since the access paths computed for `_.dropWhile` is only `<lodash>.dropWhile`, and the number of arguments and the type of the second argument can be inferred syntactically from the arguments to the call node.

Our evaluation shows that all high confidence matches are indeed true positives. The matching aims to be overapproximate, so if the second argument was passed through a function as in Fig. 4.4, where the type could not be inferred syntactically, then it would be reported as a low confidence match, since the analysis does not know whether `x` is a function. The evaluation of the analysis in Chapter 10 shows that the analysis does not have any false negatives and only a false positive rate on 14% on our 265 benchmarks. The average analysis time is around one second per client.

Interactive phase When performing code transformations, we need some approach for filtering away the false positives. If we do not filter away the false positives, we might transform some code that should not be transformed and thereby break the application we are trying to patch. We exploit the confidence information from the TAPIR results, so when analyzing the code in Fig. 4.4, the interactive phase will ask the user the question "Is the argument x of type function?" and point to the source location for the call. Since the question is about the developers own code, the question should be easy and fast to answer for the developer. When the developer has answered all questions, the set of affected locations that are true positives are passed to the transformation phase.

Transformation phase The transformation phase applies the code templates for all affected locations. In the ongoing example, the transformation phase has to apply the code template `$callee($1, $2.bind($3))`, which is the right-hand-side of $\rightsquigarrow$ in the semantic patch that was presented earlier. The transformation is applied by replacing the matched code with the code produced by the code template. Code templates are basically JavaScript expressions but with some meta-variables that refer to AST nodes relative to the matched node. In Fig. 4.4 we have matched the call node for the code `_.dropWhile([1, 2, 3], x, {})`. The meta-variable `$callee` refers to the node describing who the callee is going to be, which in this case is `_.dropWhile`. `$n` represent the `n`'th argument node, which means that `$1` is `[1, 2, 3]`, `$2` is `x`, and `$3` is `{}`. Replacing the meta-variables with their values for this match, we get the code `_.dropWhile([1, 2, 3], x.bind({}))` as shown in line 116 in Fig. 4.4. Similarly, line 112 in Fig. 4.1 shows the transformed code for that example.

Evaluation The evaluation of JSFIX in Chapter 11 shows that the code templates can express most of the breaking changes in popular npm packages. Furthermore, JSFIX can almost always correctly patch client code to work with the new version of a library. We test this by using clients whose test suite fail without patching the client code, and test whether the test suite succeeds after JSFIX has patched the code. The patches are mostly similar to what a developer would write, and 31 pull requests created based on the JSFIX patches have been accepted by the client developers. We do not find the manual process of answering questions an issue for the practicality of the approach, since only 2.7 questions are asked on average per client, and as mentioned earlier, the questions are easy and fast to answer. Using JSFIX is therefore much easier and faster for a developer than doing the update manually. Furthermore, JSFIX only takes a few seconds per client so it is fast enough to be used in practice. We therefore believe that JSFIX can save a lot of developer time, since once the semantic patches have been written for a library update, then they can be applied to all clients.

4.3 Function-Level Security Scanning

In this section we give an overview of the research paper included in Chapter 12 that presents the tool JAM that addresses the issue with too many false positives in package-level dependency scanners. In this work, we develop a scalable and modular

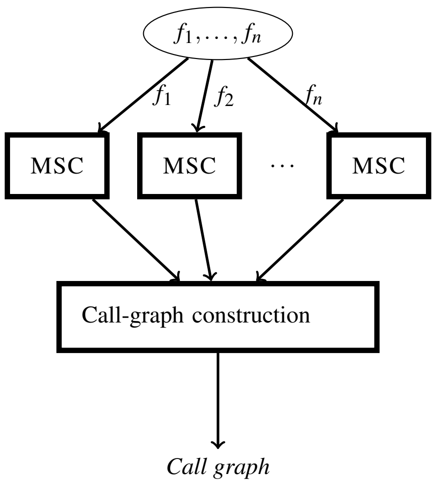


Figure 4.5: Structure of JAM.

```
lib.js:
module.exports.forEach = (iteratee) => {
  return (arr) => {
    for (var x of arr) {
      iteratee(x);
    }
  };
}
client.js:
const forEach = require('./lib.js').forEach;
forEach(x => ...)([1, 2, 3]);
```

Figure 4.6: Call graph construction example

call graph construction algorithm for Node.js applications that can be used for security scanning at the function-level.

We leverage our finding from the TAPIR analysis in Chapter 10 that module-based analysis using access paths is an accurate and scalable approach for analyzing a program. Therefore, in the call graph construction algorithm, we start by computing access path based file summaries using our module summary computer (MSC). The results from these summaries are then used for computing the call-graph in the Call-graph constructions phase as shown in Fig. 4.5. We will explain our technique using the example program in Fig. 4.6. It consists of a main file client.js, and a library file lib.js that client.js depends on. The file lib.js exports a curried `forEach` function that takes a callback function and returns another function that takes an array as argument. On invocation of the returned function, the callback function is called for each of the elements in the array. The code in client.js invokes this curried `forEach` function.

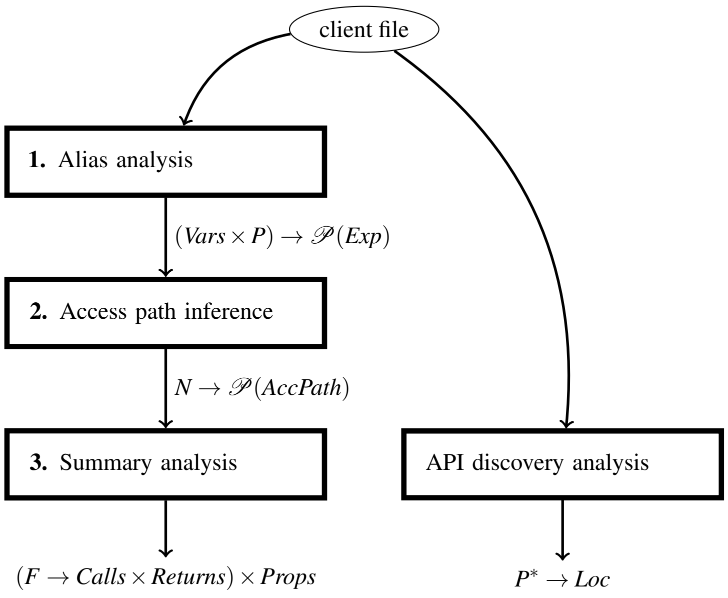

client file
1. Alias analysis
$(Vars \times P) \rightarrow \mathscr{P}(Exp)$
2. Access path inference
$N \rightarrow \mathscr{P}(AccPath)$
3. Summary analysis
API discovery analysis
$(F \rightarrow Calls \times Returns) \times Props$
$P^* \rightarrow Loc$

Figure 4.7: The module summary computer in JAM.

Module summary computer The module summary computer is shown in Fig. 4.7. The boxes on the left are static analysis phases, and the API disovery analysis is a dynamic analysis. The alias analysis is exactly as the one in TAPIR. The access path inference is similar to TAPIR, but the access path mechanism has been extended to have special access paths for functions and parameters, such that interprocedural flow can be inferred from the access paths. For example, the access path for `x` in Fig. 4.4 is unknown in TAPIR, while in JAM the access path indicates that it is the 0'th parameter of `f`. For `() => ...`, the access path in TAPIR is also unknown, while in JAM the access path describes the location of the function definition. The third phase in the static analysis in JAM is a summary analysis that categorizes the inferred access paths by function, producing results of the form $(F \rightarrow Calls \times Returns) \times Props$. The result is a map from each function to the call access paths inferred for nodes in the body of the function, and the access paths describing the values that may be returned by the function. The *Props* component is the $P \rightarrow AccPath$ part of the alias analysis results that is used for performing field-based analysis across multiple files in the call graph construction phase.

The API discovery phase uses a dynamic analysis that loads the file to be analyzed and then summarizes the exported properties into a map $P^* \rightarrow Loc$. It maps each sequence of property names on the exported object that has a function value to the location of that function's definition. In the call graph construction phase, we use this map for property reads of library objects instead of using the more imprecise field-based information. We will refer to this component as the *Exports* component of a module summary.

The computed module summaries for the code in Fig. 4.6 are shown in Figures

$$
\begin{aligned}
MS_{lib} &= (exports_{lib}, calls_{lib}, returns_{lib}, props_{lib})\\
exports_{lib} &= [\texttt{"forEach"} \rightarrow \langle\texttt{lib.js:119}\rangle]\\
calls_{lib} &= [\\
&\quad \langle\texttt{lib.js}\rangle \rightarrow \{\},\\
&\quad \langle\texttt{lib.js:119}\rangle \rightarrow \{\},\\
&\quad \langle\texttt{lib.js:120}\rangle \rightarrow \{\langle\texttt{lib.js:119}\rangle\texttt{.Param[0](...)}\}\\
&]\\
returns_{lib} &= [\\
&\quad \langle\texttt{lib.js}\rangle \rightarrow \{\},\\
&\quad \langle\texttt{lib.js:119}\rangle \rightarrow \{\langle\texttt{lib.js:120}\rangle\},\\
&\quad \langle\texttt{lib.js:120}\rangle \rightarrow \{\}\\
&]\\
props_{lib} &= []
\end{aligned}
$$

Figure 4.8: Module summary for lib.js from Fig. 4.6.

4.8 and 4.9. The access path ⟨`lib.js:119`⟩ describes the function defined at line 119 in lib.js, and ⟨`lib.js`⟩ describes the top-level code in lib.js. From the *Exports* component in the summary for lib.js, we can see that lib.js exports a function created at line 119 through the property `forEach`, and the *Returns* component for this function shows that it returns the function defined in line 120. From the *Calls* component, we can also see that the function defined in line 120 calls its first argument.

Call graph construction We will explain the call graph construction by using the example code from Fig. 4.6. Fig. 4.10 shows the final call graph, and we will now explain each step in its computation. The call graph construction starts by creating a node for each function in each of the module summaries. This results in the nodes represented by the ellipses in Fig. 4.10. Next, call edges are added from a function f to all the access paths in the *Calls* component for the function f. For access paths that describe explicit function locations, we simply add a call edge to that node. This rule adds an edge from ⟨`client.js`⟩ to ⟨`lib.js`⟩. For a sequence of property reads from another module, we use the results from the API discovery phase to lookup the corresponding function. The access path ⟨`lib.js`⟩`.forEach(`⟨`client.js:127`⟩`)` therefore results in an edge from ⟨`client.js`⟩ to ⟨`lib.js:119`⟩, since the *Exports* component for lib.js contains `"forEach"` $\rightarrow$ ⟨`lib.js:119`⟩. We create intermediary nodes for access paths where we cannot directly infer the function node. For this reason, we create the two rectangle nodes in Fig. 4.10 and the edges to them. We add edges from these intermediary nodes afterwards through a fixpoint computation.

The edges from a rectangle node where the callee is returned from another function

$$
\begin{aligned}
MS_{client} &= (exports_{client}, calls_{client}, returns_{client}, props_{client}) \\
exports_{client} &= [] \\
calls_{client} &= [\\
&\qquad \langle\texttt{client.js}\rangle \to \{ \\
&\qquad\qquad \langle\texttt{lib.js}\rangle, \\
&\qquad\qquad \langle\texttt{lib.js}\rangle\texttt{.forEach(}\langle\texttt{client.js:127}\rangle\texttt{)}, \\
&\qquad\qquad \langle\texttt{lib.js}\rangle\texttt{.forEach(}\langle\texttt{client.js:127}\rangle\texttt{)(...)} \\
&\qquad \}, \\
&\qquad \langle\texttt{client.js:127}\rangle \to \{...\} \\
&] \\
returns_{client} &= [\\
&\qquad \langle\texttt{client.js}\rangle \to \{\}, \\
&\qquad \langle\texttt{client.js:127}\rangle \to \{...\} \\
&] \\
props_{client} &= []
\end{aligned}
$$

Figure 4.9: Module summary for client.js from Fig. 4.6.

are inferred using the return summaries for the relevant functions. For the node with access path ⟨`lib.js`⟩`.forEach(`⟨`client.js:127`⟩`)(...)`, we use the return summary for all functions that are targets for a call edge that was added due to the access path ⟨`lib.js`⟩`.forEach(`⟨`client.js:127`⟩`)`. In this case, it is only the function ⟨`lib.js:119`⟩. The return summary for this function is {⟨`lib.js:120`⟩}, and therefore we add an edge from ⟨`lib.js`⟩`.forEach(`⟨`client.js:127`⟩`)(...)` to {⟨`lib.js:120`⟩} as shown in Fig. 4.10. The call edges from the rectangle node with access path ⟨`lib.js:119`⟩`.Param[0](...)` are computed using the access paths for the first argument on call edges going to ⟨`lib.js:119`⟩. The access path responsible for the only call edge to ⟨`lib.js:119`⟩ is ⟨`lib.js`⟩`.forEach(`⟨`client.js:127`⟩`)`, so we add a call edge to ⟨`client.js:127`⟩, since that is the access path for the argument. Since the rectangle nodes rely on the call graph for adding call edges, we need to recompute their callees until a fixpoint is reached. In this example, the fixpoint is already reached, but in practice multiple iterations are usually needed. For example, if resolving one of the rectangle nodes would add a new edge to ⟨`lib.js:119`⟩, we would have to recompute the node ⟨`lib.js:119`⟩`.Param[0](...)`.

JAM scales to large Node.js applications since it abstracts each module by a module summary that is fast to compute. Due to these summaries, the potentially expensive fixpoint computation only reasons about simple access paths instead of the original program, making it much more scalable. The design of JAM also allows

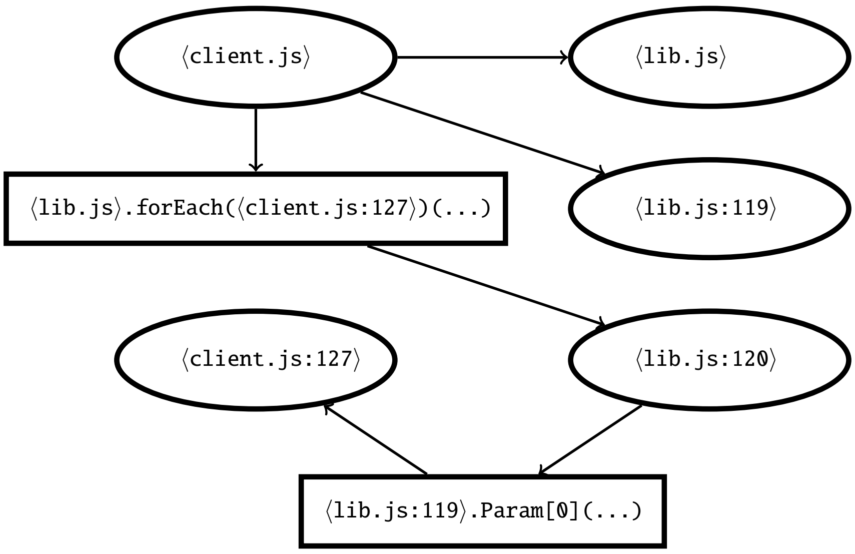

⟨client.js⟩
⟨lib.js⟩
⟨lib.js⟩.forEach(⟨client.js:127⟩)(...)
⟨lib.js:119⟩
⟨client.js:127⟩
⟨lib.js:120⟩
⟨lib.js:119⟩.Param[0](...)

Figure 4.10: The call graph computed using JAM on Fig. 4.6.

for computing call graphs modularly. Once a call graph has been computed, we can always add more module summaries, and then recompute the fixpoint computation to incorporate these new modules into the call graph. This means that we can precompute call graphs for all dependencies, and then we can compute the call graph for the full application by only computing the call graph for the application code. We can thereby save redundant computations by not computing the call graph for dependencies when the application code has been modified. This improves the scalability of the analysis even further.

We have evaluated JAM by using the produced call graphs for function-level security scanning. We compare with npm audit that is a package-level security scanner and use the same database of vulnerabilities as npm audit does. Our analysis only considers the actual vulnerable function as vulnerable, instead of the entire package as the vulnerabilities in npm audits vulnerability database does. We evaluated JAM on 12 Node.js applications with a vulnerability according to npm audit. The evaluation showed that function-level security scanning using the JAM callgraph does not miss any of the 8 true positive vulnerabilities. Furthermore, the precision increase reduces the number of false positives from 26 when using npm audit to only 5 when using JAM, i.e., the number of false positives is reduced by 81%. The 12 benchmarks vary in size from 800 to 27 000 functions. The analysis time is only a few seconds for most of the benchmarks when also computing the call graph for all dependencies. The analysis time is less than one second on average when using the modular approach where the call graphs for dependencies have been precomputed. Since the modular approach takes less than a second on average, it is fast enough to be integrated into an IDE, where it could be useful for code-navigation. To our knowledge, JAM is so far the only call graph construction algorithm that can reason precisely about the modular structure of Node.js applications, and JAM scales to real-world Node.js applications. The closest related work is that of Feldthaus et al. [48], but this work does not precisely reason about the modular structure of Node.js applications. This is because their field-based analysis mixes together all properties in the entire application, whereas JAM treats properties read from library objects precisely by using the exports summary produced by the API discovery analysis.

Chapter 5

Conclusion

JavaScript and Node.js are widely used even though they have limited tool support, which might negatively affect developer productivity. They are widely used, because they provide flexibility through very dynamic features. These dynamic features are also what makes automatic reasoning for JavaScript and Node.js a challenge. This thesis addresses this challenge by presenting new sound static analysis techniques that can reason precisely about the most popular Node.js libraries. This is an important step towards sound static analysis for Node.js applications. Furthermore, we have also presented some practical static analyses that scale to large Node.js applications. To summarize, the contributions of this thesis are:

C1 **Techniques for systematically finding soundness issues and root-causes for precision loss in a static analysis:** The techniques use delta debugging to reduce programs that shows a problematic behavior in the analysis, for instance, for programs that cannot be analyzed in a reasonable time. We also present soundness testing for finding soundness bugs by comparing the analysis results to runtime facts observed by a dynamic analysis. Combined with delta debugging, we can find small programs that showcase soundness bugs in the analysis. Lastly, we use blended analysis to detect precision bottlenecks in a static analysis. This is done by using delta debugging to find a minimal set of locations where blended analysis needs to be used to analyze a program. The analysis designer can now use these locations to find out how to improve the analysis, which might save the analysis designer time.

C2 **Sound and precise static analysis for correlated read/write pairs:** We have presented two techniques that can reason precisely about correlated read/write pairs. The first is *demand-driven value refinement* that heuristically detects when an imprecise dynamic property write is about to happen. At such a write, a backwards analysis is used to regain a relation between the property name to write to and the value to write. The backwards analysis provides precise results for correlated read/write pairs, such that these pairs are analyzed precisely in the main analysis. The second technique is *value partitioning* that through a simple change to the abstract domain of an analysis is able to efficiently track relational information between different abstract values. This technique has been shown to work for analyzing correlated read/write pairs, free variables, and type predicate functions, without incurring significant overhead in the static analysis. The two techniques are the only existing techniques that can precisely analyze the most popular npm libraries *lodash* and *underscore*, which is an important step towards analyzing real-world Node.js applications.

C3 **Scaling a static taint analysis to real-world Node.js applications:** We propose the tool NODEST that is a static taint analysis that scales to real-world Node.js applications. The scalability is achieved by only analyzing those third-party packages that are relevant for the taint analysis. The challenge in this work has been how to decide what packages are relevant. We propose three heuristics for when to analyze a package. A package should be analyzed if too much imprecision is introduced by not analyzing the package precisely, if side-effects are missed due to not analyzing the package, or if the package might be part of a taint flow. Using these heuristics, we detect the packages that need to be analyzed, while not including so many packages that the analysis cannot find the known taint flows in our benchmarks within 30 minutes. Furthermore, in our experiments, NODEST did not have any false positives.

C4 **Semi-automatically patching clients when updating a library with breaking changes:** We have presented the TAPIR analysis that automatically finds the locations that might be affected by breaking changes. TAPIR does not miss any locations according to our experiments. Furthermore, we presented JSFIX that uses an interactive phase to filter out the false positives from the TAPIR analysis, and then automatically patches the locations that are affected by breaking changes. We have shown that JSFIX can almost always patch clients successfully. The patches are very similar to the code that a developer would write, and the quality is high enough that 31 clients have accepted pull requests based on the patches made by JSFIX. JSFIX runs in a few seconds per application and asks only 2.7 questions on average per application, which means that JSFIX has good potential for saving developers a lot of time when updating dependencies with breaking changes.

C5 **Modular call graph construction and function-level security scanning:** We propose the practical static analysis tool JAM for modular call graph construction. The computed call graphs are used for security scanning by checking whether any vulnerable function is reachable in the call graph. JAM scales to real-world Node.js applications by creating module summaries for each file and construct a call graph based on these summaries. When using JAM for security scanning on 12 Node.js applications, JAM does not miss any true positives, but reduce the amount of false positives by 81%. The modular structure of JAM allows precomputation of call graphs for dependencies. When using these precomputed call graphs, the call graph construction for an application takes less than a second on average, which means that the technique is sufficiently fast for integration into an IDE.

Based on these contributions, we have supported the thesis statement: *Despite the highly dynamic features of JavaScript and Node.js, it is possible to develop static analysis techniques to accurately infer useful facts for Node.js applications. The useful facts are type-related information for detecting type errors, taint information for detecting injection vulnerabilities, call-graph information for security scanning and code-navigation, and library API usages for aiding in updating dependencies with breaking changes*.

However, the work on static analysis for JavaScript and Node.js is far from completed. For sound static analysis, this thesis has taken some of the steps to make analysis of Node.js applications feasible, by proposing techniques that can precisely reason about the most popular Node.js libraries. In the future, more work needs to be done on how to make static analysis scale to larger Node.js applications. In general, we can follow the research methodology used in this thesis, by starting to select a Node.js application that we cannot analyze, and then systematically as described in Chapter 6 find out why the analysis cannot currently analyze the application.

There are also some interesting work in trying to extend the use of value partitioning. The lightweight value partitioning approach might be used to replace some of the existing expensive context sensitivity strategies used for static analysis of JavaScript. For example, it could be interesting to investigate whether value partitioning could be used instead of parameter-sensitivity in the TAJS analysis.

Future work also involves using TAJS to find vulnerabilities in libraries. Currently, since TAJS is a whole-program analysis, a taint analysis that is built on top of TAJS can only detect vulnerabilities if the analyzed program exploits the vulnerable path, so it is not possible to prove whether a library might be vulnerable. By using a reasonably most general client [88] of a library, it might be possible for TAJS to prove whether a library is vulnerable or not.

In the area of code modification tools such as JSFIX, there is also more research to pursue. Even though JSFIX only asks few questions, it might be possible to reduce the number of questions further by grouping some similar questions together, similar to what Feldthaus and Møller [46] do for rename refactoring. Furthermore, the current practice of writing the semantic patches for JSFIX is entirely manual. As a starting point it could be interesting to investigate whether the detection patterns could be inferred by extending the work done by Møller and Torp [105] that can automatically detect breaking changes. The performance of JSFIX is also high enough for integration into an IDE, so it could be interesting to use the machinery behind JSFIX directly in an IDE. For instance, when a developer renames an exported property, a corresponding semantic patch could automatically be generated. JSFIX could then apply the patch in the rest of the application, and thereby perform the renaming automatically. This idea should also be possible for refactorings that change how exported functions are used. Combined with the call-graph generated by JAM, it should also be possible to make refactorings for functions that are not exported.

To sum up, this thesis has shown that despite the highly dynamic features of JavaScript, static analysis is promising for accurately inferring useful facts about Node.js applications. In the future, the presented analysis techniques might be used to increase developer productivity if the techniques are incorporated into the IDEs that are used by developers.

Part II

Publications

Chapter 6

Systematic Approaches for Increasing Soundness and Precision of Static Analyzers

By Esben Sparre Andreasen (Aarhus University, Denmark), Anders Møller (Aarhus University, Denmark) and Benjamin Barslev Nielsen (Aarhus University, Denmark).
Published in the Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP), June 2017.

Abstract

Building static analyzers for modern programming languages is difficult. Often soundness is a requirement, perhaps with some well-defined exceptions, and precision must be adequate for producing useful results on realistic input programs. Formally proving such properties of a complex static analysis implementation is rarely an option in practice, which raises the challenge of how to identify causes and importance of soundness and precision problems.

Through a series of examples, we present our experience with semi-automated methods based on delta debugging and dynamic analysis for increasing soundness and precision of a static analyzer for JavaScript. The individual methods are well known, but to our knowledge rarely used systematically and in combination.

6.1 Introduction

Analysis soundness Static analysis of programs written in mainstream programming languages inevitably involves approximation. Analysis designers often strive toward soundness, meaning that the analysis should consider every possible execution of the program being analyzed. Practically all analyzers deliberately treat some language features unsoundly, for example regarding reflection or native code [95].

However, unsoundness may also be caused by errors in the design or the implementation of the analysis. Such errors are easily overlooked—the analysis may produce a result, quickly and with good precision, but nevertheless a wrong result. How can the developer of a static analyzer detect such errors?

One way to ensure soundness is to make a formal proof. Sometimes this is done for key parts of the analysis design, but rarely for the entire analysis, and even more rarely for the actual implementation. One notable exception is the Verasco analyzer [75], which has been specified and proven sound using Coq. Despite the relative simplicity of that analyzer, the proof burden was massive, and the approach is hardly feasible for static analysis development in general.

Instead of requiring analyzers to be *provably* sound, we aim for making them *probably* sound, which can be achieved using thorough, automated testing. One such approach is property-based testing (i.e., quickchecking), which has been shown in previous work [101] to be an effective technique for detecting errors in static analyzers, by exploiting the generic algebraic properties of lattices and dataflow constraints. In this paper, we describe our experience with another pragmatic technique that we call *soundness testing*. The idea is simple and unsurprising: after a program has been analyzed statically, we compare the analysis results with the concrete states that are observed by a dynamic analysis of the program. If the information produced by the static analysis does not over-approximate the information obtained from the executions, a soundness error has been detected.

Analysis precision Another important aspect of static analysis design is precision. As approximation is inevitable and higher precision generally implies higher worst-case complexity, the right choice of abstractions can only be determined experimentally. Analysis precision is usually measured using some analysis client, for example the ability to prove absence of certain kinds of errors in the programs being analyzed. However, internal analysis metrics, such as sizes of points-to sets or degrees of suspiciousness of abstract values [10] may also provide valuable hints to where it may be advantageous to improve the analysis abstractions. In this paper we focus on another technique to investigate analysis precision problems, which is inspired by the work on *blended analysis* [44, 142]. A blended analysis is a static analysis that uses observations from a dynamic analysis to unsoundly approximate the program behavior. Tuned static analysis [83] is a related technique that uses unsound static pre-analysis instead of dynamic analysis. The previous work on blended and tuned analysis is about specific analysis algorithms that increase precision (by sacrificing soundness), whereas our goal here is to systematically identify opportunities for improving an analysis design.

Although our approach is inspired by blended analysis, it is also related to the recently proposed process called *root-cause localization and remediation* [144] for supporting the design of JavaScript analyses. That process involves a static analysis that automatically identifies where it looses precision, and a mechanism for suggesting alternative context sensitivities for those locations based on dynamic analysis.

Delta debugging In addition to the use of soundness testing and blended analysis, soundness problems and precision problems are both amenable to *delta debugging* [148]. This is an effective technique to reduce the size of inputs (e.g. programs to be analyzed) while preserving problematic behavior, which in our case is unsoundness or low precision.

Contributions In this paper we briefly describe our experience with soundness testing and blended analysis as methods for increasing soundness and precision of the TAJS [10, 70] static analyzer for JavaScript. Both of these techniques rely on information recorded by the same dynamic analysis. We have used both soundness testing and blended analysis in combination with delta debugging to identify causes and importance of soundness and precision problems. The methods are semi-automated and tightly integrated into the TAJS infrastructure, and they have become essential tools for guiding our continuous development of the analyzer.

TAJS We believe the techniques we present are broadly applicable to static analysis development in general, but we here focus on the TAJS analyzer.[1] In brief, TAJS is a whole-program dataflow analyzer for JavaScript, aiming to infer type-related properties involving the flow of primitive values, objects, and functions in the programs being analyzed. It supports most ECMAScript 5 features, including the native library and large parts of modern browser API and HTML DOM functionality.

Regarding soundness, we face several challenges. The language itself is extremely complex, there is a substantial native library specified in the ECMAScript standard, and the browser API and HTML DOM are not only massive but also poorly documented and constantly evolving. Tiny errors in the models can easily cause serious soundness issues that may affect validity of experimental results if not detected.

Regarding precision, we (and many others) have found that some programming patterns that are common in widely used JavaScript libraries require extraordinary analysis precision, and that inadequate precision often renders even small programs unanalyzable due to avalanches of spurious dataflow [10, 80, 83, 127, 144]. Here, "unanalyzable" means, for example, that the analysis (spuriously) finds that `eval` is called with an unknown string as argument.

6.2 Basic Techniques

In this section we briefly describe the three basic techniques with examples of how we have used them in our ongoing development of TAJS.

6.2.1 Delta Debugging

Delta debugging [148] is a technique for automated debugging of programs. The essence of the technique is that a large input is reduced systematically while preserving

[1]TAJS is available at `http://www.brics.dk/TAJS/`.

```
var a, b, x;
a = {p: 0, q: 0};
b = [];
for (var p in a)
 b.push(p);
x = b[0]
a[x] = b[x];
a.p();
```

(a) Reduced program for 5 versions of underscore.js.

```
var a, x;
a = {};
a.p = 0;
b.q = 0;
for (var i = 0; i++) {
 x = Object.keys(a)[i];
 this[x] = a[x];
}
```

(b) Reduced program for 11 versions of lodash.js.

Figure 6.1: Delta debugging precision problems.

some specific buggy behavior. The output is valuable because it makes it easier to understand *why* the buggy behavior arises. A delta debugging session takes two inputs: (1) an input program to reduce, and (2) a predicate that determines if the (reduced) program exhibits some specific behavior. The output is a smaller program exhibiting this behavior. We will use the term “delta debugging” throughout this text, although a more appropriate name is the generalized concept of “cause reduction” [53]. The difference is that we are using the technique to identify causes of a wide range of analysis behaviors, and not just buggy behaviors.

Example While further developing our static determinacy analysis technique [10], we observed that several utility libraries[2], each consisting of thousands lines of code, were unanalyzable. To investigate whether the libraries contained common patterns that were problematic for TAJS to analyze, we applied delta debugging using the predicate that TAJS should not be able to analyze the library within 5 minutes. Manually inspecting the outputs that were produced for the different versions of the libraries quickly revealed a small, common pattern for each library. Fig. 6.1a shows the resulting reduced program that was common to all five different versions of the underscore.js library. The manageable size (only 8 lines) made it possible to determine the root cause of the critical precision loss: The entries of an array are mixed together since the iteration order of `for-in` loops is implementation-specific according to the ECMAScript standard. The lost precision eventually causes spurious dataflow to a large number of native functions at the method call in line 8. A similar reduced program for a common pattern in 11 versions of the lodash.js library can be seen in Fig. 6.1b.

Delta debugging in practice The JavaScript delta debugger JSDelta[3] has been used by several JavaScript research groups. JSDelta systematically simplifies a JavaScript program by performing statement deletion, sub-expression simplification, and general purpose program optimizations, such as function inlining.

[2]`http://underscorejs.org/`, `https://lodash.com/`, among others
[3]`https://github.com/wala/jsdelta`

JSDelta is integrated with TAJS through a simple Java interface. To start delta debugging, a predicate is implemented in 5–10 lines of Java code, and the delta debugging main method is executed through the IDE with the predicate and some input program. This often reduces a few thousand lines of code to less than 20 within a few hours, fully automatically. The integration through Java also enables highly specialized predicates. As an example, a specialized predicate has been used to understand surprising differences between two slightly different analysis configurations. This predicate was defined to determine whether one configuration would lead to a particular kind of flow graph node being processed significantly more often than with the other configuration.

Although a delta debugger in principle can produce output that contains problems that are not present in the input, we have found that situation to be rare. In practice, the output usually exhibits the problem that was also present in the input, which makes the approach useful for understanding analysis limitations.

6.2.2 Soundness Testing

We use the term *soundness testing* to denote the process of checking whether observations in a concrete execution of a program are subsumed by the results computed by the static analysis. Soundness testing has been applied in various ways to many static analyzers. A notable example of this is a study of the consequences of *deliberate* unsoundness in the Clousot analyzer [27]. An interesting conclusion in that work is that Clousot often encounters unsoundness in practice but nevertheless rarely misses alarms.

Value logging An important design choice when performing soundness testing is deciding what information to include from the dynamic executions. We have chosen a simple approach based on *value logs*, which consist of the values of expressions that are computed during the execution of a program. Other options include recording the call graph [144], statement traces [142], or state snapshots [47, 118]. We have found the simpler value logs sufficient for our purposes.

An example program and (a simplified version of) its value log produced by our tool can be seen in Fig. 6.2. The property access on Line 145 of the program is represented by the first two lines of the value log. It has been logged that the property access occurred on the object allocated at position `1:8` in the program (`BASE`), and that the result is the string “`foo,bar`” (`PROP`). Similarly, the call to `split` on Line 146, is represented by the three last lines of the value log.

A notable design choice for our value logs is that we abstract away from execution order. This allows us to eliminate duplicate entries, which leads to a considerable reduction of the sizes of the logs.

Another important choice is that the value logs do not contain information about call stacks or scope chains of function objects. As mentioned, TAJS is partly context-sensitive, but it is extremely difficult to implement a faithful mapping from, for example, runtime call stacks to the abstract notion of contexts used by the static

```
144 var o = { p: 'foo,bar' };
145 var s = o.p;
146 var a = s.split(',');
```

```
f.js:18:9 BASE OBJECT(f.js:17:9)
f.js:18.9 PROP STRING("foo,bar")
f.js:19:9 BASE STRING("foo,bar")
f.js:19:9 CALLEE BUILTIN(String.prototype.split)
f.js:19:9 ARG0 STRING(",")
```

Figure 6.2: A program (top) and its value log (bottom).

analysis. This means that when checking subsumption of the concrete values and the abstract values, we can only report a soundness error if a given concrete value is not subsumed by the corresponding abstract values for *all* contexts.

Value logging in practice We obtain value logs with a dynamic analysis implemented using Jalangi [128]. The program of interest is instrumented and executed such that observations about runtime values are recorded. When the execution ends, the observations are post-processed into a value log, which is then persisted for reuse. The value log also contains metadata, for example a checksum of the program code so that we can easily detect if the program has been modified and the value log should be recreated.

The logging mechanism also supports different environments, enabling the creation of value logs for plain ECMAScript applications, Node.js applications, and browser-based applications. For example, if a log file is missing for a browser-based application, a browser is spawned to load the instrumented application, making it easy to manually interact with it and decide when to stop recording.

Example As an example of a failing soundness test, consider the code and the value log in Fig. 6.2. If the analysis is missing a model for the `split` property of string objects, then our soundness testing tool fails with the following report:

```
Soundness testing failed for 1/5 checks:
 - CALLEE on program Line 146:
  - concrete: BUILTIN(String.prototype.split)
  - abstract: {undefined}
```

In this case, it is easy for the analysis designer to spot and fix the root cause of the unsoundness. All that is needed is a model of the built-in function `String.prototype.split`.

```
Soundness testing succeeded for 5/5 checks
```

Soundness errors can easily spread in less obvious ways: a missing assignment to a field can cause the soundness check of the subsequent reads of that field to fail because of an unsound value rather than missing dataflow. Such extraneous soundness errors can make it harder to deduce the root cause of unsoundness. Furthermore, there may be multiple root causes of a failing soundness test, which can also make it harder to identify a single one of them. In Section 6.3.1 we present a technique for remedying these situations.

Soundness testing in practice Soundness testing is integrated into TAJS' regression test system and has been successful in uncovering many subtle soundness bugs. Initially, we found bugs in the core parts of the analysis, but recently mostly in the models of the huge, complex, and constantly evolving native libraries. For this reason we are planning to apply deeper checks of values originating from the native libraries, for example, not just checking that a value expected to be an object (rather than a primitive value) really is an object, but also that the object has the right properties.

Soundness errors sometimes result in highly inaccurate analysis results, which may be difficult to notice without soundness testing. However, as also observed by Christakis et al. [27], unsoundness can be benign, in the sense that it sometimes influences only a few nearby statements and not the remainder of the program, nor the analysis output.

In TAJS, we maintain a catalog of known soundness errors, which are then ignored when running the soundness tests. Most of these known errors have been added to the catalog because they have been classified as benign. This catalog helps to document the unsoundness in TAJS, as advocated by the soundiness manifesto [95]. It also helps prioritizing which soundness errors to fix.

At the time of writing, the main regression test suite of TAJS contains approximately 2 200 successfully soundness tested JavaScript programs, comprising 900 000 individual soundness checks for 100 000 syntactic locations. Only around 100 of the soundness checks fail, due to around 20 different soundness bugs that are caused by, for example, inadequate modeling of the HTML DOM.

6.2.3 Blended Analysis

An easy way to increase the precision of a static analysis is to replace parts of the abstract states with concrete states obtained by a dynamic analysis. While this is obviously not sound in general, it is sound relative to the execution path taken by the dynamic analysis.

The idea is not new. Blended analysis [44, 142] for Java and JavaScript allows the analysis to follow the control flow of the concrete execution. The TamiFlex tool [18] uses the same approach to handle Java reflection. Dynamic determinacy analysis [127] for JavaScript is based on a similar idea but retains soundness by only using dynamic information that is valid for all executions.

We apply this technique by leveraging the value logs that we already have from soundness testing as described in Section 6.2.2. When analyzing a program, the associated value log can be queried for the concrete values at a program location. The abstract value for that program location can then be refined by *intersecting* (technically, applying the greatest lower bound) with the abstraction of the concrete values. Another option would be to *replace* the abstract value with the abstraction of the concrete

```
876 ...
877 eval(code); // code is unknown
878 ...
```

```
f.js:877:1 ARG0 STRING("print('Same')")
```

Figure 6.3: A program with `eval` and a line from its value log.

values, but since the value logs do not record any control flow information, that would generally be less precise.

In practice, our value logs are more detailed than presented in Section 6.2.2. We record some relational information, for example, at a property write, we log the base object, the property value, and the value to be written. Thereby, when refining, for example, the abstract value being written, we can ignore concrete values that apply to other abstract objects and other property values, which increases precision.

Example As a TamiFlex-like example, a static analysis for JavaScript can use blended analysis for the argument to `eval`.[4] Consider the program and its associated value log in Fig. 6.3. Without having support for determining that the variables `x` and `y` always have the same value, the analysis is able to evaluate the `eval` call as the code `print('Same')`.

A similar use of blended analysis that enables focused prototype analysis design is to obtain call and points-to graphs from the value log instead of approximating them soundly.

Blended analysis in practice The use of blended analysis makes it possible to circumvent challenging language or library features, allowing the analysis designer to proceed with other aspects of the analysis.

We mostly use blended analysis in combination with other techniques, as we describe in Sections 6.3.2 to 6.3.4. However, we have also used the approach to investigate “best-case scenarios” for analyzing large JavaScript applications that are beyond reach of all existing sound JavaScript analyzers. By applying blended analysis aggressively—at *all* program locations—we can test the analyzer for fundamental scalability problems and logical implementation errors. Any errors that are detected in such a scenario also exist without enabling blended analysis but may be more difficult to find without it.

6.3 Combining the Basic Techniques

The basic techniques introduced in the previous section can be combined to create some particularly powerful techniques that guide the design of static analyses.

[4]We note that TAJS is able to handle some common occurrences of `eval`.

```
880 var i, s;
881 i = "0";
882 s = i++;
```

```
Soundness testing failed:
 - VAR 's' on program Line 882:
  - concrete: NUMBER(0)
  - abstract: {STRING("0")}
```

Figure 6.4: Reduced program with a subtle soundness error.

6.3.1 Soundness Testing and Delta Debugging

As stated in Section 6.2.2, soundness testing can expose soundness bugs, but it is often difficult to locate the cause of a failing soundness test if the program being analyzed is large or if the bug causes many soundness checks to fail. Delta debugging is extremely useful in these cases. Each iteration of this delta debugging process works as follows. (1) run the program concretely to obtain a value log, (2) analyze the program, (3) perform soundness testing of the result from step 2 using the value log from step 1. The delta debugging predicate is that step 3 results in one or more failing soundness checks. Delta debugging then automatically produces a small program containing a soundness error.

If one wants to target a specific soundness error, then the predicate can be refined to consider only that particular error. Nevertheless, any reduced program with a soundness error is valuable even after the error has been fixed, since their small sizes make them useful as fast regression tests.

Example Fig. 6.4 shows a reduced version of an unsoundly analyzed program, together with the failing soundness test. This small program was produced starting from a large program that at that time had thousands of soundness failures. The reduced program exposed that the value of a postfix expression in JavaScript is, perhaps surprisingly, the number-coerced value and not the original value. In this example, it turned out that the exposed soundness bug was benign, and after fixing it the original program still had thousands of soundness failures. However, repeating the process quickly revealed that those failures all had the same root cause and could also easily be fixed.

6.3.2 Soundness Testing and Blended Analysis

By combining soundness testing and blended analysis, it is possible to detect soundness errors even in programs that are unanalyzable (in the sense described in Section 9.1) when using the ordinary analysis! Using the same dynamic information for the two purposes, any failures that are detected during the soundness testing must be due to unsoundness in the analysis, and *not* due to the under-approximation introduced by the use of blended analysis.

```
1489 ...
1490 _.mixin = function(obj) {
1491  _.each(_.functions(obj), function(name) {
1492   var func = _[name] = obj[name];
1493   _.prototype[name] = function() {
1494    func.apply(_, arguments);
1495   };});};
1496 _.mixin(_);
1497 ...
```

Figure 6.5: Excerpt from problematic underscore.js code.

Example Consider the soundness error below:

```
Soundness testing failed for 43/3932 checks:
 - PROP on program line 542:
  - concrete: BUILTIN(Symbol.unscopables)
  - abstract: {undefined}
```

It reveals that the program being analyzed uses the `Symbol` ECMAScript 6 feature, which was not yet fully modeled in TAJS at the time this test was run. Without the use of blended analysis, the program was unanalyzable due to inadequate analysis precision, and it would have been difficult to tell that the feature was not just encountered due to spurious dataflow.

6.3.3 Delta Debugging and Blended Analysis

We can also combine delta debugging and blended analysis. This time, delta debugging is not instantiated with a program, but instead with a set of program locations where blended analysis is allowed. This combination of techniques gives a way of finding a minimal set of locations that need to be handled precisely by the static analysis.

Delta debugging is initiated with the set of all locations in the program to be analyzed and the predicate that TAJS can analyze the program, for example within one minute, by applying blended analysis in the current set of locations. The outcome is a reduced set of locations where blended analysis is critical. Manually inspecting those locations often gives good hints for improving the analysis design.

We find that the resulting number of locations is usually below 5, which supports the claim that few root causes of imprecision can render the analysis result useless [144].

Examples Using this technique to investigate causes of precision problems when analyzing various small applications of the underscore.js library resulted in the following automatically generated report.

```
underscore-1.8.3.js needs more precision at:
- PROPERTY WRITE at line 1492
```

The relevant piece of code is shown in Fig. 6.5 (line numbers have been modified to match the figure). Blended analysis was only needed in a single location, which means that improving TAJS to be able to handle this particular location precisely was the key to analyze the entire program. TAJS' problem with underscore.js was that the abstract value of `name` was an unknown string, so each property of `obj` was conservatively written to each property of the library object, thereby introducing a critical loss of precision.

Further in our investigation involving a more complicated application of the library, we got this report:

```
underscore-1.8.3.js needs more precision at:
- PROPERTY WRITE at line 1492
- CALL at line 1494
```

This time, one additional location needed more precision. The problem was that `func` could be every element in `obj`, which would cause TAJS to conservativly model calls to every function of `obj`, making the program unanalyzable.

Using this technique to systematically investigate precision problems with a range of applications of the library, we obtained an overview of the various precision bottlenecks, which was useful for prioritizing our effort and designing useful improvements of the analysis abstractions.

6.3.4 Combining All Three Techniques

As discussed in Section 6.3.2, combining blended analysis and soundness testing makes it possible to detect soundness errors even with programs that cannot be analyzed by TAJS. It is not always easy to identify the root cause of such a soundness error, but again we can make use of delta debugging to automatically produce a small program that is analyzed unsoundly. Compared to the combination of soundness testing and delta debugging described in Section 6.3.1, we now use the value log that is created in each delta debugging step both for soundness testing and for blended analysis.

Example As mentioned in Section 6.3.3, TAJS could not analyze simple applications of the underscore.js library, meaning that without blended analysis, we could not use those JavaScript programs to detect soundness errors. By combining the three techniques we not only detected a soundness error, but we also obtained a reduced program containing the error. The reduced program and the soundness test report are shown in Fig. 6.6. It turned out that the analysis did not properly support accessing properties of the special `arguments` object outside of its declaring function. The reduced program made it easy to locate the cause. The single fix reduced thousands of failing soundness checks to zero, since the soundness error influenced the dataflow in the rest of the program.

```
1503 function f(){
1504  return arguments;
1505 }
1506 f().p;
```

```
Soundness testing failed:
- PROP on program line 1506:
  - concrete: UNDEFINED
  - abstract: {}
```

Figure 6.6: Identifying the cause of unsoundness from an unanalyzable program.

6.4 Conclusion

We have presented our experience with soundness testing, blended analysis, and delta debugging for systematically guiding improvements of soundness and precision of the TAJS static analyzer. Both soundness testing and blended analysis build on top of value logs obtained by dynamic analysis. Other useful techniques, such as quickchecking and suspiciousness metrics, are described elsewhere [10, 101],

Our experience can be summarized as the following recommendations to static analysis developers:

- Use dynamic analysis to record value logs for all benchmark programs. The value logs are useful for improving both soundness and precision as the analysis design and implementation evolves.
- Use soundness testing as an integrated part of the development, and maintain a catalog of known soundness issues. When soundness errors appear, use delta debugging to quickly identify the cause.
- When precision problems appear, use blended analysis to investigate how alternative analysis abstractions may help. Combining blended analysis with delta debugging can often automatically locate the critical places where extra precision is needed.
- By combining soundness testing, blended analysis, and delta debugging, it is possible to quickly identify soundness errors even for programs that are unanalyzable due to insufficient analysis precision.

Chapter 7

Static Analysis with Demand-Driven Value Refinement

By Benno Stein (University of Colorado Boulder, USA), Benjamin Barslev Nielsen (Aarhus University, Denmark), Bor-Yuh Evan Chang (University of Colorado Boulder, USA) and Anders Møller (Aarhus University, Denmark). Published in the Proceedings of the ACM on Programming Languages, Volume 3, Issue OOPSLA, October 2019.

