Practical Dynamic Symbolic Execution for JavaScript

Blake William Loring

Submitted in fulfillment for the degree of
Doctor of Philosophy

Information Security Group
Royal Holloway, University of London

Declaration of Authorship

The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Information Security Group as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment. Blake Loring, June 2020

Acknowledgements

To begin, I would like to thank my supervisor Prof. Dr. Johannes Kinder, whose perseverance and guidance was essential throughout my PhD. Next, I would like to thank Duncan Mitchell and Ronny Ko, with whom I had interesting and fruitful research collaborations. I would also like to thank Claudio Rizzo, Roberto Jordaney, and Carlton Shepherd, for their help and advice. I would also like to extend my thanks to Nick Sullivan, and Ben Livshits, of Cloudflare and Brave, for giving me the opportunity to take part in exciting and useful internships. The experience I gained from both of these opportunities was invaluable in subsequent work. Finally, I would like to thank my family, without which I would never have had the time or confidence required to embark on a PhD journey.

Abstract

In this thesis we develop a practical and scalable approach for dynamic symbolic execution (DSE) of JavaScript programs and prove its effectiveness by implementing ExpoSE, our new DSE engine. ExpoSE uses program instrumentation to implement DSE, enabling analysis of both web applications and Node.js software while also allowing quick support for the latest JavaScript standards. We detail novel encodings for regular expressions, objects, and arrays which allow ExpoSE to analyze programs out of reach of prior work. In particular, we present the first complete encoding for ES6 regular expressions, including symbolic support for capture groups and backreferences. We show the effectiveness of our design through two case studies. In the first study we show that our approach is able to generate a suite of supplementary conformance tests for JavaScript standard library methods that further the official JavaScript testing suite Test262. Test cases are generated through symbolic exploration of polyfill implementations and verified with differential testing. In the second case study we use DSE to automatically deduce what conditions trigger resource loading, enabling our new speculative loading approach Oblique, a proxy which reduces page load times by sending resources before a client requests them.

Contents

1 Introduction **1**

1.1 The Problem 5
1.2 Example: A Motivating XSS Vulnerability 7
1.3 An Architecture for Practical Symbolic Execution of JavaScript 9
1.4 Thesis Overview 11
1.5 Previously Published Work 13
1.6 Open Source Work 13

2 Background **15**

2.1 The JavaScript Programming Language 16
2.1.1 The Type System 17
2.1.2 Classes in JavaScript 19
2.1.3 Built-in Objects 23
2.1.4 The Event Loop 23
2.2 Program Analysis 25
2.2.1 JavaScript Analysis Tools 28
2.2.2 Symbolic Execution 30
2.2.3 Example: A Simple Symbolic Execution 30
2.2.4 Dynamic Symbolic Execution 32
2.2.5 Applications of Symbolic Execution 35
2.2.6 Unique Challenges for Symbolic Execution of JavaScript 37

2.3 Satisfiability Modulo Theories and Program Analysis 38
2.4 Source Code Instrumentation 41
2.4.1 Instrumenting JavaScript 41
2.5 Example: A Prototype DSE Engine By Instrumentation . . . 42

3 ExpoSE: Practical Symbolic Execution of JavaScript 47
3.1 Introduction 48
3.2 Example: Symbolic Execution With ExpoSE 50
3.3 General Architecture 53
3.4 Test Case Parallelism 54
3.5 Test Case Isolation 55
3.6 Source Code Instrumentation 55
3.7 Symbolic Values and the Type System 57
3.7.1 Example: Concolic Values in JavaScript 57
3.7.2 Overview 59
3.7.3 Strings 60
3.7.4 Arrays and Objects 62
3.7.5 Untyped Symbols 62
3.7.6 Type Coercions 63
3.8 Modelling Functions 63
3.8.1 Modelled APIs 64
3.9 Supporting Web Analysis 66
3.10 Test Case Selection 67
3.11 Coverage Calculation 69
3.12 Evaluation 71
3.12.1 Application Support 72
3.12.2 Test Case Selection 73

4 Sound Regular Expression Semantics for ExpoSE 81
4.1 Introduction 82

4.2 ECMAScript Regex . . . 84
4.2.1 Methods, Anchors, Flags . . . 84
4.2.2 Capture Groups . . . 86
4.2.3 Backreferences . . . 87
4.2.4 Operator Evaluation . . . 88
4.3 Overview . . . 89
4.3.1 The Word Problem and Capturing Languages . . . 89
4.3.2 Regex In Dynamic Symbolic Execution . . . 90
4.3.3 Modeling Capturing Language Membership . . . 91
4.3.4 Refinement . . . 93
4.4 Modeling ES6 Regex . . . 94
4.4.1 Preprocessing . . . 94
4.4.2 Operators and Capture Groups . . . 96
4.4.3 Backreferences . . . 99
4.4.4 Backreferences and Overlapping Capture Groups . . . 102
4.4.5 Modeling Non-Membership . . . 105
4.5 Matching Precedence Refinement . . . 106
4.5.1 Matching Precedence . . . 106
4.5.2 CEGAR for ES6 Regular Expression Models . . . 106
4.5.3 Termination . . . 109
4.5.4 Soundness . . . 109
4.6 Implementation . . . 110
4.6.1 Using our Encoding in ExpoSE . . . 110
4.6.2 Modeling the Regex API . . . 112
4.7 Evaluation . . . 114
4.7.1 Surveying Regex Usage . . . 115
4.7.2 Improvement Over State of the Art . . . 117
4.7.3 Breakdown of Contributions . . . 119
4.7.4 Effectiveness on Real-World Queries . . . 122
4.7.5 Threats to Validity . . . 124