Abstract

Static analysis tools for JavaScript must strike a delicate balance, achieving the level of precision required by the most complex features of target programs without incurring prohibitively high analysis time. For example, reasoning about dynamic property accesses sometimes requires precise relational information connecting the object, the dynamically-computed property name, and the property value. Even a minor precision loss at such critical program locations can result in a proliferation of spurious dataflow that renders the analysis results useless.

We present a technique by which a conventional non-relational static dataflow analysis can be combined soundly with a value refinement mechanism to increase precision on demand at critical locations. Crucially, our technique is able to incorporate relational information from the value refinement mechanism into the non-relational domain of the dataflow analysis.

We demonstrate the feasibility of this approach by extending an existing JavaScript static analysis with a demand-driven value refinement mechanism that relies on backwards abstract interpretation. Our evaluation finds that precise analysis of widely used JavaScript utility libraries depends heavily on the precision at a small number of critical locations that can be identified heuristically, and that backwards abstract interpretation is an effective mechanism to provide that precision on demand.

7.1 Introduction

Although the many dynamic features of the JavaScript programming language provide great flexibility, they also make it difficult to reason statically about dataflow and control-flow. Several research tools, including TAJS [70], WALA [133], SAFE [91], and JSAI [80], have been developed in recent years to address this challenge. A notable trend is that analysis precision is being increased in many directions, including high degrees of context sensitivity [10], aggressive loop unrolling [115], and sophisticated abstract domains for strings [8, 96, 116], to enable analysis of real-world JavaScript programs.

The need for precision in analyzing JavaScript is different from other programming languages. For example, it is widely recognized that, for Java analysis, choosing between different degrees of context sensitivity is a trade-off between analysis precision and performance. With JavaScript, the relationship between precision and performance is more complicated: low precision tends to cause an avalanche of spurious dataflow, which slows down the analysis and often renders it useless [10, 133].

Unfortunately, uniformly increasing analysis precision to accommodate the patterns found in JavaScript programs is not a viable solution, because the high precision that is critical for some parts of the programs may be overkill for others. For example, the approach taken by SAFE performs loop unrolling indiscriminately whenever the loop condition is determinate [115], which is often unnecessary and may be costly. Another line of research attempts to address this problem by identifying specific syntactic patterns known to be particularly difficult to analyze and applying variations of trace partitioning to handle those patterns more precisely [84, 85, 133].

In this work, we explore a different idea: instead of pursuing ever more elaborate abstract domains, context-sensitivity policies, or syntactic special-cases, we augment an existing static dataflow analysis by a novel *demand-driven value refinement* mechanism that can eliminate spurious dataflow at critical places with a targeted backwards analysis. Our approach is inspired by Blackshear et al. [16], who introduced the use of a backwards analysis to identify spurious memory leak alarms produced by a Java points-to analysis. We extend their technique by applying the backwards analysis *on-the-fly*, to avoid critical precision losses during the main dataflow analysis, rather than as a post-processing alarm triage tool. Also, our technique is designed to handle the dynamic features of JavaScript that do not appear in Java.

We find that demand-driven value refinement is particularly effective for providing precise *relational* information, even though the abstract domain of the underlying dataflow analysis is non-relational. Such relational information is essential for the precise analysis of many common dynamic language programming paradigms, especially those found in widely-used libraries like Underscore[1] and Lodash[2] that rely heavily on metaprogramming.

An important observation that enables our approach is that the extra precision

[1]`https://underscorejs.org/`
[2]`https://lodash.com/`

is typically only critical at very few program locations, and that these locations can be identified during the main dataflow analysis by inspecting the abstract values it produces.

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

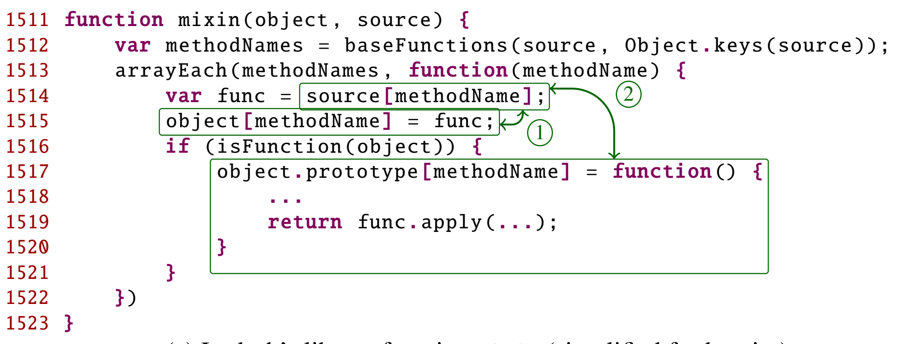
- We present the idea of demand-driven value refinement as a technique for soundly eliminating critical precision losses in static analysis for dynamic languages (Section 7.4). For clarity, the presentation is based on a dataflow analysis framework for a minimal dynamic programming language (Section 7.3).
- We present a separation logic-based backwards abstract interpreter, which can answer value refinement queries to precisely refine abstract values and provide relational precision to the non-relational dataflow analysis as an abstract domain reduction. This backwards analysis is first described for the minimal dynamic language (Section 7.5) and then for JavaScript (Section 7.6).
- We empirically evaluate our technique using an implementation, TAJS$_{\text{VR}}$, for JavaScript (Section 12.7). We find that demand-driven value refinement can provide the necessary precision to analyze code in libraries that no existing static analysis is able to handle. For example, the technique enables precise analysis of 266 of 306 test cases from the latest version of Lodash (the most depended-upon package in the npm repository), all of which are beyond the reach of other state-of-the-art analyses.

7.2 Motivating Example

The example program in Fig. 7.1 is an excerpt from Lodash 4.17.10. It consists of a library function `mixin` (Fig. 7.1a) and a simple use of `mixin` (Fig. 7.1b) originating from the bootstrapping of the library. The `mixin` function copies all methods from the `source` parameter into the `object` parameter. If `object` is a function, the methods are also written to the prototype of `object`, such that instantiations of `object` (using the keyword **`new`**) also have the methods. The function `baseFor` invokes the `iteratee` function on each property of the given object. The `mixin` function is called in line 1534, where the first argument is the `lodash` library object and the second argument is an object, created using `baseFor`, that consists of all properties of `lodash` that are not already in `lodash.prototype`. The purpose of this call to `mixin` is to make each method that is defined on `lodash`, for example `lodash.map`, available on objects created by the `lodash` constructor.

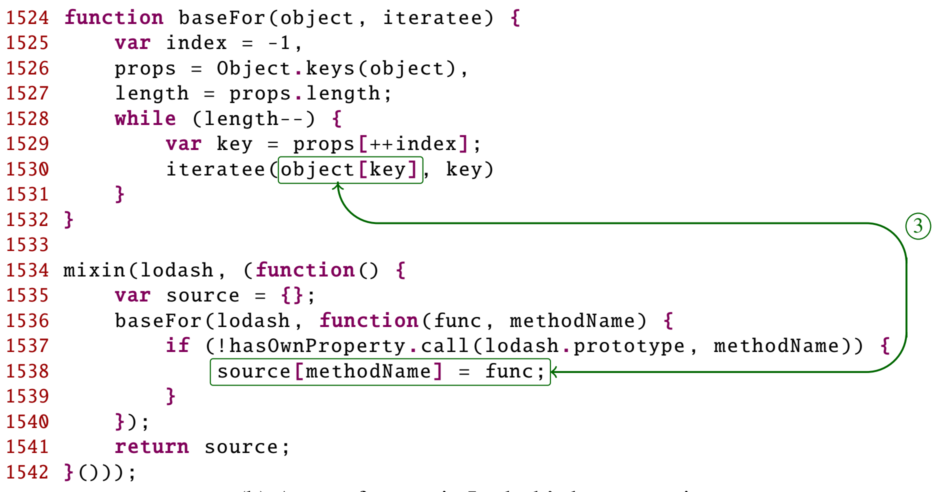
This code contains three dynamic property read/write pairs – indicated by the labels ①, ②, and ③ in Fig. 7.1 – where relational information connecting the property name of the read and write is essential to avoid crippling loss of precision.

All three labelled read/write pairs are instances of a metaprogramming pattern called *field copy or transformation (FCT)* [84, 85], which consists of a property read operation `x[a]` and a property write operation `y[b] = v` where the property name `b`



```
function mixin(object, source) {
    var methodNames = baseFunctions(source, Object.keys(source));
    arrayEach(methodNames, function(methodName) {
        var func = source[methodName];
        object[methodName] = func;
        if (isFunction(object)) {
            object.prototype[methodName] = function() {
                ...
                return func.apply(...);
            }
        }
    })
}
```

(a) Lodash's library function `mixin` (simplified for brevity).



```
function baseFor(object, iteratee) {
    var index = -1,
    props = Object.keys(object),
    length = props.length;
    while (length--) {
        var key = props[++index];
        iteratee(object[key], key)
    }
}

mixin(lodash, (function() {
    var source = {};
    baseFor(lodash, function(func, methodName) {
        if (!hasOwnProperty.call(lodash.prototype, methodName)) {
            source[methodName] = func;
        }
    });
    return source;
}()));
```

(b) A use of `mixin` in Lodash's bootstrapping.

Figure 7.1: Excerpts from the Lodash library. Dynamic property writes that require relational information to analyze precisely are highlighted by arrows connecting them to corresponding dynamic property reads.

is a function of the property name `a` and the written value `v` is a function of the read value `x[a]`. This pattern is a generalization of the *correlated read/write* pattern [133], which requires that the property names are strictly equal. Our technique attempts to generalize such syntactic patterns by identifying imprecise dynamic property writes *semantically* during the dataflow analysis. The benefits of this semantic approach are twofold: it avoids the brittleness of syntactic patterns, and it only incurs additional analysis cost where needed, rather than at all locations matching a syntactic pattern.

Analysis precision at dynamic property read and write operations is known to be critical for static analysis for JavaScript programs [10, 84, 85, 133]. If the abstract values of the property names are imprecise, then a naive analysis will mix together the values of the different properties, which often causes a catastrophic loss of precision. In the case of Lodash, such a naive analysis of the bootstrapping code would essentially cause all the library functions to be mixed together, making it impossible to analyze any realistic applications of the library.

Existing attempts to address this problem are unfortunately insufficient. WALA [133], SAFE$_{LSA}$ [115], and TAJS [10] use context-sensitivity and loop-unrolling to attempt to obtain precise information about the property names, but fail to provide precise values of the variables `methodName` (at lines 1514, 1515, and 1538) and `key` (at line 1530).[3]

The CompAbs analyzer [84, 85] takes a different approach that does not require precise tracking of the possible values of `methodName` and `key`. Instead, it attempts to syntactically identify correlated dynamic property reads and writes and applies trace partitioning [122] at the relevant property read operations. However, CompAbs fails to identify any of the three highlighted read/write pairs in Fig. 7.1 due to the brittleness of syntactic patterns. While it might be possible to detect ② syntactically, the trace partitioning approach is insufficient for that read/write pair, since the value in this case flows through a free variable (`func`) that is shared across all partitions. As a result, CompAbs fails to analyze the full Lodash library with sufficient precision and ends up mixing together all properties of the `lodash` object.

Triggering Demand-Driven Value Refinement Our approach is able to achieve sufficient precision for all three dynamic property read/write pairs in the example without relying on brittle syntactic patterns to identify such pairs.

The key idea underlying our technique is to detect *semantically* when an imprecise property write is about to occur during the dataflow analysis, at which point we apply a targeted value refinement mechanism to recover the relational information needed to precisely determine which values are written to which heap locations. More specifically, when the analysis encounters a dynamic property write `obj[p] = v` and has imprecise abstract values for `p` and `v`, we decompose the abstract value of `v` into a set of more precise partitions and then query the refinement mechanism to determine, for each partition, the possible values of `p`. Now, instead of writing the imprecise value of `v` to all the property names of `obj` that match the imprecise value of `p`, we write each partition of `v` only to the corresponding refined property names of `obj`, thereby recovering relational information between `p` and `v`.

For example, suppose that the dataflow analysis reaches the dynamic property write of ③ (at line 1538) with an abstract state mapping the property name `methodName` to the abstract value denoting any string and the value `func` to the abstract value that abstracts all functions in `lodash`. We then decompose `func` into precise partitions – one for each of the functions – and query the value refinement mechanism for the corresponding sets of possible property names. Recovering that relational information, we obtain a unique function for each property name, such that the `lodash.map` function is assigned to `source["map"]`, `lodash.filter` is assigned to `source["filter"]`, etc.,

[3]For example, achieving sufficient precision for `methodName` at lines 1514 and 1515 requires that the analysis can infer the precise length and contents of the `methodNames` array, which is beyond the capabilities of those analyzers, even if using an (unsound) assumption that the order of the entries of the array returned by `Object.keys` is known statically.

instead of mixing them all together. This technique handles case ① analogously, by detecting the imprecision semantically at the dynamic property write.

For property read/write pair ②, however, the value to be written is a precise function (the anonymous function in lines 1517–1520) and the imprecise value `func` is a free variable declared in an enclosing function. In this case, value refinement is not triggered until `func` is called on line 1519, at which point we apply the same value refinement technique as above and recover the necessary relational information to precisely resolve the target of that call, as described in further detail in Section 7.6.2.

Unlike abstraction-refinement techniques (see Section 7.8), this mechanism is able to recover relational information and use it to regain precision in the dataflow analysis without any modifications to its non-relational abstract domain and without restarting the entire analysis.

Value Refinement using Backwards Analysis Our value refinement mechanism is powered by a goal-directed backwards abstract interpreter. Given a program location ℓ, a program variable y, and a constraint ϕ, it computes a bound on the possible values of y at ℓ in concrete states satisfying ϕ. We refer to the forward dataflow analysis as the *base analysis* and the backwards analysis as the *value refiner*.

For example, if asked to refine the variable `methodName` at the location preceding line 1538, under the condition that `func` is the `lodash.map` function, our value refiner can determine that `methodName` must be `"map"`. In doing so, the value refiner provides targeted information about the relation between `func` and `methodName` to the base analysis.

Intuitively, the value refiner works by overapproximately traversing the abstract state space backwards from the given program location, accumulating symbolic constraints about the paths leading to that location. The traversal proceeds along each such path until a sufficiently precise value is obtained for the desired program variable. In this process, the value refiner takes advantage of the current call graph and the abstract states computed so far by the base analysis. The resulting abstract value thereby overapproximates the possible values of y at ℓ, for all program executions where ϕ is satisfied at ℓ and that are possible according to the call graph and abstract states from the base analysis. As we argue in Sections 7.4 and 12.7, the value refinement mechanism is sound even though it relies on information from the base analysis that has not yet reached its fixpoint.

7.3 A Simple Dynamic Language and Dataflow Analysis

To provide a foundation for explaining our demand-driven value refinement mechanism, in Section 7.3.1 we define the syntax and concrete semantics of a small dynamic language. This language is designed for simplicity and clarity of presentation and is meant to illustrate some of the core challenges in dynamic language analysis without the complexity that arises from a tested core calculus for JavaScript such as λ_{JS} [56].

$$
\begin{array}{rlll}
\text{variables} & x,y,z \in \mathit{Var} & & \\
\text{primitives} & p \in \mathit{Prim} & ::= & \texttt{undef} \mid \texttt{true} \mid \texttt{false} \\
 & & & \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \dots \\
 & & & \mid \texttt{"foo"} \mid \texttt{"bar"} \mid \dots \\
\text{statements} & s \in \mathit{Stmt} & ::= & x\texttt{=\{\}} \mid x\texttt{=}y \mid x\texttt{=}p \\
 & & & \mid x\texttt{=}y \oplus z \mid \texttt{assume}\ x \\
 & & & \mid x\texttt{[}y\texttt{]=}z \mid x\texttt{=}y\texttt{[}z\texttt{]} \\
\text{operators} & \oplus & ::= & + \mid - \mid = \mid \neq \mid \dots \\
\text{locations} & \ell \in \mathit{Loc} & & \\
\text{control edges} & t \in \mathit{Trans} & ::= & \ell \rightarrow_s \ell'
\end{array}
$$

Figure 7.2: Concrete syntax for a simple dynamic language.

We then define our analysis over this minimal language in Section 7.3.2 and describe the extensions needed to handle the full JavaScript language in Section 7.6.

7.3.1 A Simple Dynamic Language

The syntax and denotational semantics of our core dynamic language are shown in Fig. 7.2 and Fig. 7.3.

A program in this language is an unstructured control-flow graph represented as a pair $\langle \ell_0, T \rangle$ of an initial location $\ell_0 \in \mathit{Loc}$ and a set of control-flow edges $T \subseteq \mathit{Trans}$. A program location $\ell \in \mathit{Loc}$ is a unique identifier. A memory address $m \in \mathit{Mem}$ is either a program variable x or an object property (a, p) where a is the address of an object and p is a primitive value. Concrete states $\sigma \in \mathit{State}$ are partial functions from memory addresses to values, which are either object addresses or primitives.

We write ε for the empty state and use the notation $\sigma[m \mapsto v]$ to denote a state identical to σ except at location m where v is now stored, and σm to denote the value stored at m in σ or `undef` if no such value exists. We also assume a helper function $\mathsf{fresh}(\sigma)$ that returns an object address that is fresh with respect to state σ.

The denotation of a statement s is a partial function $[\![s]\!]$ from states to states. The collecting semantics of a program $\langle \ell_0, T \rangle$ is defined in terms of the denotational semantics as a function $[\![\_]\!]_{\langle \ell_0, T \rangle} : \mathit{Loc} \rightarrow \mathscr{P}(\mathit{State})$ that captures the reachable state space of the program, as the least solution to the following constraints:

$$
\varepsilon \in [\![\ell_0]\!]_{\langle \ell_0, T \rangle}
$$
$$
\forall \sigma \in [\![\ell]\!]_{\langle \ell_0, T \rangle} \text{ and } \ell \rightarrow_s \ell' \in T : [\![s]\!](\sigma) \in [\![\ell']\!]_{\langle \ell_0, T \rangle}
$$

The first constraint says that the empty state ε is reachable at the initial location. The second constraint defines the successor states according to the denotational semantics.

$$\begin{aligned}
\text{object addresses} \quad & a \in \mathit{Addr} & \\
\text{memory addresses} \quad & m \in \mathit{Mem} & = \mathit{Var} \cup (\mathit{Addr} \times \mathit{Prim}) \\
\text{values} \quad & v \in \mathit{Val} & = \mathit{Addr} \cup \mathit{Prim} \\
\text{states} \quad & \sigma \in \mathit{State} & = \mathit{Mem} \hookrightarrow \mathit{Val}
\end{aligned}$$

$$\begin{aligned}
[\![\cdot]\!] \; &: \; \mathit{Stmt} \to \mathit{State} \hookrightarrow \mathit{State} \\
[\![x\texttt{=\{\}}]\!](\sigma) &= \sigma[x \mapsto \mathsf{fresh}(\sigma)] \\
[\![x\texttt{=}y]\!](\sigma) &= \sigma[x \mapsto \sigma y] \\
[\![x\texttt{=}p]\!](\sigma) &= \sigma[x \mapsto p] \\
[\![x\texttt{=}y \oplus z]\!](\sigma) &= \sigma[x \mapsto \sigma y \oplus \sigma z] \\
[\![\texttt{assume}\; x]\!](\sigma) &= \sigma \quad \text{if } \sigma x = \texttt{true} \\
[\![x\texttt{[}y\texttt{]=}z]\!](\sigma) &= \sigma[(\sigma x, \sigma y) \mapsto \sigma z] \quad \text{if } \sigma x \in \mathit{Addr} \\
& \qquad\qquad\qquad\qquad\quad \text{and } \sigma y \in \mathit{Prim} \\
[\![x\texttt{=}y\texttt{[}z\texttt{]}]\!](\sigma) &= \sigma[x \mapsto \sigma(\sigma y, \sigma z)]
\end{aligned}$$

Figure 7.3: Denotational semantics and concrete domains.

7.3.2 Dataflow Analysis

We now describe a basic dataflow analysis for this minimal dynamic language, which we will extend in Section 7.4 with support for demand-driven value refinement.

The analysis is expressed as a monotone framework [76] consisting of a domain of abstract states and monotone transfer functions for the different kinds of statements. Programs can then be analyzed by a fixpoint solver computing an overapproximate abstract state for each program location.[4]

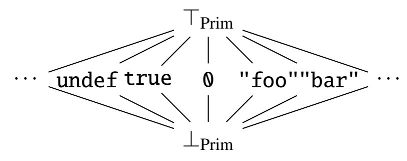
The analysis domain we use is the lattice L described in Fig. 7.4, which is a simplified version of the one used by TAJS [70]. For each program location, an element of L provides an abstract state, which maps abstract memory addresses to abstract values. Object addresses are abstracted using, for example, allocation-site abstraction [24]. The domain of abstract values, $\widehat{Val}$, is a product of two sub-domains describing references to objects and primitive values, respectively, using the constant propagation lattice to model the latter.

We write $\hat{a} \prec_1 \hat{v}$ if the first component of the abstract value $\hat{v}$ contains the abstract object address $\hat{a}$, and similarly, $\hat{p} \prec_2 \hat{v}$ means that the concretization of the abstract primitive value $\hat{p}$ is a subset of the concretization of $\hat{v}$. We use $\hat{v}_1 \sqcup \hat{v}_2$ to denote the least upper bound of $\hat{v}_1$ and $\hat{v}_2$.

[4]We assume the reader is familiar with the basic concepts of abstract interpretation [33], including the terms *abstraction*, *concretization*, *collecting semantics*, and *soundness*.

$$\hat{a} \in \widehat{Addr}, \hat{A} \subseteq \widehat{Addr}$$

$$\hat{p} \in \widehat{Prim} \quad =$$



$$\begin{aligned}
\hat{m} \in \widehat{Mem} &= Var \cup (\widehat{Addr} \times Prim) \\
\hat{v} \in \widehat{Val} &= \mathscr{P}(\widehat{Addr}) \times \widehat{Prim} \\
\hat{\sigma} \in \widehat{State} &= \widehat{Mem} \to \widehat{Val} \\
L &= Loc \to \widehat{State}
\end{aligned}$$

Figure 7.4: Dataflow analysis lattice.

The semantics of each control edge $\ell \to_s \ell'$ is modeled abstractly by the transfer function $\mathscr{T}_{\ell \to_s \ell'} : \widehat{State} \to \widehat{State}$. When the statement s is a dynamic property write $x[y]=z$, the transfer function is defined as follows:

$$\mathscr{T}_{\ell \to_s \ell'}(\hat{\sigma})(\hat{m}) = \begin{cases} \hat{\sigma}\hat{m} \sqcup \hat{\sigma}z & \text{if } \hat{m} = (\hat{a}, p) \wedge \hat{a} \prec_1 \hat{\sigma}x \wedge p \prec_2 \hat{\sigma}y \\ \hat{\sigma}\hat{m} & \text{otherwise} \end{cases}$$

In other words, the analysis models such an operation by weakly[5] updating all the affected abstract memory addresses with the abstract value of z. Due to the limited space, we omit descriptions of the remaining analysis transfer functions for other kinds of statements; it suffices to require that they soundly overapproximate the semantics [31].

The refinement mechanism directly involves the fixpoint solver. As customary in dataflow analysis, we assume the fixpoint solver uses a worklist algorithm to determine which locations to process next each time a transfer function has been applied [82]. A worklist algorithm relies on a map $dep : Loc \to \mathscr{P}(Loc)$, such that $dep(\ell)$ contains all direct dependents of ℓ, in our case the successors $\{\ell' \mid \ell \to_s \ell' \in Trans\}$. For a worklist algorithm to be sound, it must add all the locations $dep(\ell)$ to the worklist when the abstract state at ℓ is updated.

Example Fig. 7.5 shows a program fragment with a correlated property read/write pair like in Section 12.2. Assume $\hat{\sigma}$ is an abstract state where x and y point to distinct objects ($\hat{a}_{\text{x}}$ and $\hat{a}_{\text{y}}$, respectively), p is any primitive value, the x object has three

[5]Our implementation for JavaScript uses a more expressive heap abstraction that permits strong updates [24, 70] in certain situations.

$$\begin{tikzcd}\ell_1 \arrow[r, "\texttt{t = x[p]}"] & \ell_2 \arrow[r, "\texttt{y[p] = t}"] & \ell_3\end{tikzcd}$$

Figure 7.5: A program fragment with a correlated property read/write pair.

properties (named "a", "b", and "c") with different values, and the y object is empty:

$$\begin{array}{ll}
\hat{\sigma}\texttt{x} = (\{\hat{a}_{\texttt{x}}\}, \bot_{\text{Prim}}) & \hat{\sigma}(\hat{a}_{\texttt{x}}, \texttt{"a"}) = (\{\hat{a}_{\texttt{xa}}\}, \bot_{\text{Prim}}) \\
\hat{\sigma}\texttt{y} = (\{\hat{a}_{\texttt{y}}\}, \bot_{\text{Prim}}) & \hat{\sigma}(\hat{a}_{\texttt{x}}, \texttt{"b"}) = (\{\hat{a}_{\texttt{xb}}\}, \bot_{\text{Prim}}) \\
\hat{\sigma}\texttt{p} = (\emptyset, \top_{\text{Prim}}) & \hat{\sigma}(\hat{a}_{\texttt{x}}, \texttt{"c"}) = (\emptyset, \top_{\text{Prim}}) \\
 & \hat{\sigma}(\hat{a}_{\texttt{y}}, p) = (\emptyset, \bot_{\text{Prim}}) \quad \text{for all } p \in \mathit{Prim}
\end{array}$$

The transfer function for `t = x[p]` with this abstract state at initial program location ℓ_1 yields an abstract state at ℓ_2 that maps `t` to $(\{\hat{a}_{\texttt{xa}}, \hat{a}_{\texttt{xb}}\}, \top_{\text{Prim}})$. Next, the transfer function for `y[p] = t` as defined above results in an abstract state at ℓ_3 where every property of the abstract object $\hat{a}_{\texttt{y}}$ has the same abstract value as `t`, meaning that they all may point to any of the two abstract objects $\hat{a}_{\texttt{xa}}$ and $\hat{a}_{\texttt{xb}}$ and have any primitive value. Consequently, the analysis result is very imprecise. In the following section, we explain how the basic dataflow analysis can be extended with demand-driven value refinement to avoid the precision loss.

7.4 Demand-Driven Value Refinement

In this section, we introduce the notion of a *value refiner* (Section 7.4.1), discuss when and how to apply value refinement during the base analysis (Section 7.4.2), and explain how to integrate a value refiner into the base analysis in a way that allows the value refiner to benefit from the abstract states that are constructed by the base analysis (Section 7.4.3).

7.4.1 Value Refinement

A *value refiner* is a function

$$R : \mathit{Loc} \times \mathit{Var} \times \mathit{Constraint} \to \mathscr{P}(\widehat{\mathit{Val}})$$

that, given a program location $\ell \in \mathit{Loc}$, a program variable $y \in \mathit{Var}$, and a constraint $\phi \in \mathit{Constraint} = \mathit{Var} \times \widehat{\mathit{Val}}$ yields a set of abstract values that are possible for y at ℓ in states that satisfy ϕ. We refer to an invocation of the value refiner function by the base analysis as a *refinement query*. For the refinement queries we need, a constraint is simply a pair of a program variable and an abstract value, written $z \mapsto \hat{v}$, specifying that the variable z has value $\hat{v}$.

We require R to be *sound* in the sense that it overapproximates all possible behaviors of the program according to the collecting semantics: for every state

$\sigma \in [\![\ell]\!]_{\langle \ell_0, T\rangle}$ where ℓ is a location in the program $\langle \ell_0, T\rangle$ and the abstraction of σ satisfies the constraint ϕ, the value σy is in the concretization of an abstract value in $R(\ell, y, \phi)$ for any y.

In Section 7.5 we present a specific value refiner; for the remainder of the current section we can think of the value refiner as a black-box component with the above properties.

7.4.2 Using Value Refinement in Dataflow Analysis

Value refinement can in principle be invoked whenever the base analysis detects that a potentially critical loss of precision is about to happen, to provide more precise abstract values. As discussed in Section 12.2, such precision losses often occur in connection with dynamic property writes, so we here focus on that kind of operation. In Section 7.6.2, we consider value refinement also at variable read operations.

First, we define a helper function $Part : \widehat{Val} \to \mathscr{P}(\widehat{Val})$ that partitions an abstract value into a set of abstract values, each containing at most one abstract memory address:

$$Part(\hat{A}, \hat{p}) = \{\ (\{\hat{a}\}, \bot_{\mathrm{Prim}}) \mid \hat{a} \in \hat{A}\ \} \cup \begin{cases} \{(\emptyset, \hat{p})\} & \text{if } \hat{p} \neq \bot_{\mathrm{Prim}} \\ \emptyset & \text{otherwise} \end{cases}$$

Continuing the example from Section 7.3.2, we have:

$$Part(\{\hat{a}_{\mathsf{xa}}, \hat{a}_{\mathsf{xb}}\}, \top_{\mathrm{Prim}}) = \{(\{\hat{a}_{\mathsf{xa}}\}, \bot_{\mathrm{Prim}}), (\{\hat{a}_{\mathsf{xb}}\}, \bot_{\mathrm{Prim}}), (\emptyset, \top_{\mathrm{Prim}})\}$$

We now incorporate value refinement into the base analysis by replacing the ordinary transfer function $\mathscr{T}_{\ell \to_s \ell'}$ from Section 7.3.2 by a new transfer function $\mathscr{T}^{\mathsf{VR}}_{\ell \to_s \ell'}$. The domain of the base analysis remains unchanged (unlike traditional abstraction refinement techniques). The new transfer function is defined as follows when s is a dynamic property write statement x[y]=z:

$$\mathscr{T}^{\mathsf{VR}}_{\ell \to_s \ell'}(\hat{\sigma})(\hat{m}) = \begin{cases} \hat{\sigma}\hat{m} \sqcup V(\hat{\sigma}, \ell, y, z, p) & \text{if } \top_{\mathrm{Prim}} \prec_2 \hat{\sigma}y \ \wedge\ |Part(\hat{\sigma}z)| > 1 \\ & \wedge\ \hat{m} = (\hat{a}, p)\ \wedge\ \hat{a} \prec_1 \hat{\sigma}x \\ \mathscr{T}_{\ell \to_s \ell'}(\hat{\sigma})(\hat{m}) & \text{otherwise} \end{cases}$$

where the abstract value being written is

$$V(\hat{\sigma}, \ell, y, z, p) = \bigsqcup \big\{ \hat{z} \in Part(\hat{\sigma}z) \ \big|\ \exists \hat{y} \in R(\ell, y, z \mapsto \hat{z}) : p \prec_2 \hat{y} \big\}$$

This modified transfer function captures some of the key ideas of our approach, so we carefully explain each part of the definition. The first case of the transfer function definition shows when and how value refinement is used, and the second case simply falls back to the ordinary transfer function. Value refinement is applied when the analysis has an imprecise value for y (i.e., $\top_{\mathrm{Prim}} \prec_2 \hat{\sigma}y$) and an imprecise value for z that is partitioned nontrivially (i.e., $|Part(\hat{\sigma}z)| > 1$), provided that the desired abstract memory address $\hat{m}$ denotes an object property (i.e., $\hat{m} = (\hat{a}, p)$ for some abstract object address $\hat{a}$ and property name p). The abstract value $V(\hat{\sigma}, \ell, y, z, p)$ being written is then computed by issuing a refinement query $R(\ell, y, z \mapsto \hat{z})$ for each partition $\hat{z}$ of $\hat{\sigma}z$ (i.e., $\hat{z} \in \mathit{Part}(\hat{\sigma}z)$). Each of the resulting abstract values $\hat{y}$ describes a possible value of y under the constraint that z has value $\hat{z}$. In this way, rather than writing the imprecise abstract value $\hat{\sigma}z$ to all properties of $\hat{a}$, the analysis writes each of the more precise abstract values $\hat{z}$ to the corresponding refined property name p that matches $\hat{y}$ (i.e., $p \prec_2 \hat{y}$).

Example Recall that in the example from Section 7.3.2, at the dynamic property write `y[p] = t` the base analysis has imprecise abstract values for the property name `p` and for the value `t` being written. More specifically, *Part* partitions the latter into three more precise abstract values as shown above. This means that the condition is satisfied for using value refinement, so the modified transfer function then issues three refinement queries. Using the value refiner that we present in Section 7.5 yields the following results:

$$R(\ell_2, \texttt{p}, \texttt{t} \mapsto (\{\hat{a}_{\texttt{xa}}\}, \bot_{\text{Prim}})) = \{(\emptyset, \texttt{"a"})\}$$
$$R(\ell_2, \texttt{p}, \texttt{t} \mapsto (\{\hat{a}_{\texttt{xb}}\}, \bot_{\text{Prim}})) = \{(\emptyset, \texttt{"b"})\}$$
$$R(\ell_2, \texttt{p}, \texttt{t} \mapsto (\emptyset, \top_{\text{Prim}})) = \{(\emptyset, \texttt{"c"})\}$$

The transfer function then writes each refined abstract value only to the relevant property of $\hat{a}_{\texttt{y}}$, instead of mixing them all together like the ordinary transfer function. For example, the resulting state maps $(\hat{a}_{\texttt{y}}, \texttt{"a"})$ to $(\{\hat{a}_{\texttt{xa}}\}, \bot_{\text{Prim}})$. The base analysis then proceeds with this more precise abstract state.

Notice that the base analysis has only one abstract value per abstract memory address and program location, whereas the value refiner returns a *set* of abstract values at each refinement query. In the example described above, each of the refinement query results contains only one abstract value, but when applying our technique to the examples from Section 12.2, we benefit from the possibility that R can return multiple abstract values: Some methods of the `lodash` object are accessible via multiple names, for example `lodash.entries` and `lodash.toPairs` are aliases. In this case, querying the value refiner for the possible property names given that the value being written is that specific function, the result can be expressed as the set of the two strings `"entries"` and `"toPairs"` instead of the less precise single abstract value representing all possible strings.

7.4.3 Using the Base Analysis During Refinements

The value refiner can leverage the base analysis state to allow for more efficient implementation. For the dataflow analysis defined in Section 7.3.2, the refiner can read partially-computed abstract states; in our JavaScript implementation (see Section 7.6.1), the refiner also uses the partially-computed call graph. We argue that this is sound even though the base analysis has not yet reached a fixpoint when the refiner is invoked. This extended kind of a value refiner is denoted R_X where $X : Loc \rightarrow \widehat{State}$ is a lattice element of the base analysis.

For the example from Section 7.3.2, R_X can obtain the value of the variable `x` at ℓ_1 simply by looking up $X(\ell_1)(\mathtt{x})$ from the base analysis, without needing to traverse all the way back to where the value of `x` was created. Similarly, for analysis of JavaScript, using the call graph available from the base analysis allows the value refiner to narrow its exploration when stepping from function entry points to call sites.

symbolic variables	$\hat{x}, \hat{y}, \hat{z}, \text{RES}$	$\in \widehat{Var}$
symbolic expressions	$\hat{e} \in \widehat{Expr}$	$::= \hat{x} \mid \hat{v} \mid \hat{e}_1 \oplus \hat{e}_2$
symbolic stores	$\varphi \in \widehat{Store}$	$::= \hat{h} \wedge \pi \mid \varphi_1 \vee \varphi_2$
heap constraints	$\hat{h}$	$::= \text{true} \mid \mathsf{unalloc}(\hat{x}) \mid x \mapsto \hat{x} \mid \hat{x}_1[\hat{x}_2] \mapsto \hat{x}_3 \mid \hat{h}_1 * \hat{h}_2$
pure constraints	π	$::= \text{true} \mid \hat{e} \mid \pi_1 \wedge \pi_2$
valuations	$\eta \in Valua$	$: \widehat{Var} \rightarrow Val$

(a) Abstractions for value refinement. Recall from Section 7.3 that $\oplus$ ranges over binary operators, x over program variables, and $\hat{v}$ over abstract values.

$$\begin{array}{rcl}
\mathsf{eval} & : & (\widehat{Expr} \times Valua) \rightarrow \mathscr{P}(Val) \\
\mathsf{eval}(\hat{x}, \eta) & = & \{\eta \hat{x}\} \\
\mathsf{eval}(\hat{v}, \eta) & = & \gamma_{\mathsf{val}}(\hat{v}) \\
\mathsf{eval}(\hat{e}_1 \oplus \hat{e}_2, \eta) & = & \left\{ v_1 \oplus v_2 \;\middle|\; \begin{array}{l} v_1 \in \mathsf{eval}(\hat{e}_1, \eta) \\ \wedge\ v_2 \in \mathsf{eval}(\hat{e}_2, \eta) \end{array} \right\}
\end{array}$$

(b) Abstract expression evaluation function eval. The notation $\gamma_{\mathsf{val}}(\hat{v})$ refers to the concretization of $\hat{v}$.

$$\begin{array}{rcl}
\gamma(\hat{h} \wedge \pi) & = & \gamma(\hat{h}) \cap \gamma(\pi) \\
\gamma(\varphi_1 \vee \varphi_2) & = & \gamma(\varphi_1) \cup \gamma(\varphi_2) \\
\gamma(\text{true}) & = & State \times Valua \\
\gamma(\hat{e}) & = & \{(\sigma, \eta) \mid \text{true} \in \mathsf{eval}(\hat{e}, \eta)\} \\
\gamma(\pi_1 \wedge \pi_2) & = & \gamma(\pi_1) \cap \gamma(\pi_2) \\
\gamma(\mathsf{unalloc}(\hat{x})) & = & \{(\sigma, \eta) \mid \forall v : \sigma(\eta(\hat{x}), v) = \mathtt{undef}\} \\
\gamma(x \mapsto \hat{x}) & = & \{(\sigma, \eta) \mid \sigma x = \eta(\hat{x})\} \\
\gamma(\hat{x}_1[\hat{x}_2] \mapsto \hat{x}_3) & = & \{(\sigma, \eta) \mid \sigma(\eta(\hat{x}_1), \eta(\hat{x}_2)) = \eta(\hat{x}_3)\} \\
\gamma(\hat{h}_1 * \hat{h}_2) & = & \left\{ (\sigma_1 \uplus \sigma_2, \eta) \;\middle|\; \begin{array}{l} (\sigma_1, \eta) \in \gamma(\hat{h}_1)\ \wedge\ (\sigma_2, \eta) \in \gamma(\hat{h}_2) \\ \wedge\ \mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2) = \emptyset \end{array} \right\}
\end{array}$$

(c) Concretizations γ for symbolic stores φ, heap constraints $\hat{h}$, and pure constraints π. We denote by $\uplus$ the union of two partial functions with disjoint domains.

Figure 7.6: Syntax and concretizations of abstractions used for value refinement.

For this mechanism to retain analysis soundness, we slightly modify the base analysis. Recall from Section 7.3.2 that the base analysis relies on a worklist of program locations. Now, whenever a refinement query is triggered by the base analysis at some program location ℓ and the value refiner reads from the abstract state $X(\ell')$, we extend the *dep* map to record that ℓ depends on ℓ'. In this way, if the abstract state at ℓ' changes later during the base analysis, the result of the invocation of the value refiner is invalidated and eventually recomputed.[6]

As such, the soundness criterion from Section 7.4.1 for the value refiner needs to be adjusted by weakening the soundness requirement so that the value refiner needs only overapproximate those concrete program behaviors that are abstracted by the current base analysis state. We say that a trace $\ell_0 \rightarrow_{s_1} \ell_1 \rightarrow_{s_2} \cdots \rightarrow_{s_n} \ell_n$ is *abstracted* by a base analysis state $X \in L$ if, for all $k \leq n$, $([\![s_k]\!] \circ [\![s_{k-1}]\!] \circ \cdots \circ [\![s_1]\!])(\varepsilon)$ is in the concretization of $X(\ell_k)$. Then, a refiner R_X is *sound* if σy is in the concretization of an abstract value in $R_X(\ell, y, \phi)$ for every concrete state $\sigma \in [\![\ell]\!]_{\langle \ell_0, T \rangle}$ that satisfies ϕ and is the final state of a trace that is abstracted by X and ends at ℓ.

Soundness Since we require that the refiner R_X is sound with respect to those traces abstracted by the base analysis state, $\mathscr{T}^{\mathsf{VR}}_{\ell \rightarrow_s \ell'}$ is sound when that base analysis state abstracts the full concrete semantics, which is guaranteed at a fixpoint. If the base analysis state does not yet abstract the full concrete collecting semantics, then R_X's refinements are only sound so long as the information they read from the base analysis state is unchanged. However, by adding additional dependency edges wherever such a read occurs, we ensure that any refinement query that used stale information will be invalidated and recomputed. We refer to Section 7.10 for a more detailed discussion.

7.5 Backwards Abstract Interpretation for Value Refinement

In this section, we define a sound value refiner $R^{\leftarrow}$ based on goal-directed backwards abstract interpretation. The value refiner $R^{\leftarrow}$ works by exploring backwards from the abstract state where it was triggered, overapproximating the set of states from which that state is reachable using an abstract domain based on separation logic constraints. We also detail the construction of a refiner $R^{\leftarrow}_X$ that extends $R^{\leftarrow}$ to access base analysis abstract state during value refinement.

This value refiner is *goal-directed* in the sense that it only traverses the subset of the control-flow graph relevant to a given refinement query and only computes transfer functions that directly affect its constraints. As such, it is quite precise with respect to a fixed property of interest without incurring the cost of applying that same precision to the full program.

[6] To improve performance, our implementation actually tracks these extra dependencies in a more fine-grained manner, as described in Section 7.6.3.

$$\textsc{Consequence}\quad \frac{\varphi_2' \Rightarrow \varphi_2 \qquad \langle\varphi_2'\rangle\ s\ \langle\varphi_1'\rangle \qquad \varphi_1 \Rightarrow \varphi_1'}{\langle\varphi_2\rangle\ s\ \langle\varphi_1\rangle}$$

$$\textsc{Frame}\quad \frac{\langle \hat{h}_1' \wedge \pi'\rangle\ s\ \langle \hat{h}_1 \wedge \pi\rangle \qquad \mathsf{mod}(s)\cap\mathsf{fv}(\hat{h}_2)=\emptyset}{\langle \hat{h}_1' * \hat{h}_2 \wedge \pi'\rangle\ s\ \langle \hat{h}_1 * \hat{h}_2 \wedge \pi\rangle}$$

$$\textsc{Disjunction}\quad \frac{\langle\varphi_l'\rangle\ s\ \langle\varphi_l\rangle \qquad \langle\varphi_r'\rangle\ s\ \langle\varphi_r\rangle}{\langle\varphi_l' \vee \varphi_r'\rangle\ s\ \langle\varphi_l \vee \varphi_r\rangle}$$

$$\textsc{BinOp}\quad \frac{\hat{h} \ = \ y \mapsto \hat{y} * z \mapsto \hat{z}}{\langle \hat{h} \wedge \pi \wedge \hat{x} \ = \ \hat{y} \oplus \hat{z}\rangle\ x = y \oplus z\ \langle \hat{h} * x \mapsto \hat{x} \wedge \pi\rangle}$$

$$\textsc{NewObj}\quad \frac{}{\left\langle \mathsf{unalloc}(\hat{x}) \wedge \pi \wedge \left(\bigwedge_i \hat{z}_i = \mathtt{undef}\right)\right\rangle\ x = \{\}\ \left\langle x \mapsto \hat{x} * \left(\mathop{\Large *}_i \hat{x}[\_] \mapsto \hat{z}_i\right) \wedge \pi\right\rangle}$$

$$\textsc{Alias}\quad \frac{}{\langle (y \mapsto \hat{y} \wedge \pi)[\hat{y}/\hat{x}]\rangle\ x = y\ \langle x \mapsto \hat{x} * y \mapsto \hat{y} \wedge \pi\rangle}$$

$$\textsc{ReadProp}\quad \frac{\hat{h} \ = \ y \mapsto \hat{y} * z \mapsto \hat{z} * \hat{y}[\hat{z}] \mapsto \hat{x}'}{\langle (\hat{h} \wedge \pi)[\hat{x}'/\hat{x}]\rangle\ x = y[z]\ \langle \hat{h} * x \mapsto \hat{x} \wedge \pi\rangle}$$

$$\textsc{Assume}\quad \frac{\hat{h} \ = \ x \mapsto \hat{x}}{\langle \hat{h} \wedge \hat{x} \wedge \pi\rangle\ \mathtt{assume}\ x\ \langle \hat{h} \wedge \pi\rangle}$$

$$\textsc{WriteProp}\quad \frac{\hat{h} \ = \ x \mapsto \hat{x} * y \mapsto \hat{y} * z \mapsto \hat{z}}{\langle (\hat{h} \wedge \pi)[\hat{z}/\hat{z}']\rangle\ x[y] = z\ \langle \hat{h} * \hat{x}[\hat{y}] \mapsto \hat{z}' \wedge \pi\rangle}$$

$$\textsc{Constant}\quad \frac{}{\langle \mathsf{true} \wedge \pi \wedge \hat{x} = p\rangle\ x = p\ \langle x \mapsto \hat{x} \wedge \pi\rangle}$$

Figure 7.7: Rules for refutation-sound backwards abstract interpretation. We denote the set of memory locations possibly modified by a statement s by $\mathsf{mod}(s)$, the free variables of a heap constraint $\hat{h}$ by $\mathsf{fv}(\hat{h})$, and the substitution of symbolic variable $\hat{x}$ for $\hat{y}$ in a symbolic store φ by $\varphi[\hat{x}/\hat{y}]$. To simplify the presentation, without loss of generality, we assume the program has been normalized so that any statement involving multiple program variables uses distinct variables.

7.5.1 Abstract Domain

The abstract domain of $R^{\leftarrow}$ is a constraint language over memory states. The syntax and semantics of this constraint language are given in Fig. 7.6. A *symbolic store* φ is a disjunctive normal form expression over heap constraints $\hat{h}$ and pure constraints π. Each clause represents a symbolic store, while the top-level disjunction permits case-splitting.

Heap constraints $\hat{h}$ are defined using an intuitionistic separation logic [69] wherein a single-cell heap constraint (i.e., $x \mapsto \hat{x}$ or $\hat{x}_1[\hat{x}_2] \mapsto \hat{x}_3$) holds for any heap containing that cell, not just for those heaps comprised only of that single cell. This results in a monotonic logic in which heap constraints $\hat{h}$ are preserved under heap extension. That is, if an intuitionistic separation logic assertion $\hat{h}$ holds for some concrete state σ, then $\hat{h}$ must also hold for all extensions σ' of σ. This succinctly supports a goal-directed analysis, in which we want to infer information only about some sub-portion of the heap.

Pure constraints π are either symbolic expressions $\hat{e}$ or conjunctions thereof; the pure constraint $\hat{e}$ holds whenever it could possibly evaluate to `true` according to the abstract expression evaluation function eval. By defining pure constraints over abstract values $\hat{v}$ rather than concrete values v, we will be able to seamlessly integrate information from the abstract state of the base analysis during refinement, as discussed in Section 7.4.3.

We denote by $\varphi \wedge \varphi'$ conjunction and re-normalization to DNF after alpha-renaming free symbolic variables in φ' such that all memory addresses are mapped to the same symbolic variable by both symbolic stores. For example, $(x \mapsto \hat{x} \wedge \hat{x} > 0) \wedge (x \mapsto \hat{y} \wedge \hat{y} < 5)$ reduces to $x \mapsto \hat{x} \wedge (\hat{x} > 0 \wedge \hat{x} < 5)$.

7.5.2 Backwards Abstract Interpretation

We define the analysis in terms of *refutation sound* [16] Hoare triples of the form $\langle \varphi \rangle\ s\ \langle \varphi' \rangle$, which are given in Fig. 7.7.

Refutation soundness is similar to the standard definition of soundness for Hoare logic (i.e., partial correctness), but in the opposite direction: a triple is refutation sound if and only if $((\sigma', \eta) \in \gamma(\varphi') \wedge [\![s]\!](\sigma) = \sigma') \Rightarrow (\sigma, \eta) \in \gamma(\varphi)$ holds for all σ, σ', and η. That is, a triple is refutation sound if any concrete run through s ending in a state satisfying the postcondition φ' must have started in a state satisfying the precondition φ.

These triples are best read from postcondition to precondition, since that is the natural direction in which to understand refutation soundness and the direction in which the analysis actually applies them.

The first three rules in Fig. 7.7 are integral structural components of the system that are not specific to any particular statement form. The CONSEQUENCE rule allows the analysis to strengthen preconditions and weaken postconditions, making explicit our notion of refutation soundness and allowing other rules to materialize heap cells as needed; FRAME enables local heap reasoning; and DISJUNCTION splits reasoning over each disjunct of a symbolic store.

The remaining rules abstract the concrete semantics of their respective statement forms. READPROP, WRITEPROP, and ALIAS transfer any postcondition constraints on their left-hand-side to precondition constraints on their right-hand-side; NEWOBJ constrains any properties of the allocated object to be `undef` and asserts that the object is now unallocated, which ensures (by separation) that no such properties can be framed out by FRAME; BINOP and CONSTANT use pure equality constraints to precisely model the concrete semantics; and ASSUME directly encodes the assumption into a pure constraint.

Our value refiner $R^{\leftarrow} : \mathit{Loc} \times \mathit{Var} \times \mathit{Constraint} \rightarrow \mathscr{P}(\widehat{\mathit{Val}})$ is based on backwards abstract interpretation using the judgment $\langle \varphi \rangle\ s\ \langle \varphi' \rangle$. We introduce a distinguished symbolic variable RES to represent the value that is being refined as we move backwards through the program. Note that $R^{\leftarrow}$ is implicitly parameterized by a specific program $\langle \ell_0, T \rangle$, but does *not* have access to abstract states from the base analysis as discussed in Section 7.4.3; integration with the base analysis will be detailed in Section 7.5.3.

Given a refinement query $R^{\leftarrow}(\ell, x, y \mapsto \hat{v})$, we first encode the inputs as a symbolic store $x \mapsto \text{RES} * y \mapsto \hat{y} \wedge \hat{y} = \hat{v}$. Then, we algorithmically apply the Hoare triples from Fig. 7.7 backwards from ℓ in T to compute a symbolic store for each backwards-reachable location, using a standard worklist algorithm to ensure correctness and minimize redundant work and applying the widening technique of Blackshear et al. [16] to compute fixpoints over loops.

Due to refutation soundness, it is sound for the backwards abstract interpreter to stop at any point. Since each successive application of a rule from Fig. 7.7 computes an abstract precondition that a concrete execution must satisfy to reach the given abstract postcondition, the constraints on RES grow more precise the more of the program is analyzed but are overapproximate every step of the way. As such, the *stopping criterion* of $R^{\leftarrow}$ can be tuned, offering a tradeoff between refinement precision and performance. In our implementation for JavaScript, we stop the backwards traversal along a path if sufficient precision has been reached for the refinement variable RES, meaning that its abstract value is either a singleton set of object addresses or a non-$\top_{\text{Prim}}$ abstract primitive.

This analysis continues either until we reach a least fixpoint in the symbolic store domain (partially ordered under implication) or the stopping criterion is fulfilled for all symbolic stores in the worklist. At that point, we compute an upper bound on the value of RES in all remaining symbolic stores and return the corresponding set of abstract values.

Example Recall that the base analysis issues three refinement queries for the example from Section 7.4.2, the first one being $R^{\leftarrow}(\ell_2, \texttt{p}, \texttt{t} \mapsto (\{\hat{a}_{\texttt{xa}}\}, \bot_{\text{Prim}}))$. This query is encoded as the initial symbolic store $\texttt{p} \mapsto \text{RES} * \texttt{t} \mapsto \hat{t} \wedge \hat{t} = (\{\hat{a}_{\texttt{xa}}\}, \bot_{\text{Prim}})$ at the program location ℓ_2. From there, $R^{\leftarrow}$ uses the CONSEQUENCE and READPROP rules from Fig. 7.7 to construct the triple

$$\begin{gathered} \left\langle\ \texttt{p} \mapsto \text{RES} * \texttt{x} \mapsto \hat{x} * \hat{x}[\text{RES}] \mapsto \hat{t} \ \wedge\ \hat{t} = (\{\hat{a}_{\texttt{xa}}\}, \bot_{\text{Prim}})\ \right\rangle \\ \texttt{t = x[p]} \\ \left\langle\ \texttt{p} \mapsto \text{RES} * \texttt{t} \mapsto \hat{t} \ \wedge\ \hat{t} = (\{\hat{a}_{\texttt{xa}}\}, \bot_{\text{Prim}})\ \right\rangle \end{gathered}$$

which precisely models the dynamic property read `t= x[p]` and yields a precondition symbolic store expressing a refinement of the value of `p` when `x[p]` has value $(\{\hat{a}_{\texttt{xa}}\}, \bot_{\text{Prim}})$. We continue this example in the following section to show how the value refiner reaches the final result $\{(\emptyset, \texttt{"a"})\}$.

Soundness By refutation soundness, applying the Hoare rules from Fig. 7.7 backwards soundly overapproximates the states from which the refinement location is reachable. By exhaustively tracking the value of RES on all backward abstract paths from the refinement location, $R^{\leftarrow}$ therefore computes an overapproximation of the variable being refined with respect to the concrete collecting semantics. We refer to Section 7.11 for a proof sketch.

7.5.3 Integration of Base Analysis State

We now extend $R^{\leftarrow}$ to leverage abstract state from the base analysis as described in Section 7.4.3, thereby constructing a value refiner $R^{\leftarrow}_{X}$ that is parameterized by a base analysis abstract state X. In particular, we describe a procedure by which a symbolic store φ and base analysis state X can be combined to compute a refinement that the symbolic store φ is not able to on its own. Essentially, when $R^{\leftarrow}_{X}$ has a symbolic store φ refining a property's name under a constraint on that property's value, it accesses the base analysis state to determine possible property names satisfying those constraints and then returns that set. We refer to this procedure as *property name inference*.

In more detail, this mechanism works as follows. If, during the backwards abstract interpretation as described for $R^{\leftarrow}$ in the previous section, the current symbolic store φ matches

$$x \mapsto \hat{x} * \hat{x}[\text{RES}] \mapsto \hat{y} * \hat{h} \ \wedge \ \hat{y} = \hat{v} \wedge \pi$$

for some x, $\hat{x}$, $\hat{y}$, $\hat{v} \neq \texttt{undef}$, $\hat{h}$, and π, then property name inference is applied. Intuitively, this condition means that the abstract value of the RES property of x is $\hat{v}$, which allows $R^{\leftarrow}_{X}$ to determine the desired set of property names by reading the base analysis state at that location. We define a function infer-prop-names to compute that refinement:

$$\mathsf{infer\text{-}prop\text{-}names}(\varphi, \hat{\sigma}) = \left\{ \ p \ \middle| \ \begin{array}{c} \exists \hat{a} : \hat{a} \prec_1 \hat{\sigma} x \ \wedge \ (\hat{a}, p) \in \mathrm{dom}(\hat{\sigma}) \\ \wedge \ \hat{\sigma}(\hat{a}, p) \sqcap \hat{v} \neq (\emptyset, \bot_{Prim}) \end{array} \right\}$$

Intuitively, infer-prop-names checks each property $(\hat{a}, p)$ on the object x, returning the names p of those properties whose abstract value intersects with $\hat{v}$.

The property name inference mechanism thus refines the abstract *names* of object properties. Our implementation uses the same idea to also refine abstract *values* of properties, which we return to in Section 7.6.2.

Example Continuing the example from Section 7.5.2, φ is of the form specified above, so the analysis applies property name inference to compute a refinement for `p`. Computing the refinement infer-prop-names(φ,$X(\ell_1)$) gives the names of those properties in $X(\ell_1)$ that satisfy φ, meaning that the property value intersects with $(\{\hat{a}_{\mathsf{xa}}\}, \bot_{\mathsf{Prim}})$. In this case, `"a"` is the only such property name according to the value of $X(\ell_1)$ given in Section 7.3.2, so the refiner returns $\{(\emptyset, \texttt{"a"})\}$. In this simple example, a single step backwards suffices before the stopping criterion is fulfilled, due to the integration of the base analysis state, but multiple steps are often needed in practice.

7.6 Instantiation for JavaScript

Our implementation, TAJS$_{\text{VR}}$,[7] generalizes to JavaScript the ideas presented in the previous sections for the simple dynamic language. As base dataflow analysis TAJS$_{\text{VR}}$ uses the existing tool TAJS, extended as explained in Section 7.4.3. The other main component of the implementation is the value refiner, built from scratch and based on the design given in Section 7.5. The two components are implemented separately – the base analysis in Java (approximately 2500 lines of code on top of TAJS) and the refiner in Scala (approximately 2400 lines of code) – and communicate only through a minimal interface that allows the base analysis to issue refinement queries and the refiner to read partially-computed base analysis state, request control-flow information to traverse the program, or perform property name inference as described in Section 7.5.3. The implementation and experimental data are available at `https://www.brics.dk/TAJS/VR`.

7.6.1 A Value Refiner for JavaScript

Many JavaScript language features that are not directly in the minimal dynamic language are straightforward to handle. However, `for-in` loops, interprocedural control flow, and prototype-based inheritance are nontrivial and require some additional machinery in the backwards analysis.

`for-in` loops In order to handle `for-in` loops efficiently, the refiner analyzes the loop body under contexts corresponding to the properties of the loop object in the base analysis state.

That is, upon reaching the exit of a `for-in` loop, the value refiner queries the base analysis state for a set of property names on the loop object, generating one context for each of them and one additional context as a catch-all for all other property names. Then, it analyzes the loop body once per context before joining the results and continuing backwards from the loop entry.

Interprocedural control flow In order to soundly navigate interprocedural control flow in the value refinement analysis, we rely on the partially-computed call graph from the base analysis while maintaining a stack of return targets where possible. That is, when reaching a call site, the analysis pushes that location (along with any locally-scoped constraints) onto a stack before jumping to the exit of all possible callees in the base analysis call graph. Then, upon reaching a function entry point, it pops a stack element and jumps to the corresponding call site or, if the stack is empty, jumps to all callers of the current function in the base analysis call graph. When using an unbounded stack, it analyzes function calls fully context-sensitively and therefore relies on a timeout to ensure termination, but k-limiting the stack height would ensure termination (without a timeout) while analyzing function calls with k-callstring context sensitivity.

[7]**TAJS** with demand-driven **V**alue **R**efinement

Prototype-based inheritance Handling prototype-based inheritance is more complicated since the semantics of a dynamic property access depend not only on the values of the object and property name but also on the prototype relations and properties of other objects in the program.

Our implementation reasons about prototype-based inheritance by introducing "prototype constraints" at property reads to keep track of prototype relationships between relevant symbolic variables. These constraints are manipulated as the analysis evaluates other property writes and modifications to the prototype graph; for example, when encountering a property write under a prototype constraint, the analysis splits into a disjunction on whether or not the write is to the memory location whose read produced the prototype constraint. This allows the analysis to reason about prototype semantics, even in programs that dynamically modify the prototype graph.

7.6.2 Functions with Free Variables

As mentioned in Section 12.2, the dynamic property read/write pair ② in the example in Fig. 7.1a differs from ① and ③, because the values flow from the property read to the associated write via a free variable that is declared in an enclosing function. It is critical that the analysis does not mix together the different functions of the `source` object. For example, in clients of Lodash, the function value `lodash([1, 2]).map` is the one created in line 1517 where `methodName` is `"map"`. If the program contains a call to that function, then at the call `func.apply(...)` in line 1519, the analysis must have enough precision to know that `func` is the same function as `source["map"]`.

We achieve that degree of precision by adjusting the base analysis as follows. At the dynamic property write in line 1517, the analysis detects that in the current abstract state, the property name (i.e. `methodName`) is imprecise and that the value being written denotes a function that contains a free variable with an imprecise value as noted in Section 12.2. The analysis then annotates the abstract value being written with the memory address of `methodName` and the current program location ℓ_{1517} for later use. Every property of `object.prototype` then has this single annotated abstract value.

When the analysis later encounters a property read operation that yields such an annotated value, the value is modified to reflect the property name, which can now be resolved. For example, at an expression `lodash([1, 2]).map`, the resulting abstract value describes a function that has been created at a point where the value of `methodName` was `"map"`.

If that function is called, the analysis reaches line 1519 and then issues the refinement query $R_X^{\leftarrow}(\ell_{1517}, \texttt{func}, \texttt{methodName} \mapsto \texttt{"map"})$ to learn the possible values of `func` at line 1517 under the constraint that `methodName` has the value `"map"`. When the value refiner reaches the dynamic property read in line 1514 during the backwards analysis, it can use the base analysis state to read `source["map"]` (as hinted in Section 7.5.3), which provides the desired precise value for `func`.

Note that with this mechanism, our base analysis does not only trigger refinement at dynamic property writes as described in Section 7.4.2, but also for variable reads of imprecise free variables as described above.

7.6.3 Performance Improvements

Recall from Section 7.4.3 that a location ℓ is added to the worklist when a refinement query at ℓ has accessed the abstract state $X(\ell')$ and that abstract state has changed. Our implementation uses a more fine-grained notion of dependencies by keeping track of the individual dataflow facts instead of entire abstract states, such that ℓ is only added to the worklist when the state change at ℓ' invalidates the dataflow facts that were previously accessed from ℓ'.

Additionally, our implementation caches refinement query results, exploiting the fact that the result of a query $R_X(\ell, y, \phi)$ depends only on the three parameters and the dataflow facts in X that are accessed by the value refiner.

7.7 Evaluation

We evaluate the demand-driven value refinement technique by considering the following research question:

> Can TAJS_{VR} analyze programs that other state-of-the-art tools are unable to analyze soundly and with high precision?

To provide insights into why the mechanism is effective when analyzing real-world programs, we also investigate how many value refinement queries are issued, how often the value refiner is able to produce more precise results than the base analysis, and how much analysis time is typically spent on value refinement.

7.7.1 Comparison with State-of-the-Art Analyzers

We compare TAJS_{VR} with two existing state-of-the-art analysis tools: TAJS [10, 70] and CompAbs [84, 85]. TAJS is the base dataflow analysis upon which TAJS_{VR} is built; it is designed for JavaScript type analysis but performs no value refinement. CompAbs – described in further detail in Sections 12.2 and 7.8 – is a tool built on top of SAFE [91] that attempts to syntactically identify problematic dynamic property access patterns and applies trace partitioning at those locations.

We evaluate each tool on three sets of benchmarks: a series of micro-benchmarks designed as minimal representative examples of dynamic property manipulation patterns, a collection of evaluation suites drawn from other JavaScript static analysis research papers, and the unit test suites of two popular JavaScript libraries that are unanalyzable by the existing static analysis tools. All experiments have been performed on an Ubuntu machine with 2.6 GHz Intel Xeon E5-2697A CPU running a JVM with 10 GB RAM. Collectively, the results indicate that the relational information provided by value refinement is critical for the analysis of challenging JavaScript programs.

Table 7.1: Micro-benchmarks that check how state-of-the-art analyses handle various dynamic property access patterns. A ✗ indicates that the analysis mixes together the properties of the object being manipulated, while a ✓ indicates that it is sufficiently precise to keep them distinct. The CF, CG, AF, and AG benchmarks are drawn directly from [84], while M1, M2, and M3 are isolated and distilled from the read/write patterns presented in Section 12.2.

Benchmark		TAJS	CompAbs	TAJS$_{\text{VR}}$
CF	(`for-in` loop over statically known set of properties)	✓	✓	✓
CG	(`while` loop over statically known set of properties)	✓	✓	✓
AF	(`for-in` loop over statically unknown properties)	✗	✓	✓
AG	(`while` loop over statically unknown properties)	✗	✓	✓
M1	(indirect field copy, distilled from ① in Fig. 7.1)	✗	✗	✓
M2	(field copy through closure, distilled from ② in Fig. 7.1)	✗	✗	✓
M3	(interprocedural field copy, distilled from ③ in Fig. 7.1)	✗	✗	✓

Micro-Benchmarks Following the approach of Ko et al. [84], we first evaluate TAJS$_{\text{VR}}$ on a series of small benchmarks containing dynamic property access patterns known to be difficult for static analysis. Source code for these benchmarks can be found in Ko et al. [84] (CF, CG, AF, and AG) or at `https://www.brics.dk/TAJS/VR` (M1, M2, and M3).

The results, shown in Table 8.1, are as expected: TAJS only handles the benchmarks CF and CG where the property names are known statically, and CompAbs fails to successfully analyze any of M1, M2 or M3: M1 and M3 because the relevant read/write pair is not detected by the syntactic patterns, and M2 because the trace partitioning mechanism does not distinguish closures on free variables within the partitions. TAJS$_{\text{VR}}$ handles all seven programs precisely by the use of demand-driven value refinement.

Library Benchmarks Prior work has found that the analysis of highly dynamic, metaprogramming-heavy libraries is a major hurdle for the analysis of realistic JavaScript programs in the wild [10, 115, 133]. As such, we evaluate TAJS$_{\text{VR}}$ against TAJS and CompAbs on a corpus of challenging real-world library benchmarks, drawn from the benchmark suites of past JavaScript analysis research works as well as from library unit test suites. We selected these benchmarks to evaluate our approach on programs that other researchers find important and to demonstrate that our approach enables the analysis of programs beyond the reach of existing analyzers.

From past works, we analyze the jQuery tests found in Andreasen and Møller [10], the Prototype.js and Scriptaculous tests from Wei et al. [144], and the test suite (excluding 7 benchmarks that require additional modelling of Firefox add-ons) used by Kashyap et al. [80] and Dewey et al. [42] (referred to collectively as “JSAI tests” henceforth).

In addition, we analyze the unit test suites for two heavily used functional utility libraries: Underscore (v1.8.3, 1548 LoC) Lodash3 (v3.0.0, 10785 LoC), and Lodash4 (v4.17.10, 17105 LoC). The unit tests (664 in total) provide comprehensive coverage

Table 7.2: Analysis results for real-world benchmarks drawn from previous evaluations of JavaScript analysis tools [10, 80, 144] and additional library unit test suites. A test is a "Success" if the analysis terminates with dataflow to the program exit within a 5 minute timeout, and times are averaged across all successfully analyzed tests.

		TAJS		**CompAbs**		**TAJS$_{\text{VR}}$**	
Benchmark		Success	Time	Success	Time	Success	Time
JQuery	(71 tests)	7%	14.4s	0%	-	7%	17.2s
JSAI tests	(29 tests)	86%	12.3s	34%	32.4s	86%	14.3s
Prototype	(6 tests)	0%	-	33%	23.1s	83%	97.7s
Scriptaculous	(1 test)	0%	-	100%	62.0s	100%	236.9s
Underscore	(182 tests)	0%	-	0%	-	95%	2.9s
Lodash3	(176 tests)	0%	-	0%	-	98%	5.5s
Lodash4	(306 tests)	0%	-	0%	-	87%	24.7s

and illustrate realistic use-cases of the two libraries. We select Lodash and Underscore because they are the two most-depended-upon packages in NPM[8] that do not require platform-specific modelling for Node.js and are unanalyzable by TAJS without the use of value refinement. In total, more than 100,000 NPM packages depend on one or both libraries (excluding transitive dependencies), so they represent a significant hurdle for analysis of Node.js modules and applications. Both Lodash3 and Lodash4 are included since their codebases are substantially different and they present distinct challenges for static analysis.

A summary of the results of each tool on these library benchmark is given in Table 7.2. We say that a program is *analyzable* when analysis terminates within 5 minutes, and with dataflow to the program exit. In our experiments, we find that increasing this time budget does not allow the tools to successfully analyze many more tests, due to the all-or-nothing nature of dynamic language analysis: analyzers are generally either precise enough to analyze a program quickly, or they lose precision at some key location, leading to a proliferation of spurious dataflow that renders the analysis results useless and cannot be recovered from regardless of time budget. This phenomenon has also been observed by Jensen et al. [70], Park and Ryu [115], and Ko et al. [84]. For some of the tests, the CompAbs tool does terminate quickly but has no dataflow to the program exit, which is unsound for these programs. In comparison, TAJS$_{\text{VR}}$ passes extensive soundness testing as explained below.

Demand-driven value refinement enables TAJS$_{\text{VR}}$ to efficiently analyze many benchmarks that TAJS cannot handle. The Prototype and Scriptaculous libraries are unanalyzable by TAJS, but the relational information provided by value refinement allows TAJS$_{\text{VR}}$ to successfully analyze the Scriptaculous test and five of six Prototype tests. For the tests that CompAbs can analyze, it is faster than TAJS$_{\text{VR}}$. Less than 5% of the analysis time for TAJS$_{\text{VR}}$ is spent performing value refinement for those benchmarks, so refinement is not the primary reason for this difference; we believe

[8]At time of publication, Lodash ranks #1 and Underscore #14, per `https://npmjs.com/browse/depended`.

that the reason is rather that CompAbs uses a more precise model of the DOM, which is used heavily in both libraries.

TAJS$_{\text{VR}}$ is furthermore able to analyze 92% (611/664) of the Underscore, Lodash3, and Lodash4 unit tests, none of which are analyzable by either TAJS or CompAbs, in 13 seconds on average. This result is explained in part by the analysis behavior on the micro-benchmarks above. Since the M1, M2, and M3 micro-benchmarks are extracted from Lodash library bootstrapping code and neither TAJS nor CompAbs can reason precisely about them, it follows that neither tool can precisely analyze the library test cases. The result also indicates that the relational information provided by value refinement – and, correspondingly, TAJS$_{\text{VR}}$'s ability to analyze the M1, M2, and M3 micro-benchmarks – is integral to the precise analysis of libraries like Underscore and Lodash.

Manual triage shows that the library unit tests that TAJS$_{\text{VR}}$ fails to analyze are mostly due to challenges orthogonal to dynamic property access operations. Some of the tests involve complex string manipulations, some are caused by insufficient context sensitivity in the base dataflow analysis, and most of the remaining ones could be handled by improving TAJS' reasoning at type tests in branches. Also, our value refiner fails to provide sufficiently precise answers for approximately 0.02% of queries, as discussed in Section 11.6.2.

For those benchmarks that TAJS can handle without demand-driven value refinement, TAJS$_{\text{VR}}$ provides similar results to TAJS, both in terms of precision and performance. Because the static determinacy technique by Andreasen and Møller [10] enables TAJS to analyze many of jQuery's dynamic property writes precisely, the jQuery test cases where TAJS$_{\text{VR}}$ fails are unanalyzable for reasons unrelated to dynamic property accesses and so value refinement is rarely triggered. The results for the JSAI tests are analogous: since TAJS can reason precisely about them without value refinement for the most part, TAJS$_{\text{VR}}$ yields similar results to TAJS and never triggers refinement. More data about the refinement queries issued for these benchmarks are presented in Section 11.6.2. Overall, the results indicate that extending an analysis with demand-driven value refinement does not add significant cost in situations where the base analysis is already sufficiently precise.

Precision We measure TAJS$_{\text{VR}}$ analysis precision with respect to type analysis and call graph construction, following the methodology of prior JavaScript analysis works [10, 115] that have established these metrics as useful proxies for precision. In these measurements, we treat locations context-sensitively, counting the same location once per context under which it is reachable. At each variable or property read in a program successfully analyzed by TAJS$_{\text{VR}}$, we count the number of possible types for the resulting value; in 99.48% of cases, the value has a single unique type, and the average number of types per read is 1.009 (of course, the actual value must be at least 1 at every read). Similarly, we measure the number of callees per callsite, finding that 99.95% of calls have a unique callee.

For analysis of a library to be useful it is also important that the library object itself is analyzed precisely, such that properties of the library yield precise values when referenced in client programs. To verify that is the case, we check all methods of library objects (i.e., properties that contain functions) in programs successfully analyzed by TAJS$_{\mathrm{VR}}$. We find that 99.44% of such methods contain a unique function, indicating that TAJS$_{\mathrm{VR}}$ successfully avoids mixing together the library methods.

These numbers clearly demonstrate that in the situations where the critical precision losses are avoided and the analysis terminates successfully, the analysis results are very precise. This degree of precision may enable analysis clients such as program optimizers and verification tools; however, developing such client tools is out of scope of this work.

Soundness Testing To increase confidence that TAJS$_{\mathrm{VR}}$ is sound, we have applied the empirical soundness testing technique of Andreasen et al. [11]. The technique checks whether the analysis result overapproximates all values observed in every step of a concrete execution. For example, if the program at some point in the execution writes the number 42 to a property of an object, then the analysis must at that point have an abstract value that overapproximates that concrete value. Since most of the benchmarks do not require user interaction, a single concrete execution for each suffices to get good coverage. In total, more than 7.8 million pairs of concrete and abstract values are tested. Only 117 of them fail, all for the same reason: one Lodash4 test uses `Array.from` in combination with ES6 iterators, which is not fully modeled in the latest version of TAJS.

Scalability Compared to Trace Partitioning When analyzing the initialization code of Lodash4, TAJS$_{\mathrm{VR}}$ only issues value refinement queries at the dynamic property writes in ① and ③ in Fig. 7.1. The precise information provided by these queries can also be gained using the trace partitioning technique from CompAbs at the correlated reads of ① and ③. To compare the scalability of value refinement with that of trace partitioning (isolated from the choice of where to issue value refinement queries or apply trace partitioning), we have implemented an extension of TAJS that is hardcoded to perform trace partitioning at exactly those two reads.

TAJS$_{\mathrm{VR}}$ analyzes the initialization code of Lodash4 in 19 seconds while TAJS with hardcoded trace partitioning takes 222 seconds. This result indicates that value refinement scales better than trace partitioning even when partitioning is applied only at the necessary locations.

7.7.2 Understanding the Effectiveness of the Value Refiner

Table 8.3 shows statistics about the value refinement queries that are issued when analyzing the benchmark suites.[9] In summary, these results demonstrate that value refinement queries are being triggered at only a small fraction of all the dynamic property write operations, and that the backwards abstract interpreter described in

[9]More granular experimental results can be found in Section 7.12.

Table 7.3: Summary of value refinement behavior for library tests. “Ref. locs” is the total number of program locations where refinement queries are issued out of the total number of property writes with dynamically-computed property names; “Avg. queries” is the average number of queries issued per test; “Success” is the percentage of queries where the value refiner produces a result more precise than the base analysis state for the requested memory address; “Refiner time” is the percentage of the total analysis time spent by the value refiner; “Avg. query time” is the average time spent by the value refiner on each query; “Avg. locs visited” is the average number of program locations visited in each invocation of the value refiner; “Inter.” is the percentage of the queries where the value refiner visits multiple functions; and “PNI” is the percentage of queries where the value refiner uses property name inference (Section 7.5.3).

	Ref. locs	Avg. queries	Success (%)	Refiner time (%)	Avg. query time (ms)	Avg. locs visited	Inter. (%)	PNI (%)
JQuery (71)	5 / 138	1.13	87.5	0.1	13.57	7.1	2.86	90.00
JSAI tests (29)	0 / 2705	-	-	-	-	-	-	-
Prototype (6)	4 / 69	188.17	100.0	2.5	13.08	39.98	48.10	97.61
Scriptaculous (1)	2 / 92	601.00	100.0	3.4	13.21	36.91	42.26	99.33
Underscore (182)	5 / 32	267.84	99.98	22.4	2.43	5.05	0.10	99.76
Lodash3 (176)	12 / 132	475.28	99.99	47.2	5.46	10.47	40.22	99.90
Lodash4 (306)	7 / 123	1284.04	99.97	52.0	10.01	10.09	25.75	99.67

Sections 7.4 and 7.6.1 is an effective value refiner: it efficiently computes highly precise refinements on the base analysis state in the vast majority of cases, often spending only a few milliseconds and visiting only a small part of the program.

Semantic Triggers for Refinement Value refinement is triggered (meaning that the first case in the definition of the modified transfer function $\mathscr{T}^{\mathsf{VR}}_{\ell \to_s \ell'}$ applies) at a total of only 35 program locations across the 7 benchmark groups. This is a low number compared to the sizes of the benchmarks (which contain a total 3291 property writes with dynamically computed property names), but as we have seen in Section 11.6.1, adequate relational precision at those 35 locations is critical for successful analysis of library clients.

The value refiner is invoked many times for the benchmarks that TAJS cannot analyze without refinement but quite rarely in the benchmarks (JSAI and JQuery) that TAJS can handle alone. As discussed in Section 11.6.1, this is because we trigger refinement semantically only at imprecise dynamic property writes, but TAJS has sufficient precision to avoid imprecise writes without applying refinement in some benchmarks. As for the large number of refinement queries for the other benchmarks, recall that each computation of $\mathscr{T}^{\mathsf{VR}}_{\ell \to_s \ell'}$ issues multiple refinement queries for the same memory locations under different constraints.

Effectiveness of Backwards Abstract Interpretation The table shows that over 99% of refinement queries are successful, in the sense that the value refiner computes a more precise abstract value for the queried memory location than the base analysis alone. Each invocation of the value refiner is limited to 2 seconds, and all unsuccessful queries reported in Table 8.3 are due to this timeout. Even though we invoke the value refiner often, in most cases less than half of the total analysis time is spent performing value refinement, and that fraction of time only exceeds 4% for the Underscore and Lodash experiments. This low performance cost of refinement is due to the fact that most queries are answered in a few milliseconds and only require the backwards analysis to visit quite few program locations: the average for every one of the seven benchmark groups is below 40 locations, even in programs containing thousands of lines of code. This indicates that the "goal-directed" nature of the backwards abstract interpreter – the fact that it reasons only about the portion of the heap relevant to the query at hand and traverses only the subset of the control-flow graph needed to answer that query – is integral to its effectiveness. Still, many queries require interprocedural reasoning, especially for Lodash, Prototype, and Scriptaculous.

We additionally observe that property name inference (the mechanism described in Section 7.5.3, in which we leverage the abstract states of the base analysis when performing value refinement) is used by almost all queries. That mechanism is not only used often by TAJS_{VR}, it is essential to its success; an extra experiment shows that if we disable property name inference, then TAJS_{VR} fails on all of the same tests as TAJS in Table 7.2.

A manual investigation has shown that in situations where the value refiner fails to provide precise results, the relevant code does not require relational information between the property name in the write and the value to be written. A typical example is the code snippet `result[pad + h[hIndex]] = args[argsIndex++]` from Lodash3. Since there is no relation between `hIndex` and `argsIndex`, the value refinement mechanism is unable to precisely resolve the dynamic property write operation and the value being written. Conversely, when the refinement does hinge on relational information, the value refiner is precise and effective in solving the queries.

7.8 Related Work

There is a wealth of research on techniques for static analysis of JavaScript applications, much of which has focused on the development of general analysis frameworks such as TAJS [70], SAFE [91], WALA [68], and JSAI [80].

Notably, much recent work has focused on improving analysis precision with various algorithmic innovations. The static determinacy technique by Andreasen and Møller [10] and the loop-sensitive analysis by Park and Ryu [115] both employ specialized context-sensitivity policies to achieve greater precision, especially in loop bodies and for free variables in closures. The correlation tracking points-to analysis by Sridharan et al. [133] and the composite abstraction by Ko et al. [84, 85] target the dynamic behaviors exhibited by particularly troublesome syntactic patterns and apply extra precision wherever those syntactic patterns are detected.

All of the JavaScript analysis tools mentioned above are whole-program analyzers, while successful analyses for other languages typically achieve scalability by modularity. For dynamic programming languages like JavaScript, it is hard to achieve modularity without involving complex specifications of the program components. Moreover, even a modular analysis inevitably has to reason about the extremely difficult pieces of code that exist in libraries like those studied in this paper.

Other researchers have developed abstractions to better model JavaScript's idiosyncratic heap semantics. Cox et al. [37] have introduced a complex relational abstraction to reason more precisely and efficiently about open objects, and Gardner et al. [51] and Santos et al. [125, 126] have designed a separation logic-based program logic and verification engine to succinctly express and verify heap properties. These approaches are modular but require complex, manually provided specifications, whereas TAJS_{VR} is fully automatic.

In the area of demand-driven and refinement analysis, much of the prior work has targeted pointer analyses for languages like Java [59, 94, 131, 132]. These works each employ some form of abstraction refinement to increase analysis precision as needed, successively tightening their overapproximations of the forward concrete semantics. Similarly, the technique by Gulavani and Rajamani [58] symbolically propagates error conditions backwards to generate hints for choosing between joining or widening in a finite powerset domain. Like standard counterexample-guided abstraction-refinement [13, 30, 66], these techniques alternate between deriving counterexamples and restarting the fixpoint analysis with a refined abstraction. In contrast, our approach interleaves refinement queries within a single fixpoint analysis and refines the abstract values instead of the analysis abstraction.

Several recent techniques use refutation sound backwards analyses for refinement. For example, Blackshear et al. [16] use a backwards abstract interpreter to refute false alarms from a base pointer analysis, and Cousot et al. [36] infer preconditions by backwards symbolic propagation of program assertions. These works are distinct from the well-known backward techniques for weakest-precondition analysis [22, 50, 99], which are underapproximate and use a forward soundness condition that is dual to refutation soundness (as described in Section 7.5.2). See also Ball et al. [14] for a more detailed treatment of these distinct soundness properties; in their terminology, the forwards soundness condition corresponds to a $must^{+}$ transition while refutation soundness corresponds to a $must^{-}$ transition.

Our work is most similar to that of Blackshear et al. [16], but generalizes their technique in several ways. Whereas their tool is used after-the-fact for alarm triage only, our demand-driven analysis runs during the execution of the base analysis; we generalize from boolean refutation queries of the form "is this abstract state reachable?" to value refinement queries of the form "which values can this memory location hold at this abstract state?"; and we extend their technique from Java to JavaScript, where dynamic property access introduces significant new analysis challenges for the value refiner.

Combining separate static analyses has also been the subject of much research, going back to the introduction of the reduced product abstract domain by Cousot and Cousot [32]. Much of the recent work on combinations of analyses has similarly focused on combining abstract domains while leaving the analysis algorithm itself largely unchanged, essentially composing analyses in parallel [23, 35, 92, 139]. Other works have composed analyses in series, for example to guide context-sensitivity policies using a pre-analysis [111] or to filter spurious alarms from a coarse whole-program analysis with a targeted refutation analysis [16]. Our framework, in which a forward analysis communicates and interleaves with a backward analysis that is triggered on demand when high precision is needed, cannot easily be expressed within these existing formalisms for combining abstract domains or sequencing analyses.

7.9 Conclusion

We have presented a novel program analysis mechanism, *demand-driven value refinement*, and shown how it can be used to soundly regain lost precision during the execution of a dataflow analysis for JavaScript programs. The mechanism is particularly effective at providing precise relational information, even when the base dataflow analysis employs a non-relational abstract domain.

In experiments using our implementation TAJS$_{\text{VR}}$, we have demonstrated that demand-driven value refinement is more effective than a state-of-the-art alternative technique that relies on syntactic patterns and trace partitioning, when analyzing widely used JavaScript libraries. These results suggest that demand-driven value refinement is a promising step towards fast and precise static analysis for real-world JavaScript programs.

7.10 Soundness of Demand-Driven Value Refinement

First, we argue that the base analysis remains sound when switching to the modified transfer functions $\mathscr{T}^{\mathsf{VR}}_{\ell \to_s \ell'}$ using R, provided that R is sound. Second, we extend this result from R to R_X.

First, note that if $R(\ell, y, z \mapsto \hat{z})$ always returns $\hat{\sigma}(y)$, then the definition of $\mathscr{T}^{\mathsf{VR}}_{\ell \to_s \ell'}$ yields a node transfer function that is equivalent to the original one $\mathscr{T}_{\ell \to_s \ell'}$. Therefore, $\mathscr{T}^{\mathsf{VR}}_{\ell \to_s \ell'}$ is sound provided that it is sound to use an actual refiner, instead of one always returning $\hat{\sigma}(y)$.

Since R is sound by assumption, we have – per the definition of refiner soundness in Section 7.4.1 – that for every state $\sigma \in [\![\ell]\!]_{\langle \ell_0, T \rangle}$ where ℓ is a location in the program $\langle \ell_0, T \rangle$ and the abstraction of σ satisfies the constraint $z \mapsto \hat{z}$, the value σy is in the concretization of an abstract value in $R(\ell, y, z \mapsto \hat{z})$ for any y. That is, the refinement result over-approximates the possible concrete values of y at ℓ under the given constraint. As such, so long as the partitioning $\mathit{Part}(\hat{\sigma}z)$ overapproximates z, $V(\hat{\sigma}, \ell, y, z, p)$ overapproximates the possible values written to $\hat{m}$ and the base analysis therefore remains sound.

On the other hand, if the base analysis receives new dataflow that invalidates the partitioning (i.e., such that $\hat{\sigma}z$ is no longer covered by the partitioning) at ℓ, then $V(\hat{\sigma}, \ell, y, z, p)$ may not be overapproximate. However the dataflow update at ℓ causes the refinement to rerun with a new partitioning that does cover the new $\hat{\sigma}z$.

All such new dataflow will eventually be processed, so eventually the partitioning $\mathit{Part}(\hat{\sigma}z)$ will overapproximate z at ℓ. Thus, from trace partitioning, the refinement result is sound with respect to the collecting semantics and $V(\hat{\sigma}, \ell, y, z, p)$ therefore bounds the possible value written to $\hat{m}$.

We will now extend this to show that it is sound to use R_X instead of R, meaning that the value refiner can use dataflow facts from the base analysis.

Note that, by the definition of R_X soundness in Section 7.4.3, R_X overapproximates the full collecting semantics when the base analysis has abstracted all traces. As such, soundness with R_X follows from soundness with R once the base analysis has abstracted all traces. Therefore, we only need to argue the case where the base analysis issues a refinement query at ℓ_r, R_X accesses the base analysis' state at ℓ_d, and there exists a concrete trace $\ell_0 \to_{s_1} \cdots \to_{s_d} \ell_d \to \cdots \to_{s_r} \ell_r$.

If all partial concrete traces to ℓ_d are abstracted by the base analysis state (as defined in Section 7.4.3), then the abstract state read at ℓ_d overapproximates the possible concrete states at ℓ_d and the refinement result is sound. Otherwise, there exists a trace $\ell_0 \to_{s_1} \cdots \to_{s_d} \ell_d$, that is not yet abstracted by the base analysis. When this trace gets abstracted by the base analysis, we will have new dataflow at ℓ_d. At that point, ℓ_r will be added to the worklist due to the updated worklist dependency map of Section 7.4.3. That is, ℓ_r will always be re-added to the worklist if a dataflow fact it depended on was from a not-yet-overapproximating base analysis state.

Thus, in the final transfer function execution at ℓ_r, all the dataflow facts used by the value refiner will be overapproximate with respect to the collecting semantics. Therefore, the refinement result is overapproximate as well by refiner soundness, so

$V(\hat{\sigma}, \ell, y, z, p)$ bounds the possible concrete values written to $\hat{m}$ and the base analysis therefore remains sound when using the modified transfer functions $\mathscr{T}^{\mathsf{VR}}_{\ell \to_s \ell'}$ and the refiner R_X.

7.11 Soundness of the Value Refiner $R^{\leftarrow}$

Let us write $v \models \hat{v}$ for v being in the concretization of $\hat{v}$. Then, following the definition of refiner soundness given in Section 7.4, it suffices to show, for all variables x, program locations ℓ, and constraints $y \mapsto \hat{v}$, that if a concrete state σ such that $\sigma y \models \hat{v}$ is in the collecting semantics $[\![\ell]\!]_{\langle \ell_0, T \rangle}$, then there exists a corresponding $\hat{u}$ in the refinement $R^{\leftarrow}(\ell, x, y \mapsto \hat{v})$ such that $\sigma x \models \hat{u}$.

Take any ℓ, x, $y \mapsto \hat{v}$, and σ, supposing that $\sigma y \models \hat{v}$ and $\sigma \in [\![\ell]\!]_{\langle \ell_0, T \rangle}$. Since σ is in the concrete semantics at ℓ, there exists a trace leading to that state: $\ell_0 \to_{s_0} \ell_1$, ... , $\ell_n \to_{s_n} \ell \in T^*$ where $([\![s_n]\!] \circ [\![s_{n-1}]\!] \circ \cdots \circ [\![s_0]\!])(\varepsilon) = \sigma$.

Let φ be the initial abstract store given in the definition of $R^{\leftarrow}$. That is, let $\varphi = x \mapsto \text{RES} * y \mapsto \hat{y} \wedge \hat{y} = \hat{v}$. Note that, by construction, there exists $(\sigma, \eta) \in \gamma\varphi$ such that $\eta(\text{RES}) = \sigma x$.

Let φ_i be the symbolic store computed at ℓ_i by the fixpoint algorithm. By refutation soundness, it must be the case that $\langle \varphi_n \rangle s_n \langle \varphi \rangle$, $\langle \varphi_{n-1} \rangle s_{n-1} \langle \varphi_n \rangle$, and so on through $\langle \varphi_0 \rangle s_0 \langle \varphi_1 \rangle$.

Transitively applying the refutation soundness property for each of these triples yields the implication $(\sigma, \eta) \in \gamma\varphi \Rightarrow (\varepsilon, \eta) \in \gamma\varphi_0$. Therefore, there exists $(\varepsilon, \eta) \in \gamma\varphi_0$ with $\eta(\text{RES}) = \sigma x$.

Therefore, the upper bound $\bigsqcup_{(\sigma,\eta) \in \gamma(\varphi_0)} \beta(\eta(\text{RES}))$ (where $\beta : \text{Val} \to \widehat{\text{Val}}$ denotes value abstraction) on the value of RES that is computed for the symbolic store φ_0 overapproximates σx and thus the value refinement $R^{\leftarrow}(\ell, x, y \mapsto \hat{u})$ soundly overapproximates σx.

7.12 Supplementary Experimental Data

Table 8.3 presented accumulated statistics for all of the refinement queries performed during the analysis of the library benchmarks. In this section, we provide more granular details about the individual refinement locations. We have manually investigated all of the program locations where refinement queries are issued and categorized them into one of the following categories:

1. Intraprocedural correlated read/write pattern
2. Interprocedural correlated read/write pattern
3. No correlated read/write pattern
4. Dataflow through free variable

In addition, we make note of two other meaningful characteristics of refinement locations:

a. Transformation of the read, e.g. `target[key] = f(source[key])`

b. Conditional property write

The numbered categories 1, 2, and 3 indicate whether a correlated read/write is present and, if so, whether it spans a procedure boundary. Category 4 indicates that the memory location being refined is a free variable in the function where refinement is triggered.

The lettered categories describe extra characteristics about the property read/write patterns: "a" patterns transform the read property value before writing it, and "b" patterns perform the property write conditionally only when a predicate is satisfied.

A particular refinement location belongs to one of the numbered categories and any number of the lettered categories.[10]

The results of the investigation are shown for Underscore, Lodash3, Lodash4, Prototype, Scriptaculous, and jQuery in tables 7.4, 7.5, 7.6, 7.7, 7.8, and 7.9 respectively.

The tables each have the same structure and provide the following information: "Ref. loc" is the source location in the program where refinement queries are issued, written as "line:column"; "Queries" is the total number of refinement queries issued; "Success" is the percentage of queries where the value refiner produces a result more precise than the base analysis state for the requested memory address; "Avg. time" is the average time spent by the value refiner on each query; "Avg. locs" is the average number of program locations visited per refinement query; "Inter." is the percentage of the queries where the value refiner visits multiple functions; "PNI" is the percentage of queries where the value refiner applies property name inference (see Section 7.5.3); "Test cases" is the number of test cases that issue refinement queries; and "Category" is a categorization of the refinement location as defined above.

In total, the refiner fails to provide precise results at seven locations, where four of them do not belong to a correlated property read/write. However, 24 of the 35 refinement locations are in correlated property read/write pairs (i.e. categories 1 and 2), indicating that such patterns are often a reason for loss of precision by the base analysis. Of the 24 correlated property read/writes, we see that 19 are intraprocedural and 5 are interprocedural.

Note that, for the two locations in Table 7.6 categorized as "2 or 3", the categorization depends on the invocation context of the function containing the refinement location. In our experiments, most of the refinement queries at locations 2002:7 and 2573:9 are issued in correlated read/write contexts (i.e. in category 2).

The tables also show that the refiner almost always provides precise results when analyzing correlated property read/write patterns, whether those patterns are intraprocedural or interprocedural. We also see that property name inference is used

[10]With the exception of the locations categorized "2 or 3", where the categorization depends on the invocation context of the containing function, as described below.

Table 7.4: Statistics for all refinement queries issued in the 182 Underscore test cases.

Ref. loc	Queries	Success	Avg. time	Avg. locs	Inter.	PNI	Test cases	Category
1492:18	48412	100.0%	2ms	5	0.0%	100.0%	182	1
1493:7	114	100.0%	9ms	9	0.0%	0.0%	25	1
22:9	24	87.5%	30ms	13	4.8%	95.2%	1	1
1037:38	56	85.7%	13ms	39	100.0%	100.0%	2	1, b
108:54	141	100.0%	1ms	5	0.0%	100.0%	11	1

Table 7.5: Statistics for all refinement queries issued in the 176 Lodash3 test cases.