4.8 Related Work 126

5 Systematic Generation of Conformance Tests 129
5.1 Introduction 130
5.2 Conformance Testing using PolyFills 132
5.2.1 Polyfills 132
5.2.2 Architecture 133
5.2.3 Test Case Generation 133
5.2.4 Test Case Executor 135
5.3 Representing Symbolic Data Structures in JavaScript 136
5.3.1 Motivation 137
5.3.2 Symbolic Objects 138
5.3.3 Mixed Type Arrays 142
5.3.4 Optimized Support for Homogeneous Typed Arrays 145
5.4 Evaluation 147
5.4.1 Test Case Generation 148
5.4.2 Executing Our Test Cases 149
5.4.3 Test Suite Coverage 152
5.5 Related Work 157

6 Accelerating the Web with Symbolic Execution 161
6.1 Introduction 162
6.2 Proxies and Speculative Loading 164
6.3 Overview 166
6.4 Analysis of Client-Side JavaScript 166
6.4.1 Symbolic Execution of Web Applications 167
6.4.2 Detecting Resource Dependencies 168
6.4.3 Example: Find Resource Dependencies with ExpoSE 170
6.5 Proxy Implementation 173
6.6 Example: Oblique in Action 174

6.7 Evaluation 175
6.7.1 Methodology 176
6.7.2 Browser Performance 177
6.7.3 Performance of ExpoSE 179
6.7.4 Conclusions 181

7 Conclusions **183**

Bibliography **187**

List of Figures

2.1 An illustration of source code instrumentation. 42
2.2 A prototype DSE engine by instrumented callbacks. 45

3.1 The ExpoSE architecture. 53
3.2 ExpoSE search strategy performance on `minimist`. 76
3.3 ExpoSE search strategy performance on `body-parser`. . . . 77
3.4 ExpoSE search strategy performance on `validator`. 78

5.1 Test Case Generator Overview. 134
5.2 Test Case Executor Overview. 136
5.3 Illustration of symbolic object modeling. 139
5.4 Illustration of symbolic array modeling 143

6.1 An overview of the approach architecture. 167
6.2 Our new ExpoSE architecture. 169
6.3 The page load times of our evaluated webpages when compared to a standard browser. 177
6.4 The page load times of our evaluated webpages when compared to a standard browser, Vroom, and RDR. 178
6.5 A cumulative distribution of Oblique, Vroom and RDR. . . . 180
6.6 The distribution of cache misses across websites in our evaluation. 180

List of Tables

3.1 Instrumented language operations. 56
3.2 Type Coercion Rules in ExpoSE. 64
3.3 Compatibility with the top 1,000 most depended libraries on NPM. 73
3.4 Total coverage for each search strategy. 74

4.1 JavaScript regular expression API. 85
4.2 Regular expression operators, separated by classes of precedence. 88
4.3 Models for regex operators. 96
4.4 Modeling backreferences. 100
4.5 Regex usage by NPM package. 116
4.6 Feature usage by unique regex. 116
4.7 Statement coverage with our approach (**New**) vs. [79] (**Old**) and the relative increase (**+**) on popular NPM packages (**Weekly** downloads). **LOC** are lines loaded and **RegEx** are regular expression functions symbolically executed. 118
4.8 Breakdown of how different components contribute to testing 1,131 NPM packages, showing number (**#**) and fraction (**%**) of packages with coverage improvements, the geometric mean of the relative coverage increase from the feature (**Cov**), and test execution rate. 121

4.9 Solver times per package and query. 123

5.1 Automatically generated test cases by built-in method. . . . 149
5.2 Test case summaries for 5 built-in implementations. 150
5.3 Branch coverage for systematically generated conformance tests and Test262 at a call depth of 3. 156
5.4 Coverage improvements of automatically generated tests at various call depths by built-in method implementation. . . . 157

6.1 A resource dependency table after symbolic execution. . . . 172

1 Introduction

The JavaScript programming language is widely used. Created as a scripting language for early websites, it quickly became the only widely supported language by browsers. Later, the release of Node.js popularised stand alone JavaScript interpreters, allowing for JavaScript programs to run outside of the browser. The convenience of using a single language for frontend and server-side software increased popularity further, with tools like Node.js now powering many high-profile systems [59].