Ref. loc	Queries	Success	Avg. time	Avg. locs	Inter.	PNI	Test cases	Category
10682:7	16	100.0%	4ms	18	100.0%	0.0%	8	4
2358:11	2	0.0%	-	-	0.0%	0.0%	1	3
3739:11	2	0.0%	-	-	0.0%	0.0%	1	2, b
10540:11	32800	100.0%	8ms	15	100.0%	100.0%	176	2
10084:9	49781	100.0%	3ms	7	0.0%	100.0%	176	1
3720:11	217	99.1%	25ms	5	0.0%	100.0%	2	1
2331:11	41	100.0%	50ms	67	100.0%	4.9%	1	1, a, b
10086:11	28	100.0%	7ms	18	100.0%	0.0%	8	4
2388:9	6	66.7%	39ms	75	100.0%	0.0%	1	1, a
2386:9	2	0.0%	-	-	0.0%	0.0%	1	1, a
1547:11	752	100.0%	14ms	40	100.0%	100.0%	2	1, a, b
2827:9	2	0.0%	-	-	0.0%	0.0%	1	3

in most cases. As such, the refinements also depend on the precision of the current analysis state; for instance, at Lodash3 location 2388:9, we see that they succeed for most but not all refinements. The failing refinements are issued in the same test case as the location 2358:11 refinement. Since the value refiner fails to precisely refine at 2358:11, imprecision is introduced to the analysis state which is propagated to 2388:9, meaning that the base analysis state is not precise enough to perform property name inference. The refinement at line 2386:9 fails for the same reason. For some of the other cases where refinements is sometimes but not always successful, it is typically because the refinements at that location are highly variable. For instance, category "a" and "b" locations often require reasoning about a callback function that is provided in the test case, so successful refinement depends heavily on whether the refiner is able to reason precisely about that function.

The tables also show that most successful refinements happen quite close to the refinement location. The refinement that visited the most locations visited 161 locations on average, and most refinements involved visiting less than 50 locations. This demonstrates that the loss of precision is often proximate to the critical imprecise property write. However, even though most refinements visit relatively few locations, 18 of the 35 refinement locations nonetheless require interprocedural reasoning.

Table 7.6: Statistics for all refinement queries issued in the 306 Lodash4 test cases.

Ref. loc	Queries	Success	Avg. time	Avg. locs	Inter.	PNI	Test cases	Category
15702:9	291417	100.0%	5ms	7	0.0%	100.0%	306	1
16840:11	99498	100.0%	8ms	17	100.0%	100.0%	306	2
2573:9	500	89.6%	361ms	97	100.0%	97.1%	14	2 or 3
15704:11	84	100.0%	51ms	14	0.0%	0.0%	10	4
16983:7	45	97.8%	32ms	40	100.0%	0.0%	12	4
2002:7	1209	94.7%	1229ms	161	100.0%	0.0%	16	2 or 3
12808:13	164	100.0%	4ms	5	0.0%	100.0%	1	1

Table 7.7: Statistics for all refinement queries issued in the 6 Prototype test cases.

Ref. loc	Queries	Success	Avg. time	Avg. locs	Inter.	PNI	Test cases	Category
145:7	1102	100.0%	13ms	40	46.8%	100.0%	6	1
7155:5	13	100.0%	3ms	5	100.0%	0.0%	6	4
7157:5	4	100.0%	4ms	5	100.0%	0.0%	3	4
3380:9	10	100.0%	3ms	5	100.0%	0.0%	3	1, a

Table 7.8: Statistics for all refinement queries issued in the Scriptaculous test case.

Ref. loc	Queries	Success	Avg. time	Avg. locs	Inter.	PNI	Test cases	Category
145:7	597	100.0%	13ms	37	41.9%	100.0%	1	1
7155:5	4	100.0%	4ms	5	100.0%	0.0%	1	4

Table 7.9: Statistics for all refinement queries issued in the JQuery test cases.

Ref. loc	Queries	Success	Avg. time	Avg. locs	Inter.	PNI	Test cases	Category
368:6	8	87.5%	85ms	26	0.0%	0.0%	8	1
8369:4	5	0.0%	-	-	0.0%	0.0%	5	1, a
5188:4	2	0.0%	-	-	100.0%	0.0%	1	3
5200:4	63	100.0%	5ms	5	0.0%	100.0%	1	1
3650:3	2	0.0%	-	-	0.0%	0.0%	1	3

Chapter 8

Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript

By Benjamin Barslev Nielsen (Aarhus University, Denmark) and Anders Møller (Aarhus University, Denmark). Published in Published in the Proceedings of the 34th European Conference on Object-Oriented Programming, ECOOP 2020, November 2020.

Abstract

In static analysis of modern JavaScript libraries, relational analysis at key locations is critical to provide sound and useful results. Prior work addresses this challenge by the use of various forms of trace partitioning and syntactic patterns, which is fragile and does not scale well, or by incorporating complex backwards analysis. In this paper, we propose a new lightweight variant of trace partitioning named value partitioning that refines individual abstract values instead of entire abstract states. We describe how this approach can effectively capture important relational properties involving dynamic property accesses, functions with free variables, and predicate functions. Furthermore, we extend an existing JavaScript analyzer with value partitioning and demonstrate experimentally that it is a simple, precise, and efficient alternative to the existing approaches for analyzing widely used JavaScript libraries.

8.1 Introduction

JavaScript programs are challenging to analyze statically due to the dynamic nature of the language. One of the main obstacles is the presence of *dynamic property access* operations that allow objects to be manipulated using object property names that are dynamically computed strings. A typical pattern that has received much attention is *correlated read/write pairs* [133], a simple variant of which looks as follows:

```
t = x[p];  ...  y[p] = t;
```

At run-time, this code copies a property whose name is the value of `p` from the `x` object to the `y` object. If the static analysis does not know precisely the string value of `p`, then the properties of `x` will be mixed together in `y`. Experience with analyzers such as WALA [127, 133, 144], SAFE [91, 115], JSAI [80], and TAJS [10, 70, 137] has shown that when analyzing real-world JavaScript code, including jQuery, Lodash, Underscore and other widely used libraries, such situations often cause an avalanche of spurious dataflow that makes the analysis results useless. If, for example, `x` is the object `{ m1: f1, m2: f2, ..., m10: f10 }` where `f1`, `f2`, ..., `f10` are functions, then any subsequent function call, for example `y.m3(...)`, will be treated by the analysis as a call to any of the 10 functions.

Several analysis techniques have been proposed to address this challenge. The techniques based on correlation tracking [133], static/dynamic determinacy [10, 127], and loop sensitivity [115] aim to increase precision by the use of context sensitivity or loop unrolling to ensure that the analysis has precise information about `p` in the example above. Although this approach works well in many cases, the aggressive use of context sensitivity or loop unrolling can be expensive on analysis time. Even more importantly, it falls short when p is not determinate (i.e., when its value is not fixed even when the call context is known).

An important step forward is the approach used in the CompAbs analyzer, which is built on SAFE [85]: Even if `p` is imprecise, the loss of precision at the property write operation can be avoided by applying trace partitioning [122] at the property read operation, based on which properties exist on `x`. Intuitively, it is often not necessary to have precise information about `p`; instead we can refine the current abstract state into a collection of more precise partitions, one for each of the 10 properties of `x` (plus one extra for the case where `p` is none of those strings, but let us ignore that for now), and after the property write operation merge them again. (The same idea was used earlier in TAJS, but only at `for-in` loops, not at dynamic property reads [10].) This approach, however, also has drawbacks. Trace partitioning is expensive, so it must be used scarcely: in the example, the code between the dynamic property read and the dynamic property write is essentially analyzed 10 times. For this reason, CompAbs relies on a syntactic pre-analysis to recognize different kinds of correlated read/write pairs for guiding the creation and merging of partitions.

Recent work [137] has shown that the syntactic pre-analysis approach of CompAbs is too fragile, for example, it is incapable of analyzing the Lodash library (see Section 12.2), and demand-driven value refinement has been proposed as an alternative. Instead of relying on context sensitivity, loop unrolling, or trace partitioning, that approach applies, during the analysis when encountering a dynamic property write operation with an imprecise property name, a separate backwards analysis to regain the relation between the property name and the value to be written. Although demand-driven value refinement has been shown to work quite well in practice, building a backwards analysis for the full JavaScript language and its standard library is a major endeavor, so developing simpler alternatives is desirable.

Our approach builds upon the observation from CompAbs that sufficient precision can be obtained using trace partitioning based on the properties of the object being read. Our key insight is that we do not need to partition the entire abstract state as done by CompAbs: It suffices to only partition the abstract values for the property name `p` and the value being read `x[p]` in the above example. This means that instead of analyzing the code 10 times, we only analyze it once, but using partitioned abstract values that retain the correlation between `p` and `x[p]`. The partitioned abstract values are introduced at `t = x[p]` and used at `y[p] = t` by means of specialized transfer functions. We refer to this variant of trace partitioning as *value partitioning*. Since partitioning individual abstract values does not increase the analysis complexity as much as partitioning entire states, it becomes feasible to apply value partitioning more extensively, at every dynamic property read where the property name is imprecise, thereby obviating the need for the syntactic pre-analysis.

In this paper we present a theoretical framework for value partitioning, together with three instantiations: *property-name partitioning* (which is the one used in the example above), *free-variable partitioning* (to improve precision for free variables of closures), and *type partitioning* (to improve precision for predicate functions). Additionally, we extend the static analyzer TAJS with all three kinds of value partitioning and demonstrate that the approach is effective for analyzing popular JavaScript libraries. Value partitioning is a lightweight alternative to the existing approaches to relational analysis for JavaScript: Compared to CompAbs-style trace partitioning it avoids many redundant computations caused by similarities between different partitions, and compared to demand-driven value refinement it avoids the need for creating a separate backwards analysis.
In summary our contributions are:

- Value partitioning: a general static analysis technique that is capable of reasoning about relations between abstract values.
- Three instantiations of value partitioning, which tackle different challenges in static analysis for JavaScript, each involving relational properties:
  - property-name partitioning: relations between dynamically computed object property names and values;
  - free-variable partitioning: relations between functions and their free variables; and
  - type partitioning: relations between arguments and return values of predicate functions.
- Experimental results: We show that value partitioning makes TAJS more precise than CompAbs [85] for several real-world JavaScript libraries, including Lodash, which is the most widely used library. The resulting precision is comparable to (and in case of the Lodash4 benchmark group substantially higher than)

```
 1 function mixin(object, source) {
 2   baseFor(source, function (func, methodName) {
 3     if (!isFunction(func))
 4       return;
 5     object[methodName] = func;
 6     if (isFunction(object))
 7       object.prototype[methodName] = function() {
 8          ...
 9          func.apply(...);
10       }
11   });
12 }
13 
14 function baseFor(source, iteratee) {
15   Object.keys(source).forEach(function (key) {
16     iteratee(source[key], key);
17   });
18 }
19 
20 // usage of mixin during initialization
21 mixin(lodash, lodash);
```

Figure 8.1: Motivating example based on code from the Lodash library.

that of demand-driven value refinement [137], without the need for a separate backwards analysis.

8.2 Motivating Example and Overview

Fig. 11.1 shows a small code example based on Lodash (version 4.17.10), which is the most depended-upon of all npm packages.[1] Lines 1–12 define the function `mixin`, which copies all function properties from `source` to `object`. If `object` is a function, a new function (which on invocation calls the function to be copied) is also copied to `object.prototype`, such that instantiations of `object` (using the keyword `new`) also will have these functions. In line 21, which is executed during the initialization of Lodash, `mixin` is called with the library object as both arguments. The function `mixin` uses a helper function `baseFor` defined in lines 14–18. It is called with `source` and a callback function defined in lines 2–11. The `baseFor` function then gets all the object property names from the `source` object using `Object.keys`, and the callback function is called (line 16) for each property name and corresponding property value. Line 3 checks whether `func` is a function. If so, the function is copied to `object[methodName]` in line 5. Note that `func` actually is the value `source[methodName]`. Line 6 checks whether `object` is also a function and if so, a new function is declared and written to `object.prototype[methodName]` in line 7. When invoked, that new function calls `func` using `func.apply(...)` in line 9.

[1]Lodash (`https://lodash.com/`) has more than 115 000 dependents in npm and more than 27 million weekly downloads as of May 2020.

Such complex code is not unusual in modern JavaScript libraries. For a static analysis reasoning about the dataflow in this code, the correlation between `methodName` and `func` is critical. An analysis that loses track of this correlation will mix together all the properties of the library object `lodash` when analyzing the call `mixin(lodash, lodash)` in line 21. As a consequence, if the program being analyzed contains a call to, for example, `lodash.map`, that will be treated by the analysis as a call to any of Lodash's more than 100 different functions, not only the actual `map` function, thereby triggering an avalanche of spurious dataflow.

Existing approaches Existing JavaScript analyzers do not have precise information about the value of `key` in line 16, for various different reasons. (Most importantly, `Object.keys` produces an array of property names in unspecified order.) Previous work has suggested two approaches to analyze such code precisely even when `key` is imprecise. The CompAbs [85] approach uses trace partitioning guided by syntactic patterns. If trace partitioning is used at the dynamic property read operation in line 16, the abstract state is partitioned into a set of refined abstract states corresponding to the properties of the `source` object. This way the value of `key` is precise in each of those states, and the call in line 16 is analyzed separately for each of them. Trace partitioning, however, is expensive, so CompAbs limits the use of trace partitioning according to certain syntactic patterns. At this specific dynamic property operation, CompAbs chooses not to apply trace partitioning and fails to detect that the relation between `methodName` and `func` is important.

The second approach is demand-driven value refinement [137], which can analyze the example code with sufficient precision to avoid mixing together the Lodash functions. With this approach, the analysis detects imprecision at the dynamic property write in line 5: `methodName` is an imprecise string and `func` can be many different functions. It then queries a backwards abstract interpreter asking for the possible value of `methodName` for each of the functions. The backwards analysis returns a precise property name for each function and thereby enables the dynamic property write operation to be modeled precisely. For the dynamic property write in line 7, the function defined in lines 7–10 is written to all properties of `object.prototype`, but the abstract value being written is augmented, such that the value of `methodName` remains precise. When reading `func` in line 9, the backwards analysis is queried to get the value of `func` relative to the value of `methodName`, thereby retrieving a precise value for `func`. This ensures the desired precision, but the approach requires a complicated backwards analysis.

Value partitioning We will now informally explain how value partitioning can provide similar precision as demand-driven value refinement, but without the need for a backwards abstract interpreter. With traditional trace partitioning, as used by, for example, CompAbs, the analysis can track multiple abstract states for each program point, such that the different abstract states cover different assumptions about the execution paths that lead to that point. (Correlation tracking [133], determinacy-based analysis [10, 127], and loop sensitivity [115] can also be viewed as variations of trace partitioning.) The key idea behind value partitioning is that we can obtain a similar effect as trace partitioning by instead performing the partitioning at the level of individual abstract values. In principle, the resulting abstract domain is isomorphic to a traditional trace partitioning domain, but this approach provides more flexibility for using different kinds of partitioning for different parts of the abstract states. This general idea can be instantiated in multiple ways to track different kinds of relational properties. We next describe three instantiations that enable precise analysis of challenging JavaScript code, including the Lodash example.

Property name partitioning One instantiation is property name partitioning, which performs partitioning at dynamic property reads, similar to the CompAbs technique, but on abstract values instead of abstract states. To illustrate this mechanism by example, consider the read operation in line 16 and the correlated write operation in line 5. Assume for simplicity that the `source` object has only two properties, `{ map: f1, trim: f2 }` where `f1` and `f2` are functions, and `methodName` is an abstract value that overapproximates all valid property names. When reading `source[methodName]`, an analysis without value partitioning will read all the properties of `source`. When using value partitioning, we instead partition this value according to the property names of `source`, meaning that we obtain a value $[t_1 \mapsto \texttt{f1}, t_2 \mapsto \texttt{f2}, t_3 \mapsto \texttt{undefined}]$ where t_1, t_2, and t_3 represent different partitions.[2] Intuitively, t_1 represents the execution traces where the property name being read is `map`, t_2 similarly represents traces where the property name being read is `trim`, and t_3 represents all other traces. We similarly write the partitioned value $[t_1 \mapsto \texttt{"map"}, t_2 \mapsto \texttt{"trim"}, t_3 \mapsto \mathsf{AnyString}]$ to `methodName`.[3] In this way, the resulting abstract state retains the correlation between the values of `methodName` and `source[methodName]`.

Later the analysis reaches the write operation `object[methodName] = func`, with an abstract state where `methodName` is $[t_1 \mapsto \texttt{"map"}, t_2 \mapsto \texttt{"trim"}, t_3 \mapsto \mathsf{AnyString}]$ and `func` is $[t_1 \mapsto \texttt{f1}, t_2 \mapsto \texttt{f2}, t_3 \mapsto \texttt{undefined}]$. Since the property name and the value to be written have the same partitions, we can perform the dynamic property write separately for each partition, meaning that `f1` is written to the `map` property, and analogously for the other two partitions, thereby avoiding mixing together the properties.

Since the partitioning is performed at the value level, unlike traditional trace partitioning we do not need any extra call contexts to the callback function defined in line 2, so the overhead of value partitioning is negligible, even when the correlated read/write pairs span multiple functions. For this reason, we can apply property name partitioning at all dynamic property reads where the property name is imprecise, without the use of syntactic patterns.

[2] In JavaScript, reading an absent property yields the special value `undefined`.

[3] $\mathsf{AnyString}$ is an abstract value that represents any string. In practice we instead use a slightly more precise abstract value representing $\mathsf{AnyString} \setminus \{\texttt{"map"}, \texttt{"trim"}\}$.

Free variable partitioning A second instantiation of value partitioning is for handling free variables more precisely. In the example, this is useful for `func` in line 9, which is a free variable in the function defined in lines 7–10. At that function definition, we partition both the resulting abstract function value ℓ and the abstract value of `func` according to the existing partitioning of `func`, intuitively to be able to distinguish functions created with different values of the free variable. This means that the function value being written at the dynamic property write in line 7 is $[t_1 \mapsto \ell_{t'_1}, t_2 \mapsto \ell_{t'_2}, t_3 \mapsto \ell_{t'_3}]$ where $\ell_{t'_1}$ represents the function created at a point where `func` is `f1` (i.e., that point is at the end of a t_1 trace), and similarly for the other partitions. At the same time, the value of `func` becomes $[t_1 \mapsto \texttt{f1}, t_2 \mapsto \texttt{f2}, t_3 \mapsto \texttt{undefined}, t'_1 \mapsto \texttt{f1}, t'_2 \mapsto \texttt{f2}, t'_3 \mapsto \texttt{undefined}]$ where the three new partitions t'_1, t'_2, and t'_3 denote the new partitioning we have made (one abstract value can thus have multiple partitionings simultaneously). Using the property name partitioning mechanism described above, at the dynamic property write in line 7, $\ell_{t'_1}$ is written to the `map` property of `object.prototype`, and similarly for the other properties.

We can exploit the free variable partitioning information when the function is later called. Assume the analysis encounters a call to the `map` method. The abstract value of `lodash.prototype.map` is then $\ell_{t'_1}$. We now use t'_1 as a context in ordinary context sensitive analysis of the function, so that when reaching `func` in line 9, it suffices to consider only the t'_1 partition of `func`, which yields the precise value `f1`, so again, we successfully avoided mixing together the properties.

Type partitioning The above two uses of value partitioning are sufficient for analyzing the motivating example without critical precision losses, but we can make the analysis even more precise using a third variant. The function named `isFunction` used in the branch condition in line 6 is a typical example of a *predicate function*, i.e., a one-parameter function that returns a boolean, in this case testing whether the value passed in is a function. Assume the abstract value of the argument `object` is `fun1`|`obj2`, meaning that it represents either a function `fun1` or a non-function object `obj2`. With a simple analysis, the abstract return value and hence the branch condition is Bool representing any boolean value, so the analysis does not know that `object` cannot be `obj2` inside the branch. This causes the analysis to spuriously raise a type error when writing to `object.prototype` in line 7.

Type partitioning avoids that imprecision as follows. Type partitioning is triggered at any call to a function with one argument, and partitions that argument according to its types. In this case, the value of `object` is partitioned into $[a \mapsto \texttt{fun1}, b \mapsto \texttt{obj2}]$. The result value from `isFunction` then becomes $[a \mapsto \mathsf{true}, b \mapsto \mathsf{false}]$, which we can exploit using ordinary control sensitivity [57] (also called type refinement [79]) at the 'true' branch such that `object` in line 7 will only be `fun1` and not `obj2`.

Overview In Section 12.3 we give a brief introduction to the analysis domain of TAJS. Section 8.4 explains the general value partitioning mechanism, and Section 8.5 details the three instantiations: property name partitioning, free variable partition-

$r_1[r_2] \leftarrow r_3$:	Writes r_3 to the property named r_2 of the object r_1
$r_1 \leftarrow r_2[r_3]$:	Reads the property named r_3 of the object r_2 to r_1
$r_1 \leftarrow x$:	Reads the value of the variable x to r_1
$x \leftarrow r_1$:	Writes r_1 to the variable x
$r_1 \leftarrow c$:	Assigns the constant c to r_1
$r_1 \leftarrow \mathsf{function}(x)\{\cdots\}$:	Creates a closure for the function and stores it in r_1
$\mathsf{if}(r_1)$:	Conditionally propagates dataflow (to model `if` and `while`)
$r_1 \leftarrow r_2(r_3)$:	Calls the function r_2 with argument r_3 and stores the result in r_1
$r_1 \leftarrow r_2 \oplus r_3$:	Computes the binary operation $r_2 \oplus r_3$ and stores the result in r_1

Figure 8.2: The main flow graph instructions in TAJS.

$$\begin{aligned} n \in N &: \mathit{Nodes} \\ c \in C &: \mathit{Contexts} \\ p \in P &: \mathit{Property\ names} \\ \ell \in L = N \times C &: \mathit{Locations} \end{aligned} \qquad \begin{aligned} X \in \mathit{AnalysisLattice} &= L \to \mathit{State} \\ \sigma \in \mathit{State} &= (L \to \mathit{Obj}) \times \mathit{Registers} \\ o \in \mathit{Obj} &= P \to \mathit{Value} \\ r \in \mathit{Registers} &= R \to \mathit{Value} \\ v \in \mathit{Value} &= \mathit{Prim} \times \mathscr{P}(L) \end{aligned}$$

Figure 8.3: Simplified abstract domain.

ing, and type partitioning. Section 12.7 describes our experimental evaluation, and Section 12.8 discusses related work.

8.3 Background: The TAJS Analyzer

In this section we give a brief introduction to a heavily simplified version of the analysis domain and program representation used in TAJS [10, 70], which lays the foundation for our extensions in the following sections.

TAJS is an open-source dataflow analysis tool for JavaScript built as a monotone framework [76]. A JavaScript program is represented as a control flow graph for each function, with nodes representing primitive instructions of the different kinds listed in Fig. 8.2. Each instruction operates on registers, which can be thought of as special local variables. For simplicity, we ignore `this` and receiver objects at calls, and we assume all functions have only one parameter. As an example, the single JavaScript statement `y[p] = x[p]` is represented as six flow graph nodes as shown in Fig. 8.4.

The components of the abstract domain are summarized in Fig. 8.3. A *location* is a pair of a node and a context. The contexts allow for context sensitivity (using the context-sensitivity strategy described by Andreasen and Møller [10]). The main abstract domain, *AnalysisLattice*, is a lattice that maps locations to abstract states, where each state contains abstract values of object properties and registers. Objects

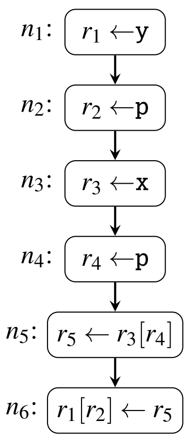


Figure 8.4: Fragment of a control flow graph, for the single statement `y[p] = x[p]`.

are modeled using context-sensitive allocation-site abstraction [24, 110], so abstract object addresses are simply locations.[4] Functions are special kinds of objects. Abstract values are modeled using a product of a constant-propagation lattice [82] named *Prim* of primitive values (strings, numbers, etc.) and a powerset lattice of object addresses.

The analysis is control sensitive by pruning infeasible dataflow at if nodes. This includes not only eliminating flow along unreachable branches, for example when a branch condition is definitely false [141], but also filtering abstract values based on the branch condition [57, 79]. As an example, the JavaScript code `if(z)` is represented by two primitive instructions, $r_6 \leftarrow$ `z` and $\text{if}(r_6)$. In the 'true' branch, not only r_6 but also `z` must have the value `true`.[5] To track the connection between r_6 and `z`, a simple intraprocedural must-equals analysis is performed alongside the main dataflow analysis. We leverage this mechanism in Section 8.5, for example to obtain the information that r_2, r_4, and `p` must have the same value at the property read operation in Fig. 8.4 (unless a property accessor changes `p`). To keep Fig. 8.3 simple, we omit the must-equals information in the description of the *State* lattice.

In the following sections, with a slight abuse of notation we let $\sigma(r)$ denote the value of register r in state σ, and similarly, $\sigma(\texttt{x})$ denotes the value of variable `x`. Also, we use the notation $\sigma(r) := \ldots$ to describe the operation of writing a given value to register r and also to the variables and registers that are equal to r according to the must-equals information. If $\ell \in L$ is a location representing an object address, we sometimes write ℓ for the abstract value $(\bot, \{\ell\}) \in \mathit{Value}$. Similarly, for abstract values that represent primitive values only, we omit the location sets, for example, `"foo"` denotes the abstract value $(\texttt{"foo"}, \emptyset) \in \mathit{Value}$.

[4]TAJS models absence/presence of object properties and uses two artificial properties DEFAULTNUMERIC and DEFAULTOTHER to model properties with unknown numeric/non-numeric names; we ignore that here.

[5]In actual JavaScript, the value must be *truthy*, which also includes nonempty strings, nonzero numbers, and objects.

$$\begin{aligned}
t \in T &\; : \textit{Partition tokens} \\
o \in \textit{Obj} &= P \to \textit{PartitionedValue} \\
r \in \textit{Registers} &= R \to \textit{PartitionedValue} \\
pv \in \textit{PartitionedValue} &= T \hookrightarrow \textit{Value}
\end{aligned}$$

Figure 8.5: Extension of the abstract domain for value partitioning.

We omit many details of TAJS, including the definitions of the concretizations of the lattice elements, the definitions of the transfer functions for the different instructions, how values of variables are being stored in special activation objects, and how a call graph is built during the analysis. Analyzing full JavaScript also requires reasoning about prototypes, scope chains, implicit type conversions, exceptions, the standard library, property accessors (getters and setters), and much more. It suffices to know that the resulting abstract states soundly overapproximate the possible program behavior [31].

A *trace* is a concrete execution of the program expressed as a finite sequence of pairs (ℓ, γ) where ℓ is a location and γ is a concrete state, starting at the program entry point with the initial call context in an empty state. The semantics of a program is defined as a set of traces. The *collecting semantics* is the program semantics projected onto the program locations: Given a location ℓ, the collecting semantics for ℓ, denoted $[\![\ell]\!]$, is the set of states that appear at ℓ in the set of traces defined by the program semantics. The analysis result is thus a lattice element $X \in \textit{AnalysisLattice}$ such that $[\![\ell]\!]$ is a subset of the concretization of $X(\ell)$ for all locations $\ell \in L$.

8.4 Value Partitioning

To prepare the analysis for value partitioning, we introduce a set T of partition tokens and replace occurrences of *Value* by *PartitionedValue* in the abstract domain, as shown in Fig. 8.5. A *partitioned value* is a partial map from partition tokens to ordinary values. We use the notation $[t_1 \mapsto v_1, \ldots, t_k \mapsto v_k]$ (or set-builder notation like $[t_i \mapsto v_i \mid i = 1, \ldots, k]$) to denote the partitioned value that maps t_i to v_i for each $i = 1, \ldots, k$ and is undefined for all other partition tokens.

The partition tokens play a similar role as in trace partitioning [122], but at the level of abstract values. (We explain the differences between value partitioning and traditional trace partitioning in more detail in Section 12.8.) A partition token intuitively represents a set of execution traces. The special token ANY represents all traces, so the partitioned value $[\text{ANY} \mapsto v]$ has the same meaning as the ordinary value v in the original abstract domain. As an invariant, all partitioned values we use are defined for the token ANY.[6] We extend partitioned values to be total functions $pv\colon T \rightarrow \mathit{Value}$ by defining $pv(t) = pv(\text{ANY})$ when $t \notin \text{dom}(pv)$.[7]

Assume $X \in \mathit{AnalysisLattice}$ is the result of analyzing a given program, $\sigma = X(\ell)$ is the abstract state at some location ℓ, and $[\dots, t \mapsto v, \dots] = \sigma(r)$ is the partitioned value of some register r. The meaning of such a partitioned value is that for any trace that ends at ℓ and is in the set of traces represented by t, the concrete value of r is in the concretization of the abstract value v.

A *covering*[8] at a location ℓ is a set of partition tokens where the union of the sets of traces they represent is the set of all traces that lead to ℓ. This means that if $\sigma(\mathtt{x}) = [\dots, t_1 \mapsto v_1, ..., t_k \mapsto v_k, \dots]$ where $\sigma = X(\ell)$ for some program variable `x` at location ℓ where $\{t_1, \dots, t_k\}$ is a covering, then for every concrete state in $[\![\ell]\!]$, the value of `x` is in the concretization of at least one of the abstract values $v_1, \dots, v_k$. For the initial abstract state at the program entry, all partitioned values use the trivial covering $\{\text{ANY}\}$.

Now that we have generalized the abstract domain, it is easy to adjust all transfer functions for the different kinds of nodes to operate on partitioned values instead of ordinary values. As an example, the original transfer function for $r_1 \leftarrow r_2 \oplus r_3$ updates a given abstract state σ by $\sigma(r_1) := \sigma(r_2) \oplus \sigma(r_3)$ (where $\oplus$ applied to abstract values works as in constant propagation).[9] When switching to the domain with partitioned values, we simply replace $\sigma(r_2) \oplus \sigma(r_3)$ by $[t \mapsto pv_2(t) \oplus pv_3(t) \mid t \in \text{dom}(pv_2) \cup \text{dom}(pv_3)$ where $pv_2 = \sigma(r_2)$ and $pv_3 = \sigma(r_3)]$. The other transfer functions and least-upper-bound are adapted similarly.

A small example can illustrate how partitioning can make the analysis relational. Assume the binary operation is equality, $r_1 \leftarrow r_2 == r_3$, and that we have two partitions, t_1 and t_2, where both registers r_2 and r_3 have the value 42 in partition t_1, and both have the value `"foo"` in partition t_2. With partitioning, the value of r_1 becomes $[t_1 \mapsto \mathtt{true}, t_2 \mapsto \mathtt{true}]$ (i.e., definitely true), whereas without partitioning, r_2 and r_3 both have the value `42|"foo"`, so the value of r_1 becomes AnyBool (i.e., true or false).

To get any advantage of the new abstract domain, we of course need to modify specific transfer functions to selectively introduce partition tokens and further exploit the extra information available regarding relational properties between values. We show how that can be accomplished in Section 8.5. Those mechanisms rely on some general operations for manipulating the partitions in partitioned values. Most importantly, we use an operation $\uplus$ when introducing new coverings: $pv_1 \uplus pv_2$ where $pv_1, pv_2 \in \mathit{PartitionedValue}$ denotes the combined partitioned value. For each token that is only present in one of pv_1 or pv_2, the new value will be the value for that token,

[6]When we define a partitioned value $[t_i \mapsto v_i \mid i = 1, \dots, k]$ without an ANY token, an ANY partition is implicitly created with value $v_1 \sqcup \cdots \sqcup v_k$.

[7]In trace partitioning terminology, this use of ANY corresponds to a simple pre-ordering of partition tokens.

[8]For formal definitions of the notions of traces and coverings, see Rival and Mauborgne [122]. Basing our approach on partitions instead of coverings (a *partition* is a covering where all the trace sets are disjoint) could improve precision but would complicate the analysis without much practical benefit.

[9]The actual TAJS analysis also models implicit type conversions.

$$
\begin{array}{rcll}
T & ::= & \text{ANY} & \text{(Section 8.4)} \\
 & \mid & \text{VAL}\langle N, R, \mathit{Value}\rangle & \text{(Section 8.5.1)} \\
 & \mid & \text{FUN}\langle F, C, T\rangle & \text{(Section 8.5.2)} \\
 & \mid & \text{TYPE}\langle N, R, \mathit{Types}\rangle & \text{(Section 8.5.3)} \\
\\
\mathit{Types} & ::= & \mathsf{undefined} \mid \mathsf{null} \mid \mathsf{number} \mid \mathsf{string} \mid \mathsf{boolean} & \\
 & \mid & \mathsf{object} \mid \mathsf{array} \mid \mathsf{function} \mid \mathsf{regexp} &
\end{array}
$$

Figure 8.6: Partition tokens used by property name partitioning, free variable partitioning, and type partitioning.

and for each token shared by pv_1 and pv_2, the new value will be the join of the two respective values.

8.5 Three Instantiations of Value Partitioning

We now present three instantiations of the value partitioning framework. Each of them targets a category of relational properties that are relevant to analysis of JavaScript libraries. Each instantiation introduces a family of partition tokens, as shown in Fig. 8.6, along with some modification of the analysis transfer functions. Each partition token represents a set of traces, as explained in the following.

8.5.1 Property Name Partitioning

The first use of value partitioning is for improving precision at correlated object property read/write operations as in the motivating example.

Partition tokens for property name partitioning We introduce a family of partition tokens, $\text{VAL}\langle n,r,v\rangle$, where $n \in N$, $r \in R$, and $v \in \mathit{Value}$. Such a token represents the set of traces where at the last occurrence of n, the value of register r is v. In all $\text{VAL}\langle n,r,v\rangle$ tokens we use in property name partitioning, the node n is a property read node (i.e., of the form $r_1 \leftarrow r_2[r_3]$), the register r is the one holding the property name in that instruction (i.e., r_3 in $r_1 \leftarrow r_2[r_3]$), and the value v is a property name (i.e., an element of P).[10]

As an example, assume the code from Fig. 8.4 appears inside a loop, and consider the following two traces that both end at n_6:

$$\tau_a = \cdots (n_1, \gamma_{1a})(n_2, \gamma_{2a})(n_3, \gamma_{3a})(n_4, \gamma_{4a})(n_5, \gamma_{5a})(n_6, \gamma_{6a})$$

[10]In JavaScript, property names are either strings, which we model in the sub-lattice *Prim*, or symbols, which can be modeled as special heap locations.

$$\sigma(r_3) := \begin{cases} \sigma(r_3) \uplus [\textsc{val}\langle n, r_3, p\rangle \mapsto p \mid p \in \textsc{propNames}(\sigma(r_2))] \\ \qquad \uplus [\textsc{val}\langle n, r_3, \textsc{other}\rangle \mapsto \mathsf{AnyString}] & \text{if } \sigma(r_3)(\textsc{any}) = \mathsf{AnyString} \\ \sigma(r_3) & \text{otherwise} \end{cases}$$

$$\sigma(r_1) := \begin{cases} [\textsc{val}\langle n, r_3, p\rangle \mapsto \textsc{readProp}(\sigma(r_2)(\textsc{any}), p) \mid p \in \textsc{propNames}(\sigma(r_2))] \\ \qquad \uplus [\textsc{val}\langle n, r_3, \textsc{other}\rangle \mapsto \texttt{undefined}] & \text{if } \sigma(r_3)(\textsc{any}) = \mathsf{AnyString} \\ [\textsc{any} \mapsto \textsc{readProp}(\sigma(r_2)(\textsc{any}), \sigma(r_3)(\textsc{any}))] & \text{otherwise} \end{cases}$$

Figure 8.7: Introduction of partitioned values at a dynamic property read node n of the form $r_1 \leftarrow r_2[r_3]$.

and

$$\tau_b = \cdots(n_1, \gamma_{1b})(n_2, \gamma_{2b})(n_3, \gamma_{3b})(n_4, \gamma_{4b})(n_5, \gamma_{5b})(n_6, \gamma_{6b})$$

where each "$\cdots$" is a trace prefix leading from the program entry point to this part of the code, $\gamma_{1a}, \ldots, \gamma_{6b}$ are concrete states, and τ_a is a prefix of τ_b. The last occurrence of n_5 (which is the instruction $r_5 \leftarrow r_3[r_4]$) is emphasized in each of the traces. Also assume that the value of the register r_4 is `"foo"` in γ_{5a} and `"bar"` in γ_{5b}. Note that r_4 is the register holding the property name at the n_5 instruction. In this situation, the token $\textsc{val}\langle n_5, r_4, \texttt{"foo"}\rangle$ represents τ_a but not τ_b.

Dynamic property reads Fig. 8.7 shows the modified transfer function for read-property nodes, $r_1 \leftarrow r_2[r_3]$. The function $\textsc{readProp}(v_1, v_2)$ looks up the abstract value of properties named v_2 in the abstract objects pointed to by v_1 in the current state σ.[11] Property name partitioning is triggered if the property name is not precise (here modeled as $\mathsf{AnyString}$), so in that case we partition the property name r_3 with respect to the properties that appear in the abstract objects pointed to by r_2 (expressed as $\textsc{propNames}(\sigma(r_2))$), and perform the property read for each partition to obtain a partitioned value for r_1. We use the artificial abstract value $\textsc{other} \in \mathit{Value}$ to represent all other properties; for that partition, the result value becomes `undefined`.[12] If the property name r_3 is already precise (corresponding to the 'otherwise' cases), there is no need to introduce new partitions, so in that case r_3 is unmodified and the result value r_1 is obtained directly using $\textsc{readProp}$ and the $\textsc{any}$ partition token.

Recall that a $\textsc{val}\langle n, r, p\rangle$ token represents the set of traces where at the last occurrence of n, the value of register r is v. To respect this property we need to remove all existing $\textsc{val}\langle n, \_, \_\rangle$ tokens from the abstract state before applying the transfer function for dynamic property reads. (This is safe because every abstract value still has other coverings, in particular $\{\textsc{any}\}$.)

[11] Reading an object property is a nontrivial operation in JavaScript because of prototypes, getters, and implicit type conversions. Importantly, the value partitioning mechanism is orthogonal to such JavaScript technicalities.

[12] In our implementation we use a more precise string lattice, which allows us to express more precisely that $\sigma(r_3)$ for the $\textsc{val}\langle n, r_3, \textsc{other}\rangle$ partition is $\mathsf{AnyString} \setminus \textsc{propNames}(\sigma(r_2))$, i.e., any string except for the property names that are covered by other partitions. See also Footnote 3.

```
for each $t \in \textsc{chooseCommonCovering}(\sigma(r_2), \sigma(r_3))$:
    $\textsc{writeProp}\big(\sigma(r_1)(\textsc{any}), \sigma(r_2)(t), \sigma(r_3)(t)\big)$
```

Figure 8.8: Exploiting partitioned values at a dynamic property write node, $r_1[r_2] \leftarrow r_3$.

Notice that for both r_3 and r_1, the new partitions use the partition tokens $\textsc{val}\langle n, r_3, p\rangle$ where n is the read-property node. Evidently, the new partition tokens form a covering. Also, this new transfer function respects the interpretation of the newly added $\textsc{val}\langle n, r, p\rangle$ tokens, and due to the partitioning, the resulting abstract states maintain the relation between the involved object property names and values.

Dynamic property writes Next, we modify the transfer function for dynamic property writes, $r_1[r_2] \leftarrow r_3$, as shown in Fig. 8.8, to take advantage of the partitionings introduced at dynamic property reads. The function $\textsc{writeProp}(v_1, v_2, v_3)$ writes v_3 to the properties named v_2 in the objects referred to by v_1.[13] The function $\textsc{chooseC-ommonCovering}$ finds a covering shared by the property name $\sigma(r_2)$ and the value to be written $\sigma(r_3)$. (An example is given below.) If multiple such coverings exist, a largest one (i.e., one with the largest number of partition tokens) is selected.[14] Recall that the two values always share the $\{\textsc{any}\}$ covering, which will be used if no other covering exist. When a covering has been chosen, the value is written to the appropriate object property for each partition, thereby exploiting the relational information. In case the $\{\textsc{any}\}$ covering is chosen, the transfer function behaves as the original version without value partitioning.

Example To better understand property name partitioning, we now explain the mechanism in more detail on the example given in Fig. 8.4. Let us assume that $\sigma(\texttt{p}) = [\textsc{any} \mapsto \mathsf{AnyString}]$, $\sigma(\texttt{x}) = [\textsc{any} \mapsto \mathit{obj}_2]$ and $\sigma(\texttt{y}) = [\textsc{any} \mapsto \mathit{obj}_1]$ where in state σ, obj_1 is the location of an empty abstract object and obj_2 is the location of an abstract object with two properties, `{foo: 1, bar: 2}`. This means when analyzing the read property node $r_5 \leftarrow r_3[r_4]$ we have $\sigma(r_3) = [\textsc{any} \mapsto \mathit{obj}_2]$ and $\sigma(r_4) = [\textsc{any} \mapsto \mathsf{AnyString}]$. Since the property name r_4 is imprecise, the first case in each definition in Fig. 8.7 applies, meaning that value partitioning is triggered. Since $\textsc{propNames}(\sigma(r_2)) = \{\texttt{"foo"}, \texttt{"bar"}\}$, we update r_4 such that $\sigma(r_4)$ equals $[\textsc{val}\langle n, r_4, \texttt{"foo"}\rangle \mapsto \texttt{"foo"}, \textsc{val}\langle n, r_4, \texttt{"bar"}\rangle \mapsto \texttt{"bar"}, \textsc{val}\langle n, r_4, \textsc{other}\rangle \mapsto \mathsf{AnyString}]$, where n is the read property node. Recall from Section 12.3 that the operation $\sigma(r_4) := \ldots$ does not only modify r_4 but also the must-equals variables and

[13]We omit the details of how the implementation of $\textsc{writeProp}$ in TAJS handles strong/weak updates, setters, and implicit type conversions. Importantly, the value partitioning mechanism is orthogonal to such JavaScript technicalities.

[14]Multiple coverings can arise if, for example, the same property name is used at two different property read operations. We choose the largest covering based on the heuristic that fine-grained coverings lead to higher precision than coarse-grained coverings. The most important consequence of this heuristic is that we avoid the $\{\textsc{any}\}$ covering if others are available. In case of multiple largest ones, an arbitrary one is selected among them.

registers, meaning that this partitioned value is also written to r_2 and `p`. The value being read gets the same partitions, such that $\sigma(r_5)$ becomes $[\text{VAL}\langle n, r_4, \texttt{"foo"}\rangle \mapsto 1, \text{VAL}\langle n, r_4, \texttt{"bar"}\rangle \mapsto 2, \text{VAL}\langle n, r_4, \text{OTHER}\rangle \mapsto \texttt{undefined}]$.

When reaching the property write operation $r_1[r_2] \leftarrow r_5$, the state σ contains $\sigma(r_2) = [\text{VAL}\langle n, r_4, \texttt{"foo"}\rangle \mapsto \texttt{"foo"}, \text{VAL}\langle n, r_4, \texttt{"bar"}\rangle \mapsto \texttt{"bar"}, \text{VAL}\langle n, r_4, \text{OTHER}\rangle \mapsto \mathsf{AnyString}]$ and $\sigma(r_5) = [\text{VAL}\langle n, r_4, \texttt{"foo"}\rangle \mapsto 1, \text{VAL}\langle n, r_4, \texttt{"bar"}\rangle \mapsto 2, \text{VAL}\langle n, r_4, \text{OTHER}\rangle \mapsto \texttt{undefined}]$. We now apply the transfer function from Fig. 8.8. The two values $\sigma(r_2)$ and $\sigma(r_5)$ share two coverings: $\{\text{ANY}\}$ and $\{\text{VAL}\langle n, r_4, \texttt{"foo"}\rangle, \text{VAL}\langle n, r_4, \texttt{"bar"}\rangle, \text{VAL}\langle n, r_4, \text{OTHER}\rangle\}$. Since the second covering is largest, that one is picked by $\text{CHOOSECOMMONCOVERING}(\sigma(r_2), \sigma(r_5))$. We therefore perform three writes corresponding to the abstract assignments obj_1`["foo"]=1`, obj_1`["bar"]=2`, and obj_1`[AnyString]=undefined`; notably, the properties `foo` and `bar` are not mixed together.

8.5.2 Free Variable Partitioning

We now explain how to leverage value partitioning to gain precision for free variables, such as `func` in line 9 in the motivating example from Fig. 11.1.

Extending the abstract domain The first step is to extend the abstract domain as shown in Fig. 8.9. The *Value* component in *PartitionedValue* is replaced by *FPValue*, which is a product of *FunctionPartitions* and *Value*. The component *FunctionPartitions* is a set of partition tokens, which we use for tracking which partitions the functions described in the *Value* component may have been declared in. (For instance, for the motivating example from Fig. 11.1, the function declared in lines 7–10 was created in the partitions t'_1, t'_2, and t'_3 so the corresponding abstract values become[15] $(\{t'_1\}, \ell)$, $(\{t'_2\}, \ell)$, and $(\{t'_3\}, \ell)$, where ℓ denotes the location for the created closure.) To preserve this information when analyzing calls to such functions, we also augment the set of contexts to include this information (replacing C by C' in *AnalysisLattice*). The *FunctionPartitions* set is empty for values and contexts that do not use free variable partitioning.

Next, we introduce a new kind of partition tokens, and we then describe how elements of *FunctionPartitions* are created at function expressions and used at read variable nodes.

Partition tokens for free variable partitioning We introduce a new kind of partition tokens, $\text{FUN}\langle f, c, t\rangle$, where f is a function, $c \in C'$ is a context, and $t \in T$ is a partition token. A trace is represented by such a token if (1) the trace ends at a program location that belongs to a closure that was created when the trace up to that point was a t trace, and (2) that point in the trace is in function f in context c. (For instance, in the motivating example, a trace ending in line 9 where the currently executed closure was

[15]These three abstract values are denoted $\ell_{t'_1}$, $\ell_{t'_2}$, and $\ell_{t'_3}$, respectively, in the motivating example in Section 12.2.

$$\begin{aligned}
pv \in \mathit{PartitionedValue} &= T \hookrightarrow \mathit{FPValue} \\
fv \in \mathit{FPValue} &= \mathit{FunctionPartitions} \times \mathit{Value} \\
fp \in \mathit{FunctionPartitions} &= \mathscr{P}(T) \\
\mathit{AnalysisLattice} &= C' \times N \to \mathit{State} \\
c \in C' &= \mathit{FunctionPartitions} \times C
\end{aligned}$$

Figure 8.9: Extensions of the abstract domain for free variable partitioning.

$$pv(t) \;=\; \begin{cases} pv(t) & \text{if } t \in \mathrm{DOM}(pv) \\ \bot & \text{otherwise if } t = \mathrm{FUN}\langle f,c,t'\rangle \wedge \\ & \qquad\qquad \exists c',t''\colon c \neq c' \wedge \mathrm{FUN}\langle f,c',t''\rangle \in \mathrm{DOM}(pv) \\ pv(\mathrm{ANY}) & \text{otherwise} \end{cases}$$

Figure 8.10: Redefining how partitioned values are extended to total functions, exploiting free variable partitioning.

created in line 7 at the end of a t_1 trace can be represented by $\mathrm{FUN}\langle f,c,t_1\rangle$, where f is the function at lines 2–11 and c is the context for the call to that function.) We only allow such partition tokens to appear in abstract values of variables that are declared in f. Intuitively, we use these partition tokens to obtain a form of heap specialization (also called heap cloning or context sensitive heap) [110] for the activation objects of f.[16]

An important property is that if the abstract value of a variable `x` declared in a function f contains partition token $\mathrm{FUN}\langle f,c',t''\rangle$ for some c',t'' but not $\mathrm{FUN}\langle f,c,t'\rangle$ for any c,t' where $c \neq c'$, then f has not been invoked with context c in any trace represented by $\mathrm{FUN}\langle f,c,t'\rangle$. This means that it is safe to redefine how partitioned values are extended to total functions as shown in Fig. 8.10. The only difference between the new and the original definition from Section 8.4 is the second case, where $\bot$ is returned to indicate that the set of traces for the given partition is empty due to the above mentioned property being satisfied.

Function definitions Assume the analysis reaches a function definition node, $r_1 \leftarrow$ function$(\cdots)\{\cdots\}$, while analyzing a function f in context c, and that the function being defined has free variables $\mathtt{x}_1,\ldots,\mathtt{x}_n$ that are declared in f (i.e., as parameters or local variables). Note that f is the function containing the function definition node being analyzed, not the function being defined. Let ℓ denote the location of the newly created closure according to the original transfer function without free variable

[16]Local variables and arguments are stored as properties on activation objects, which are created on each invocation.

$$LC = \textsc{chooseCovering}(\sigma(\mathtt{x}_1),\ldots,\sigma(\mathtt{x}_n))$$

$$\sigma(\mathtt{x}_i) := \begin{cases} \sigma(\mathtt{x}_i) \uplus [\textsc{fun}\langle f,c,t\rangle \mapsto \sigma(\mathtt{x}_i)(t) \mid t \in LC] & \text{if } LC \subseteq \mathrm{dom}(\sigma(\mathtt{x}_i)) \\ \sigma(\mathtt{x}_i) & \text{otherwise} \end{cases}$$

$$\sigma(r_1) := \begin{cases} [t \mapsto (\{\textsc{fun}\langle f,c,t\rangle\} \cup fp, \ell) \mid t \in LC] & \text{if } LC \subseteq \mathrm{dom}(\sigma(\mathtt{x}_i)) \text{ for some } \mathtt{x}_i \\ [\textsc{any} \mapsto (\emptyset, \ell)] & \text{otherwise} \end{cases}$$

Figure 8.11: Introduction of partitioned values at a function definition node $r_1 \leftarrow$ $\mathsf{function}(\cdots)\{\cdots\}$.

partitioning. We now partition both the resulting function value of register r_1 and the values of $\mathtt{x}_1,\ldots,\mathtt{x}_n$ as shown in Fig. 8.11.

First, we use a function $\textsc{chooseCovering}$ that finds a largest covering, denoted *LC*, among the values of $\mathtt{x}_1,\ldots,\mathtt{x}_n$. (If multiple such coverings exist, an arbitrary one is selected among them, as before.)

For each $\mathtt{x}_i$ for $i = 1,\ldots,n$, if the current value of $\mathtt{x}_i$ contains the covering *LC*, we add $\textsc{fun}\langle f,c,t\rangle \mapsto \sigma(\mathtt{x}_i)(t)$ to the value of $\mathtt{x}_i$ for each $t \in LC$. (This evidently respects the meaning of $\textsc{fun}\langle f,c,t\rangle$ tokens informally described in the beginning of the section.) Otherwise, if the current value of $\mathtt{x}_i$ does not contain *LC*, we leave $\mathtt{x}_i$ unmodified.

For the result register r_1, we augment the function location ℓ by the same partition tokens. If at least one free variable has been partitioned (i.e., $LC \subseteq \mathrm{dom}(\sigma(\mathtt{x}_i))$ for some $\mathtt{x}_i$), then for each of the partition tokens $t \in LC$, the value of r_1 becomes the augmented value $(\{\textsc{fun}\langle f,c,t\rangle\} \cup fp, \ell)$ where *fp* is the set of function partitions in the current context c. By augmenting the value using the $\textsc{fun}\langle f,c,t\rangle$ token, the information about the partitioning is available when ℓ is later invoked, which is explained below. (The function partitions *fp* of the current context describe how the current function was declared in an outer scope, so by inheriting those, the partitioning also works for multiple layers of nested functions.) Otherwise, if none of the free variables have been partitioned, register r_1 is assigned the partitioned value $[\textsc{any} \mapsto (\emptyset, \ell)]$, which is equivalent to the original transfer function without free variable partitioning.

Function calls At a function call $r_1 \leftarrow r_2(r_3)$ where $\sigma(r_2)$ is an augmented function value (fp, v) (i.e., *fp* is a set of partition tokens introduced at function definitions and v refers to the set of closures that may be invoked), we use *fp* to augment the context for each callee. (The set of augmented contexts C' contains the *FunctionPartitions* component exactly for this purpose.) Assume for simplicity that v refers to a single closure location so we only have one callee. By augmenting the context, when analyzing the body of the callee we retain the information about the partitions where the callee closure was created, which we can exploit when reading its free variables as explained next.

$$\sigma(r_1) := \begin{cases} [\text{ANY} \mapsto \bigsqcup\{\sigma(\mathtt{x})(t) \mid t \in \text{dom}(\sigma(\mathtt{x})) \cap fp\}] & \text{if } \text{dom}(\sigma(\mathtt{x})) \cap fp \neq \emptyset \\ \sigma(\mathtt{x}) & \text{otherwise} \end{cases}$$

Figure 8.12: Exploiting partitioned values at a read variable node, $r_1 \leftarrow \mathtt{x}$.

Variable reads Fig. 8.12 shows the updated transfer function for read variable nodes $r_1 \leftarrow \mathtt{x}$, where we read a variable `x` in a calling context with function partitions *fp*. The set of function partitions *fp* tells us which partitions the current closure may have been created in. For this reason, if the abstract value of `x` contains partition tokens that are also in *fp*, we can obtain a covering for `x` by considering only those partition tokens. If there is no such partition token, we just read the value of `x` as in the original transfer function.

As an example, assume $\sigma(\mathtt{x}) = [\text{FUN}\langle f,c,t\rangle \mapsto 1, \text{FUN}\langle f,c',t'\rangle \mapsto 2,\ldots]$ and $fp = \{\text{FUN}\langle f,c,t\rangle\}$. The value of `x` tells us that `x` must be a local variable in function f which may have been called in contexts c and context c', and that `x`'s value is 1 or 2, respectively. Since $fp = \{\text{FUN}\langle f,c,t\rangle\}$, we know that the current function is defined inside the lexical scope of f in context c, meaning that the value of `x` must be 1.

Examples To better understand free variable partitioning, we provide two examples. The first example (Fig. 8.13) shows how free variable partitioning can preserve precision when a function is called in multiple contexts, in a way that resembles traditional heap specialization [110]. The second example (Fig. 8.14) shows how free variable partitioning can preserve the precision of free variables partitioned with property name partitioning.

In Fig. 8.13, lines 22–26 define a function `f` that returns a closure, which on invocation returns the argument passed to `f`. Lines 28 and 29 call `f` with the arguments `"foo"` and `"bar"` and store the returned closures in the variables `foo` and `bar`, respectively. Line 31 calls the closure stored in `bar` and asserts that the resulting value is not the string `"foo"`. The two calls to `f` are analyzed in different contexts c and c' (due to the context sensitivity mechanism mentioned in Section 12.3, as `"foo"` and `"bar"` are determinate values). For the invocation `bar()`, the resulting value is the value of the free variable `v` in the closure stored in `bar`. If not using heap specialization, the two concrete activation objects at the two calls to `f` would be modeled by a single abstract object, so the free variable `v` would have the imprecise abstract value AnyString. To reason precisely about the assertion in line 31, the analysis has to distinguish the value of `v` at the two calls. The baseline TAJS analyzer accomplishes this by the use of heap specialization [10], which provides two different abstract activation objects for the calls to `f`, so the two values `"foo"` and `"bar"` are kept separate.

With free variable partitioning we obtain the same degree of precision as with heap specialization in this situation. Since `v` is a free variable in the closure created in line 23, we apply the top cases in the transfer functions shown in Fig. 8.11

```
22 function f(v) {
23   return function g() {
24     return v;
25   }
26 }
27
28 var foo = f("foo");
29 var bar = f("bar");
30
31 assert(bar() != "foo");
```

Figure 8.13: Free variable partitioning example with different contexts.

```
32 var o1 = {x: 1, y: 2};
33 var o2 = {};
34 Object.keys(o1).forEach(
35   function h(p) {
36     var v = o1[p];
37     o2[p] = function j() {
38       return v;
39     }
40   }
41 );
42 assert(o2.y() != 1);
```

Figure 8.14: Free variable partitioning example with partitioned argument.

with $LC = \{\text{ANY}\}$. This means that `v` after the call to `f("foo")` will have the value $[\text{ANY}{\mapsto}\texttt{"foo"},\text{FUN}\langle f,c,\text{ANY}\rangle \mapsto\texttt{"foo"}]$ and the value written to the `foo` variable is $(\{\text{FUN}\langle f,c,\text{ANY}\rangle\},\ell_g)$ where ℓ_g is the location of the closure created in line 23. For the call to `f("bar")`, the value for `v` will similarly be $[\text{ANY}{\mapsto}\texttt{"bar"},\text{FUN}\langle f,c',\text{ANY}\rangle \mapsto\texttt{"bar"}]$ and the value written to `bar` is $(\{\text{FUN}\langle f,c',\text{ANY}\rangle\},\ell_g)$. Note the difference in the context part of the FUN token (c at the `"foo"` call and c' at the `"bar"` call), since the calls to `f` are in those two different contexts. The value of `v` then becomes $[\text{ANY}{\mapsto}\text{AnyString},\text{FUN}\langle f,c,\text{ANY}\rangle \mapsto\texttt{"foo"},\text{FUN}\langle f,c',\text{ANY}\rangle \mapsto\texttt{"bar"}]$, so that the FUN partitions preserve the precise values.

Now when analyzing `bar()`, `bar` has the value $(\{\text{FUN}\langle f,c',\text{ANY}\rangle\},\ell_g)$, which means the calling context to the function `g` is augmented with the set of function partitions $\{\text{FUN}\langle f,c',\text{ANY}\rangle\}$ as described above. When reading the free variable `v` in line 24, we use the first case in the transfer function defined in Fig. 8.12, since $\text{dom}(\sigma(\texttt{x}))\cap fp$ is $\{\text{FUN}\langle f,c',\text{ANY}\rangle\}$. This means that the resulting value from the variable read is the value $[\text{ANY}{\mapsto}\texttt{"bar"}]$, so we obtain the same precision as with heap specialization.

This first example shows how the free variable partitioning mechanism works and how it relates to heap specialization, but it does not demonstrate any precision improvements compared to the existing TAJS analyzer, which does apply heap specialization. The second example, Fig. 8.14, illustrates a simplified version of how free variable partitioning was used in the motivating example in combination with property name partitioning, which leads to a precision improvement of TAJS. Line 32 defines the object `o1` with two properties, and line 33 defines `o2` as an empty object. Lines 34–41 iterate over the properties of `o1`. For each property, it writes a function returning the value of `o1[p]` to the `p` property of `o2`. To prove that the assertion at line 42 always holds, it is critical that the values of `v` are not mixed together in the iterations.

Using property name partitioning at line 36, the value of `v` becomes $[\text{VAL}\langle n,r,\texttt{"x"}\rangle \mapsto 1, \text{VAL}\langle n,r,\texttt{"y"}\rangle \mapsto 2]$ and the value of `p` becomes $[\text{VAL}\langle n,r,\texttt{"x"}\rangle \mapsto \texttt{"x"}, \text{VAL}\langle n,r,\texttt{"y"}\rangle \mapsto \texttt{"y"}]$, where n is the read property node and r is the register storing the property name. (For clarity we ignore the OTHER partition in this example.) When analyzing the closure creation at line 37, we use the top rules in Fig. 8.11 with $LC = \{\text{VAL}\langle n, r, \texttt{"x"}\rangle, \text{VAL}\langle n, r, \texttt{"y"}\rangle\}$. This means that `v` is augmented with the additional partitions $[\text{FUN}\langle \texttt{h}, c, \text{VAL}\langle n, r, \texttt{"x"}\rangle\rangle \mapsto 1, \text{FUN}\langle \texttt{h}, c, \text{VAL}\langle n, r, \texttt{"y"}\rangle\rangle \mapsto 2]$, and the value being written to `o2[p]` is $[\text{VAL}\langle n, r, \texttt{"x"}\rangle \mapsto (\{\text{FUN}\langle \texttt{h}, c, \text{VAL}\langle n, r, \texttt{"x"}\rangle\rangle\}, \ell_j)$, $\text{VAL}\langle n, r, \texttt{"y"}\rangle \mapsto (\{\text{FUN}\langle \texttt{h}, c, \text{VAL}\langle n, r, \texttt{"y"}\rangle\rangle\}, \ell_j)]$. Here, ℓ_j denotes the location of the closure created in line 37. At the dynamic property write, the property name and value to be written share the covering $\{\text{VAL}\langle n, r, \texttt{"x"}\rangle, \text{VAL}\langle n, r, \texttt{"y"}\rangle\}$, meaning that the write happens as described in Fig. 8.8, so that `o2.x` becomes $(\{\text{FUN}\langle \texttt{h}, c, \text{VAL}\langle n, r, \texttt{"x"}\rangle\rangle\}, \ell_j)$ and `o2.y` becomes $(\{\text{FUN}\langle \texttt{h}, c, \text{VAL}\langle n, r, \texttt{"y"}\rangle\rangle\}, \ell_j)$. Now when `o2.y` is called in line 42, the call to `j` is augmented with the the set of function partitions $\{\text{FUN}\langle \texttt{h}, c, \text{VAL}\langle n, r, \texttt{"y"}\rangle\rangle\}$. Therefore when reading the value `v` in line 38, according to Fig. 8.12 we only read the $\text{FUN}\langle \texttt{h}, c, \text{VAL}\langle n, r, \texttt{"y"}\rangle\rangle$ partition. The result of reading `v` is then $[\text{ANY} \mapsto 2]$, so the analysis is precise enough to prove that the assertion at line 42 holds.

8.5.3 Type Partitioning

Value partitioning can also be useful for partitioning values based on their types. Since JavaScript does not have function overloading, it is common to reflectively find the type of an argument, and based on the type run different pieces of code (as in line 3 in Fig. 11.1). This is often done through the use of predicate functions, which are one-parameter functions that return a boolean value. By partitioning the arguments at calls to predicate functions, the analysis becomes able to track the relations between the arguments and the return values, and thereby boost the control sensitivity mechanism (see Section 12.3) at branches that involve such calls. Since the analysis does not know in advance whether a function returns boolean values, we simply perform this partitioning at all function calls with one argument, without considering what values the function may return.

Partition tokens for type partitioning We introduce type partitioning tokens of the form $\text{TYPE}\langle n, r, ty\rangle$, where $n \in N$ is a call node $r_1 \leftarrow r_2(r_3)$, $r \in R$ is the argument register in n (in this case r_3), and $ty \in \mathit{Types}$ using the set of types shown in Fig. 8.6. Such a token represents the set of traces where the type of r is ty at the last occurrence of n. For example, the traces that reach line 7 in Fig. 11.1 are represented by the token $\text{TYPE}\langle n, r, \mathsf{function}\rangle$ where n is the call to `isFunction` in line 6 and r is the argument register of that call node.

Function calls Fig. 8.15 shows an addition to the transfer function for call nodes, $r_1 \leftarrow r_2(r_3)$, to partition the argument value before the call takes place. The first case applies if the argument $\sigma(r_3)$ abstractly represents values of multiple types (i.e., $|\text{TYPES}(\sigma(r_3))| > 1$, where TYPES returns the set of all the types the given abstract value may have). In this case we introduce a partition $\text{TYPE}\langle n, r_3, ty\rangle$ for each $ty \in \text{TYPES}(\sigma(r_3))$, such that the value in that partition is $\text{FILTER}(\sigma(r_3), ty)$, where

$$\sigma(r_3) := \begin{cases} \sigma(r_3) \uplus [\textsc{type}\langle n, r_3, ty\rangle \mapsto \textsc{filter}(\sigma(r_3), ty) \mid ty \in \textsc{types}(\sigma(r_3))] & \\ & \text{if } |\textsc{types}(\sigma(r_3))| > 1 \\ \sigma(r_3) & \text{otherwise} \end{cases}$$

Figure 8.15: Addition to the transfer function for a call node with one argument, $r_1 \leftarrow r_2(r_3)$. FILTER restricts a partitioned value to represent only values that match the given type, and TYPES returns the possible types of the given partitioned value.

```
function isObj(arg) {
  return typeof arg == 'object';
}
if (isObj(x)) { ... } else { ... }
```

Figure 8.16: Type partitioning example.

```
function isObj(arg) {
  if (typeof arg == 'object')
    return true;
  else
    return false;
}
if (isObj(x)) { ... } else { ... }
```

Figure 8.17: Type partitioning example with control dependent relations.

FILTER restricts $\sigma(r_3)$ to only represent values of type *ty*. Since all the possible types are represented, the new partitions together form a covering.

Recall that a TYPE$\langle n, r, ty\rangle$ token only represents information about the last occurrence of n in a given trace. To ensure this property we always remove all existing TYPE$\langle n, _, _\rangle$ tokens from the abstract state immediately before applying the modified transfer function for call node n.

Example As an example consider the code in Fig. 8.16, and assume `x` has the abstract value `fun1`|`obj2` (representing either the function `fun1` or the object `obj2`). Without type partitioning, the result of analyzing the `isObj(x)` call is the abstract value AnyBool (representing true or false), so both branches are analyzed with `x` being `fun1`|`obj2`; however, in a concrete execution, `fun1` will never flow to the 'true' branch, and `obj2` will never flow to the 'false' branch.

By using type partitioning, we partition `x` before calling the predicate function. In this example let n be the call node and let r be its argument register. Then `x` becomes $[\textsc{type}\langle n, r, \text{function}\rangle \mapsto \texttt{fun1}, \textsc{type}\langle n, r, \text{object}\rangle \mapsto \texttt{obj2}]$. Now when analyzing the body of `isObj`, the expression `typeof arg == 'object'` evaluates to the partitioned value $[\textsc{type}\langle n, r, \text{function}\rangle \mapsto \texttt{false}, \textsc{type}\langle n, r, \text{object}\rangle \mapsto \texttt{true}]$. When reaching the `if` branch, control sensitivity ensures that only the object partition flows to the 'true' branch (i.e., `x`'s value becomes $[\textsc{type}\langle n, r, \text{function}\rangle \mapsto \bot, \textsc{type}\langle n, r, \text{object}\rangle \mapsto \texttt{obj2}]$ in that branch), and only the function partition flows to the 'false' branch (i.e., `x`'s value becomes $[\textsc{type}\langle n, r, \text{function}\rangle \mapsto \texttt{fun1}, \textsc{type}\langle n, r, \text{object}\rangle \mapsto \bot]$ in that branch).

$$\mathit{State}' = \mathit{State} \times \mathscr{P}(T) \times \mathscr{P}(T)$$

Figure 8.18: Abstract states updated to keep track of dead and live partitions.

$$\sigma(r_1) := \begin{cases} [t \mapsto c \mid t \in \textsc{livePartitions}(\sigma)] \uplus [t \mapsto \bot \mid t \in \textsc{deadPartitions}(\sigma)] \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } \textsc{deadPartitions}(\sigma) \neq \emptyset \\ [\textsc{any} \mapsto c] \qquad\qquad\qquad\qquad\qquad \text{otherwise} \end{cases}$$

Figure 8.19: Updated transfer function for constant nodes, $r_1 \leftarrow c$, for improved type partitioning.

Control dependent relations Predicate functions are sometimes implemented with control dependent relations between the argument and the result, as in the example in Fig. 8.17. The example is contrived but it is not uncommon in predicate functions that the result values appear as the literals `true` or `false` in branches. With the type partitioning mechanism described above, the returned values will not be partitioned in this situation, since the partitions in `arg` do not propagate to the values `true` and `false`.

To mitigate this issue, we augment the abstract states as shown in Fig. 8.18 to keep track of partitions that must be dead or may be live (represented by the two $\mathscr{P}(T)$ components, respectively). A partition is *dead* if the set of traces it represents is empty, and it is live otherwise. (We only keep track of the live partitions in coverings where there are any dead partitions.) Since the branch condition `typeof arg == 'object'` is analyzed with a partitioned value for `arg`, by control sensitivity we know that the only traces that can reach the 'true' branch are those represented by the object partition, so we record that $\textsc{type}\langle n, r, \mathsf{object}\rangle$ is live and $\textsc{type}\langle n, r, \mathsf{function}\rangle$ is dead in that branch, and conversely in the other branch. To exploit this information, we also update the transfer function for constants, $r_1 \leftarrow c$, as shown in Fig. 8.19. Basically, it assigns $\bot$ to all dead partitions and the constant c to all live partitions. If there are no dead partitions, it behaves as usual, where the constant is written to the $\textsc{any}$ partition. When the analysis reaches `true` (line 49), we obtain the partitioned value $[\textsc{type}\langle n, r, \mathsf{function}\rangle \mapsto \bot, \textsc{type}\langle n, r, \mathsf{object}\rangle \mapsto \texttt{true}]$, and similarly when analyzing `false` (line 51) we get $[\textsc{type}\langle n, r, \mathsf{function}\rangle \mapsto \texttt{false}, \textsc{type}\langle n, r, \mathsf{object}\rangle \mapsto \bot]$. The join of these two values is $[\textsc{type}\langle n, r, \mathsf{function}\rangle \mapsto \texttt{false}, \textsc{type}\langle n, r, \mathsf{object}\rangle \mapsto \texttt{true}]$, which becomes the result of `isObj(x)`. Due to the control sensitivity mechanism, only `obj2` then flows to the 'true' branch, and only `fun1` flows to the 'false' branch in line 53.

8.6 Evaluation

We have implemented the value partitioning framework (Section 8.4) and the three instantiations (Section 8.5) on top of TAJS v0.24. Implementing the general framework in TAJS required 900 lines of code, however most of this is boilerplate code for lifting operations on ordinary abstract values to also work on partitioned values. With the general framework in place, instantiations are easy to implement: property name partitioning (Section 8.5.1), free variable partitioning (Section 8.5.2), and type partitioning (Section 8.5.3) required only around 230, 250, and 60 lines of code, respectively. We disable TAJS's `for-in` specialization technique, since it is subsumed by property name partitioning.[17] We refer to our new analysis tool as TAJS$_{\text{VALPAR}}$.[18] Using this tool we evaluate our techniques by answering the following research questions:

RQ1 How does TAJS$_{\text{VALPAR}}$ compare to existing state-of-the-art analyses for JavaScript?

RQ2 What are the effects of the three different instantiations of value partitioning?

All our experiments are conducted on an Ubuntu machine with a 2.6 GHz Intel Xeon E5-2697A CPU running a JVM with 10 GB RAM.

8.6.1 RQ1: Comparison with State-Of-The-Art Analyses

We start by comparing TAJS$_{\text{VALPAR}}$ against the current state-of-the-art analyses for JavaScript: the baseline TAJS analyzer with static determinacy [10], TAJS$_{\text{VR}}$ [137] with demand-driven value refinement, and the CompAbs analyzer [85] based on the SAFE analyzer [91]. We use the same benchmarks as those used in the evaluation of TAJS$_{\text{VR}}$, which is the most recent related work.

Micro benchmarks We first evaluate TAJS$_{\text{VALPAR}}$ against a small collection of micro benchmarks that capture some of the main challenges that appear in analysis of modern JavaScript libraries and are used in previous work [85, 137]. The benchmarks all contain dynamic read/write pairs that are variations of the pattern shown in the introduction and the motivating example. The results of the comparison are shown in Table 8.1. For these benchmarks, a test *succeeds* if it avoids mixing together properties in the dynamic read/write pairs.

The first two examples, CF and CG, are loops where the static analyses have enough information to be able to unroll all the iterations and thereby analyze the read/write patterns with precise property names. For CF, property name partitioning in TAJS$_{\text{VALPAR}}$ gives the same degree of precision without loop unrolling.

[17]The motivation for introducing `for-in` specialization in [10] was to reason about correlated read-/write pairs inside `for-in` loops. This relational information is now provided by property name partitioning.

[18]TAJS$_{\text{VALPAR}}$: **TAJS** with **Val**ue **Par**titioning

Benchmark	TAJS	CompAbs	TAJS$_{\text{VR}}$	TAJS$_{\text{VALPAR}}$
CF	✓	✓	✓	✓
CG	✓	✓	✓	✓
AF	✗	✓	✓	✓
AG	✗	✓	✓	✓
M1	✗	✗	✓	✓
M2	✗	✗	✓	✓
M3	✗	✗	✓	✓

Table 8.1: Micro-benchmarks that check how state-of-the-art analyses handle various dynamic read/write pairs that represent typical challenges in JavaScript library code. A ✗ indicates that the analysis mixes together the properties of the object being manipulated, while a ✓ indicates that it is sufficiently precise to keep them distinct. The CF, CG, AF, and AG benchmarks are drawn directly from [85], while M1, M2, and M3 are drawn directly from [137].

AF and AG are loops where the static analyses are incapable of obtaining a precise value for the property name used in the dynamic read/write pairs. TAJS fails to analyze these, but CompAbs detects the pattern syntactically and therefore applies trace partitioning to analyze the code precisely. TAJS$_{\text{VR}}$ also succeeds on these tests, because its backwards abstract interpreter is capable of providing the necessary relational information. In comparison, TAJS$_{\text{VALPAR}}$ can reason about the relational information on its own.

Both TAJS and CompAbs fail on the last three tests (M1, M2, and M3). CompAbs fails on M1 and M3 because it does not apply partitioning due to the fragility of syntactic patterns, and it fails on M2 because the partitioning does not provide the necessary precision about free variables. Again, TAJS$_{\text{VR}}$ can analyze them all, since the backwards abstract interpreter is powerful enough to reason about all the cases, whereas TAJS$_{\text{VALPAR}}$ successfully preserves the relational properties by the use of value partitioning.

These results demonstrate that for these benchmarks, TAJS$_{\text{VALPAR}}$ is capable of providing comparable precision to the demand-driven value refinement technique without the need for a complicated backwards analysis, and provides better precision than the other analyses.

Library benchmarks The next set of benchmarks is taken from the evaluation of TAJS$_{\text{VR}}$ and consists of small test cases for popular real-world libraries. The libraries include the widely used functional utility library Underscore (which has more than 20 000 dependents in npm) v1.8.3 with 1 548 LoC and the most depended-upon package Lodash (more than 115 000 dependents). We analyze both Lodash3 (v3.0.0, 10 785 LoC) and Lodash4 (v4.17.10, 17 105 LoC), since their code bases are substantially different and therefore pose distinct challenges for static analysis.

Benchmark group		**TAJS**		**CompAbs**		**TAJS$_{\text{VR}}$**		**TAJS$_{\text{VALPAR}}$**	
Underscore	(182 tests)	0	(-)	0	(-)	173	(2.9s)	173	(2.7s)
Lodash3	(176 tests)	7	(2.4s)	0	(-)	172	(5.5s)	173	(5.3s)
Lodash4	(306 tests)	0	(-)	0	(-)	266	(24.7s)	289	(26.3s)
Prototype	(6 tests)	0	(-)	2	(23.1s)	5	(97.7s)	5	(34.1s)
Scriptaculous	(1 tests)	0	(-)	1	(62.0s)	1	(236.9s)	1	(55.2s)
jQuery	(71 tests)	3	(16.0s)	0	(-)	3	(13.5s)	3	(20.4s)

Table 8.2: Analysis results for real-world benchmarks (from [137]). For each group of benchmarks and for each of the four analyzers, we show the number of tests that are analyzed successfully and (in parentheses) the average analysis time per successful test.

The other libraries, Prototype v1.7.2, Scriptaculous v1.9.0, and jQuery v1.10,[19] are popular libraries for client-side web programming.

The analysis results are shown in Table 8.2. We classify an analysis of a benchmark as successful if it terminates within 5 minutes and the analysis result to our knowledge is sound. In particular, an analysis run is considered a failure if the analysis result does not have dataflow to the ordinary exit of the program. (All the tests pass in normal execution, so an analysis result is obviously unsound if there is no dataflow to the ordinary exit.) To increase confidence in the soundness of the analysis results for TAJS$_{\text{VALPAR}}$, we apply thorough soundness testing as described at the end of this section. Increasing the time budget does not help for these benchmarks: as reported previously for JavaScript analysis tools, critical precision losses tend to cause a proliferation of spurious dataflow that drastically increases analysis time and renders the analysis results useless [70, 85, 115, 137].

The results for TAJS$_{\text{VALPAR}}$ are comparable to those of TAJS$_{\text{VR}}$, which outperforms the other analyzers. TAJS$_{\text{VALPAR}}$ succeeds in analyzing all the benchmarks that TAJS$_{\text{VR}}$ can handle, plus 24 more (one Lodash3 test and 23 Lodash4 tests). Note the substantial improvement for the Lodash4 tests: the number of Lodash4 tests that are not analyzed successfully is reduced from 40 to 17. None of the analyzers do well on the jQuery benchmarks; a preliminary manual study shows that the reasons are unrelated to relational analysis. The results are as expected, since property name partitioning and free variable partitioning are alternative techniques to provide the relational information that TAJS$_{\text{VR}}$ obtains from its demand-driven value refinement. Furthermore, value partitioning is triggered more often during the analysis, which means that the precision improvements are not limited to the few critical cases where value refinement is triggered. On top of this, type partitioning provides some additional precision beyond the capabilities of TAJS$_{\text{VR}}$.

Comparing the performance between TAJS$_{\text{VALPAR}}$ and TAJS$_{\text{VR}}$, the most significant differences are for the Prototype and Scriptaculous benchmarks. TAJS$_{\text{VALPAR}}$ is

[19]This is the version of jQuery used in [10]. Note that [85] used the older v1.4.4.

around 3–4 times faster than TAJS$_{\text{VR}}$, which is mainly because property name partitioning makes the `for-in` specialization technique in TAJS obsolete. For Underscore and Lodash3, TAJS$_{\text{VALPAR}}$ is slightly faster than TAJS$_{\text{VR}}$. This is encouraging, because analyzing dynamic property writes as the one in line 7 in Fig. 11.1 is more expensive in TAJS$_{\text{VALPAR}}$ than in TAJS$_{\text{VR}}$. In TAJS$_{\text{VR}}$ such an operation is handled as a single imprecise write (since the precision is recovered on demand), whereas TAJS$_{\text{VALPAR}}$ performs the write for each property that is copied. To soundly handle setters, all the writes happen in different states that are subsequently joined together, which causes TAJS$_{\text{VALPAR}}$ to spend some extra time at such writes. Since the analysis time is nevertheless similar, we can conclude that value partitioning is cheaper for analyzing other parts of the libraries. For Lodash4 and jQuery, TAJS$_{\text{VR}}$ is slightly faster than TAJS$_{\text{VALPAR}}$. For Lodash4, the main reason is the handling of dynamic property writes, and for the jQuery benchmarks, type partitioning adds little performance overhead as seen in Table 8.3.

Precision Previous work [10, 115, 137] established that type analysis and call-graph construction are useful metrics for measuring the precision of an analysis for JavaScript, and therefore we use these metrics to evaluate the analysis precision of TAJS$_{\text{VALPAR}}$. All locations are treated context-sensitively in these measurements, meaning that we count the same location once for each reachable context. We count the number of possible types for the resulting value in each variable or property read and find that in 99.19% of the reads, a single unique type is read, with the average number of types being 1.02. For measuring precision of the call-graph construction, we measure the number of call-sites with unique callees, and find this number to be 99.95% of all call-sites. These numbers show that when the analysis succeeds, it does so with very high precision.

Soundness Formally proving soundness of the three variants of value partitioning is out of scope of this paper, however, we will informally justify that the general approach is sound. Since general trace partitioning is known to be sound, it suffices to argue that the precision gained by value partitioning is equivalent to that obtained through trace partitioning. The key reason why this holds for property name partitioning and type partitioning is that the partition tokens represent the last occurrence of some node, meaning that if two values share partitions, they represent information about the same execution traces. This means that we could (if ignoring performance) instead have applied traditional trace partitioning, with exactly the same partition tokens and at the same nodes, resulting in the same precision. (For further discussion about the connection between value partitioning and trace partitioning, see Section 12.8.) Similarly for free variable partitioning, since the partitions are only allowed on activation objects, the precision is never higher than what would be obtained using heap specialization (where each partition would be represented by a distinct abstract activation object), and therefore soundness follows from soundness of heap specialization.

Benchmark group		**None**		**P**		**P + FV**		**P + FV + T**	
Underscore	(182 tests)	0	(-)	149	(2.0s)	173	(2.5s)	173	(2.7s)
Lodash3	(176 tests)	7	(2.4s)	167	(4.7s)	173	(5.1s)	173	(5.3s)
Lodash4	(306 tests)	0	(-)	268	(16.8s)	274	(27.7s)	289	(26.3s)
Prototype	(6 tests)	0	(-)	0	(-)	5	(32.7s)	5	(34.1s)
Scriptaculous	(1 tests)	0	(-)	0	(-)	1	(53.1s)	1	(55.2s)
jQuery	(71 tests)	3	(16.0s)	3	(15.2s)	3	(16.5s)	3	(20.4s)

Table 8.3: Analysis results for real-world benchmarks (from [137]) using different instantiations of value partitioning. "None" is without value partitioning, "P" is with property name partitioning, "P + FV" is with property name and free variable partitioning, and "F + PV + T" is with property name, free variable, and type partitioning.

Furthermore, to increase confidence in the soundness of our implementation, all the TAJS$_{\text{VALPAR}}$ results have been thoroughly soundness tested [11]. This means that the analysis results overapproximate all the dataflow facts that have been observed during concrete executions of the analyzed benchmarks. For every variable and property read observed concretely, we have checked that the concrete value is in the concretization of the corresponding abstract value in the analysis results, and similarly for property writes and function calls. All our benchmarks except one pass in total more than 7.6 million soundness tests. The one benchmark that fails is a Lodash4 test, which uses ES6 iterators in combination with `Array.from`, which is not fully supported in the latest version of TAJS and is unrelated to the use of value partitioning.

8.6.2 RQ2: Effects of the Three Instantiations

We now investigate how much each of the three uses of value partitioning contributes to the results reported in the previous section. The results from running our analysis with only some instantiations enabled can be seen in Table 8.3. The column "P" is with only property name partitioning enabled; we see that it is sufficient for analyzing many of the Underscore and Lodash test cases, but not for any of the Prototype or Scriptaculous test cases. (Without property name partitioning but with the other two instantiations enabled, the analysis is not able to analyze more benchmarks than TAJS.) The column "P + FV" uses both property name partitioning and free variable partitioning. Also enabling free variable partitioning makes the analysis capable of analyzing many additional benchmarks: more Underscore and Lodash test cases, as well as some Prototype and Scriptaculous test cases. Compared to only property name partitioning, the analysis times are higher (for the reason discussed above regarding additional state joins). The last column "P + FV + T" is with all instantiations enabled and therefore contains the same numbers as shown in Table 8.2. We see that type partitioning enables the analysis of 15 additional Lodash4 tests, without significantly increasing the analysis time.

We conclude that all three instantiations contribute to the results, where property name partitioning is the most important one, followed by free variable partitioning and then type partitioning. (TAJS already performs filtering at branches, as mentioned in Section 12.3; without that feature the effect of type partitioning would likely be larger.)

8.7 Related Work

Trace partitioning Value partitioning can be viewed as a variant of trace partitioning [122] as explained in Sections 8.4, 12.1 and 12.2, but there are some important differences. Changing the original abstract domain in Section 8.4 to support traditional trace partitioning can be done by replacing $L \to \mathit{State}$ by $L \to T \to \mathit{State}$, so that an abstract state is maintained for each partition, at every location. Thus, different locations can partition the abstract states differently. Value partitioning instead has only one abstract state per location but partitions the individual abstract values, which adds an additional degree of flexibility: different parts of each abstract state can be partitioned differently. In particular, for the large parts of the states where we are not interested in relational information, we can use the $\{\text{ANY}\}$ partitioning,[20] while for the important registers and object properties, we can have nontrivial partitions. With traditional trace partitioning, the normal transfer functions are applied for each partition, which causes redundant computations because of the similarities between the different partitions[21]; with value partitioning, we only pay a price for partitioning at operations that involve abstract values with nontrivial partitions. This is the main reason for the low overhead of the technique.

Another difference is that the partition tokens in traditional trace partitioning are actually lists of "directives" (the directives language used by Rival and Mauborgne [122] is similar to our language of tokens in Fig. 8.6), which can lead to a combinatorial explosion. By partitioning at the level of values and allowing multiple coverings in each partitioned value, we avoid the need to maintain such combinations.

Relational analysis Traditional techniques for achieving relational analysis, as exemplified by the octagon abstract domain [103], focus on numeric relations, such as, linear inequalities. To reduce the cost of this approach, a syntactic pre-analysis called variable packing is typically used for partitioning the set of program variables, and one octagon is then used for each pack instead of tracking all possible combinations of inequalities. This kind of partitioning is reminiscent of value partitioning, but with the important difference that variable packing and octagons operate on sets of program variables whereas value partitioning works on individual abstract values. In our work with analysis of JavaScript libraries, we have not encountered a critical need for tracking numeric relations.

The well-known analyzer Astrée [17] applies not only trace partitioning and octagons, but also a decision tree abstract domain that is used for tracking relations

[20] In our experiments, 99.4% of all abstract values have the trivial $\{\text{ANY}\}$ partitioning.

[21] This was shown experimentally in the work on TAJS$_{\text{VR}}$ [137, Section 7.1].

between booleans and numerical variables that affect control flow. That technique has some similarities with our type partitioning mechanism but relies on variable packing to avoid combinatorial explosions, whereas type partitioning uses the more lightweight value partitioning technique in combination with the existing control sensitivity mechanism of TAJS.

The main purpose of value partitioning is to be able to reason about relations between different parts of the abstract state (i.e., program variables and registers) at the various program points. Some literature uses the term relational analysis with a slightly different meaning: to relate information across program points, typically relations between the entry and exits of functions [21, 34].

Static analysis for JavaScript Through the last decade, several static analyzers for JavaScript have been developed, including WALA [127, 133, 144], SAFE [91, 115], JSAI [80], and TAJS [10, 70, 137]. Although we focus on TAJS, the designs of SAFE and JSAI are reasonably similar, so we believe value partitioning could also be incorporated into those tools with little effort.

As discussed in the introduction, much work has been put into improving precision of the analyses through different kinds of context sensitivity and elaborate abstract domains. The techniques include parameter sensitivity and heap context sensitivity [10], loop unrolling [115], and syntactic patterns for detecting correlated read/write pairs and guiding context sensitivity [133]. Other works have explored more expressive string abstractions to reason more precisely about property names in dynamic property accesses [8, 96, 116]. Our abstract domain extension for value partitioning has few assumptions about the underlying abstract domain, so most of these techniques can be combined with value partitioning.

Despite such precision improvement techniques, imprecision is inevitable, and only a few techniques have been designed to handle dynamic property accesses with imprecise property names, most importantly, CompAbs-style trace partitioning [85] and demand-driven value refinement [137]. Previous work has shown that demand-driven value refinement enables analysis of many more challenging benchmarks than CompAbs-style trace partitioning (as also discussed in Section 12.7), and that the trace partitioning approach causes a large amount of redundant computation [137, Section 7.1]. The fundamental drawback of demand-driven value refinement is that it requires a separate backwards abstract interpreter for not only the entire JavaScript language but also the standard library. The backwards abstract interpreter of TAJS$_{\mathrm{VR}}$ is not simply the dual of TAJS but works goal-directed and with its own abstract domain based on intuitionistic separation logic. In contrast, value partitioning directly leverages the existing forward analyzer and thereby supports both the JavaScript language and the standard library essentially for free, which makes this approach substantially easier to develop and maintain. Furthermore, value partitioning is more general (for example, it enables type partitioning), and the three instantiations we have presented lead to better precision (for the Lodash4 tests).

The HOO (heap with open objects) abstract domain [37] is a relational abstraction that is designed to reason more precisely about abstract objects whose properties cannot be known statically. That approach is highly expressive but not scalable to real-world JavaScript libraries as those considered in Section 12.7.

8.8 Conclusion

We have presented value partitioning, a static analysis technique for reasoning about relational properties. It is a lightweight alternative to traditional trace partitioning techniques that allows relational information to be incorporated into the abstract values instead of requiring separate abstract states for the partitions. We have proposed three instantiations of value partitioning in JavaScript analysis: property name partitioning, free variable partitioning, and type partitioning, which enable precise reasoning for dynamic read/write pairs, free variables, and predicate functions, respectively.

The experimental results show that extending the TAJS analyzer with the three variants of value partitioning enables precise and efficient analysis of complex JavaScript libraries including Lodash and Underscore, thereby outperforming a state-of-the-art technique that relies on trace partitioning and without requiring a complicated backwards analysis. For the libraries considered in this study, property name partitioning has the largest effect among the proposed variants.

An interesting direction for future research is to investigate whether some of the traditional context sensitivity strategies used in TAJS and other JavaScript analyzers can be reformulated as new value partitioning instantiations, to make analysis faster while retaining precision.

Chapter 9

NODEST: Feedback-Driven Static Analysis of Node.js Applications

By Benjamin Barslev Nielsen (Oracle Labs, Australia and Aarhus University, Denmark), Behnaz Hassanshahi (Oracle Labs, Australia) and François Gauthier (Oracle Labs, Australia). Published in the Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), August 2019.

Abstract

Node.js provides the ability to write JavaScript programs for the server-side and has become a popular language for developing web applications. Node.js allows direct access to the underlying filesystem, operating system resources, and databases, but does not provide any security mechanism such as sandboxing of untrusted code, and injection vulnerabilities are now commonly reported in Node.js modules. Existing static dataflow analysis techniques do not scale to Node.js applications to find injection vulnerabilities because small Node.js web applications typically depend on many third-party modules. We present a new feedback-driven static analysis that scales well to detect injection vulnerabilities in Node.js applications. The key idea behind our new technique is that not all third-party modules need to be analyzed to detect an injection vulnerability. Results of running our analysis, NODEST, on real-world Node.js applications show that the technique scales to large applications and finds previously known as well as new vulnerabilities. In particular, NODEST finds 63 true positive taint flows in a set of our benchmarks, whereas a state-of-the-art static analysis reports 3 only. Moreover, our analysis scales to Express, the most popular Node.js web framework, and reports non-trivial injection vulnerabilities.

9.1 Introduction

Node.js is a platform to run JavaScript on the server-side. Node.js applications consist of modules and are managed by the NPM package manager. In contrast to client-side JavaScript applications that run in browsers, Node.js allows direct access to filesystem, operating system resources and databases, but does not provide any security mechanism such as sandboxing of untrusted code.

To detect injection vulnerabilities, tracking user-controlled in- puts is needed. This type of analysis is known as taint analysis, which is a popular analysis to detect flows of data from untrusted sources to security-sensitive sinks. Due to the dynamic nature of JavaScript, existing dataflow analyses for statically-typed languages, such as Java, are not suitable in practice. For example, it is possible to construct an over-approximate callgraph in a pre-analysis step to refine and scale a more precise and expensive analysis for Java [61, 130]. However, constructing callgraphs for JavaScript programs requires handling dynamic dispatches, which requires type inference, which itself requires a precise callgraph. Abstract interpretation techniques that are designed to more closely model the program semantics compared to previous dataflow techniques have shown to be more precise and suitable for analyzing dynamic languages such as JavaScript [70, 80, 91]. A taint analysis can be specified as a client analysis in an abstract interpretation framework to detect vulnerable dataflows.

Abstract interpretation is a static analysis technique that computes a sound overapproximation of all possible program behaviors [31]. Statically computing all possible behaviors of a program using concrete language semantics is known to be undecidable. Therefore, abstract interpretation frameworks overapproximate concrete values and operations with abstract values and operations. Existing state-of-the-art static analysis techniques for JavaScript are conducted as a whole-program analysis, precisely analyzing all reachable code from the main entry point of the program [70, 80, 91]. These analysis techniques often fail to scale because of the highly dynamic nature of the JavaScript language. This problem is exacerbated in Node.js applications because they often consist of many NPM modules. For instance, many of the web-based Node.js applications are built upon libraries such as Express [3], which relies on many other NPM modules, making a single dependency transitively depend on about **30** modules; with an estimated **12,000** lines of JavaScript code. The standard whole-program analysis techniques often get stuck in a hard-to-analyze NPM module in the early stages of the analysis and are not able to produce any useful results.

In abstract interpretation, fine-tuning analysis precision and scalability manually is infeasible. The key idea in this paper is to automatically determine which modules in a Node.js application can be approximated[1] with a wide abstraction to improve scalability while preserving precision. By precisely analyzing a set of modules while approximating the rest, our analysis is able to scale to Node.js applications and detect injection vulnerabilities.

[1]Throughout the paper, approximating a module refers to approximating the return value of `require("m")`, which imports a module named `"m"`, without analyzing `"m"`.

Our analysis tool, NODEST, performs feedback-driven static analysis that is carried out through several iterations. It uses TAJS [70], an abstract interpretation framework for JavaScript, as the underlying analysis in each iteration. At the start of the process, the set of modules that need to be analyzed precisely includes all modules in the Node.js application and excludes all third-party modules. During each iteration, it applies heuristics to determine any new modules that need to be added to this set, or existing modules that need to be deleted from the set, and uses this information as feedback to the next iteration. Third-party modules that are not in this set are not analyzed (i.e., they are approximated) and their side effects are ignored.

We evaluated the effectiveness of our technique using benchmarks in [52, 78, 135], and additional real-world Node.js applications. We also compared our technique with the whole-program analysis in TAJS [10], to understand if NODEST helps scaling the analysis to find injection vulnerabilities in real-world applications that were not analyzable before. Our results show that NODEST scales to those applications and finds not only the previously known vulnerabilities, but also previously unknown zero-day vulnerabilities. Moreover, it achieves high precision and reports few false positives.

In summary, this paper makes the following contributions:

- We present a feedback-driven static analysis that scales to real-world Node.js applications (Section 9.4).
- We extend our feedback-driven analysis with a static taint analysis, which enables us to find non-trivial injection vulnerabilities in Node.js applications (Section 9.5).
- We evaluate our feedback-driven static taint analysis on real-world Node.js applications and report injection vulnerabilities that would otherwise be missed by a whole-program taint analysis. Moreover, we report new vulnerabilities that are not reported by existing dynamic analyses (Section 12.7).