The monopoly in web browsers and entrance into other domains has increased the demand for safe and well-tested JavaScript, but many design choices, such as the permissive type system, make reasoning about JavaScript challenging. Objects act as maps, without a predefined structure, which can be mutated after creation and arrays are not required to be homogenously typed. For example, `let x = {}; x.a = "hello";` creates a new object and then adds a new field, and `[1, false, "Hello"]` is a legal array. JavaScript interpreters accept any combination of types for operands and use type coercion rules to convert them to compatible types which leads to unintuitive errors with origins that are hard to diagnose. For example, `5 + "5"` will evaluate to `"55"`, but `5 - "5"` evaluates to `0` because adding a string to a number coerces the number to a string, but subtracting it coerces the string to a number. Coercion rules add complexity when reasoning about programs, since a developer must consider how types may be coerced at each step.

Dynamic code evaluation also makes a program hard to analyze, since the entire source code cannot be known prior to execution. In JavaScript, a program loads dependencies at runtime rather than concatenating them during compilation. These additions are done using keywords like `eval`, and `require`, or adding a new `<script>` tag to the DOM when executing inside a web browser. Code added at runtime can be generated dynamically (e.g, `eval("alert(\"Hello\")")`), or loaded from a dynamic resource (e.g., `require(window.userAgent + ".js")`) so we cannot know

what code a program will run before execution. The unpredictable structure of programs makes it difficult to reason about programs statically, since intuition about what code will be loaded by the runtime may be incorrect.

To add to these issues, laissez-faire attitudes toward quality controls on package repositories like NPM have led to the language ecosystem becoming a wild-west, with widely used packages containing malware, bugs, and poorly documented features [96, 60, 134].

The variable quality and the ease with which bugs can be added to programs motivates the creation of automated analysis tools for JavaScript. Unfortunately, the complexities of JavaScript make it difficult to analyze a program statically. A static analysis looks at the program source code or artefact in order to make decisions about correctness and highlight errors. For a static analysis to be effective, the code which will be loaded by the program needs to be known, but the dynamic nature of JavaScript does not allow this. Additionally, type coercions, objects, and arrays make the number of states in a nontrivial program infeasible for thorough static analysis.

A dynamic program analysis is not limited by the addition of code at runtime, and does not need to reason about the entire program to produce output. In a dynamic analysis, runtime information is used to decide on program correctness while a program is executed. A version of the program is created which includes runtime assertions that fail if a program is misbehaving. These assertions may be added automatically, such as bounds or type checks, or manually added by a developer to enforcing complex rules. Since the code is executed using a standard interpreter, we know that any reported errors are real. The downside of a dynamic analysis is that only the control flows exercised are considered, rather than the entire program, so errors may be overlooked.

Unit testing is one example of dynamic analysis. A developer manually creates a test case for each distinct behavior of the program that they wish to test, and includes a set of passing conditions through runtime assertions. These test cases can then be executed by an automated testing framework to make sure that the program still behaves as expected after the code has been changed. Unit testing requires that a developer manually creates a test case for each behaviour they wish to test. Since the creation of unit tests is a manual process suites often do not test a program exhaustively.

Program fuzzing is an alternate dynamic analysis approach that allows for fully automated testing of programs [121]. In a fuzzer a program is repeatedly re-executed with random inputs in order to trigger runtime errors. Fuzzing does not require manual test case creation, but the use of fully random inputs means that errors trigerred by non-trivial interactions between inputs are often overlooked. Additionally, since test case generation does not consider control flow when generating new inputs, many test cases will follow identical execution paths.

Dynamic symbolic execution (DSE) is a dynamic program analysis approach that removes the burden of manual unit test creation without the limitations of program fuzzing by exploring unique control flows in a program automatically. First, some inputs to the program are marked as symbolic, and all others are fixed. Then, a DSE engine generates a series of test cases by executing the program concretely, collecting the program trace during execution and using an SMT solver to discover alternate assignments for the symbolic inputs which would have driven the program through a different control flow. This process is repeated until all inputs are exhausted or a limit is reached. Unlike fuzzing, DSE does not execute multiple paths that explore the same control flow. Instead, each new test case in is guaranteed to execute a unique control flow.

Dynamic symbolic execution has seen success as a tool for finding common errors, such as use after free errors in low-level programming languages, since the straightforward error model for these bugs make DSE easily applicable. The success in such domains has led to many mature DSE engines for C such as KLEE [21], and DART [49], or for direct symbolic execution of binaries through tools like Angr [116]. The same engines could be used to symbolically execute interpreters while they execute a program, but the complexity of modern interpreters makes it infeasible to symbolically execute language interpreters usefully [20]. By building custom DSE engines specific to a language we can leverage type information and knowledge of language semantics to improve the performance of symbolic execution, making engines more practical.

There have been several DSE engines for JavaScript, but these cannot explore programs written in modern versions of the language. No previous symbolic execution engine has supported ES6, ES7, symbolic objects, or included full support for regular expressions. Additionally, prior work has often not supported asynchronous events, an essential element for real-world JavaScript analysis. Given these limitations with the previous state-of-the-art, we developed ExpoSE, a new DSE engine for JavaScript with a design focused on practical symbolic execution.

1.1 The Problem

There is a vast amount of JavaScript in the wild but existing testing frameworks for JavaScript are insufficient when testing real-world applications written for newer versions of the language. Our goal is to extend dynamic symbolic execution to current real-world JavaScript so that reliability and security of JavaScript in the world can be improved. In order to build an analysis framework that can scale to real JavaScript we need to address

the following issues:

- **Dynamic Typing**: The permissive dynamic type system requires careful modeling since a direct encoding of type-coercion rules is too complex for current constraint solvers.
- **Strings and Regular Expressions**: JavaScript programs make heavy use of strings and regular expressions, so models for these language features are required to meaningfully analyze most programs.
- **Objects and Arrays**: Objects and arrays are widely utilized, either as inputs to programs and libraries directly or through decoding of JSON, a common object serialization format. Symbolic support for objects and arrays is essential, but the lack of strict object schemas and support for heterogeneous arrays makes encoding challenging.
- **Language Compatibility**: JavaScript is revised frequently, and the bulk of current real-world code is now out of reach of previous engines.
- **Runtime Code Evaluation**: Dependencies cannot be known ahead of time so a symbolic execution framework must be capable of handling the addition of code during program execution.
- **Asynchronous Design**: JavaScript programs have an event driven concurrency model. Here, unfaithful treatment may cause flag errors incorrectly and lingering events may impact the results of future analysis.
- **Multiple Interpreters**: There is a large diversity in JavaScript deployments but analysis tools usually focus on a single implementation, limiting their utility.

- **Scalability**: Dynamic symbolic execution is expensive, since it requires both SMT solving and repeat program execution, but existing testing frameworks for JavaScript do not scale to take advantage of available resources.

1.2 Example: A Motivating XSS Vulnerability

We now present the following example to illustrate some complexities of JavaScript and the challenges faced when building a DSE engine for it. This website includes a cross-site scripting (XSS) vulnerability when serving users of the Chrome browser. XSS vulnerabilities are dangerous because a request can be shaped to execute arbitrary JavaScript on the client. With an XSS exploit an attacker can turn a link to the legitimate website into an attack that steals anything served to the user, including login tokens. When analyzing this example, our goal is to identify if this potential XSS exploit is reachable in practice. To discover this automatically, we must reason about asynchronous events, strings, and regular expressions.

```
1  <html>
2  <head>
3  <script src="./other-file.js" />
4  <script>
5  function getParameterByName(name) {
6   name = name.replace(/[\[\]]/g, "\\$&");
7   var regex = new RegExp("[?&]"+name+"(=([^&#]*)|&|#|$)");
8   var results = regex.exec(window.location.href);
9   if (!results) return null;
10  if (!results[2]) return "";
11  return decodeURIComponent(results[2].replace(/\+/g, " "));
12 }
13 </script>
14 </head>
15 <body>
```

```
16  ...
17  <script>
18  (function() {
19   setTimeout(() => {
20    var isChrome = /Chrome\/([0-9]+)/.exec(navigator.userAgent);
21    if (isChrome && isChrome[1] == "78" && getParameterByName("chrome_extra"
          )) {
22     eval(getParameterByName("chrome_extra"));
23    }
24   }, 0);
25  })();
26  </script>
27  ...
28  </body>
29  </html>
```

Our program includes a simple XSS vulnerability through an evaluation of a portion of the URL as JavaScript, `eval(getParameterByName("chrome_extra"))`. Because of this, navigating to `website.com?chrome_extra=alert("aah")` would print a message to the screen when visiting with a vulnerable browser. When analyzing this program, our goal is to decide if this line can be reached without constraints restricting `"chrome_extra"` to ensure it is not malicious.

We start by looking at the conditions required to reach this point in the code. We first notice that the triggering method is wrapped by `setTimeout(...,0)`, which schedules a callback for immediate execution. The purpose of immediate timeouts is to execute a block of code to be executed after initialization by scheduling it immediately in the event loop. In JavaScript, there is no facility for pre-emptive multitasking. Instead, the execution of methods is scheduled by an event loop. The event loop is widely used in real-world software so any automated analysis framework for JavaScript will need to ensure it is handled correctly.