9.2 Motivating Example

To better understand the challenges explained in Section 9.1, consider the code-snippet in Listing 9.1. This example shows a simplified Node.js application based on Express [3] that is vulnerable to a NoSQL injection attack. NoSQL, is a common term for nonrelational databases, in which queries and data are represented in JavaScript Object Notation (JSON) format. The purpose of the code is to provide information of a patient by specifying the phone number of the patient. In this example, the attacker can craft a JSON object (instead of a phone number) and send it as input through an HTTP request such that the query at line 62 satisfies all patients, instead of only a patient with a specific phone number. The attacker can thereby access the records of all patients.

```
54 var http = require("http");
55 var yaml = require("js-yaml");
56 var mongo = require("mongodb");
57 var express = require("express");
58 var app = express();
59 ...
60 app.get("/patients", function (req, res) {
61     ...
62     q = {"Mobile": req.query.val};
63     ...
64     mongo.collection.find(q, {}, function (e, r) {...});
65 });
```

Listing 9.1: Excerpt of Ankimedrec, a vulnerable Node.js application that whole program analysis fails to analyze.

The application starts by importing built-in and third-party modules between lines 54 and 57. Lines 57 and 58 instantiate the Express framework, a web framework that provides HTTP utility methods and middleware to build Node.js web applications.

When an HTTP request (`req`) comes in, it is passed to the application from Express at line 60. HTTP requests can be controlled by the attacker, hence `req` is a taint source[2]. The `query.val` property of `req` is used at line 64 through `q`, to query the MongoDB NoSQL database[3] using the `mongodb` database driver, which is imported at line 56. This application is vulnerable because an attacker-controllable (tainted) value is directly passed to `mongo.collection.find` as an argument, allowing an attacker to access sensitive data of patients with no restrictions.

To analyze this program and find the vulnerable taint flow, a whole-program abstract interpretation, such as TAJS [70], can be used, analyzing all the imported modules precisely based on the designed abstract domain and abstract operations until it reaches a fixpoint. TAJS times out while analyzing `js-yaml` at line 55. Note, however, that it is not necessary to analyze this module precisely to find the NoSQL injection at line 64. On the other hand, if `express` at line 57 is not analyzed precisely, the analysis will not be able to reason about the taint flow from an incoming HTTP request to the sink, hence failing to find the NoSQL injection vulnerability. In Section 9.5, we revisit this example and show how our feedback-driven analysis is able to scale and find the taint flow by approximating modules such as `js-yaml` while precisely analyzing modules such as `express`.

9.3 Background: static analysis in TAJS

Our work is based on TAJS, an abstract interpretation framework for JavaScript. We extend several components of the original whole-program analysis in TAJS to adapt it to our proposed feedback-driven approach. The analysis in TAJS is based on the monotone framework [76] and uses a fixpoint solver that depends on a worklist

[2]The taint source location is the allocation site for a `req` object in the `http` module.

[3]To simplify the example, we have not included the database driver setup steps.

$$\begin{aligned}
P &: \textit{property names} \\
L &: \textit{object addresses} \\
N &: \textit{nodes} \\
C &: \textit{contexts} \\
\textit{AnalysisLattice} &= N \times C \to \textit{State} \\
\textit{State} &= L \to \textit{Obj} \\
\textit{Obj} &= P \to \textit{Value} \\
\textit{Value} &= \textit{Undef} \times \textit{Null} \times \textit{Bool} \times \textit{Num} \times \textit{String} \times \mathscr{P}(L)
\end{aligned}$$

Figure 9.1: Parts of the basic abstract domain in TAJS.

algorithm. The program being analyzed is represented as a control flow graph. The abstract domain used by the analysis simulates the ECMAScript specification and, in high level, provides a callgraph and an abstract state for each context and flow graph node.

Fig. 9.1 shows the simplified definitions of the abstract domain in TAJS that are useful for understanding the new extensions proposed by our approach. *AnalysisLattice* maps node and context pairs to abstract states and the fixpoint solver needs to reach a fixpoint in this lattice. An abstract state maps *object addresses* to abstract objects that are maps from *property names* to abstract values. Abstract values are modeled by the lattice *Value*. The details of each kind of value is discussed in detail in [70].

TAJS performs static analysis using the worklist algorithm in Algorithm 2. The worklist algorithm iterates over node and context pairs until a fixed abstract state is found for each node and context pair. The analysis starts by adding the initial node and context to the worklist. An element is removed from the worklist and analyzed until there are no more elements left. After analyzing the element, the current state propagates to its successors. If the state at the successor changes by this propagation (`propagate(`(n',c)`)` is true), we add the successor to the worklist.

Modelling the module loader in TAJS The original module loader in TAJS mimics the require mechanism in Node.js [5]. Given a module name and the location from which `require` is called, it finds the first matching file following a precedence order specified by the require mechanism and runs the abstract interpretation analysis in TAJS to analyze it precisely.

9.4 Feedback-driven Analysis

In a standard whole-program analysis such as TAJS [70], the whole program is analyzed with the same level of precision across all modules. However, the analysis

Algorithm 2 Worklist algorithm in TAJS

```
1:  add the initial node and context $(n,c)$ to the worklist
2:  **while** worklist not empty **do**
3:      remove node and context $(n,c)$ from worklist
4:      analyze $(n,c)$
5:      **for** all successors $n'$ of $n$ **do**
6:          **if** propagate($(n',c)$) **then**
7:              Add $(n',c)$ to worklist
8:          **end if**
9:      **end for**
10: **end while**
```

can be tuned for certain modules that are more critical. To automatically determine which modules can be approximated with a wide abstraction to improve scalability while preserving precision, we define MS_P as a set of modules that are analyzed precisely, and MS_B as a blacklist of modules that are not allowed to be added to MS_P (e.g., modules that are hard to analyze, which can be known a priori or during the feedback-driven analysis). Both MS_P and MS_B can initially be specified by the user and they should be disjoint sets. By default MS_P and MS_B are empty sets. If MS_P contains all modules used by the application, the feedback-driven analysis will be equivalent to running the normal whole-program static analysis, and if MS_P is empty, we do not analyze any third-party modules. Determining the right MS_P and MS_B sets manually can be difficult. Therefore, we design a feedback-driven analysis to automatically update these sets.

Algorithm 3 Feedback-driven Analysis

```
1: Inputs: $MS_P$ and $MS_B$
2: $Results = NULL$
3: **while** $hasChanged(MS_P)$ **do**
4:     $Results \leftarrow$ EXTENDEDTAJS($MS_P$)
5:     $MS_P$, $MS_B$ $\leftarrow$PROCESSANALYSISRESULTS($Results$,$MS_P$,$MS_B$)
6: **end while**
7: $report(Results.TaintFlows)$
```

Algorithm 3 shows the main feedback loop of our analysis. This algorithm takes MS_P and MS_B as inputs and reports taint flows. The loop continues until MS_P reaches a fixpoint. At each iteration, we run an extended version of the standard analysis in TAJS[4]. We extend the module loader to load only modules that belong to MS_P. At the end of each iteration, we process the analysis results by running the algorithm shown in Fig. 9.3 for each third-party module in the application dependency tree

[4]Feedback-driven analysis is applicable for other abstract interpretation frameworks such as SAFE [91].

$$\begin{aligned}
M &: \mathit{Module\ name}\\
SL &: \mathit{Source\text{-}code\ location}\\
\mathscr{B} &= \mathit{true} \mid \mathit{false}\\
\mathit{Value}' &= \mathit{Value} \times \mathscr{P}(\mathit{TaggingWrapper})\\
\mathit{Obj}' &= \mathit{Obj} \times \mathscr{P}(\mathit{TaggingWrapper})\\
\mathit{TaggingWrapper} &= \mathit{Tagging} \times \mathscr{B} \times \mathscr{B}\\
\mathit{Tagging} &= \mathit{Module}(M)\\
&\mid \mathit{SideEffect}(M)\\
&\mid \mathit{ImpreciseWrite}(M)\\
&\mid \mathit{Taint}(SL)
\end{aligned}$$

Figure 9.2: Extensions to the abstract domain in TAJS. Highlighted parts are taint extensions introduced in Section 9.5.

(including transitive dependencies). This algorithm performs heuristics to update MS_P and MS_B sets at the end of each iteration. If these sets are modified, we run the EXTENDEDTAJS analysis again using the updated MS_P.[5] Fig. 9.3 is described in detail in Section 9.4.2.

9.4.1 Extending the Static Analysis in TAJS

In this section, we describe our extensions to the abstract domain in TAJS, how the module loader is modified to approximate modules that are not in MS_P, and changes to the underlying analysis in TAJS to handle values that originate from approximated modules.

Abstract domain extension Fig. 9.2 shows the extensions added to the abstract domain in TAJS. We extend abstract values and abstract objects with a *TaggingWrapper* set. *TaggingWrapper* consists of *Tagging*, which provides additional information about the origin of an abstract value. *Tagging* can have three values: (1) *Module*(`M`) specifies that the abstract value originates from module `M`; (2) *SideEffect*(`M`) specifies that module `M` might have caused side-effects on the abstract value (which have been unsoundly ignored because we ignore side-effects for approximated modules); and (3) *ImpreciseWrite*(`M`) specifies that an abstract value (v) is written in an imprecise dynamic property write, i.e., $o[p] = v$ where p is approximated due to not analyzing `M`.

[5]Note that the EXTENDEDTAJS analysis does not reuse analysis results from the previous iterations.

Joining and propagating *TaggingWrappers* *TaggingWrappers* are joined by set-union and all built-in models are updated to propagate *TaggingWrappers*.

Updated module loader Before loading a third-party module `m` (line 1 in Listing 9.2), the updated module loader checks if `m` is in MS_P. If `m` is in MS_P, it is analyzed, otherwise it is approximated with the *tagging*: $\mathit{Module}(\texttt{m})$. Note that Node.js built-in modules are always analyzed.

```
1 var m = require('m');
2 var obj = {};
3 m.f(function(g) {
4   obj.a = g;
5 });
6 obj.a();
7 // rest of application
8
```

Listing 9.2: Approximating callbacks.

Calls to approximated functions To understand how the analysis handles the values passed to the modules that are not analyzed, consider the code-snippet in Listing 9.2. In this example, because `m` is not in MS_P, it is not analyzed, i.e., the value of the variable `m` is approximated. If we do not analyze the call to the callback function passed as argument to `m.f` at line 3, we get a definite type error[6] at line 6 (because `obj.a` is `undefined`) and the dataflow to the rest of the application will be missed. Therefore, the analysis analyzes the call to the callback function with an approximated argument (`g`) labeled with the same tag used for `m.f`, which is the same tag used for `m`[7]. Next, the approximated value is written to `obj.a` at line 4, and the type of `obj.a` at line 6 is resolved to a value of any type including function. As a result, the analysis is able to continue analyzing the rest of the application.

Next, we show how the feedback-driven analysis uses the new extension to the abstract domain to identify the modules that should be analyzed, i.e., included in MS_P.

9.4.2 Post-processing Analysis Results

At the end of each iteration in Algorithm 3, we post-process the analysis results to determine whether the module sets MS_P and MS_B should be modified as depicted in Fig. 9.3. Recall that this is done for each third-party module in the application dependency tree (including transitive dependencies). Given a module `m`, if it is in MS_B, we end the post-processing step for this module because modules are never removed from MS_B. If `m` is not in MS_B, but in MS_P, we check if `m` has been too expensive to analyze. The predicate `timesOut(m)` is satisfied when `m` has taken a large fraction of the analysis time. In this case, `m` is moved from MS_P to MS_B.

[6] All abstracted executions end in a `TypeError`.

[7] $\mathit{Module}(\texttt{m})$

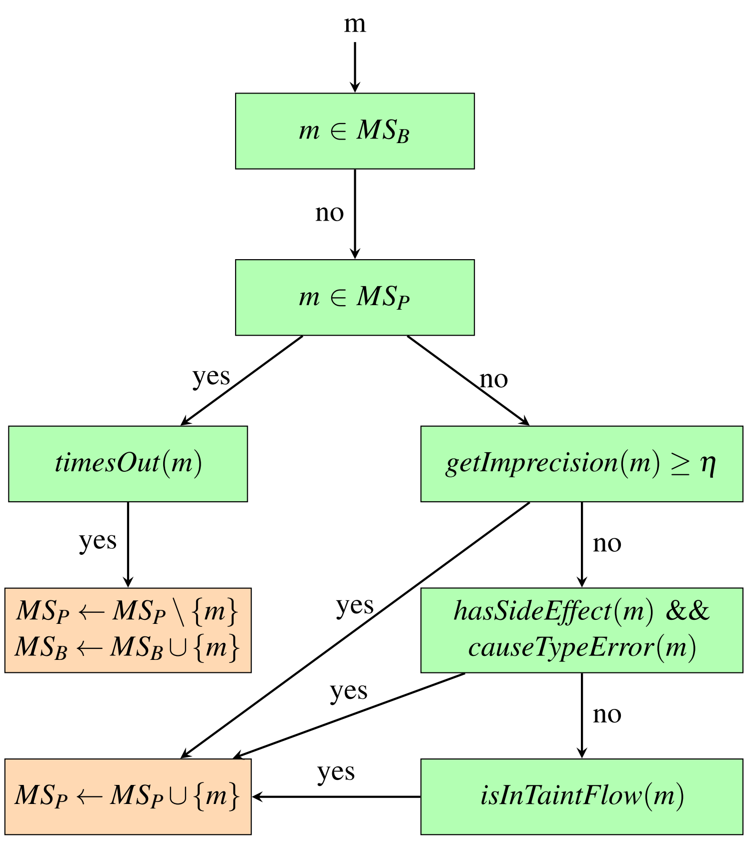

m
$m \in MS_B$
no
$m \in MS_P$
yes
no
$timesOut(m)$
$getImprecision(m) \geq \eta$
yes
no
$MS_P \leftarrow MS_P \setminus \{m\}$
$MS_B \leftarrow MS_B \cup \{m\}$
yes
$hasSideEffect(m)$ && $causeTypeError(m)$
yes
no
$MS_P \leftarrow MS_P \cup \{m\}$
yes
$isInTaintFlow(m)$

Figure 9.3: Flowchart for post-processing analysis results. Rectangular boxes indicate modifications to MS_B or MS_P, while the other boxes indicate predicates used by our heuristics. Post-processing terminates for an input module, m, when there is no transition to follow.

If `m` is neither in MS_B nor in MS_P, we apply heuristics to determine whether `m` should be added to MS_P. The first heuristic is a precision heuristic which checks if not analyzing a module results in a large precision loss. In this heuristic, the function `getImprecision(m)` returns the number of source locations in which the $ImpreciseWrite$(`m`) tag (see Fig. 9.2) is read. If this number exceeds the preconfigured threshold η, `m` is added to MS_P. Otherwise, we apply our side-effect heuristic. The predicate `hasSideEffect(m) && causeTypeError(m)` checks whether a $SideEffect$(`m`) tag ends up in a definite type error. If that is the case, we add `m` to MS_P. The last predicate (`isInTaintFlow(m)`) is related to the taint analysis, which is used as a client analysis in this paper, and is explained in Section 9.5.

To better understand the side-effect heuristic, consider the code-snippet in Listing 9.3, which uses the `setPrototypeOf` module to set the properties of the `router` object to the `proto` object. The program imports the module `setPrototypeOf`, defines a function and writes it to the `proto` variable from lines 2 to 8, and adds a property to `proto` at line 9. From lines 3 to 6, the function `router` is defined and the function `setPrototypeOf` is called with `router` and `proto` passed as arguments. This function call adds `proto` to the `router` object's prototype chain, which makes the properties of `proto` accessible through the `router` function object. In this example, the `router.handle` function called at line 4 is the function `proto.handle` defined at line 9. If the analysis fails to resolve `router.handle` correctly, it stops due to a definite type error at line 4.

Now we explain how our feedback-driven approach handles this example. Initially, the `setPrototypeOf` module is not included in MS_P. Because `setPrototypeof` is approximated, the analysis does not analyze it at line 6. Instead, it adds a side-effect tag[8], to the `router` and `proto` objects. The `router` function is returned at line 7 and because we do not overapproximate side-effects of `setPrototypeOf`, when the `router` function is called, the analysis cannot resolve `router.handle`, and the side-effect tag ends up in a definite type error. Therefore, `setPrototypeOf` is added to MS_P to be analyzed in the next iteration.

```
 1 var setPrototypeOf = require('setPrototypeOf');
 2 var proto = module.exports = function() {
 3   function router(req, res, next) {
 4     router.handle(req, res, next);
 5   }
 6   setPrototypeOf(router, proto);
 7   return router;
 8 };
 9 proto.handle = function handle(req, res, next) {...}
10
```

Listing 9.3: Side-effect heuristic example.

Termination of feedback-driven analysis Our feedback-driven analysis in Algorithm 3 is guaranteed to reach a fixpoint. Each iteration is guaranteed to terminate because the underlying analysis (EXTENDEDTAJS) is guaranteed to terminate and PROCESSANALYSISRESULTS runs in linear time with respect to the modules used by the application. There is an upper bound for the number of iterations in the feedback loop in Algorithm 3 and because each iteration terminates, the entire algorithm is guaranteed to terminate. Note that MS_P is modified in each iteration, but a module is added to MS_P at most once during the entire feedback-driven analysis: when a module is moved from MS_P to MS_B, it remains in MS_B until the end of the analysis. Assuming that the number of third-party modules used by an application is n, at most n modules can be added to MS_P, resulting in at most n iterations. In iterations where no modules are added to MS_P (and MS_P has not reached a fixpoint), a module is moved to MS_B. Note that at most n modules can be added to MS_B, resulting in at most n extra iterations. Therefore, our feedback-driven analysis is guaranteed to terminate after at most $2n$ iterations.

[8] *SideEffect*(`setPrototypeOf`)

Algorithm 4 Optimized worklist algorithm in TAJS

```
Input: $\theta$
add the initial node and context $(n,c)$ to the worklist
wlPhase = ORDINARY
postponedWorklist = $\emptyset$
visitationCounter = $\emptyset$
while worklist not empty do
    remove node and context $(n,c)$ from worklist
    if wlPhase == MAX-COV then
        if visitationCounter.get($((n,c))$) > $\theta$ then
            Add $(n,c)$ to postponedWorklist
            continue
        end if
        visitationCounter.count($((n,c))$)
    end if
    analyze $(n,c)$
    for all successors $n'$ of $n$ do
        if propagate($((n',c))$) then
            Add $(n',c)$ to worklist
        end if
    end for
    if worklist empty and wlPhase == MAX-COV then
        worklist = postponedWorklist
        Empty postponedWorklist
        wlPhase = ORDINARY
    else if wlPhase == ORDINARY and switchPhase() then
        visitationCounter.reset()
        wlPhase = MAX-COV
    end if
end while
```

9.4.3 Optimized Worklist Algorithm

To scale the static analysis to Node.js applications and increase code coverage, in addition to the feedback-driven analysis discussed in this section, we design an optimized worklist algorithm that aims to increase the coverage of worklist items before the analysis times out. This is done by introducing a new phase in the worklist algorithm, as shown in Algorithm 4. In this new phase, if a (n,c) pair is visited more than θ times, it is postponed to be processed in the ordinary worklist phase. When the worklist is empty, the ordinary worklist phase continues with the postponed worklist items.

The highlighted parts of Algorithm 4 show the new extensions to the standard worklist algorithm in Algorithm 2. The algorithm is performed in two phases: (1)

ORDINARY, and (2) MAX-COV. The current phase is stored in wlPhase, which is initially set to ORDINARY (line 3). postponedWorklist (line 4) is a list containing worklist items that have been postponed during the MAX-COV phase. visitationCounter (line 5) is a map from (n,c) to an integer number, describing how many times (n,c) is visited in the current MAX-COV phase. During the MAX-COV phase, lines 9 to 13 make sure that a (n,c) pair is not analyzed more than θ times. Lines 21 to 28 switch between the ORDINARY and MAX-COV phases: Lines 22 to 24 switch from MAX-COV to ORDINARY when the worklist is empty and postponedWorklist is assigned to the ordinary worklist to be processed later in the ORDINARY phase; Lines 26 to 27 switch from ORDINARY to MAX-COV based on the analysis execution time, making sure that multiple phases of MAX-COV are performed before the analysis times out.

9.5 Static Taint Analysis

In this section, we explain how our feedback-driven analysis can be extended to perform static taint analysis as a client analysis.

9.5.1 Incorporating Taint Analysis

To support taint analysis, we make slight modifications to the abstract domain (Section 9.4.1) and add one more heuristic to the post-processing algorithm (Fig. 9.3) in our feedback-driven analysis.

Abstract domain extensions The highlighted parts of Fig. 9.2 show the modifications needed to support static taint analysis. *TaggingWrapper* is extended with two boolean flags: the first one indicates whether the value is tainted or not, and the second one indicates whether the value is a sink. To understand why we need these additional flags, consider the code: `var x = require("M").f(eval)`, where *M* is neither a taint source nor sink. The value returned from `require("M").f(eval)` might be a taint sink because `eval` is a sink. However, the tag present in the return value would still be *Module*(`M`). Therefore, by adding these flags we can precisely distinguish which abstract values originating from *M* might be taint sources or sinks. We also added the *Tagging* type, *Taint*(*SL*), specifying that the value is tainted by a taint source located at the source location *SL*.

Post-processing extension To make sure we do not miss any taint flows because of not analyzing a module in the feedback-driven approach, we use the last heuristic (taint heuristic) in the post-processing step as shown in Fig. 9.3. The taint heuristic adds a module `m` to MS_P when `isInTaintFlow(m)` predicate holds. Intuitively, we add `m` to MS_P if it is either the source or the sink in a taint flow. Therefore, the predicate holds in two cases: (1) `m` is the *sink*: a tainted value flows to a sink with the tag *Module*(`m`), or (2) `m` is the *source*: a tainted value with the tag *Module*(`m`) flows to a sink and the sink is not approximated due to not analyzing a different module.

9.5.2 Taint Analysis Configuration

We perform a syntactic analysis that identifies sources and sinks in the application and third-party modules using method signatures. The source and sink definitions are provided as configurations by the user. For instance, if `eval` is marked as a sink, the syntactic analysis looks for occurrences of `eval` in the source-code files. It analyzes all files that are reachable through `require` function calls, where the argument is a constant string. If the syntactic analysis does not find any taint sources or sinks in a module `m` or its dependencies, loading module `m` will yield an approximated value tagged with $(\mathit{Module}(\mathtt{m}),\mathit{false},\mathit{false})$, which indicates that the approximated value is neither a taint source nor a sink.

9.5.3 Revisiting the Motivating Example

In this section, we show how our feedback-driven analysis extended with taint analysis finds the taint flow in Listing 9.1. In this example, our syntactic analysis marks the `http` request object allocated through the `express` module as source and `mongodb` as sink. Initially, MS_P is empty, so third-party modules added between lines 54 to 57 are approximated, and MS_B contains `mongodb`.

In the first iteration of the analysis, `express` is not analyzed, therefore `app` at line 58 is approximated. Because tainted data enters the application from the `express` module, it is labeled with a taint tag. At line 60, `app.get`, `req`, and `res` are all approximated and labeled with the same tag as `app`[9]. The `req` object flows to the variable `q`, ultimately reaching `mongo.collection.find`. This flow indicates that a tainted value has reached a sink. Because of the taint heuristic discussed earlier in this section, `express`, the approximated module from which the taint tag is originated, is added to MS_P.

In the next iteration of the analysis, MS_P contains `express`. To simplify this example, we skip the iterations where the analysis adjusts the MS_P and MS_B sets to analyze the `express` module. In the last iteration, MS_P contains the necessary modules to precisely identify that `req` is an `http` request object, so `req.query.val` is tainted. As `req.query.val` flows to `mongo.collection.find`, a taint flow is reported from an incoming http request to a NoSQL sink.

9.6 Evaluation

We implemented our feedback-driven static taint analysis of Node.js applications in a tool called NODEST. To evaluate our technique we used a Windows 7 machine with an Intel Core i5-5300 CPU @ 2.3 GHz, and a JVM with 10GB memory. We use the benchmarks from [52, 78, 135], which are the existing program analysis frameworks for Node.js that detect injection vulnerabilities. We also analyze two more Express-based applications, `mongo-express` and `ankimedrec` which have not been analyzed

[9] $(\mathit{Module}(\mathtt{express}),\mathit{true},\mathit{false})$

by previous works. We have responsibly disclosed all the new vulnerabilities found by NODEST to the developers.

The benchmarks from [78, 135] have small test-drivers for npm modules/applications that exercise injection vulnerabilities. We use the test-drivers for some of these benchmarks in our evaluation. For applications that depend on networking libraries, such as the `http` module, we do not need the test-drivers because our analysis over-approximates all the incoming requests. Our evaluation answers the following three research questions:

RQ1: Can static taint analysis detect taint flows in simple Node.js modules with high precision?

RQ2: Is NODEST able to improve the scalability of a whole-program static taint analysis without missing any known taint flows?

RQ3: How important is the optimized worklist algorithm for scaling static analysis of Node.js applications?

9.6.1 RQ1 - Precision

We will answer the first research question by comparing our static analysis with prior dynamic analysis works that detect taint flows. The static analysis in this experiment is the whole-program analysis in TAJS [10] extended with taint support. We use the benchmarks from [135] that are modules (not applications) for this experiment as well as the benchmarks used in [78]. All the benchmarks from [135] contain at least one known vulnerability. We exclude the modules in which exploiting the vulnerability requires interaction with the file-system. The [78] benchmarks contain both vulnerable and benign npms. The latter are used to test precision. Table 9.1 shows that NODEST finds the vulnerabilities in 22 out of 25 modules and does not reach a fixpoint in the remaining 3. The analysis does not report false positives in any of the 25 benchmarks. These results indicate that static analysis is suitable for detecting taint flows for Node.js modules, because the analysis reaches a fixpoint for almost all of these benchmarks and has high precision.

9.6.2 RQ2 - Scalability and Accuracy

To answer the second research question, we use those benchmarks from [135] that are applications, most of which depend on the Express [3] framework. We also test against `mongoosify` and `modulify`, which are two of the three modules that could not be analyzed by the whole-program analysis in Table 9.1. We do not include the third module because it has no dependencies. Furthermore, we evaluate our technique on one application from [52], `NodeGoat-v1.1`, and two new applications, `Ankimedrec` and `mongo-express`. All of these applications are based on Express [3] and use a MongoDB database [4].

For the Express-based applications, we have added a couple of source-code transformations to remove unsupported ES6 features [2]. Alternatively, we could

Table 9.1: Results for whole-program analysis of module benchmarks. Analysis timeout is five minutes.

Module	No FN	No FP	Reached fixpoint
os-uptime	✓	✓	✓
chook-growl-reporter	✓	✓	✓
growl	✓	✓	✓
os-env	✓	✓	✓
fish	✓	✓	✓
mlog	✓	✓	✓
node-os-utils	✓	✓	✓
gm	✓	✓	✓
mongo-parse	✓	✓	✓
mongoosify	✗	✓	✗
printer	✓	✓	✓
kerb_request	✓	✓	✓
mixin-pro	✓	✓	✓
pidusage	✓	✓	✓
modulify	✗	✓	✗
system-locale	✓	✓	✓
mol-proto	✗	✓	✗
libnotify	✓	✓	✓
pomelo-monitor	✓	✓	✓
systeminformation	✓	✓	✓
node-wos	✓	✓	✓
git2json	✓	✓	✓
office-converter	✓	✓	✓
mongoosemask	✓	✓	✓
cocos-utils	✓	✓	✓

have used an existing source-code transformation tool, such as Babel [1], to analyze such features. We chose not to invest time setting up Babel, since very few locations required transformations. Using Babel instead should not affect the analysis results. Moreover, we have annotated a couple of functions in Express to enable additional parameter-sensitivity [10]. Note that the underlying abstract interpretation analysis could be improved to avoid such transformations, however, the goal of this evaluation is to test the effectiveness of our feedback-driven analysis. Therefore, we use the underlying analysis as it is.

Feedback-driven analysis setup The feedback-driven analysis presented in Section 9.4 relies on the following configurations: MS_P is initially empty except for `mongoosify` and `modulify`, for which MS_P is defined as {`mongoosify`} and {`modulify`} respectively, because they require test-drivers to exercise the injection vulnerabilities.

Table 9.2: Results for feedback-driven analysis of Node.js modules/applications.

Application	Identify module sets (hh:mm:ss)	Analysis time (mm:ss)	#Installed modules	$\lvert MS_P \rvert$	Eval TP/FP	Exec TP/FP	NoSQL TP/FP
NodeGoat	01:11:12	Times out	109	9	3/0	0/0	5/0
keepass-dmenu	00:09:49	00:54	13	1	0/0	1/0	0/0
ankimedrec	01:40:05	Times out	67	16	0/0	0/0	30/0
mongui	00:44:24	Times out	148	22	6/0	0/0	11/0
codem-transcode	00:01:00	Times out	31	3	0/0	1/0	0/0
Mock2easy	01:05:54	Times out	429	15	0/0	0/0	0/0
mongoosify	00:03:10	00:13	2	1	1/0	0/0	0/0
modulify	00:00:00	00:13	25	1	1/0	0/0	0/0
mongo_edit	00:20:50	Times out	25	9	1/0	0/0	0/0
mongo-express	01:04:36	Times out	103	19	0/0	0/0	2/0
mqtt-growl	00:05:01	21:22	74	5	0/0	1/0	0/0
Total	-	-	-	-	12/0	3/0	48/0

MS_B is initialized with the following database modules (whose APIs are used as taint sinks): `mongodb`, `monk` and `sqlite3`. After the analysis runs for more than 50 seconds, the `timesOut(m)` predicate in Fig. 9.3 is triggered if a file in `m` spends more than 90% of the analysis time. For the precision predicate in Fig. 9.3 (`getImprecision(m) >=` η), we use $\eta = 20$.[10] We use $\theta = 2$ in Algorithm 4 and `switchPhase` triggers three times after 150, 800 and 1600 seconds. Note that these parameters have been chosen by one or two trial and errors, and they are not tuned to our benchmarks. We do not expect slight perturbations of these parameters to affect the analysis results significantly.

Feedback-driven analysis results Table 9.2 shows the results of running NODEST with a 30-minute timeout for each analysis iteration. Note that the analysis might terminate before reaching the timeout if any of the post-processing heuristics are satisfied. "Identify module sets" is the time spent to identify the module sets and "Analysis time" is the execution time for the last iteration, in which analysis is performed on the final version of module sets (module sets do not change in this iteration). "#Installed modules" is the number of third-party modules that are installed[11] during the installation of the application. $|MS_P|$ is the size of MS_P after reaching a fixpoint. "Eval", "Exec" and "NoSQL" indicate the number of true positive and false positive taint flows found for `eval`, `require("child_process").exec`, and NoSQL sinks provided by the database modules, respectively. We manually investigated the reported flows to determine whether they are true positives. However, the true positive taint flows might not be exploitable due to non-trivial sanitization.

NODEST is able to find taint flows in 10/11 benchmarks. We can also see that NODEST does not report any false positives for these benchmarks. Even though the analysis does not reach a fixpoint in the final iteration in 7 applications, it is still

[10]We have observed that lower thresholds can trigger at local precision losses (as compared to the precision loss spread throughout the application).

[11]We use `npm ls -prod` to count the number of unique modules.

able to report true positive taint flows. As shown in this table, identifying module sets for `modulify` takes no time. The reason is that the taint flow in this application involves only its main module, which is included in MS_P initially, and the rest of the modules that are not analyzed are not affecting the taint flow. Therefore, the modules in MS_P do not need to change. Comparing the number of installed modules with the size of MS_P after reaching a fixpoint, we see that our feedback-driven analysis skips analyzing many modules, which makes our analysis more scalable. As an example, by installing `mongui`, 148 modules are installed, which are too many for a static analysis tool such as TAJS to scale. NODEST is able to identify 22 modules ($|MS_P|$) out of these 148 modules that are sufficient to detect the taint flows. Furthermore, there is no direct correlation between the time spent to identify module sets and the size of MS_P after reaching a fixpoint. The reason is that the duration of each iteration might vary a lot across different benchmarks.

Feedback-driven vs whole-program analysis Next, we compare NODEST with the whole-program analysis in TAJS [70] on the same benchmarks to determine if the feedback-driven analysis improves the scalability without missing taint flows. Results for the whole-program analysis can be seen in Table 9.3. The timeout in this experiment is 30 minutes. This table shows that the whole-program analysis only reaches a fixpoint in one application (note that NODEST reaches a fixpoint in 4). We also observe that the whole-program analysis only finds taint flows in 4 applications, whereas NODEST finds taint flows in 10 applications. All the applications, except for `mqtt-growl`, in which the whole-program analysis finds taint flows has at most 31 installed modules, indicating that scalability of the whole-program analysis is correlated with the size of the application. The whole-program analysis finds a strict subset of the taint flows found by NODEST, which indicates that the feedback-driven analysis does not introduce any false negatives, even though it skips the analysis of some modules. Therefore, we conclude that our feedback-driven analysis is able to improve the scalability of the underlying static taint analysis without missing any known taint flows.

XSS injection vulnerabilities Because some of our benchmarks are Express-based web applications, where reflected XSS[12] is common, we also conducted experiments to see if NODEST is able to detect XSS injection vulnerabilities. The taint source for XSS vulnerabilities is `http` request object and sinks are `http` response functions. In an Express application, sources are created in the framework and sinks are often used both in the framework and application. Therefore, Express applications are likely to have common taint flows. To distinguish such results, manual investigation is required. Below, we summarize our findings for this category of vulnerabilities.

NODEST reports XSS taint flows in six applications, which are all introduced by `express`. In these taint flows, the attacker-controllable input enters the application

[12]In a reflected XSS, attacker-controllable input that comes through an `HTTP` request is sent back to the client side through an `HTTP` response.

Table 9.3: Results for whole-program analysis.

Application	Analysis time (mm:ss)	Eval TP/FP	Exec TP/FP	NoSQL TP/FP
NodeGoat	Times out	0/0	0/0	0/0
keepass-dmenu	Times out	0/0	0/0	0/0
ankimedrec	Times out	0/0	0/0	0/0
mongui	Out of memory	0/0	0/0	0/0
codem-transcode	Times out	0/0	1/0	0/0
Mock2easy	Times out	0/0	0/0	0/0
mongoosify	Times out	0/0	0/0	0/0
modulify	Times out	1/0	0/0	0/0
mongo_edit	Times out	1/0	0/0	0/0
mongo-express	Times out	0/0	0/0	0/0
mqtt-growl	25:44	0/0	1/0	0/0
Total	-	2/0	2/0	0/0

through `req.query.url` and is sent back to the client side through `res.end` if the request is invalid. However, the tainted value is sanitized, so it might not be exploitable. We also found other unique true positive XSS taint flows in four applications. Except for one application, the rest of the taint flows are not sanitized. NODEST reported false positive taint flows (infeasible dataflow) for only one application. The average time for inspecting and classifying the analysis results for XSS vulnerabilities was around 10 to 15 minutes.

New taint flow reports NODEST is able to find new taint flows that are not reported by existing tools. It reports multiple `eval` vulnerabilities in `mongui`, which are not exercised and reported by existing dynamic analysis tools [52, 135]. In general, it is known that static analysis can achieve better coverage compared to dynamic analysis because the latter requires specific inputs at runtime that exercise these taint flows to be able to report them. Furthermore, all the NoSQL and XSS taint reports except for `NodeGoat`[13] are new and have not been reported before. Apart from Affogato [52], NODEST is the only tool that detects NoSQL and XSS taint flows.

9.6.3 RQ3 - Optimized Worklist Algorithm

To answer the third question, we performed the experiments in RQ2 without using the extended worklist algorithm (see Section 9.4.3). Table 9.4 shows the reported taint flows for the feedback-driven analysis (FD) and the whole-program analysis (WP), in both of which the optimized worklist algorithm is disabled. Note that the optimized worklist algorithm makes no difference in reaching a fixpoint in our benchmarks. As

[13]The deliberately vulnerable application from OWASP [6].

Table 9.4: Taint flows found using feedback-driven (FD) and whole-program (WP) analysis without optimized worklist.

	Eval		Exec		NoSQL	
Application	FD	WP	FD	WP	FD	WP
NodeGoat	3	0	0	0	4	0
keepass-dmenu	0	0	1	0	0	0
ankimedrec	0	0	0	0	0	0
mongui	0	0	0	0	0	0
codem-transcode	0	0	1	1	0	0
Mock2easy	0	0	0	0	0	0
mongoosify	1	0	0	0	0	0
modulify	1	0	0	0	0	0
mongo_edit	1	1	0	0	0	0
mongo-express	0	0	0	0	2	0
mqtt-growl	0	0	1	1	0	0
Total	6	1	3	2	6	0

shown in Table 9.4, without using the optimized worklist, the feedback-driven analysis reports 15 taint flows (sum of FD columns) while the whole-program analysis reports 3 flows (sum of WP columns). On the other hand, using the optimized worklist helps the feedback-driven analysis to report 63 taint flows (see Table 9.2) while the whole-program analysis reports 4 flows in total (see Table 9.3). Therefore, we conclude that the optimized worklist algorithm improves the scalability of static taint analysis of Node.js applications by increasing coverage and reporting more true positive taint flows.

9.7 Case Studies

In this section, we describe two case studies to show how our feedback-driven analysis is able to find non-trivial taint flows in complex Node.js applications. In the first case study, both the source and sink are in third-party modules, i.e., in the code not analyzed in the first iteration of the feedback-driven analysis (Section 9.7.1). The second case study shows an example of a complicated taint flow, which depends on multiple requests and a specific timing between these requests, which motivates the use of static instead of dynamic analysis (Section 9.7.2).

9.7.1 Mqtt-growl

Listing 9.4 shows a simplified version of `mqtt-growl`, which has a remote code execution vulnerability (attacker controllable input reaches an `exec` sink). This code-snippet is simply importing three third-party modules from lines 1 to 3 (`mqtt`, `growl`

```
 1 var mqtt = require('mqtt')
 2   , growl = require('growl')
 3   , _ = require('underscore');
 4 growl = _.throttle(growl);
 5 mqtt.f(function (message) {
 6   growl(message);
 7 });
 8
 9 // underscore.js
10 _.throttle = function(func) {
11   return function() {
12     func.apply(this, arguments);
13   }
14 }
15
```

Listing 9.4: Simplified version of mqtt-growl.

and `underscore`) and the rest of the code does not have any sources or sinks, hence does not seem to be vulnerable in the first look. However, by combining these three modules, it enables an exploitable taint flow starting from a source in `mqtt`, passing through `underscore`, and finally reaching a sink in `growl`.

Our feedback-driven analysis goes through four iterations to identify all the modules that need to be analyzed and detect the vulnerable taint flow. Initially, all the three third-party modules from lines 1 to 3 are approximated (not analyzed). Note that even though `mqtt`, and hence `mqtt.f` from line 5 to 7 is approximated, the callback function passed as argument is called by the analysis and analyzed with approximated argument (`message`). Therefore, line 6 is reachable in all the iterations. The first iteration adds `underscore` to MS_P (the set of modules that need to be analyzed) because it has caused `growl(message)` at line 6 to be approximated while this function call is potentially part of a taint flow (see the taint heuristic in Section 9.5). The second iteration adds the `growl` module to MS_P because by analyzing `underscore`, the analysis reaches line 12 where `func` is approximated due to not analyzing `growl` and `arguments` is tainted. In the third iteration, the call to `func` at line 12 is resolved to the `growl` function object, and analyzed. Now, the tainted value passed to `growl` is approximated because of not analyzing `mqtt`, which includes the taint source (`message`). The tainted value flows to an `exec` sink and therefore, `mqtt` is identified to be potentially part of a taint flow and added to MS_P. Finally, in the fourth iteration, the analysis reports a taint flow from a source in `mqtt` to the `exec` function in `growl`.

9.7.2 Codem-transcode

`Codem-transcode` is a video transcoder, which receives requests through a simple HTTP API. The application has a XSS vulnerability, which depends on multiple requests and a specific timing between these requests. Static analysis is suitable for finding this vulnerability because it overapproximates all possible orderings of incoming requests. NODEST is the first tool that reports this vulnerability. The application has a route for posting jobs (`POST /jobs`) and a route for getting all the jobs that are registered, but not completed (`GET /jobs`). A very simplified version of the

```
 1 var slots = [];
 2 // POST /jobs
 3 postNewJob = function(request, response) {
 4   var postData = "";
 5   request.on('data', function(d) { postData += d; })
 6   request.on('end', function() { processPostedJob(postData); } );
 7 }
 8 processPostedJob = function(postData) {
 9   var job = Job.create(JSON.parse(postData));
10   slots.push(job);
11   ...
12 }
13 // GET /jobs
14 getJobs = function(request, response) {
15   var content = { jobs: slots };
16   response.end(JSON.stringify(content), 'utf8');
17 }
18
```

Listing 9.5: Simplified version of `Codem-transcode`.

implementation is shown in Listing 9.5. Lines 3 to 7 define the function that is called upon receiving a `POST` request to `/jobs` route. This function registers event listeners on the `request` object. It uses the `data` event to collect the data. Once all data is received (i.e., the end event is triggered), it calls `processPostedJob(postData)` to create a new job (`Job` object) for `postData`. Note that `postData` is tainted because the attacker can control it through HTTP requests. The code not shown in `processPostedJob` actually processes the job asynchronously and upon finishing the job, removes it from the `slots` array. Each `Job` object (that is processed but not finished yet) can be retrieved by a `GET` request to the `/jobs` route, which is handled through the function defined from line 14 to 17. As it can be seen, this function reflects back the tainted data to the client side, making the application vulnerable to an XSS attack. This example shows the advantage of using static analysis over dynamic analysis as the latter requires a test driver to trigger this behavior, while our analysis is able to find this taint flow by overapproximating the incoming requests and their orders.

9.8 Related Work

Static analysis for JavaScript has a rich history, and different static analysis frameworks have been developed over the years. Because of its highly dynamic nature, however, the JavaScript language is very difficult to analyze both precisely *and* efficiently. In this work, we presented an approach to scale static analysis for JavaScript and enable precise taint analysis of Node.js applications.

Static analysis of JavaScript In order to be practical, most, if not all static analysis frameworks for JavaScript [55, 70, 80, 91, 97] allow for precision and scalability trade-offs through context, field, heap, and loop sensitivity tuning [84, 117, 143]. When sensitivity tuning reaches its limit, however, auxiliary strategies have to be used to achieve acceptable precision and scalability. For example, the work by Madsen et al. [97] specifically addresses the issue of computing points-to analysis for JavaScript applications that depend on complex frameworks and libraries. To avoid the need for manually written *stubs*, their approach performs a use analysis that automatically infers points-to specifications. The work in [133] addresses the challenge of scaling points-to analysis for JavaScript through correlation tracking, a lightweight pre-analysis that identifies field reads and writes that must refer to the same property. Using correlation information, the subsequent points-to analysis avoids introducing spurious points-to edges, which leads to a sparser points-to graph.

In a constant quest to overcome the scalability challenges of JavaScript analysis, others have developed hybrid analyses, where the core static analysis uses dynamic information to restrict its search space, often at the cost of soundness. Wei and Ryder coined the term *blended analysis* to designate static analyses that use some dynamic analysis results as part of their computation. The work in [142] uses dynamic analysis to resolve calls to `eval` and to build a call graph that is then used by a static taint analysis [68]. Similarly, Tripp et al. [140] use concrete `location`, `referrer`, `URL`, `DOM` element values to perform partial evaluation that enables more precise string and taint analyses. While the aforementioned works all focused on JavaScript code embedded in web pages, the work in [118] specifically targets JavaScript web applications, and proposes the use of execution environment *snapshots*, as a lighter alternative to dynamic trace collection, to inform the subsequent static analysis. On the other hand, Andreasen et al. [11] used blended analysis to pinpoint root causes of imprecision and the work in [144] proposes to use dynamic information *after* the static analysis, to help pinpoint and fix root causes of imprecision, and to specialize the analysis to the code of interest. Compared to these works, while our approach is not sound, its unsoundness is more principled than relying on concrete executions, hence gaining better coverage.

Taint analysis for JavaScript Previous work on static taint analysis for JavaScript relied either on points-to analysis [54, 68], or on information flow tracking [29, 62, 63, 124]. To the best of our knowledge, this is the first paper on abstract interpretation-based taint analysis for JavaScript. Other work has also explored dynamic taint analysis approaches [28, 52, 78, 128] to circumvent the precision and scalability challenges of static analysis at the cost of completeness.

9.9 Conclusion

We have presented a feedback-driven static analysis that improves the scalability of an existing state-of-the-art whole-program static analysis for Node.js applications. By automatically identifying the third-party modules of an application that need to be analyzed, we detect taint flows in critical modules. We evaluated our tool, NODEST, on existing benchmarks and additional real-world Node.js applications. The results show that our feedback-driven static analysis scales well and is able to find previously known and new zero-day injection vulnerabilities with high precision. NODEST can statically analyze real-world applications that no other static analysis tool has been able to analyze before. In particular, it is able to analyze the Express framework and report non-trivial taint flows.

Chapter 10

Detecting Locations in JavaScript Programs Affected by Breaking Library Changes

By Anders Møller (Aarhus University, Denmark), Benjamin Barslev Nielsen (Aarhus University, Denmark), and Martin Toldam Torp (Aarhus University, Denmark). Published in the Proceedings of the ACM on Programming Languages, Volume 4, Issue 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.

10.1 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

[1] `http://modulecounts.com/` (April 2020)

80 other packages [152]. Many libraries, especially the most widely used ones, are frequently updated. New features are added, security flaws and other bugs are fixed, and outdated functionality is removed. The npm system uses semantic versioning,[2] which is a version numbering scheme that distinguishes between major updates that may contain breaking changes and minor and patch updates that usually can be adapted immediately. For most libraries, a changelog is maintained, documenting the changes, especially the ones that may require attention from the client developers.

Previous work has shown that there is typically a considerable delay, called a technical lag, between a release of a new version of a library and the corresponding update of a client [150]. 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[3] and *lodash-migrate*.[4] Developing and maintaining such specialized migration tools is difficult, which may explain why only a couple of the top 20 most depended upon npm packages[5] come with such tools. These migration tools typically work by injecting runtime warnings when affected library features are encountered in the running client, which means that they require extensive client tests to detect all the relevant places that may need updating. For some libraries, a compatibility layer is provided for major updates, for example *rxjs-compat*,[6] which can be used as temporary workarounds until the client code has been properly adapted. Finally, extensive migration guides are available for some popular libraries, such as *express*, as supplements to the changelogs, but without tool support.

Tools such as *npm audit* and Github automatically notify their users if their code depends on npm module versions that contain known security vulnerabilities. Despite the good intentions of that approach, it is often too coarse-grained to be really useful. Although it may draw attention to the need for updating the client code, it still leaves the burden of finding the relevant parts of the code to the client developer. Moreover,

[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

it very often gives false alarms because the vulnerable parts of the library are not being used by the client.