Next, we take a look at the callback function that will be triggered when

the timeout event executes. A regular expression, `/Chrome\/([0-9]+)/.exec(navigator.userAgent)`, is used to identify if the browser is a Chrome instance and extract the version number. If the browser is Chrome version 78 then the query string parameter `"chrome_extra"` is extracted and executed as JavaScript. The query parameter is extracted from the query portion of the URL using `getParameterByName`, which contains another regular expression to extract the parameter. Regular expressions are string matching tools that test whether a string matches a specified format, and, if it does, extract relevant portions of it. Regular expressions are heavily used in JavaScript, appearing in 35% of the packages on NPM (Chapter 4). Unlike classical regular expressions, which can be expressed as deterministic finite automaton, regular expressions in JavaScript may be non-regular due to their support for backreferences.

To detect the vulnerability automatically, our analysis framework will need to correctly handle the type system, event loop, and also support strings and regular expressions with capture groups. As such, even in this simple program, the complexities of JavaScript make automated analysis challenging. In real-world applications issues are generally non-obvious, so manual analysis is impractical. With an automated analysis framework obscure bugs and vulnerabilities can be identified automatically and flagged for developer review.

1.3 An Architecture for Practical Symbolic Execution of JavaScript

JavaScript implementations are diverse but generally implement the core specification laid out in the ECMAScript standard [38] while also including some implementation specific features. Previous symbolic execution engines for JavaScript tethered themselves to a single implementation,

limiting the reach of the engine to a small subset of JavaScript. Additionally many of these engines are closely tied to specific interpreter versions, limiting their ability to upgrade when newer versions of JavaScript are released. We aim to solve this with our new architecture for symbolic execution of JavaScript which separates the analyser into three components, a distributor, executor, and solver. To avoid tying engines to specific JavaScript interpreters we represent symbolic state through a generalized symbolic core and rewrite programs to use it through program instrumentation. Through this we decouple the symbolic core from the interpreter specific frontend, which allows an engine to support multiple JavaScript interpreters using the same symbolic interpreter. Where implementations differ we can include implementation specific models or custom instrumentation rules through the interpreter specific frontend. Additionally, our architecture can support different language versions without change to the symbolic core through custom instrumentors. We define a state transfer mechanism which requires communication only at the start and end of test case execution. Using this mechanism we can isolate test cases by executing them in independent processes. The straightforward interface of the transfer mechanism also allows us to execute many test cases in parallel.

To demonstrate this architecture we develop ExpoSE, the first DSE engine for JavaScript which supports both Node.js and web browser-based symbolic execution. ExpoSE is a state of the art dynamic symbolic execution engine for JavaScript with symbolic support for complex data types, including strings, arrays, and objects. To show that our approach scales to software in the wild we present two case studies. Through these case studies, we show that our approach is practical when symbolically executing real world JavaScript. We also demonstrate that symbolic execution for JavaScript has novel uses outside of bug finding.

1.4 Thesis Overview

The contributions in this thesis center around making symbolic execution more practical for real world JavaScript through a new architecture and novel encodings for JavaScript regular expressions, arrays, and objects. We then demonstrate that a symbolic execution engine for JavaScript has novel uses outside of test case generation. Chapter 2 covers JavaScript, program analysis, and SMT, the background knowledge required to develop a DSE engine for JavaScript. In the next two chapters, Chapter 3, and Chapter 4, we describe our architecture for practical symbolic execution and detail a novel encoding approach for extended regular expressions. The final two chapters, Chapter 5, and Chapter 6, demonstrate that a practical symbolic execution engine for JavaScript has uses outside of traditional error checking and demonstrate that our strategy can be applied to real-world code. In this thesis we present the following contributions:

(**C1**) An architecture for scalable symbolic execution of JavaScript with a straightforward interface to add new interpreter frontends and an interpreter agnostic core. We demonstrate this architecture through ExpoSE, a DSE engine which can symbolically execute Node.js programs and web pages (Chapter 3).

(**C2**) An encoding approach for extended regular expressions where matching operations are represented through string equality, concatenation, and regular constraints, resolving ambiguity through a CEGAR refinement scheme. We use this to develop the first complete encoding for ES6 regular expressions, a widely used JavaScript feature (Chapter 4)

(**C3**) A practical encoding for JavaScript arrays and objects that enables straightforward symbolic modeling of concrete field lookups and under-approximate analysis of symbolic field lookups (Chapter 5).

(**C4**) A method for systematically testing specification conformance of multiple black-box standard library implementations by symbolically executing white-box implementations and deciding on correctness through differential testing. We use this approach to automatically generate supplementary test cases for Test262 which improves test coverage and find 17 bugs in `mdn-polyfill` (Chapter 5).

(**C5**) A method for finding dynamic dependencies within webpages through symbolic execution of page-loads with symbolic request headers. We demonstrate that this approach can be used to improve resource dependency resolution proxies (Chapter 6).

Digging down further, in Chapter 2 we motivate the creation of dynamic analysis tools for JavaScript and introduce dynamic symbolic execution using satisfiability modulo theories, demonstrating how these tools can explore programs automatically. In Chapter 3, we introduce the overall architecture of our symbolic execution engine and describe the modeling and exploration strategies used. We show that ExpoSE is capable of analyzing current JavaScript through two experiments, each evaluating a unique component of the engine. In Chapter 4, we present a novel approach to represent JavaScript regular expressions in modern SMT solvers. Here, we formalize an encoding for ES6 regular expressions. Our specification includes encodings for backreferences, assertions, and capture groups. In this work, we also address the issue of matching precedence in regular expressions. This issue arises because the order of matching can impact behavior in extended regular expressions. In Chapter 5, we use DSE to generate supplementary conformance tests for the JavaScript standard library, using several JavaScript implementations of standard library methods to drive interesting test case generation and differential testing as an oracle for test case correctness. We use the new test suite to supplement Test262, a JavaScript implementation conformance test suite, finding

bugs in a popular standard library implementation and improving overall coverage. In our second case study, Chapter 6, we use ExpoSE to find resource dependencies in web applications, enabling a novel constraints based approach to resource dependency resolution proxying.

1.5 Previously Published Work

Work in this thesis has also appeared in the following works:

1. ExpoSE was first introduced in: Blake Loring, Duncan Mitchell, and Johannes Kinder. "ExpoSE: Practical Symbolic Execution of Standalone JavaScript". In: *Proc. Int. SPIN Symp. Model Checking of Software (SPIN)*. ACM, 2017, pp. 196–199, a short paper covering the engine design and an initial support for regular expressions (C1).

2. Our ES6 regular expression encoding was published in: Blake Loring, Duncan Mitchell, and Johannes Kinder. "Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 425–438 (C2).

3. Our work on web acceleration using DSE will appear in NSDI 2021 (C5).

1.6 Open Source Work

We have published many of the tools developed through this work open-source, so that they can be inspected and further expanded by interested parties. These tools are all designed for natural extension, and we hope

that they will form the foundation of future research and practical DSE applications.

- ExpoSE: A Dynamic Symbolic Execution engine for JavaScript [1].
- Z3Javascript: A modification of the Z3 symbolic execution engine which a user can incorporate into Node.js and Electron applications. This tool comes with automatic support for regular expression encoding, including our CEGAR refinement strategy [2].
- A static analysis tool for JavaScript which is capable of quickly scanning NPM packages for feature usage [3].
- A conformance test generator and evaluator for JavaScript standard library methods, which we used to find bugs in several polyfill packages [4].
- A resource dependency development tool, which can take a website address and use ExpoSE to automatically which request header constraints will result in a resource loading [5].

[1] https://github.com/ExpoSEJS/
[2] https://github.com/ExpoSEJS/z3javascript
[3] https://github.com/ExpoSEJS/PLDI-Artifact
[4] https://github.com/ExpoSEJS/conformance_test_generator/
[5] https://github.com/ExpoSEJS/ExpoSE/tree/features/browser/

2 Background

We begin by discussing JavaScript, focusing on elements of the language which have inhibited meaningful program analysis in prior work, such as prototypal inheritance, the dynamic type system, and regular expressions. Once we have introduced JavaScript, we move onto an introduction to program analysis and dynamic symbolic execution. Next, we address how satisfiability modulo theories (SMT) can be used to represent the logic of a program trace, and how we can use this in program analysis. Finally, we discuss source code instrumentation and how it can be used to enable symbolic execution.

2.1 The JavaScript Programming Language

To understand why JavaScript analysis is challenging, we first take a close look at the language. JavaScript's original use-case, the web browser, heavily influenced its design. Netscape introduced JavaScript as a scripting language for web pages in 1995, but the initial scope of the language was limited, with a focus on simple tasks inside the browser. Since then the scope of the language has been expanded considerably but the core design has remained the same. JavaScript is a dynamic language, enforcing type rules during execution. The language includes a permissive type system that marshals conflicting types, rather than halting with runtime errors. This type system is often cited as a weakness because type errors quickly propagate making them difficult to diagnose. The initial specification was informal, with no standardization between vendors, but in 1996 ECMA began formally specifying JavaScript [38].

The features which make JavaScript difficult to manually reason about also limit program analysis. In this section, we describe the features of JavaScript which hinder meaningful program analysis and how language revisions can impact the development of analysis tools.

Node.js Node.js is a JavaScript framework based upon the Google V8 JavaScript engine. It is popular and often touted for the time-saving benefits of having a single codebase spanning the client and server-side. Distributions of Node.js usually include NPM, a JavaScript package manager. NPM is the largest repository of non browser-based JavaScript, with over half a million open source packages. Node.js and NPM enjoy a near-monopoly on browserless deployments of JavaScript. NPM has a laissez-faire attitude towards quality controls, where anybody can publish a package with little vetting. The lack of oversight has led to a language ecosystem renown for error-prone applications, unintuitive packages, questionable long term reliability, and malware [96, 60, 134, 95, 83].