Techniques and tools used for other programming languages are difficult to adapt to JavaScript because of the dynamic nature of the language. As an example, for Java, simply recompiling the client code often reveals which program locations require attention, in the form of static type errors. Specialized tools, such as gofix[7] for Go and Coccinelle [77, 114] 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 [88, 137].

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 12.7), 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+ [105], 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 11.6.2). In that situation, it may take a long time for the client developer to realize that no changes in the client code are needed.

Our key insight is that it is possible to express breaking change patterns in a way that permits accurate and efficient pattern matching based on lightweight static

[7]`https://golang.org/cmd/fix/`

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 12.7). 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 10.3), which has guided the design of our approach.

- We propose a simple pattern language for describing the API access points that are involved in breaking changes, and we provide an accompanying program analysis tool, named TAPIR,[8] for locating which parts (if any) of the client code may be affected by breaking changes (Sections 10.4–10.6).

- 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 12.7).

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.

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

2: *Removed category names from module paths*

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

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

51: *Removed 17 aliases*
Removed `_.all` *in favor of* `_.every`
Removed `_.any` *in favor of* `_.some`
⋮

Figure 10.1: 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.

10.2 Motivating Example

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

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

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

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

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

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 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 10.1 (see the full list on `https://brics.dk/tapir/`). Evidently, finding these relevant items among all the 54 entries in the changelog is a major effort if done manually, even for someone deeply familiar with the *postal* source code. Interestingly, the second one (about removed `thisArg` params) is relevant for *postal* despite being described as "largely unused" in the changelog.

Using TAPIR, the push of a single button will list all of the 9 places in *postal*'s source that had to be changed when updating the *lodash* dependency, due to breaking changes in the library API. As we explain in Section 12.7, 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 10.2. A manual inspection reveals that 2 of the 11 matches reported are false positives (meaning that those locations in the program are actually not affected by any of the breaking changes). As part of its output, TAPIR shows its confidence, and those two cases are indeed matches with low confidence. All the other locations are true positives that require small changes to adapt to the new version of the library. Conversely, all the changes made manually by the *postal* developer to adapt to *lodash* 4.0.0 are correctly detected by TAPIR (disregarding a couple of places where *postal* accesses internal functionality of *lodash* that is not part of its public API and therefore not mentioned in the changelog).

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

10.3 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 10.1.) 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 10.1; a closer inspection reveals that this one affects 64 different functions.) This means that even for manual use by the client developers, the existing informal changelogs do not provide enough information to be able to safely adapt the client code.

To learn about the nature of the collected real-world breaking changes, we grouped them into a number of categories. The collection of packages and the number of changes belonging to each category are listed in Table 10.1.

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

Table 10.1: Breaking changes in npm packages.

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

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 (**Sig.**) 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 10.4.

Most importantly, we see from Table 10.1 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.

10.4 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 10.5 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 12.7, a collection of patterns for a library

```
            Pattern ::= import(D)? Glob
                      | read PropertyPathPattern
                      | write PropertyPathPattern
                      | call(R)? AccessPathPattern (Filter)*
               Glob ::= (GlobElement)*
        GlobElement ::= Filename | * | /**/ | /
                      | { Glob , ... , Glob }
  AccessPathPattern ::= < Glob >
                      | { AccessPathPattern , ... , AccessPathPattern }
                      | ( AccessPathPattern \ AccessPathPattern )
                      | AccessPathPattern ()
                      | AccessPathPattern **
                      | PropertyPathPattern
PropertyPathPattern ::= AccessPathPattern . Property
                      | AccessPathPattern .{ Property , ... , Property }
             Filter ::= [ Int , Int ] | [ Int ,]
                      | Int : Type | Int : { Type , ... , Type }
               Type ::= string | number | boolean
                      | undefined | object | array | function
                      | function[ Int ] | Literal
```

Figure 10.3: Pattern language for describing API access points.

update is typically relatively small and quite easy to write, even without expert knowledge of the library, so it is also possible that, for example, a client developer performs this task. This resembles the practice for TypeScript declaration files,[9] which are also often contributed to the community by other programmers than the JavaScript library developers themselves. In situations where comprehensive changelogs are not available, existing tools can be used for detecting the breaking changes in the libraries [105].

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 10.6, to automatically find the places in the client code that are affected by the specified breaking changes.

10.5 A Pattern Language for Describing API Access Points

Figure 10.3 shows the syntax of our pattern language for describing API access points of interest. There are four kinds of patterns.

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

First, an *import* pattern, written `import` q where q is a Unix path-like pattern (also known as a glob pattern), matches client code that imports a module with a name described by q. Such patterns are intended for describing the **Module** breaking changes from Section 10.3. A glob consists of a file system path that, when matched against a concrete file system, results in a set of matched files or directories. A glob can also contain wildcards, such as the * that matches all files, ** that matches all directories recursively, and $\{i_1, \ldots, i_n\}$ that matches the union of the globs i_1 to i_n. The specialized import pattern `importD` is described at the end of this section.

Example 1 The pattern

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

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

A *read* pattern, written `read` p, matches property read operations in the client code that match the property path pattern p. Most of the **Property** changes from Section 10.3 can be described by read patterns. A *write* pattern, `write` p, similarly matches property write operations. We introduce a notion of access path patterns to be able to characterize the relevant dataflows:

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

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

- p`.`f matches all property read and write operations E`.`f in the client code (for a JavaScript expression E and a property name f) where p recursively matches E. (In principle this kind of pattern also matches dynamic property accesses where the property name is dynamically computed and may evaluate to f, but see Section 10.6.)
- p`.{`f_1`, ... ,`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 10.6 we show how to approximate this statically using a simple alias analysis.

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

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

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

Example 3 The following write pattern describes the breaking change in the *async* version 3.0.0 update that was mentioned in Section 10.3.

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

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

A *call* pattern, `call` $p\ f_1\ \ldots\ f_n$, matches function/method/constructor call operations that match p and also satisfy each of the filters $f_1,\ldots,f_n$. The call patterns are designed to handle the **Function** changes from Section 10.3. We use a notion of filters to describe the changes that are conditional on the number of arguments or the argument types.

- `[`n`,`m`]` restricts the matching to calls with between n and m arguments.
- `[`n`,]` restricts the matching to calls with at least n arguments.
- n`:`t restricts the matching to calls where the n'th argument has type t.
- n`:{`t_1`,`$\ldots$`,`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 10.6 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 10.3.

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

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

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

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

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

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

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

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

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

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

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

```
importD rxjs
```

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

10.6 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 10.2.

10.6.1 Phase 1: Alias Analysis

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

$$\alpha_s \colon \mathit{Var} \cup \mathit{Prop} \rightarrow \mathscr{P}(\mathit{Exp})$$

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

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

Example 9 For a chain of assignments such as `x = require('lib')` and `y = x`, the analysis infers $\alpha_s(\texttt{x}) = \{\texttt{require('lib')}\}$ and $\alpha_s(\texttt{y}) = \{\texttt{x}\}$; it does not model transitive

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

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

dataflow, so $\alpha_s(y)$ does not contain `require('lib')`. This may seem like a serious weakness for an alias analysis, but we compensate when the alias analysis results are being used in the second phase, which we explain later.

The analysis completely ignores dynamic property write assignments, the flow of objects at function calls and returns, exceptions, etc. The analysis design is inspired by Feldthaus et al. [48] 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 12.8 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.

```
19 const _ = require('lodash');
20 const libAsync = require('async');
21 const fs = require('fs').promises;
22
23 (async function () {
24   const lines = await libAsync.map(['file1', 'file2'],
25     async (file) => {
26       const lns = (await fs.readFile(file)).split('\n');
27       const idxLns = _.map(lns, (ln, idx) => idx + ':' + ln);
28       await fs.writeFile(file, idxLns.join('\n'));
29       return lns.length;
30     });
31   console.log('Added ' + _.sum(lines) + ' line numbers');
32 })();
```

The example uses the `map` function from the *async* library in line 24 for asynchronously applying the transformation to both files. The insertion of the line numbers is done in line 27 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 24 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*

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

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

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

Figure 10.4: Access path inference.

module:

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

10.6.2 Phase 2: Access Path Inference

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

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

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 12.3.

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

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

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

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

- Otherwise, AP returns $\{\texttt{U}\}$ 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 24 computes to the singleton set $\{\texttt{<async>.map}\}$, and the access paths for the `map` read on line 27 computes to the set $\{\texttt{<lodash>.map}\}$.

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

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

The result of the alias analysis contains $\alpha_s(\texttt{f}) = \{\texttt{require('lib').fun}\}$ (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 $AP(\texttt{x.f}) = \{\texttt{<lib>.fun}, \texttt{U.f}\}$. The access path `U.f` appears spuriously because the analysis is flow-insensitive and ignores parameter passing at calls.

Example 13 For the following code,

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

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

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

since

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

and

$$\begin{aligned}AP_{\{\mathtt{x}\}}(\mathtt{x.f}) &= \{ap.\mathtt{f} \mid ap \in AP_{\{\mathtt{x}\}}(\mathtt{x})\} \cup lookup_{\{\mathtt{x}\}}(\mathtt{f})\\ &= \{ap.\mathtt{f} \mid ap \in lookup_{\{\mathtt{x}\}}(\mathtt{x})\} \cup \bigcup_{E \in \alpha_s(\mathtt{f})} AP_{\{f\}}(E)\\ &= \emptyset.\end{aligned}$$

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

Example 14 The API access path pattern

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

matches both the access paths

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

and

```
U.saturated
```

but not

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

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

Based on our experience, it is generally quite easy to decide whether to use p or $p?$. One can always choose the $p?$ variant if in doubt since it overapproximates the other one, but it may result in more false positives in the pattern matching in phase 3. However, as we show in the evaluation, the number of false positives is not as problematic as one might expect, since variable and property names tend not to overlap much across different libraries. Overall, this design choice of introducing the relaxed form of detection patterns is a pragmatic compromise between simplicity of the pattern language and scalability and accuracy of the analysis; we return to this discussion in Section 12.8.

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

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

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

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

Figure 10.5: Pattern matching relation $p \succ t$ where p is an access path pattern and t is an access path.

10.6.3 Phase 3: Pattern Matching

The third phase of the TAPIR analysis matches the API access point patterns against the client code. The matching is defined as a relation $\succ$ between access path patterns and access paths as seen in Figure 10.5. The inference rules directly reflect the descriptions of the access path patterns from Section 10.5 (the only exception being the two UNCERTAIN rules for handling $p?$ as discussed in Section 10.6.2).

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

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

Example 15 Consider the JavaScript code and the `read` pattern from Example 10. Based on the access paths computed for this client code (see Example 11), TAPIR finds a match on line 27, but not on line 24.

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)

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

against the access path pattern using $\succ$ exactly as with `call` and `write` patterns. For all calls where an access path matches, TAPIR matches the arguments of the call against the filters of the pattern. The number of arguments to the call can be extracted directly from the AST.[13] For type filters, TAPIR also attempts to extract the argument type directly from the argument AST node: If the argument is a literal, then the type is simply the type of that literal; otherwise, TAPIR conservatively assumes that the type filter matches (but with low confidence; see Section 10.6.4). For a `callR` pattern to match, the result of the call must not be discarded.

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

```
39 var _ = require('lodash');
40 ...
41 _.each(_.keys(this.cache), function (cacheKey) {
42   ...
43 }, this);
44
```

The access paths of `_.each` on line 41 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 41 instead been a call to the `dropWhile` function (which is also affected by the `thisArg` breaking change), then a type filter would also be required in the pattern:

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

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

10.6.4 Reporting Analysis Confidence

As a convenient extra feature of the static analysis, it can classify each match as either *high confidence* or *low confidence*, as shown in the example output in Figure 10.2. 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 12.7 that the confidence classification has this property.)

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

As explained in Section 10.6, the analysis is designed such that it is unlikely to have false negatives, but it may have false positives meaning that it can report spurious matches. Despite the simplicity of the analysis, there are only three sources of likely false positives: the relaxed form of patterns p? (see Section 10.6.2), the filters at call patterns (see Example 17), and inference of multiple access paths for an expression. This gives us a simple confidence classification mechanism. A match between a pattern p and an expression E is classified as low confidence if

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

and otherwise it is a high confidence match.

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

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

```
function f(x) {
  ...
  x.drain = ...
  ...
}
```

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

Example 19 Assume p is the pattern

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

from Example 17 and E is the call expression in this function:

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

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

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

```
55 var program = require('commander');
56 var UglifyJS = require('../tools/node');
57 ...
```

```
options.parse = program.parse;
...
UglifyJS.parse(...)
```

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

10.7 Evaluation

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

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

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

Benchmark selection To answer the research questions, we base our experiments on the 10 major updates of library packages from the npm registry that we also considered in our preliminary study in Section 10.3. 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 10.2–10.5.

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.

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

Table 10.2: Overview of detection patterns.

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

For measuring recall we are interested in client packages whose test suite fails after switching to the new version of the library (without making any changes to the client code). For each of the libraries, we attempted to extract at least 10 such clients (a few more for *lodash* since that update contains many breaking changes). For some libraries, we could not find that many clients, typically because the latest major update is so old that there are few packages that ever depended upon versions before it, or because the breaking changes are relatively benign and therefore unlikely to cause any test failures. As result, we obtained 115 such clients (Table 10.4). For measuring precision and performance, we collected 150 additional clients (Table 10.5) whose test suites are unaffected by switching to the new version of the library.

10.7.1 RQ1 (Expressiveness)

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

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

Table 10.3: Detection pattern statistics.

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

When performing the experiments for RQ2, we discovered 5 breaking changes in *rxjs*, 2 in *express*, 2 in *winston*, 1 in *bluebird* and 1 in *node-fetch* that were not mentioned in the changelogs. We have also written patterns for those changes. The total number of breaking changes for each library is shown in the **#BC** column in Table 10.2.

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 10.2), 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 10.3. 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 10.5, the patterns are generally short and simple. Table 10.3 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

Table 10.4: Experimental results for clients with test suites that fail after the update.

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

of each pattern according to the grammar of Figure 10.3. For example, the path in Example 1 has length 1, the path in Example 3 has length 4, and the path in Example 4 has length 3. The mean length of each pattern is only 3.25, which shows that patterns are typically quite small.

With this length metric, we choose to count a property path pattern with multiple property names as one (and similarly for type filters containing multiple types, and for globs at import patterns), because those do not contribute much to the pattern complexity. For example, the `thisArg` breaking change of *lodash* affects 64 different methods, which leads to a property path pattern with 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 (`NF`) are used frequently. Type filters (`TF`), 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 10.6, 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.

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

Recall For TAPIR to be useful in practice, the recall must be high, as discussed in Section 12.1: 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 10.4.

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

Table 10.5: Experimental results for clients with test suites that are unaffected by the update.

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

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 10.4 and 10.5).

For some of the breaking changes (primarily belonging to the behavioral category from Table 10.1), TAPIR cannot decide with certainty whether a call is affected by the breaking change, either because the pattern language is too coarse-grained, or because the client application is itself a library, which makes it impossible to determine if some client uses that library in a way that involves the breaking change. We conservatively mark alarms of this form as true positives as long as TAPIR points to the correct function and it satisfies the filters, but we report them separately as **TP**$_B$ in Tables 10.4 and 10.5 since we acknowledge that they may not actually be true positives in all cases.

As shown in Table 10.4, for the group of clients whose test suites detect breaking changes in the libraries, 86% of all alarms are true positives (**Prec.**), 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**$_{\neg B}$) point to places in the client source code that with certainty have to be patched to update the benchmark.

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

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

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

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

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

Confidence 54% of the alarms in the affected test suite clients and 57% of the alarms in the unaffected test suite clients are marked as high confidence by the analysis mechanism described in Section 10.6.4. We have not seen any high confidence alarms in the false positive category, which shows that for around half of the reported alarms, in practice the user does not even have to consider the possibility of false positives. Notice that low confidence alarms do not coincide with the uncertainty alarms ($\mathbf{TP}_B$). A low confidence alarm occurs because the analysis is too imprecise to either prove or disprove that a source location is affected by a breaking change. An uncertain alarm occurs whenever TAPIR lacks information on how the potentially affected API is used, or the detection pattern language is too coarse-grained to capture the constraints for the breaking change.

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

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

10.8 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 [19, 43, 86, 100, 104]. Dig and Johnson [43] 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 10.3.

Multiple studies have investigated why clients do not keep their dependencies up-to-date [41, 150]. 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 [26, 38, 45, 77, 93, 107, 112, 151]. Several techniques also include mechanisms for patching the affected client code [26, 38, 77, 107, 112] and even how to automatically infer broken APIs from the diff between library versions [38, 45, 93, 107]. 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 [151]. While Python is also dynamically typed, it is still significantly different from JavaScript. The API usage in Python is sufficiently static for PyCompat to be able to identify the relevant call sites in the client code directly from the AST. For JavaScript, a more sophisticated mechanism, such as the access path analysis of TAPIR, is required to achieve good precision.

Lightweight static analysis Creating sound context-sensitive dataflow analyses for JavaScript is notoriously difficult due to the highly dynamic language features [88, 137]. For that reason, using lightweight, unsound analysis techniques has been explored by previous work, for example to help with refactoring [46] and for constructing call graphs [48]. Common for these approaches is that they are unsound but work well in practice. As shown in Section 12.7, the static analysis of TAPIR belongs to this family of analyses.

A disadvantage of our pattern language is that pattern writers must consider whether to use the relaxed variant of patterns (i.e., p?). As we argue in Section 12.7, 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. [48] 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.

10.9 Conclusion

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

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

Once the affected locations have been found in the client code, the next task for the developer is to update those parts of the code (if any), to adapt to the breaking changes in the library. That task can possibly also be partly automated, which we will explore in future work. We also plan to investigate how to automatically generate API access point patterns for breaking changes, possibly using NoRegrets+ [105] as a starting point.

Chapter 11

Semantic Patches for Adaptation of JavaScript Programs to Evolving Libraries

By Benjamin Barslev Nielsen (Aarhus University, Denmark), Martin Toldam Torp (Aarhus University, Denmark) and Anders Møller (Aarhus University, Denmark).

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.

11.1 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 [106]. Inspired by the Coccinelle tool [112, 113], 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 the following:

- 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 11.3 and 11.4).

[1] `https://dependabot.com/`

- We propose an interactive mechanism for filtering away false positives from the detection pattern matches reported by TAPIR (Section 11.5).

- 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 12.7).

11.2 Motivating Example

The *rxjs* library,[2] 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 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*.[3] 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,[4] by no means a small task.

Figure 11.1 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 61, in fact it is not imported at all. This is because the `timer` function (`Observable.timer` in line 69) in rxjs 6.0.0 should be accessed directly from `"rxjs"` as can be seen by the import in line 67. Therefore line 74 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 62 is replaced with the import of `Subject` in line 67. The imports in lines 63–64 add properties to `Observable` and rxjs observables (through

[2]`https://www.npmjs.com/package/rxjs`
[3]`https://github.com/jeffbski/redux-logic`
[4]Counted using Git's notion of additions and deletions.

```
61 - import { Observable } from 'rxjs/Observable';
62 - import { Subject } from 'rxjs/Subject';
63 - import 'rxjs/add/observable/timer';
64 - import 'rxjs/add/operator/takeUntil';
65 + import { defaultIfEmpty, take, takeUntil, tap }
66 +        from 'rxjs/operators';
67 + import { timer, Subject } from 'rxjs';

68 - const c$ = (new Subject()).take(1);
69 - Observable.timer(warnTimeout)
70 -   .takeUntil(c$.defaultIfEmpty(true))
71 -   .do(() => {...})
72 -   .subscribe();
73 + const c$ = (new Subject()).pipe(take(1));
74 + timer(warnTimeout).pipe(
75 +   takeUntil(c$.pipe(defaultIfEmpty(true))),
76 +   tap(() => {...})
77 + ).subscribe();
```

Figure 11.1: 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.

`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 65 and 67. Line 68 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 73 in the patched code. For the same reason, lines 69–72 have been updated to use `.pipe` in lines 74–77. 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 [146], 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. 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 11.1, JSFIX does not ask any questions. However, suppose the analysis were too imprecise to determine that `c$` on line 70 is an rxjs observable, then JSFIX would have asked this question:

> *src/createLogicAction$.js, 70:14:70:36:*
> *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).

11.3 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 11.4. As output, JSFIX produces a transformed version of the client that, under certain assumptions described in Section 11.5, preserves the semantics of the old client code but now uses the new version of the library.

The analysis phase uses the TAPIR [106] 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 11.5 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.

11.4 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 21 Lines 78–79 in the following program use the `max` function from the *lodash* library.

```
78     var _ = require('lodash');
79  -  _.max(coll, iteratee, thisArg);
80  +  _.max(coll, iteratee.bind(thisArg));
```

The optional third argument on line 79, `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 79 has to be transformed by inserting a call to `bind` as shown on line 80.

Example 22 Lines 81–83 in the program below use the *async* library's `queue` data structure, which holds a queue of tasks (asynchronous functions) to be processed.

```
81     var async = require('async');
82     var q = async.queue(...);
83  -  q.drain = () => console.log("Done");
84  +  q.drain(() => console.log("Done"));
```

On line 83, 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 84.

Example 23 Lines 85–86 below import the `find` and `map` functions from the *rxjs* library.

```
85  -  import find from "rxjs/operator/find"
86  -  import map from "rxjs/operator/map"
87  +  import {find, map} from "rxjs/operators"
```

$$
\begin{array}{rcl}
\alpha \in \mathit{Expression} & ::= & \ldots \\
& | & \$\ \mathit{RefElement}\ (\texttt{:}\ \mathit{RefElement})^{*} \\
& | & <\mathit{ModuleElement}^{+}> \\
& | & \#\ i\ \mathit{Replacer}^{?} \\
\mathit{RefElement} & ::= & \texttt{prop}\ \mathit{Replacer}^{?} \mid \texttt{value} \mid \texttt{base} \\
& | & \texttt{callee} \mid i \mid \texttt{args}\ \mathit{Selector}^{?} \\
\mathit{Replacer} & ::= & [\ s_1 \Rightarrow s'_1, \ldots, s_n \Rightarrow s'_n\ ] \\
\mathit{Selector} & ::= & [\ j, k\ ] \mid [\ j\ ,] \\
\mathit{ModuleElement} & ::= & s \mid / \mid \#\ i\ \mathit{Replacer}^{?}
\end{array}
$$

Figure 11.2: 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 *rxjs*, these import paths `"rxjs/operator/find"` and `"rxjs/operator/map"` are no longer available. Instead, clients must import the `find` and `map` functions from `"rxjs/operators"` as demonstrated by the transformed import on line 87.

To automate the transformation of the client code, we define a suitable notion of semantic patches. A *semantic patch*, $\rho \rightsquigarrow \alpha$, models a breaking change and consists of a *detection pattern* ρ that identifies the affected part of the library API (the affected location), and a *code template* α 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 11.5. The detection patterns are identical to the patterns used by the API access point detection tool 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 11.3, 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 21–23, we need a mechanism for specifying the required transformations. For this purpose, we introduce the notion of a *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 ρ. 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 21) or, in some cases, the kind of the matched AST node has to change (as in Example 22, 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 83 is the same function passed to the `drain` function on line 84.

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 ρ. For example, if ρ 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 11.2 shows the grammar for code templates. A code template is a JavaScript expression (*Expression*) that can contain some special constructs explained in the following. The '...' in the first production refers to the ordinary syntactic constructs of JavaScript expressions.

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 ρ. 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 24 In Figure 11.3, 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

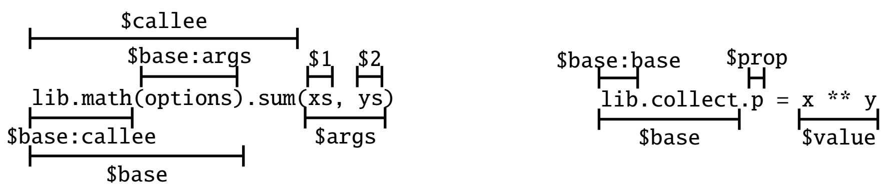


Figure 11.3: AST reference examples.

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

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

Example 25 Consider the call expression on the left of Figure 11.3. We can obtain a reference to the individual arguments of `sum` using the argument index reference, for example, `$2` refers to `ys`. We can similarly select slices of the arguments. For example, `$args[1, 2]` results in the slice `xs, ys`, and `$args[-2, -2]` results in the slice `xs`. 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 S1.[5]

Example 26 Continuing Example 21, the calls to `max` that have to be transformed are exactly those calls where `max` is called with three arguments and the second argument is a function. Hence, the following semantic patch will automate the update of the `max` calls:[6]

```
call <lodash>.max [3, 3] 2:function ⤳ $callee($1, $2.bind($3))
```

The detection pattern matches the calls to the `max` function on the *lodash* module object, where `max` is called with exactly three arguments and the second argument has type `function`. The code template specifies that those calls must be transformed such that the method call (referenced by `$callee`) has the same first argument (`$1`), but where the second argument is the result of calling `bind` on the old second argument (`$2`) passing the old third argument (`$3`) as an argument to `bind`. Hence, the transformation that is applied is exactly the one shown on line 80 in Example 21.

Example 27 Continuing Example 22, the following semantic patch expresses the required transformation:

```
write <async>.queue().drain ⤳ $base.drain($value)
```

[5]Examples whose name begin with 'S' can be found in the supplementary material (Section 11.9).

[6]The detection pattern of the semantic patch can easily be extended to also match many other functions affected by the `thisArg` breaking change, as shown in the supplementary material (breaking change number 4 for *lodash*).

The detection pattern matches writes of the `drain` property on objects returned by calls to the `queue` function on the *async* module object. These writes are transformed such 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 28 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 ⇒ some, all ⇒ 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 23) or using `require` (as in Example 21). 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 29 Continuing Example 23, 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 detection pattern 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 30 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:

```
call <uuid>.{parse,unparse} ⇝ <uuid-parse>.$prop($args)
```

Semantic patches can also be useful for post-processing transformed code to improve its readability and performance as demonstrated by Example S2.

11.5 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 12.7).

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 31 Consider the code

```
const f = (x, y) => x.map(y);

```

and the following semantic patch for *rxjs* version 6:

```
call <rxjs>?**.map ⇝ $base.pipe(<rxjs/operators>.map($args))
```

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 though 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 **CALL**) occurs when TAPIR is unable to determine if constraints on arguments (so-called call filters) are satisfied.

Example 32 Consider the following code:

```
90  var _ = require('lodash');
91  const f = (x, y) => _.pick(x, y);
92
```

The `pick` function from *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 `pick` with calls to `pickBy` when upgrading to *lodash* version 4. This transformation is captured by the following semantic patch:

```
call <lodash>.pick [2, 2] 2:function ⇝ $base.pickBy($args)
```

For this example, JSFIX will automatically generate the question "*Is the argument* `y` *of type function in line 91?*", 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 `y` is of type function in line 91.

While the primary purpose of the interactive phase is to filter away spurious AST nodes to avoid redundant or wrong transformations, 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 S3), and (2) a breaking change has relatively minor implications that some clients may prefer not to fix (see Example S4). We refer to these two categories as **EXTRA** and **MINOR**, respectively.

Note that it is only the authors of the semantic patches who need to understand the semantic patch language – the client developers who run JSFIX only need to respond to the questions in the interactive phase about the behavior of the client code.

In the end, the correctness of the applied transformations of the client code relies on three assumptions: (1) The semantic patches correctly model all the breaking changes in the library, (2) the user answers correctly in the interactive phase, and (3) TAPIR has no false negatives (i.e., it does not miss any matches) when analyzing the client code. The evaluation presented in the following section shows that these assumptions can be satisfied in a realistic setting involving real-world libraries and clients. Furthermore, even if the transformations may not be 100% correct and require manual review for some clients, the automatically transformed client code can still serve as a good starting point compared to the traditional fully manual practice.

11.6 Evaluation

We have implemented JSFIX in only 1 200 lines of TypeScript. Apart from the detection analysis and transformation phase as presented in Section 11.4, 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 Do the applied transformations make clients that are affected by breaking changes in a library compatible with the new version of the library, and 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 [106] 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. 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 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 confidently merge the pull requests.

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 [106]), shown in the first column of Table 11.1.[7]

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.[8] 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. 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.

11.6.1 RQ1 (Expressiveness of transformations)

A total of 324 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 [106], 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.

A summary of the results for each library update is shown in Table 11.1, 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 S5 and S6.

[7]We 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.

[8]In the evaluation of TAPIR, 115 clients were considered, but some of them are written in languages that compile to JavaScript, which means that JSFIX cannot patch the source code of those clients.

Table 11.1: Experimental results for RQ1.

				No code template		
Library	**BC**	**Patterns**	**Temp**	**U**	**NGP**	**?**
lodash 4.0.0	51	121	121	0	0	0
async 3.0.0	4	7	6	1	0	0
express 4.0.0	18	24	23	1	0	0
chalk 2.0.0	3	3	3	0	0	0
bluebird 3.0.0	8	12	7	2	3	0
uuid 3.0.0	1	1	1	0	0	0
rxjs 6.0.0	26	56	54	0	2	0
core-js 3.0.0	26	41	35	0	5	1
node-fetch 2.0.0	9	9	7	0	2	0
winston 3.0.0	23	29	25	1	3	0
redux 4.0.0	2	2	1	0	1	0
mongoose 5.0.0	14	19	14	2	2	1
Total	**186**	**324**	**297**	**7**	**18**	**2**

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 [106]). 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 297 of the 324 detection patterns. The remaining 27 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 S5); (2) for 18 patterns, according to our knowledge of the breaking change, no general patch exists that will work for all clients (see Example S6); (3) for 2 patterns, the breaking change was not documented sufficiently well for us to write a correct template.

For the 297 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 30). 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 33 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*, the `test` function 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:

```
call <async>.whilst ⇝                                            (11.1)
$callee( function() {                                            (11.2)
  const cb = arguments[arguments.length - 1];                    (11.3)
  const args = Array.prototype.slice.call(arguments,0,           (11.4)
               arguments.length-1);                              (11.5)
  try {                                                          (11.6)
    cb(null, $1.apply(this, args));                              (11.7)
  } catch (e) {                                                  (11.8)
    cb(e, null);                                                 (11.9)
  }                                                              (11.10)
}, $2, $3)                                                       (11.11)
```

The transformation pattern wraps the test function in a new function (lines 11.2–11.11), which extracts the new callback (line 11.3), calls the old test function, and passes the result to the callback (line 11.7). 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 S6). 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 transformation patterns 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.

11.6.2 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).

Table 11.2: Experimental results for RQ2 and RQ3 for the client test suite experiment.

Library	**C**	**Tr**	✓	✗	**−CT**	**Time**	OBJ	CALL	EXTRA	MINOR
lodash	13	70	13	0	0	1.76	6	2	3	4
async	8	10	8	0	0	2.29	2	0	0	0
express	10	18	7	0	3	1.19	11	0	0	2
chalk	10	54	10	0	0	0.21	0	0	0	0
bluebird	9	18	9	0	0	0.28	3	0	18	3
uuid	1	2	1	0	0	0.14	0	0	0	0
rxjs	6	200	5	1	0	2.20	40	0	0	0
core-js	8	28	8	0	0	6.25	2	0	0	0
node-fetch	8	7	7	0	1	0.46	7	1	6	17
winston	8	36	7	0	1	0.80	16	14	7	7
redux	4	4	3	1	0	0.33	0	0	0	3
mongoose	4	4	4	0	0	0.29	5	0	5	1
Total	**89**	**451**	**82**	**2**	**5**	**1.53**	**92**	**17**	**39**	**37**

The results of this experiment are shown in Table 11.2: "**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 11.6.3.

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 S5. 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 major updates, 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 11.1. 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,

Table 11.3: Experimental results for RQ2 and RQ3 for the pull request experiment.

Library	PR	Acc	Rej	Tr	Time	OBJ	CALL	EXTRA	MINOR
lodash	10	1	0	25	2.01	5	10	7	3
async	10	2	1	2	0.63	0	0	0	0
express	4	1	0	17	5.89	13	0	0	2
chalk	10	3	0	0	0.78	0	0	0	0
bluebird	10	3	1	2	1.21	0	0	24	1
uuid	10	3	0	0	0.95	0	0	0	0
rxjs	10	3	0	112	1.18	106	0	0	0
core-js	10	4	0	140	17.46	5	0	0	0
node-fetch	10	3	1	3	0.24	11	0	20	14
winston	10	2	0	19	1.12	34	50	10	4
redux	10	2	0	0	3.59	0	0	0	6
mongoose	10	0	1	5	1.81	16	6	18	3
Total	**114**	**27**	**4**	**325**	**2.92**	**190**	**66**	**79**	**33**

which means that JSFIX could patch all of the breaking changes in the 6 clients.

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 Table 11.3. 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 11.6.3.

Of the 114 pull requests created, 43 involved one or more transformations. 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 33, 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 S7 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.

11.6.3 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 11.6.2. The number of questions in each of the four categories OBJ, CALL, EXTRA, and MINOR from Section 11.5 is shown in the last four columns of Tables 11.2 and 11.3. 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, concerning the breaking changes that often have minor implications, the answers are more subjective, which means that our answers may diverge from what the client maintainer would have chosen. 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 Tables 11.2 and 11.3 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 S4). 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, 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.

We found it easy and fast to answer the questions, and since the client developers are familiar with their code, it should be even easier for them (as discussed in Section 11.5). Therefore, we do not consider 553 questions as a concern for the usefulness of JSFIX. Notice only 2.8 questions are asked per client and 0.7 questions per transformation on average. 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.

11.7 Related Work

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 [112, 113], which is designed to adapt Linux drivers to breaking changes in the kernel. It has also been adapted to Java [77]. 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 makes heavy use of the fact that C and Java are statically typed, unlike JavaScript.

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 [38, 45, 89, 93, 107, 151]. Most of these approaches are for Java, where each member of a class has a fully qualified name, which makes it tractable to 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 [151]. 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 [119], 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.

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 [19, 41, 74, 100, 121]. A few tools have also been designed for automatically detecting breaking changes in library updates [20, 100, 105]. 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 [105], it might be possible to derive the detection pattern part of the semantic patches automatically, which may reduce the overhead of writing semantic patches.

11.8 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 [106] 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.

11.9 Additional Examples

Example S1 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:

```
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 S2 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`.

```
- x.map(...).filter(...)
+ import {map, filter} from "rxjs/operators";
+ x.pipe(map(...)).pipe(filter(...))
```

That transformation is performed by applying the following semantic patch twice:

```
call <rxjs>**.{map,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.

```
- x.pipe(map(...)).pipe(filter(...))
+ 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 35 is also sensible for `pipe` calls that were not inserted by JSFIX. Nevertheless, JSFIX only applies such post-processing patches to clean up code it has inserted, to avoid unnecessary transformations that are unrelated to breaking changes.

Example S3 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.[9] 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, JSFIX will ask the user "*Is the first argument a function that calls its callback with more than two arguments?*".[10] 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 S4 The *node-fetch* library is a polyfill library implementing the browser HTTP library `window.fetch`. In the update of *node-fetch* to version 2, the `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 simple workaround for this breaking change is expressed by the following semantic patch:

```
call <node-fetch>?**.json ⇝ ($base.status === 204 ?  {} :  $base.json())
```

The transformed code is a ternary expression that only calls `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, semantic patches can be marked as

[9]The 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.

[10]Deciding such properties fully automatically is beyond the capabilities of any existing static analysis for JavaScript.

"low priority". For such semantic 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".

Example S5 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:

```
36 var app = require('express')();
37 app.use(app.router);
38 // middleware
39 app.use(function(req, res, next) { ... });
40 ...
41 // routes
42 app.get('/' ...);
43 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 37 must be removed. However, due to the change in the ordering semantics, the call to `app.use` on line 39 must move below the calls to `app.get` and `app.post` on lines 42 and 43. 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 S6 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 by replacing `getAll` with `get` and then splitting the resulting string at all commas:

```
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.

Example S7 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. 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[11] content manager. The *contentful-cli* package has more than 13 000 weekly downloads. Below are three excerpts of the transformation of *contentful-cli* made by JSFIX.

```
44 -  const { Observable, Subject } = require('rxjs/Rx')
45 +  const { map, filter } = require('rxjs/operators')
46 +  const { merge, Subject } = require('rxjs')
```

```
47 -  const loggingData$ = scopedEvents$
48 -    .map(({ payload }) => { ... })
49 -    .filter(message => ...)
50 +  const loggingData$ = scopedEvents$.pipe(
51 +    map(({ payload }) => { ... }),
52 +    filter(message => ...)
53 +  )
```

```
54 -  const ls$ = Observable.merge(...lss)
55 +  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 operators, such as `map` and `filter`, from methods on observable objects into independent functions that must be called through `pipe`, as previously demonstrated in Example S2. 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`.

[11]https://www.contentful.com/

Chapter 12

Modular Call Graph Construction for Security Scanning of Node.js Applications

By Benjamin Barslev Nielsen (Aarhus University, Denmark), Martin Toldam Torp (Aarhus University, Denmark) and Anders Møller (Aarhus University, Denmark).

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 an existing tool and with zero false negatives. The experiments also show that the analysis time can be 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 [81, 152]. 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 [87].

As security vulnerabilities in libraries are frequently discovered [40, 135, 136, 147, 149, 152], 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 [152]. Another study has found that 12% of the available packages have a release that directly relies on a release of a package that contains a vulnerability listed in Snyk's security reports [40], and if taking transitive dependencies and more security reports into account the percentage is likely much higher. (A related study [90] 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 [147]. 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 (CG) 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 [123]. It is well known that call graphs have many applications for a variety of development tools [48]. 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 needs to be *sound* when applied to real-world applications (we do not require theoretical soundness guarantees, but if the constructed CG misses call edges that may appear in concrete executions then security issues may be overlooked), reasonably *precise* (if having too many edges it is no better than the existing tools that only look at the package dependency structure), and *efficient* (so that the tool can be integrated into existing development processes).

[1] `https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A9-Using_Components_with_Known_Vulnerabilities`

[2] `https://dependabot.com/`

[3] `https://docs.npmjs.com/cli/audit`

[4] `https://snyk.io/`

Besides security scanning, other possible applications of the call graphs include change impact analysis [7, 60], which may be useful for finding out how breaking changes in library updates affect client code [39, 105]. 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 [48, 97].

Multiple approaches for constructing call graphs for JavaScript programs already exist (see Section 12.8), but none of them take the module structure of Node.js applications into account. 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 $\mathscr{G}_{BC}$ 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 $\mathscr{G}_{BC}$ 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 graphs for security scanning (Section 12.6).
- We demonstrate experimentally 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 (Section 12.7).

12.2 Motivating Example

Consider the npm application *writex*[6] 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 for every occurrence of a vulnerable dependency, and some appear through several

[5]**Ja**vaScript **m**odule analyzer

[6]`https://www.npmjs.com/package/writex`

dependency chains. For example, a prototype pollution vulnerability[7] 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.[8] 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 89 of the modules (spanning 49 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.

The modular analysis approach 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 2.2s to 0.7s.

[7] `https://www.npmjs.com/advisories/1523`
[8] `https://www.npmjs.com/advisories/118`

12.3 Key Challenges

To understand some of the challenges with computing call graphs for JavaScript applications, we describe two examples.

Example 34 Consider the code below consisting of the two modules `lib1.js` and `client1.js`:

`lib1.js`:
```
56 module.exports.filter = (iteratee) => {
57   return (arr) => {
58     const res = [];
59     for (var x of arr) {
60       if (iteratee(x))
61         res.push(x);
62     }
63     return res;
64   };
65 }
```
`client1.js`:
```
66 const filter = require('./lib1.js').filter;
67 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 56. 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 66.[9] 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 [87]. It is crucial for static call graph analysis to know the structure of the module interfaces, and we use a dynamic analysis for this purpose (see Section 12.4).

The second challenge is that a higher-order function is used; the `filter` function on line 56 takes a function as argument and also returns a function. The analysis should be able to determine that the call to `iteratee` on line 60 is really a call to the arrow function on line 67, and that the call to the value returned from `filter` on line 67 (blue parentheses) is really a call to the function on lines 57–64. Our call graph analysis keeps track of all these functions and where they are being called.

Example 35 Consider the following application consisting of `lib2.js` and `client2.js`:

`lib2.js`:

[9]This module system is known as CommonJS. The standardized ES6 module system is also supported by JAM but is rarely used in practice.

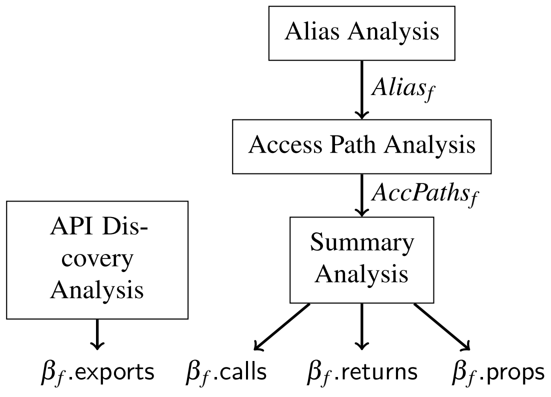


Figure 12.1: Module summary construction.

```
68 function Arit () { ... }
69 Arit.prototype.sum = (x, y) => x + y;
70 Arit.prototype.mul = (x, y) => x * y;
71 ...
72 module.exports.Arit = Arit;
73
```

`client2.js`:

```
74 const lib = require('./lib2');
75 const arit = new Arit();
76 ... 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 75. On line 76, the `sum` method is called on `arit`, resulting in an invocation of the function defined on line 69.

For the call graph analysis to resolve the call on line 76, 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 [88, 109, 137]. As we explain in Section 12.4, we instead resort to using a light-weight field-based approach that tracks what functions are written to which fields (also called properties in JavaScript) but without distinguishing individual objects. In particular, since the method call on line 76 happens on a property named `sum`, it is connected to the write to the property named `sum` on line 69. While this 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 [48].

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 *Loc*, *Prop*, *Var*, and *Exp* denote the set of all possible source code locations,[10] property names, variable names (including parameters), and program expressions, respectively. Given a single JavaScript file f as input,[11] through the processes outlined in Figure 12.1 we compute a *module summary* β_f consisting of four separate pieces of information:

- $\beta_f.\mathsf{exports}\colon \mathit{Prop}^+ \hookrightarrow \mathit{Loc}$ is an *exports summary*, where Prop^+ is a '`.`'-separated chain of property names, denoting the source location of each function exported in the file. A function written directly to the `module.exports` object (a default export), e.g., `module.exports = () =>...`, is represented by $\beta_f.\mathsf{exports}()$.

- $\beta_f.\mathsf{calls}\colon \mathit{Loc} \hookrightarrow \mathscr{P}(\mathit{AccessPath})$ 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.

- $\beta_f.\mathsf{returns}\colon \mathit{Loc} \hookrightarrow \mathscr{P}(\mathit{AccessPath})$ is a *function return summary*, which for each function definition similarly describes its possible return values.

- $\beta_f.\mathsf{props}\colon \mathit{Prop} \hookrightarrow \mathscr{P}(\mathit{AccessPath})$ 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 dynamic analysis to compute the exports summary, and a light-weight static analysis to compute the three other components, as explained next.

API Discovery Analysis The dynamic analysis works as follows, inspired by Feldthaus and Møller [47]. To generate $\beta_f.\mathsf{exports}$, the module f is first loaded and stored in a variable `m`. The analysis then scans the module object, looking for function values that appear either directly (e.g., `m.g`) or through a chain of properties (e.g., `m.x.y.z`). The source locations of those functions are stored in the exports summary map using the chain of properties as key. One might be inclined to think that using a static analysis to extract the exports summary would be an even simpler solution. However, for many modules, the module object is constructed using JavaScript's complex dynamic features that are known to be challenging to analyze statically, which is why we resort to dynamic analysis.

Access Paths The other three components of the module summary are constructed using a static analysis that involves an access path mechanism (inspired by Mezzetti et al. [100]) to describe how values are being accessed in the program (see the grammar in Figure 12.2).

[10]Source locations consists of file name, begin line, end line, begin column, and end column, and are therefore always unique for every function definition. However, for brevity we here only write ⟨file name, begin line⟩.

[11]We use the terms module and file interchangeably since a module is always stored in a single file in JavaScript.

$$\begin{array}{rcl} \mathit{AccessPath} & ::= & <\mathit{ImportPath}> \\ & | & \mathsf{FunDef}\langle f,\, l\rangle \\ & | & \mathsf{FunDef}\langle f,\, l\rangle.\mathsf{Param}[i] \\ & | & \mathit{AccessPath}\ .\ \mathit{Property} \\ & | & \mathit{AccessPath}\ (\mathscr{P}(\mathit{AccessPath}), \ldots) \\ & | & \mathsf{U} \end{array}$$

Figure 12.2: Grammar for access paths.

$$AP_V(E) := \begin{cases} \{<\!m\!>\} & \text{if } E = \texttt{require(m)} \text{ or } \texttt{import} \ldots \texttt{ from m} \\ \{ap.p \mid ap \in AP_V(E')\} \cup lookup_V(p) & \text{if } E = E'.p \\ \{ap(AP_V(E_1), \ldots, AP_V(E_n)) \mid ap \in AP_V(E')\} & \text{if } E = E'(E_1, \ldots, E_n) \text{ or} \\ & \quad E = \texttt{new}\, E'(E_1, \ldots, E_n) \\ lookup_V(x) & \text{if } E = x \text{ where } x \text{ is a non-parameter variable} \\ \{\mathsf{FunDef}\langle f, l\rangle.\mathsf{Param}[n]\} \cup lookup_V(x) & \text{if } E = x \text{ where } x \text{ is the } n\text{'th parameter in} \\ & \quad \text{a function created at line } l \text{ in file } f \\ \{\mathsf{FunDef}\langle f, l\rangle\} & \text{if } E \text{ is a function definition at line } l \text{ in file } f \\ AP_V(E_1) \cup AP_V(E_2) & \text{if } E = E' \mathbin{?} E_1 : E_2 \text{ or } E = E_1 \mathbin{||} E_2 \text{ or} \\ & \quad E = E_1 \mathbin{\&\&} E_2 \\ \{\mathtt{U}\} & \text{otherwise} \end{cases}$$

$$lookup_V(z) := \begin{cases} \bigcup_{E \in Alias_f(z)} AP_{V \cup \{z\}}(E) & \text{if } z \notin V \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 12.3: Access path computation.

- $< m >$ describes a module object obtained using, e.g., `require("`*m*`")`.
- $\mathsf{FunDef}\langle f,\, l\rangle$ describes a function value originating from a function definition in file f at line l.
- $\mathsf{FunDef}\langle f,\, l\rangle.\mathsf{Param}[i]$ describes a value of the i'th parameter of a function described by $\mathsf{FunDef}\langle f,\, l\rangle$.
- $ap\texttt{.}p$ describes the values of p properties of objects described by ap.
- $ap\texttt{(}S_1, \ldots, S_n\texttt{)}$ describes the return values of calls to functions described by ap where the i'th argument is described 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.