Future JavaScript ECMA publishes revisions of the JavaScript specification frequently. These updates have included substantial syntax changes, such as arrow function, classes, and promises [38]. Constant revision to the specification is a challenge for any practical program analysis tool, which has to support each new feature in order to analyze any new programs. Evidence of this can be seen in prior work, which can only analyze programs written in previous versions of the language [111, 73, 108, 115].

2.1.1 The Type System

Enforcement of type rules can be either static or dynamic. Statically typed languages enforce correct typing during program compilation, typically by following explicit type annotations in the source code. Static type systems are often considered safer, since a compiler can verify that there are no violations of type rules ahead of time, but add extra work for developers through explicit type annotation in source code. Dynamically typed languages, such as JavaScript, do not include static type annotations in the source code and only enforce type rules during program execution. The re-

laxed enforcement increases the flexibility for the developer, enabling programming paradigms that would be impractical in statically typed programming languages.

In the ES6 standard the JavaScript type system allows for values to be one of six different types [38]. Unlike other languages, JavaScript has two distinct representations of emptiness, `undefined`, and `null`. The boolean type can either be `true` or `false`. The values of the type `number` are represented internally as 64-bit IEEE floating point values or 32 bit integers. Developers are given little control over the internal representation of numbers but can force coercion through bitwise operations which truncate numbers to integers. Strings are built-in and come with support for a variety of common operations, such as indexOf and regular expression matching. Objects are a reference type which map string keys to JavaScript values. Arrays behave similarly to objects, but permit numeric and string field names and have additional utility methods. Functions are special executable instances of objects and are first-class; i.e., they can be treated as values, stored in objects and passed as function arguments or results. The final type, Symbol, is a more recent addition to the specification. We initialize each Symbol with an identifier, but each new instance is unique in the program. As these symbols are unique, the equality operator will only return true when referencing the same Symbol, even if the compared one that initialized with the same key. Symbols are often used to ensure the correct behavior of field lookups in programs where type-coercion may introduce ambiguity.

Despite constant revisions, the need for backward compatibility ensures that the language remains burdened by its informal roots. The dynamic type system is one component of the language where these issues are evident. While recent revisions have reduced developer burdon through features like async/await and classes, the type system still relies on unintuitive type-coercions. The language rarely generates exceptions due to

typing, instead coercing values to common types using a set of built-in rules. These rules often lead to unintuitive code. For example, `"5" - 5 == 0`, but `"5" + 5 == "55"`. The specification is also littered with historical artefacts, such as `typeof(null) == "object"`.

The language specification defines the coercion rules `ToBoolean`, `ToNumber`, `ToString`, and `ToObject`. These rules are used throughout the specification when defining operation behaviour or standard library methods. Algorithm 1 and Algorithm 2 detail the coercion rules of any value to a boolean or a string. In these examples we see that the specification covers conversion from most types automatically, and usually does not throw an exception. The full set of type coercion rules can be found in the ECMA specification [38].

Some attempts have been made to address the issues with JavaScript. Languages such as TypeScript [129] and Dart [33] are based upon JavaScript, but introduce optional static typing. Such systems often lead to less error-prone code when employed effectively; however they introduce many of the pain points associated with statically typed languages, and the existence of `any` types allows developers negate the benefits. Using a new language can increase development overhead as all developers need to learn the new syntax and design paradigms. Other approaches, such as Flow [41] make less significant changes to the language and attempt to use type inference to identify typing errors. Unfortunately, due to the complexity of the JavaScript type coercions, Flow is generally imprecise and often unable to produce useful output.

2.1.2 Classes in JavaScript

In JavaScript an objects schema is not fixed and new fields can be added at any time. Instead of fixed classes which dictate what fields must exist on an object, class constructors in JavaScript are as utility functions, taking

Algorithm 1: ToBoolean(argument)

```
1  if argument is undefined then
2  |  return false
3  if argument is null then
4  |  return false
5  if argument is boolean then
6  |  return argument
7  if argument is number then
8  |  if argument is +0, -0 or NaN then return false;
9  |  return true
10 if argument is string then
11 |  if argument.length == 0 then return false;
12 |  return true
```

a fresh object and adding the desired fields. Here, prototypal inheritance can be used to define the initial fields of a new object.

Class constructors are functions. When we call a method with the `new` keyword the JavaScript interpreter will initialize a new object instance. It then executes the method with the `this` value referencing the created object. Upon termination of the function the new object is returned. For example, the following program constructs a new object using the method `A`:

```
1  function A(arg) {
2    this.arg = arg;
3  }
4  var a = new A("Hello");
5  console.log(a.arg);
```

In this program `A` is defined as a new function and the variable `a` is initialised to a new instance of the class constructed with the argument `"Hello"`. The value of `a.arg` is then printed. Here the program prints `"Hello"`.