As an example, the access path $\mathsf{FunDef}\langle\mathsf{lib1}, 56\rangle.\mathsf{Param}[0]$ describes the value of the `iteratee` parameter to the `filter` function on line 56 in Example 34.

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

$$Alias_f\colon \big(Var \cup Prop\big) \to \mathscr{P}(Exp)$$

for the file f, such that if the value of an expression E is written to a variable or property x, then $E \in \mathit{Alias}_f(x)$.

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 $\mathit{Alias}_f(x)$ or $\mathit{Alias}_f(p)$, 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:

$$\mathit{AccPaths}_f\colon \mathit{Exp} \to \mathscr{P}(\mathit{AccessPath})$$

The map is computed by the *AP* function as shown in Figure 12.3 for all expressions in f. The subscript V in *AP* and in the *lookup* auxiliary function ensure 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 $.p$.[12] 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 using a parameter read access path. For a function definition expression, *AP* creates the corresponding FunDef 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.

Summary Analysis To form the function call summary β_f.calls, the access paths of each expression according to $\mathit{AccPaths}_f$ are grouped according to the function definition containing the expression. (For an expression in a nested function, we here only consider the inner-most function.) We use the special access path FunDef$\langle f$, *Main*$\rangle$ to refer to the function that wraps the analyzed file f.[13]

The function return summary β_f.returns is similarly computed by grouping the access paths assigned to each expression within each return statement of the enclosing function.

Finally, the object property summary β_f.props is constructed as $\beta_f.\mathsf{props}(p) = \bigcup_{E \in \mathit{Alias}_f(p)} \mathit{AccPaths}_f(E)$ for each property p.

Example 36 Continuing Example 34, we obtain the module summaries β_{client1} and β_{lib1}. Since the `filter` function is called in the outer-most scope of the `client1.js` file,

[12] The analysis ignores dynamic property reads, but since it is field-based this has little effect on its recall (see Section 12.5).

[13] Every Node.js module is wrapped in a function upon load of the module.

the call of `filter` is recorded as follows.

$$\beta_{\mathsf{client1}}.\mathsf{calls}(\langle \mathsf{client1}, \mathit{Main}\rangle) = \{\texttt{<}\mathsf{lib1}\texttt{>}.\mathsf{filter}(\ldots), \ldots\}$$

The `filter` function is exported by lib1 on line 56, so that fact is recorded in the exports summary:

$$\beta_{\mathsf{lib1}}.\mathsf{exports}(\mathsf{filter}) = \langle \mathsf{lib1}, 56\rangle$$

Furthermore, the return summary of the `filter` function records the access path of the function returned:

$$\beta_{\mathsf{lib1}}.\mathsf{returns}(\langle \mathsf{lib1}, 56\rangle) = \{\mathsf{FunDef}\langle \mathsf{lib1}, 57\rangle\}$$

Example 37 Continuing Example 35, the function defined on line 69 is written to the property `sum`, and the function defined on line 70 is written to the property `mul`. Therefore the object property summary for the module lib2 is the following:

$$\beta_{\mathsf{lib2}}.\mathsf{props}(\mathsf{sum}) = \{\mathsf{FunDef}\langle \mathsf{lib2}, 69\rangle\}$$
$$\beta_{\mathsf{lib2}}.\mathsf{props}(\mathsf{mul}) = \{\mathsf{FunDef}\langle \mathsf{lib2}, 70\rangle\}$$

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, $\beta.\mathsf{calls}$ (omitting the module name) denotes the combined call summary and is computed by
$\beta.\mathsf{calls}(loc) = \bigcup_{f \in M} \beta_f.\mathsf{calls}(loc)$
for all $loc \in Loc$ 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 $\mathscr{G} = (V, E, \beta, \alpha, \rho)$ with nodes $V \subseteq Loc$ corresponding to function definitions and edges $E \subseteq Loc \times Loc \times AccessPath$ represent the call edges, β is the combined module summary, and α and ρ are explained below. Each edge is annotated with the access path of the function being called. We use these annotations when resolving calls to higher-order function arguments.

Computing the call graph amounts to solving the constraints of Figure 12.4 and Figure 12.5. The constraints involve three relations: E, which contains the call graph edges, $\alpha\colon AccessPath \times AccessPath \times AccessPath$, which is used when resolving calls to parameters, return values, and values stored in object properties, and $\rho\colon AccessPath \times AccessPath$, which is used when resolving calls of function return values. We will explain α and ρ in detail later. We use the notation $n \underset{E}{\overset{ap}{\rightsquigarrow}} n'$ as a shorthand for $(n, n', ap) \in E$, and similarly $ap \underset{\alpha}{\overset{ap'}{\sim}} ap''$ means $(ap, ap', ap'') \in \alpha$, and $ap \underset{\rho}{\sim} ap'$ means $(ap, ap') \in \rho$.

The call graph computation solves the constraints by iteratively extending E, α, and ρ until a fixed point is reached. Such a fixed point is guaranteed to exist since E,

α, or ρ always increases in size and there are finitely many access paths in the module summaries.

The three constraints in Figure 12.4 only depend on the function call summaries ($\beta.\mathsf{calls}$), and not on E, α, or ρ, so they are all resolved in the first iteration of the algorithm.

- *static-resolve* models calls where a function at $\langle f, l\rangle$ calls another function defined at $\langle f, l'\rangle$ in the same file. The fact that the call is to a function in the same file is extracted directly from the access path.
- *module-export* models calls where a function at $\langle f, l\rangle$ calls a function with access path $<m>\ldots g$, the module m resolves[14] to the file f', and the $\ldots g$ part of the access path is in the exports summary of f'. If p is empty, i.e., $ap = <m>()$, then the default exported function is extracted from the exports summary. Notice, this constraint will only resolve function calls that are either directly exported by f' or on a chain of objects exported by f', since these are the only functions captured by the exports summary. For exported functions not satisfying these constraints, we resort to using the less precise object property summary (as explained below).
- *α-create* connects a function definition $\mathsf{FunDef}\langle f, l\rangle$ to an access path ap' if $ap'(\ldots)$ appears in the function call summary for $\langle f, l\rangle$. Determining the function definitions ap can resolve to is handled by the constraints in Figure 12.5 explained below.

Example 38 Consider the call to `filter` on line 67 marked with red parentheses in Example 34. Based on the module summaries presented in Example 36, the constraint *module-export* applies. Therefore the following edge is created, corresponding to the call to `filter`.

$$\langle \mathsf{client1}, Main\rangle \overset{\mathsf{<lib1>.filter(\ldots)}}{\underset{E}{\rightsquigarrow}} \langle \mathsf{lib1}, 56\rangle$$

The constraints presented in Figure 12.5 model calls to functions returned by other functions (*α-return-call*), calls to function parameters (*α-param-call*), and calls to functions stored in object properties (*α-prop-call*). These constraints are applied iteratively since, for example, for the expression `x()()`, resolving the second call depends on the result of the first call. If, through a series of iterations, we get that $\mathsf{FunDef}\langle f, l\rangle \overset{x}{\underset{\alpha}{\sim}} \mathsf{FunDef}\langle f', l'\rangle$, then the *$\alpha$-edge* constraint ensures that an edge is added between $\langle f, l\rangle$ and $\langle f', l'\rangle$. The annotated access path x is preserved by all the constraints to ensure that the edge created by *α-edge* is annotated with the access path of the function being called.

- *α-return-call* ensures that if some access path ap is related to an access path representing the call of a return value ($ap'(\ldots)$), then we relate ap' to the access paths representing the return values of the functions related to ap'. The ρ relation is used to retrieve these functions. It is updated using the three constraints *ρ-init-α*,

[14]The *resolve* function in this constraint is similar to the `require.resolve` function from Node.js.

$$[\textit{static-resolve}]\quad \frac{ap = \mathsf{FunDef}\langle f, l'\rangle(\dots) \in \beta.\mathsf{calls}(\langle f, l\rangle)}{\langle f, l\rangle \overset{ap}{\underset{E}{\rightsquigarrow}} \langle f, l'\rangle}$$

$$[\textit{module-export}]\quad \frac{ap = <\!m\!> \overbrace{\dots g}^{p}(\dots) \in \beta.\mathsf{calls}(\langle f, l\rangle) \quad f' = \textit{resolve}(m) \quad \langle f'', l''\rangle = \beta_{f'}.\mathsf{exports}(p)}{\langle f, l\rangle \overset{ap}{\underset{E}{\rightsquigarrow}} \langle f'', l''\rangle}$$

$$[\alpha\textit{-create}]\quad \frac{ap = ap'(\dots) \in \beta.\mathsf{calls}(\langle f, l\rangle) \qquad \text{where } \textit{static-resolve} \text{ and } \textit{module-export} \text{ do not apply}}{\mathsf{FunDef}\langle f, l\rangle \overset{ap}{\underset{\alpha}{\sim}} ap'}$$

Figure 12.4: Basic analysis constraints.

ρ*-init-E* and ρ*-reflexive* of Figure 12.5, and it ensures that once an access path *ap* has been resolved to a function $\mathsf{FunDef}\langle f, l\rangle$, then any other access path which relates to *ap* in α is related to $\mathsf{FunDef}\langle f, l\rangle$ through ρ.

- α*-param-call* ensures that if *ap* is related to a call to the *n*'th parameter in a function $\mathsf{FunDef}\langle f, l\rangle$, then we relate the call of the parameter to all access paths that flow into the *n*'th argument at all call sites to $\mathsf{FunDef}\langle f, l\rangle$.
- α*-prop-call* ensures that if *ap* is related to some call of a property *q*, then we relate the call of *q* to all access paths which *q* might refer to according to the object property summary $\beta.\mathsf{props}(q)$.

Example 39 Continuing Example 34, let us now consider how the call in blue parentheses on line 67 to `filter`'s return value is resolved. From $\beta_{client}.\mathsf{calls}$ we get the access path of this call as $<\!\mathsf{lib1}\!>.\mathsf{filter}(\dots)(\dots)$. Since neither *static-resolve* nor *module-export* matches, and the access path ends with a (), the constraint α*-create* applies:

$$\mathsf{FunDef}\langle \mathsf{client1}, Main\rangle \overset{<\mathsf{lib1}>.\mathsf{filter}(\dots)(\dots)}{\underset{\alpha}{\sim}} <\!\mathsf{lib1}\!>.\mathsf{filter}(\dots)$$

Furthermore, we saw in Example 36 that the return summary contains the fact that `filter` returns the function $\mathsf{FunDef}\langle\mathsf{lib1}, 57\rangle$. By the ρ*-init-E* constraint, we also have that

$$<\!\mathsf{lib1}\!>.\mathsf{filter} \underset{\rho}{\sim} \mathsf{FunDef}\langle\mathsf{lib1}, 56\rangle$$

so by the constraint α*-return-call* we have

$$<\!\mathsf{lib1}\!>.\mathsf{filter}(\dots) \overset{<\mathsf{lib1}>.\mathsf{filter}(\dots)(\dots)}{\underset{\alpha}{\sim}} \mathsf{FunDef}\langle\mathsf{lib1}, 57\rangle$$

which finally by α*-transitive* and α*-edge* ensures that an edge is added to *E* between the caller and the callee:

$$\langle \mathsf{client1}, Main\rangle \overset{<\mathsf{lib1}>.\mathsf{filter}(\dots)(\dots)}{\underset{E}{\rightsquigarrow}} \langle\mathsf{lib1}, 57\rangle$$

$$[\alpha\textit{-return-call}]\quad \frac{ap' \underset{\rho}{\sim} \mathsf{FunDef}\langle f,l\rangle \qquad ap \overset{x}{\underset{\alpha}{\sim}} ap'(\ldots) \qquad ap'' \in \beta_{f'}.\mathsf{returns}(\langle f,l\rangle)}{ap'(\ldots) \overset{x}{\underset{\alpha}{\sim}} ap''}$$

$$[\alpha\textit{-param-call}]\quad \frac{ap'' \in ap'[n] \qquad ap \overset{x}{\underset{\alpha}{\sim}} \mathsf{FunDef}\langle f,l\rangle.\mathsf{Param}[n] \qquad \_ \overset{ap'}{\underset{E}{\rightsquigarrow}} \langle f,l\rangle}{\mathsf{FunDef}\langle f,l\rangle.\mathsf{Param}[n] \overset{x}{\underset{\alpha}{\sim}} ap''}$$

$$[\alpha\textit{-prop-call}]\quad \frac{ap \overset{x}{\underset{\alpha}{\sim}} ap'.q \qquad ap'' \in \beta.\mathsf{props}(q)}{ap'.q \overset{x}{\underset{\alpha}{\sim}} ap''} \qquad [\alpha\textit{-edge}]\quad \frac{\mathsf{FunDef}\langle f,l\rangle \overset{x}{\underset{\alpha}{\sim}} \mathsf{FunDef}\langle f',l'\rangle}{\langle f,l\rangle \overset{x}{\underset{E}{\rightsquigarrow}} \langle f',l'\rangle}$$

$$[\alpha\textit{-transitive}]\quad \frac{ap \overset{x}{\underset{\alpha}{\sim}} ap' \qquad ap' \overset{x}{\underset{\alpha}{\sim}} ap''}{ap \overset{x}{\underset{\alpha}{\sim}} ap''} \qquad [\rho\textit{-init-}\alpha]\quad \frac{ap \overset{x}{\underset{\alpha}{\sim}} \mathsf{FunDef}\langle f,l\rangle \qquad ap \neq \mathsf{FunDef}\langle f',l'\rangle}{ap \underset{\rho}{\sim} \mathsf{FunDef}\langle f,l\rangle}$$

$$[\rho\textit{-init-E}]\quad \frac{\langle f,l\rangle \overset{ap(\ldots)}{\underset{E}{\rightsquigarrow}} \langle f',l'\rangle}{ap \underset{\rho}{\sim} \mathsf{FunDef}\langle f',l'\rangle} \qquad [\rho\textit{-reflexive}]\quad \frac{}{\mathsf{FunDef}\langle f,l\rangle \underset{\rho}{\sim} \mathsf{FunDef}\langle f,l\rangle}$$

Figure 12.5: Iterative analysis constraints.

Example 40 Let us now consider the call to `iteratee` colored brown on line 60 in Example 34. The access path of the call is $\mathsf{FunDef}\langle\mathsf{lib1},56\rangle.\mathsf{Param}[0](\ldots)$. Again, by α*-create*:

$$\mathsf{FunDef}\langle\mathsf{lib1},56\rangle \overset{\mathsf{FunDef}\langle\mathsf{lib1},56\rangle.\mathsf{Param}[0](\ldots)}{\underset{\alpha}{\sim}} \mathsf{FunDef}\langle\mathsf{lib1},56\rangle.\mathsf{Param}[0]$$

The α*-param-call* constraint says that the parameter call access path is related to the access paths of the 0'th argument of calls to $\mathsf{FunDef}\langle\mathsf{lib}, 56\rangle$. The only such call is the call to `filter` on line 67 with access path $<\!\mathsf{lib1}\!>.\mathsf{filter}(\{\mathsf{FunDef}\langle\mathsf{client1},67\rangle\})$ (we have omitted the arguments in the access path previously due to space constraints). So by the α*-param-call* constraint:

$$\mathsf{FunDef}\langle\mathsf{lib1},56\rangle.\mathsf{Param}[0] \overset{\mathsf{FunDef}\langle\mathsf{lib1},56\rangle.\mathsf{Param}[0](\ldots)}{\underset{\alpha}{\sim}} \mathsf{FunDef}\langle\mathsf{client1},67\rangle$$

which by α*-transitive* and α*-edge* results in the edge:

$$\langle\mathsf{lib1},56\rangle \overset{\mathsf{FunDef}\langle\mathsf{lib1},56\rangle.\mathsf{Param}[0](\ldots)}{\underset{E}{\rightsquigarrow}} \langle\mathsf{client1},67\rangle$$

Example 41 Consider the call to `sum` on line 76 in Example 35. From $\beta_{client2}.\mathsf{calls}$ we have that the access path of `sum` is $<\!\mathsf{lib2}\!>.\mathsf{Arit}().\mathsf{sum}()$. Again, the constraint

α-create applies:

$$\mathsf{FunDef}\langle \mathsf{client2.js}, Main\rangle \overset{\texttt{<lib2>.Arit().sum(...)}}{\underset{\alpha}{\sim}} \mathsf{<lib2>.Arit().sum}$$

This entry triggers the *α-prop-call* rule, which says that <lib2>.Arit().sum should be related to all functions in the object property summary for `sum`. From Example 37 we know that the only such function is the one defined on line 69.

$$\mathsf{<lib2>.Arit().sum} \overset{\texttt{<lib2>.Arit().sum(...)}}{\underset{\alpha}{\sim}} \mathsf{FunDef}\langle \mathsf{lib2.js}, 69\rangle$$

which by *α-transitive* and *α-edge* results in the following edge:

$$\langle \mathsf{client2.js}, Main\rangle \overset{\texttt{<lib2>.Arit().sum(...)}}{\underset{E}{\rightsquigarrow}} \langle \mathsf{lib2.js}, 69\rangle$$

Example 42 The call graph analysis is completely modular, as mentioned in Section 12.1. Let us consider the example from the introduction. We assume a call graph $\mathscr{G}_{BC}$ 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 α, ρ, and E relations from $\mathscr{G}_{BC}$. The analysis then computes the module summaries for every module in A_1 and initializes the constraints of Figure 12.4 with these summaries. The constraints from Figure 12.5 are now solved using the combined module summaries from B, C, and A_1. Because $\mathscr{G}_{BC}$ represents a partial result of $\mathscr{G}_{A_1BC}$, we can compute $\mathscr{G}_{A_1BC}$ faster with $\mathscr{G}_{BC}$ precomputed, as demonstrated in Section 12.7.

Analysis Extensions The analysis described so far does not have support for built-in functions, getters, and setters. 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 $ap \overset{x}{\underset{\alpha}{\sim}} ap'.q$ from Figure 12.5 also adds $ap'.q \overset{x}{\underset{\alpha}{\sim}} ap''$ for any access path ap'' that represents callbacks to a built-in function q. For example, if q is `map`, the analysis adds the above entry to α for each $ap'' \in x[0]$ since `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 handling getters and setters, 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 β.getters and β.setters. For each property q that is read or written, we add $\mathsf{FunDef}\langle f, l\rangle \overset{x}{\underset{\alpha}{\sim}} ap'$ to α for each $ap' \in \beta.\mathsf{getters}(q)$ for getters and $ap' \in \beta.\mathsf{setters}(q)$ for setters.

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 *α-prop-call* constraint, we have added a heuristic where the object property summary is only

$$\begin{array}{rcl}
\textit{VulnDescription} & ::= & (\textit{AdvisoryID}, \textit{PackageName}, \\
 & & \textit{VersionRange}, \textit{API-Pattern}) \\
\textit{API-Pattern} & ::= & \{\ \textit{API-Pattern}, \ldots, \textit{API-Pattern}\ \} \\
 & \mid & <\textit{ImportPath}> \\
 & \mid & \textit{API-Pattern}\ \texttt{.}\ \textit{Property} \\
 & \mid & \textit{API-Pattern}\ \texttt{()}
\end{array}$$

Figure 12.6: Vulnerable API detection pattern language. *AdvisoryID* is a number, and *ModuleName* and *VersionRange* are strings.

mixed between directly related packages. With this heuristic, the lookup $\beta.\mathsf{props}(q)$ in α*-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, the experimental results (see Section 12.7) remain sound, and precision is improved.

Soundness Assumptions The call graph analysis is not theoretically sound. Below we list the potential sources of unsoundness [95].

- The analysis ignores dynamic property reads/writes, but since the analysis is field-based, the analysis results are not affected much by this [48].
- The adjacent packages heuristic can result in missing edges if values flow between packages that are not directly linked in the package dependency graph. However, as such flows occur rarely in practice, the analysis result remains sound for practical purposes.
- 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.
- For the export summary generation, we assume that loading a module produces the same result every time. While this assumption is not enforced by Node.js, we have found that it generally holds. The only exception may be some modules where the result is platform dependent, which we do not address in this work.

12.6 Security Scanning using Call Graphs

In order to use the call graph for security scanning, the analysis has to know which call graph nodes represent vulnerable functions. We describe known security vulnerabilities from the npm vulnerability database[15] using a simple pattern language,

[15]`https://www.npmjs.com/advisories`

$$
\mathit{findNodes}(p) \;:=\; \begin{cases} \bigcup_{p' \in \{p_1,\ldots,p_n\}} \mathit{findNodes}(p') & \text{if } p = \{p_1,\ldots,p_n\} \\ \beta_{\mathit{resolve}(m)}.\mathsf{exports}(q) & \text{else if } p \;=\; \text{<m>}\overbrace{\ldots g}^{q} \\ \{\langle f,l\rangle \mid ap \in \beta.\mathsf{props}(q) \wedge ap \underset{\rho}{\sim} \mathsf{FunDef}\langle f,l\rangle\} & \text{else if } p \;=\; p'.q \\ \bigcup_{\langle f,l\rangle \in \mathit{findNodes}(p')} \{\langle f',l'\rangle \mid ap \in \beta_f.\mathsf{returns}(\langle f,l\rangle) & \text{else if } p \;=\; p'() \\ \qquad\qquad\qquad \wedge\; ap \underset{\rho}{\sim} \mathsf{FunDef}\langle f',l'\rangle\} & \end{cases}
$$

Figure 12.7: Algorithm for finding vulnerable functions from API patterns.

and use the function *findNodes* (see Figure 12.7) to convert these patterns to concrete source locations.[16]

The grammar of the pattern language is shown in Figure 12.6. A vulnerability description (*VulnDescription*) 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. An *API-Pattern* can express a disjunction of *API-Pattern*s ({*API-Pattern*, ..., *API-Pattern*}), the return value from loading a module (< *ImportPath* >), the value read from a property (*API-Pattern* **.** *Property*), and the return value of a function (*API-Pattern* **()**). The language of API patterns resembles the language of access paths (Figure 12.2) but is designed for easily identifying API functions, whereas access paths are used only internally by the analysis.

If the application depends on some version of the vulnerable package in the vulnerable version range, then the function *findNodes* is used to find the source locations of the vulnerable functions. It uses the vulnerability descriptions, the module summaries, and ρ 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 *findNodes* for each subpattern. For property read sequences on a module object without any calls, $p \;=\; \text{<}m\text{>}\overbrace{\ldots g}^{q}$, the function source locations are extracted from the exports summary of the module (this rule also applies when q is empty, i.e., $p \;=\; \text{<}m\text{>}$). 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 $\beta.\mathsf{props}$ and then the concrete source locations of these access paths from ρ. For calls to returned values, $p = p'()$, the source locations represented by p' are extracted by calling *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 ρ (similar to the α*-return-call* rule of Figure 12.5).

The security scanner can then check whether these functions are reachable in the call graph from the entry node of the application.[17] If any of the functions are

[16]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.

[17]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.

Table 12.1: Experimental results.

				npm audit		Jam			
Name	**Functions**	**Modules**	**Packages**	**T**	**F**	**T**	**F**	**Full**	**Mod.**
makeappicon	6 165 (623)	1 393 (43)	13 (12)	0	3	0	2	4.15	0.96
toucht	6 479 (54)	1 560 (2)	25 (1)	0	3	0	0	0.70	0.67
spotify-terminal	8 259 (54)	783 (3)	106 (1)	0	4	0	0	0.73	0.67
ragan-module	839 (594)	85 (79)	61 (59)	1	0	1	0	1.44	0.59
npm-git-snapshot	898 (291)	120 (53)	41 (40)	1	0	1	0	1.10	0.52
nodetree	1 557 (142)	15 (8)	4 (4)	0	4	0	0	0.87	0.46
jwtnoneify	27 703 (3 167)	1 869 (194)	93 (61)	0	4	0	3	600.91	2.88
foxx-framework	4 334 (677)	261 (105)	68 (58)	1	0	1	0	2.37	0.85
npmgenerate	1 638 (513)	266 (30)	23 (20)	2	0	2	0	2.35	0.65
smrti	1 228 (731)	121 (115)	64 (62)	1	0	1	0	1.99	0.61
writex	4 177 (864)	187 (89)	53 (49)	1	4	1	0	2.22	0.71
openbadges-issuer	6 043 (602)	1 366 (125)	69 (66)	1	4	1	0	6.00	0.66
Total	69 320 (8 312)	8 026 (846)	620 (433)	8	26	8	5	624.83	10.23

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 3 000 lines of TypeScript code, using *acorn*[18] and *acorn-walk*[19] for parsing JavaScript files and traversing ASTs, and NodeProf [138] for the API discovery analysis. We evaluate the approach by answering the following research questions.

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 *npm audit* approach that is based on package-level dependencies?

RQ2: 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 repository 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

[18]`https://www.npmjs.com/package/acorn`

[19]`https://www.npmjs.com/package/acorn-walk`

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*.

To answer RQ2, 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 Example 42). Finally, we compute the call graph for the entire application using the aggregated call graph of the dependencies.

Our experiments have been run on an Ubuntu machine with a Xeon E5-2697A CPU with 10GB RAM for the analysis.

Results for RQ1 (Precision and Recall)

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 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, "**T**") and alarms about a vulnerable library function that is never used by the application (false positives, "**F**").

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 the remaining 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 the analysis misses 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 69 320 functions, 8 026 modules, and 620 packages (including duplicates used by several applications). According to the computed call graphs, only 8 312 (12%) of the functions, 846 (11%) of the modules, and 433 (70%) of the packages are reachable, which gives an indication of the overall potential of call-graph-based security scanning.

Results for RQ2 (Analysis Time and Modularity)

The “**Full**” column shows the number of seconds it takes JAM to compute the call graphs for the applications including all dependencies.[20] The analysis time varies from less than one second for the *toucht* application to around 10 minutes for *jwtnoneify*. 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 constraint *α-param-call* from Figure 12.5 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 long analysis time and the imprecise call graph, the security scanner is still more precise than *npm audit* for this application.

[20]The NodeProf dynamic analysis framework, which we use for the API discovery analysis, has a startup time of 2–3 seconds. Once the analysis has started, the export summaries are usually generated almost instantly. With further implementation effort it should be possible to optimize this part considerably by using the same NodeProf instance to generate export summaries for multiple modules. For this reason, we exclude the time it takes to generate the exports summary from the reported analysis time. The time spent on the actual security scanning is also excluded since it is negligible once the call graph has been computed.

Nevertheless, investigating this outlier in more detail and improving its analysis time is an interesting challenge for future work.

The "**Mod.**" column in Table 12.1 shows the analysis time in seconds, 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 almost all applications. The only exception is (again) *jwtnonify* where the modular analysis takes 2.88s, but that is still 200× faster than the full call graph analysis.

We conclude that the JAM full call graph analysis is efficient for most benchmarks. For the few exceptions, the modular approach ensures that all benchmarks are analyzed fast, 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 [39, 40, 81, 90, 135, 146, 147, 149, 152]. In particular, Zapata et al. [147] 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. [67] and Hejderup et al. [65] 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 [64] 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 [120] is a similar approach for Java. It has recently been adapted to JavaScript [25], however, that work uses a simple program analysis that ignores most JavaScript language constructs. Mininode [87] 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 [48, 68, 70, 91, 97, 98]. 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., [48, 97, 98]) are fast but tend to miss many call edges, whereas abstract-interpretation-based analyzers (e.g., TAJS [70] and SAFE [91]) do not yet scale to real-world Node.js applications.

As explained in Section 12.5, one of the key components of JAM is the field-based analysis approach by Feldthaus et al. [48]. We also take inspiration from the notion of access paths by Mezzetti et al. [100] to model module interfaces. The modular approach of JAM is inspired by componential analysis [49], 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 [95]. 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 [109] 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. [136] 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,[21] 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 [7, 60] are designed for browser-based applications, not for reasoning about dependencies between modules in Node.js applications.

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.

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 applications of the call graphs, for instance, change impact analysis as well as code navigation and refactoring in IDEs.

[21]`https://lgtm.com/`

Bibliography

[1] Babel. `https://babeljs.io/`, 2019. 165

[2] ES6 Features. `http://es6-features.org`, 2019. 164

[3] Express web framework. `https://www.npmjs.com/package/express`, 2019. 152, 153, 164

[4] MongoDB database. `https://www.mongodb.com/`, 2019. 164

[5] Module Loader in Node.js. `https://github.com/nodejs/node/blob/master/lib/module.js`, 2019. 155

[6] OWASP vulnerable Node.js application. `https://github.com/OWASP/NodeGoat`, 2019. 168

[7] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Hybrid DOM-sensitive change impact analysis for JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 321–345. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi: 10.4230/LIPIcs.ECOOP.2015.321. 233, 251

[8] Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. Combining string abstract domains for JavaScript analysis: An evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017*, volume 10205 of *Lecture Notes in Computer Science*, pages 41–57, 2017. 31, 88, 149

[9] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, 1994. 29

[10] Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *OOPSLA*, 2014. 31, 32, 34, 43, 76, 77, 78, 86, 88, 90, 91, 107, 108, 109, 110, 113, 122, 126, 128, 138, 143, 145, 146, 149, 153, 164, 165

[11] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017*, pages 31–36, 2017. 9, 111, 147, 172

[12] Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage. In *Proceedings of the 13th International Conference on Static Analysis*, SAS'06, page 221–239, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540377565. doi: 10.1007/11823230_15. URL `https://doi.org/10.1007/11823230_15`. 35

[13] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Model Checking Software, 8th International SPIN Workshop, 2001*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 2001. 114

[14] Thomas Ball, Orna Kupferman, and Greta Yorsh. Abstraction for falsification. In *Computer Aided Verification, 17th International Conference, CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2005. 114

[15] Jon Bentley. Programming pearls: Little languages. *Commun. ACM*, 29(8): 711–721, August 1986. ISSN 0001-0782. doi: 10.1145/6424.315691. URL `https://doi.org/10.1145/6424.315691`. 13

[16] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: Precise refutations for heap reachability. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 275–286, 2013. 88, 102, 103, 114, 115

[17] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 196–207. ACM, 2003. 148

[18] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, 2011. 81

[19] Aline Brito, Laerte Xavier, André C. Hora, and Marco Tulio Valente. Why and how java developers break apis. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 255–265. IEEE Computer Society, 2018. 202, 225

[20] Aline Brito, Laerte Xavier, André C. Hora, and Marco Tulio Valente. APIDiff: Detecting API breaking changes. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 507–511. IEEE Computer Society, 2018. 225

[21] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011. 149

[22] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: A powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*, pages 363–374. ACM, 2009. 114

[23] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005*, pages 147–163, 2005. 115

[24] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, PLDI 1990*, pages 296–310. ACM, 1990. 35, 94, 95, 129

[25] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Code-based vulnerability detection in Node.js applications: How far are we? *CoRR*, abs/2008.04568, 2020. 250

[26] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *1996 International Conference on Software Maintenance (ICSM '96), 4-8 November 1996, Monterey, CA, USA, Proceedings*, page 359. IEEE Computer Society, 1996. 203

[27] Maria Christakis, Peter Müller, and Valentin Wüstholz. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *VMCAI*, 2015. 79, 81

[28] Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. Linvail: A general-purpose platform for shadow execution of javascript. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 260–270. IEEE, 2016. 172

[29] Andrey Chudnov and David A Naumann. Information flow monitor inlining. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 200–214. IEEE, 2010. 172

[30] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000. 114

[31] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, POPL 1977*, pages 238–252. ACM, 1977. 25, 28, 95, 130, 152

[32] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages POPL 1979*, pages 269–282, 1979. 114

[33] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992. 94

[34] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2002. 149

[35] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006. 115

[36] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011*, pages 150–168, 2011. 114

[37] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Automatic analysis of open objects in dynamic language programs. In *Static Analysis - 21st International Symposium, SAS 2014*, pages 134–150, 2014. 114, 149

[38] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. *ACM Trans. Softw. Eng. Methodol.*, 20 (4):19:1–19:35, 2011. 203, 224

[39] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the evolution of technical lag in the npm package dependency network. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME*

2018, Madrid, Spain, September 23-29, 2018, pages 404–414. IEEE Computer Society, 2018. doi: 10.1109/ICSME.2018.00050. 233, 250

[40] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 181–191. ACM, 2018. doi: 10.1145/3196398.3196401. 232, 250

[41] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2187–2200. ACM, 2017. 203, 225

[42] Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf. A parallel abstract interpreter for JavaScript. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015*, pages 34–45. IEEE Computer Society, 2015. 108

[43] Danny Dig and Ralph E. Johnson. How do apis evolve? A story of refactoring. *Journal of Software Maintenance*, 18(2):83–107, 2006. 202

[44] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA*, 2007. 45, 76, 81

[45] Mattia Fazzini, Qi Xin, and Alessandro Orso. Automated api-usage update for android apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 204–215. ACM, 2019. 203, 224

[46] Asger Feldthaus and Anders Møller. Semi-automatic rename refactoring for javascript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 323–338. ACM, 2013. 56, 71, 203

[47] Asger Feldthaus and Anders Møller. Checking correctness of TypeScript interfaces for JavaScript libraries. In *OOPSLA*, 2014. 79, 237

[48] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript IDE services. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 752–761. IEEE Computer Society, 2013. 55, 67, 189, 203, 232, 233, 236, 245, 250

[49] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, 1999. doi: 10.1145/316686.316703. 251

[50] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM, 2002. 114

[51] Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*, pages 31–44, 2012. 114

[52] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. Affogato: Runtime Detection of Injection Attacks for Node.js. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, SOAP, pages 94–99. ACM, 2018. 34, 153, 163, 164, 168, 172

[53] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction: delta debugging, even without bugs. *STVR*, 2016. 78

[54] Salvatore Guarnieri and V Benjamin Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *USENIX Security Symposium*, volume 10, pages 78–85, 2009. 172

[55] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA, pages 177–187. ACM, 2011. 171

[56] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Proceedings*, pages 126–150. Springer, 2010. 92

[57] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 256–275. Springer, 2011. 127, 129

[58] Bhargav S. Gulavani and Sriram K. Rajamani. Counterexample driven refinement for abstract interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006*, volume 3920 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2006. 114

[59] Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. *Sci. Comput. Program.*, 58(1-2):83–114, 2005. 114

[60] Quinn Hanam, Ali Mesbah, and Reid Holmes. Aiding code change understanding with semantic change impact analysis. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*, pages 202–212. IEEE, 2019. doi: 10.1109/ICSME.2019.00031. 233, 251

[61] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP, pages 13–18. ACM, 2017. 152

[62] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of javascript. In *Computer Security Foundations Symposium*, CSF, pages 3–18. IEEE, 2012. 172

[63] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. Jsflow: Tracking information flow in javascript and its apis. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1663–1671. ACM, 2014. 172

[64] Joseph Hejderup, Moritz Beller, and Georgios Gousios. Präzi: From package-based to precise call-based dependency network analyses. Working paper, TU Delft, 2018. 250

[65] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. Software ecosystem call graph for dependency management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 101–104. ACM, 2018. doi: 10.1145/3183399.3183417. 250

[66] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70. ACM, 2002. 114

[67] Zoltán Herczeg, Gábor Lóki, and Ákos Kiss. Towards the efficient use of dynamic call graph generators of Node.js applications. In *Evaluation of Novel Approaches to Software Engineering - 14th International Conference, ENASE 2019, Heraklion, Crete, Greece, May 4-5, 2019, Revised Selected Papers*, volume 1172 of *Communications in Computer and Information Science*, pages 286–302. Springer, 2019. doi: 10.1007/978-3-030-40223-5_14. 250

[68] IBM Research. *T.J. Watson Libraries for Analysis (WALA)*, 2018. 113, 172, 250

[69] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–26. ACM, 2001. 101

[70] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *SAS*, 2009. 31, 34, 37, 39, 77, 88, 94, 95, 107, 109, 113, 122, 128, 145, 149, 152, 153, 154, 155, 167, 171, 250

[71] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *SAS*, 2010. 34

[72] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *ESEC/FSE*, 2011.

[73] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *ISSTA*, 2012. 34

[74] Kamil Jezek, Jens Dietrich, and Premek Brada. How Java APIs break - an empirical study. *Inf. Softw. Technol.*, 65:129–146, 2015. doi: 10.1016/j.infsof.2015.02.014. 225

[75] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *POPL*, 2015. 76

[76] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977. 25, 27, 94, 128, 154

[77] Hong Jin Kang, Ferdian Thung, Julia Lawall, Gilles Muller, Lingxiao Jiang, and David Lo. Semantic patches for Java program transformation (experience report). In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom.*, volume 134 of *LIPIcs*, pages 22:1–22:27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. 177, 203, 224

[78] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering*, 2018. 34, 153, 163, 164, 172

[79] Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type refinement for static analysis of JavaScript. In *DLS'13, Proceedings of the 9th Symposium on Dynamic Languages, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 17–26. ACM, 2013. 127, 129

[80] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: a static analysis platform for JavaScript. In *ESEC/FSE*, 2014. 31, 77, 88, 108, 109, 113, 122, 149, 152, 171

[81] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 102–112. IEEE Computer Society, 2017. doi: 10.1109/MSR.2017.55. 23, 232, 250

[82] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, POPL 1973*, pages 194–206. ACM Press, 1973. 25, 28, 95, 129

[83] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. Practically tunable static analysis framework for large-scale JavaScript applications. In *ASE*, 2015. 76, 77

[84] Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. Weakly sensitive analysis for unbounded iteration over JavaScript objects. In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017*, pages 148–168, 2017. 33, 88, 89, 90, 91, 107, 108, 109, 113, 171

[85] Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. Weakly sensitive analysis for JavaScript object-manipulating programs. *Softw., Pract. Exper.*, 49(5):840–884, 2019. 88, 89, 90, 91, 107, 113, 122, 123, 125, 143, 144, 145, 149

[86] Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. Classification of changes in API evolution. In *23rd IEEE International Enterprise Distributed Object Computing Conference, EDOC 2019, Paris, France, October 28-31, 2019*, pages 243–249. IEEE, 2019. 202

[87] Igibek Koishybayev and Alexandros Kapravelos. Mininode: Reducing the attack surface of Node.js applications. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, October 2020. 23, 232, 235, 250

[88] Erik Krogh Kristensen and Anders Møller. Reasonably-most-general clients for javascript library analysis. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 83–93. IEEE / ACM, 2019. 52, 71, 177, 203, 236

[89] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Chen. A4: automatically assisting android API migrations using code examples. *CoRR*, abs/1812.04894, 2018. 224

[90] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. 232, 250

[91] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: formal specification and implementation of a scalable analysis framework for ECMAScript. In *Proc. International Workshop on Foundations of Object Oriented Languages (FOOL 2012)*, 2012. 31, 88, 107, 113, 122, 143, 149, 152, 156, 171, 250

[92] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 270–282, 2002. 115

[93] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. Cid: automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 153–163. ACM, 2018. 203, 224

[94] Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 590–601, 2011. 114

[95] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 2015. 75, 81, 245, 251

[96] Magnus Madsen and Esben Andreasen. String analysis for dynamic field access. In *Proc. 23rd International Conference on Compiler Construction*, volume 8409 of *Lecture Notes in Computer Science*. Springer, 2014. 31, 88, 149

[97] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 499–509. ACM, 2013. 56, 171, 172, 233, 250

[98] Magnus Madsen, Frank Tip, and Ondrej Lhoták. Static analysis of event-driven Node.js JavaScript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 505–519. ACM, 2015. doi: 10.1145/2814270.2814272. 250

[99] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: explaining program failures via postmortem static analysis. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2004*, pages 63–72. ACM, 2004. 114

[100] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Type regression testing to detect breaking changes in node.js libraries. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 7:1–7:24, 2018. 202, 225, 237, 251

[101] Jan Midtgaard and Anders Møller. Quickchecking static analysis properties. In *ICST*, 2015. 76, 86

[102] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005. ISSN 1049-331X. doi: 10.1145/1044834.1044835. URL `https://doi.org/10.1145/1044834.1044835`. 43

[103] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006. 148

[104] Dimitris Mitropoulos, Panos Louridas, Vitalis Salis, and Diomidis Spinellis. Time present and time past: analyzing the evolution of javascript code in the wild. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pages 126–137. IEEE / ACM, 2019. 202

[105] Anders Møller and Martin Toldam Torp. Model-based testing of breaking changes in node.js libraries. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 409–419. ACM, 2019. 71, 177, 184, 204, 225, 233

[106] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. Detecting locations in javascript programs affected by breaking library changes. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi: 10.1145/3428255. URL `https://doi.org/10.1145/3428255`. 10, 206, 209, 217, 218, 219, 225

[107] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 302–321. ACM, 2010. 203, 224

[108] Benjamin Barslev Nielsen and Anders Møller. Value partitioning: A lightweight approach to relational static analysis for javascript. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 16:1–16:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPIcs.ECOOP.2020.16. URL `https://doi.org/10.4230/LIPIcs.ECOOP.2020.16`. 9

[109] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: feedback-driven static analysis of Node.js applications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 455–465. ACM, 2019. doi: 10.1145/3338906.3338933. 10, 236, 251

[110] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA, June 7-8, 2004*, pages 43–48. ACM, 2004. 35, 129, 136, 138

[111] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective x-sensitive analysis guided by impact pre-analysis. *ACM Trans. Program. Lang. Syst.*, 38(2):6:1–6:45, 2016. 115

[112] Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. Semantic patches for documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems, PLOS 2006, San Jose, California, USA, October 22, 2006*, page 10. ACM, 2006. 203, 206, 224

[113] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. SmPL: A domain-specific language for specifying collateral evolutions in linux device drivers. *Electron. Notes Theor. Comput. Sci.*, 166:47–62, 2007. 206, 224

[114] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, pages 247–260. ACM, 2008. 177

[115] Changhee Park and Sukyoung Ryu. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *ECOOP*, 2015. 31, 32, 88, 91, 108, 109, 110, 113, 122, 126, 145, 146, 149

[116] Changhee Park, Hyeonseung Im, and Sukyoung Ryu. Precise and scalable static analysis of jQuery using a regular expression domain. In *Proceedings of*

the 12th Symposium on Dynamic Languages, DLS 2016, pages 25–36. ACM, 2016. 31, 88, 149

[117] Jihyeok Park, Xavier Rival, and Sukyoung Ryu. Revisiting recency abstraction for javascript: towards an intuitive, compositional, and efficient heap abstraction. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP, pages 1–6. ACM, 2017. 171

[118] Joonyoung Park, Inho Lim, and Sukyoung Ryu. Battles with false positives in static analysis of JavaScript web applications in the wild. In *ICSE SEIP*, 2016. 79, 172

[119] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. SPOON: a library for implementing analyses and transformations of Java source code. *Softw., Pract. Exper.*, 46(9):1155–1179, 2016. 225

[120] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 2020. doi: 10.1007/s10664-020-09830-x. 250

[121] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning and impact of breaking changes in the Maven repository. *J. Syst. Softw.*, 129: 140–158, 2017. 225

[122] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5):26, 2007. 91, 122, 130, 131, 148

[123] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Trans. Software Eng.*, 5(3):216–226, 1979. doi: 10.1109/TSE.1979.234183. 232

[124] José Fragoso Santos and Tamara Rezk. An information flow monitor-inlining compiler for securing a core of javascript. In *IFIP International Information Security Conference*, pages 278–292. Springer, 2014. 172

[125] José Fragoso Santos, Petar Maksimovic, Daiva Naudziuniene, Thomas Wood, and Philippa Gardner. JaVerT: JavaScript verification toolchain. *PACMPL*, 2 (POPL):50:1–50:33, 2018. 114

[126] José Fragoso Santos, Petar Maksimovic, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional symbolic execution for JavaScript. *PACMPL*, 3(POPL):66:1–66:31, 2019. 114

[127] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. In *PLDI*, 2013. 77, 81, 122, 126, 149

[128] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE*, 2013. 80, 172

[129] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. pages 189–234, 1981. 28

[130] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 485–495. ACM, 2014. 152

[131] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016*, pages 22:1–22:26, 2016. 114

[132] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, PLDI 2006*, pages 387–400, 2006. 114

[133] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *ECOOP*, 2012. 31, 32, 88, 90, 91, 108, 113, 122, 125, 149, 172

[134] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. *Alias Analysis for Object-Oriented Programs*, page 196–232. Springer-Verlag, Berlin, Heidelberg, 2013. ISBN 9783642369452. 29

[135] C-A. Staicu, M. Pradel, and B. Livshits. SYNODE: Understanding and Automatically Preventing Injection Attacks on Node.js. In *25th Annual Network and Distributed System Security Symposium*, NDSS, 2018. 33, 153, 163, 164, 168, 232, 250

[136] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. Extracting taint specifications for JavaScript libraries. In *Proc. 42nd International Conference on Software Engineering (ICSE)*, May 2020. 232, 251

[137] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. Static analysis with demand-driven value refinement. *Proceedings of the ACM on Programming Languages (PACMPL)*, 3:140:1–140:29, 2019. 9, 122, 124, 125, 143, 144, 145, 146, 147, 148, 149, 177, 203, 236

[138] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for Node.js. In *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, pages 196–206. ACM, 2018. doi: 10.1145/3178372.3179527. 247

[139] Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. Reduced product combination of abstract domains for shapes. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013*, pages 375–395, 2013. 115

[140] Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA, pages 49–59. ACM, 2014. 172

[141] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991. 129

[142] Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for JavaScript. In *ISSTA*, 2013. 76, 79, 81, 172

[143] Shiyi Wei and Barbara G Ryder. Adaptive context-sensitive analysis for javascript. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015. 171

[144] Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. Revamping JavaScript static analysis via localization and remediation of root causes of imprecision. In *ESEC/FSE*, 2016. 76, 77, 79, 84, 108, 109, 122, 149, 172

[145] Allen Wirfs-Brock and Brendan Eich. Javascript: The first 20 years. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020. doi: 10.1145/3386327. URL `https://doi.org/10.1145/3386327`. 13

[146] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 351–361. ACM, 2016. doi: 10.1145/2901739.2901743. 208, 250

[147] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm JavaScript packages. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pages 559–563. IEEE Computer Society, 2018. doi: 10.1109/ICSME.2018.00067. 232, 250

[148] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *STE*, 2002. 44, 77

[149] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesús M. González-Barahona. On the impact of outdated and vulnerable JavaScript packages in Docker images. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 619–623. IEEE, 2019. doi: 10.1109/SANER.2019.8667984. 232, 250

[150] Ahmed Zerouali, Tom Mens, Jesús M. González-Barahona, Alexandre Decan, Eleni Constantinou, and Gregorio Robles. A formal framework for measuring technical lag in component repositories - and its application to npm. *Journal of Software: Evolution and Process*, 31(8), 2019. 176, 203

[151] Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. How do python framework apis evolve? an exploratory study. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER London, Ontario, February 18-21, 2020*. IEEE, 2020. 203, 224

[152] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 995–1010. USENIX Association, 2019. 23, 176, 232, 250