There is a second way we can define class fields. In JavaScript, the in-

Algorithm 2: ToString(argument)

```
1  if argument is undefined then
2  |  return "undefined"
3  if argument is null then
4  |  return "null"
5  if argument is boolean then
6  |  if argument is true then return "true";
7  |  return "false"
8  if argument is number then
9  |  if argument is NaN then
10 |  |  return "NaN"
11 |  if argument is -0 or +0 then
12 |  |  return "0"
13 |  if argument is less than 0 then
14 |  |  return "-" + ToString(-argument)
15 |  if argument is +∞ then
16 |  |  return "Infinity"
17 |  return PositiveRealNumberToString(argument);
18 if argument is string then
19 |  return argument
20 if argument is Symbol then
21 |  throw TypeError
22 if argument is Object then
23 |  return ToString(ToPrimitive(argument))
```

terpreter finds object fields by walking along the prototype chain, a linked list of objects. Every constructor function has a prototype field which propogates to new instances. If the interpreter looks up a field and it does not exist it will walk through the chain of prototypes until it finds the field or reaches a `null` prototype. If the whole prototype chain is explored and the field does not exist then the interpreter returns `undefined`. This can be used to predefine fields outside of a constructing method:

```
1  A.prototype.field1 = "Hello";
```

```
2  let a = new A();
3  console.log(a.field1);
```

In this example, we add the field `field1` to our classes prototype, construct a new instance of `A`, and then print the `field1` value, which will print `"Hello"`.

We can simulate object inheritance through the manipulation of these prototypes. Take the following example:

```
1  A.prototype.getArg = function() {
2    return this.arg;
3  }
4
5  function B() {
6    B.prototype.constructor.call(this, "Hello");
7  }
8  B.prototype = Object.create(A.prototype);
9  let b = new B();
10 console.log(b.getArg());
```

In this example the prototype of `B` is chained to the prototype of `A` through the method Object.create. As we construct a new instance of `B` the class constructor calls the next constructor on the prototype chain, which is `A`, with the argument `"Hello"`. After that, we call `b.getArg()`, walking the prototype chain and finding `getArg` in the prototoype of `A`. We execute this function with `b`, which has `this.arg` set to `"Hello"` by the constructor `A`.

Objects and prototypes can be modified at any point during runtime, including standard library built-ins, and we can even modify the base `Object` prototype to introduce new values to all objects in a program. This flexibility introduces ambiguity when inspecting a piece of JavaScript as it is often difficult to identify where a change is introduced and what fields will exist on new instances. The prototype inheritance chain is particularly troublesome when analyzing JavaScript statically due to the dynamic nature of changes.

Prototypal inheritance is still used in classes constructed through the

`class` keyword introduced in 2015, as the `class` keyword is just syntactic sugar.

2.1.3 Built-in Objects

The ECMA specification details every facet of the core language, including a set of built in objects and methods. Many of these built-in objects have intuitive use-cases, such as `Function`, `String`, and `Object`, which provide methods for built-in types. Others, such as `Math`, and `JSON`, act as namespaces for built-in utility methods. For example, the method `JSON.parse` converts a JSON encoded string into an object. Later specifications add objects like `Promise`, which standardizes asynchronous behaviors and paved the way for the now standardized `async` keyword, and `UintArray`, which allow for control of variable sizes.

While the standard strictly defines method behaviour, implementation is left to the vendor. This can lead to differences in behavior when comparing different implementations. Historically, developers have mitigated the impact of these inaccuracies through handwritten software checks, however as client-side JavaScript has become more integral to web applications these mitigations have become impractical. Instead, developers now use JavaScript transpiler frameworks to rewrite JavaScript software for maximum compatibility with browsers automatically. The use of these tools allows developers to maintain a high level of browser compatibility while developing large applications.

2.1.4 The Event Loop

JavaScript does not support multi-threaded concurrency, so programs cannot load resources in a background thread, but synchronous blocking operations (e.g., a network request) can prohibit the browser from rendering. To solve this, JavaScript is structured asynchronously, with programs split

into short blocks which execute when an event is triggered. Whenever a block is not executing, the interpreter will check for triggered events and yields any remaining CPU time to other components, such as the browser renderer. Event callbacks are initialized during program startup, but can be modified at any point during execution. For example, the following program updates a HTML element every second:

```
1 var x = 0;
2 setInterval(() => {
3   x += 1; document.body.innerHTML = x;
4 }, 1000);
```

Here, we use the function `setInterval` to call an event handler repeatedly at an interval of 1000ms. `setInterval` is called during initialization but the callback it is supplied with is called by the event loop each time the interval event triggers. If we instead used a while loop to achieve the same thing, then the view in the browser would not be updated, since JavaScript would always be running and the browser renderer would not have time to repaint the page. Since blocking operations would cause responsiveness issues, asynchronous events are often chained together to form complex behaviours. For example, in the following program, a button click event will trigger a network request, which will update the page contents when it is completed:

```
1 document.getElementById("btn").onclick = function() {
2   request("data").then((data) => {
3     document.getElementById("content").innerHTML = data;
4   });
5 };
```

Modern JavaScript also supports the `async` and `await` keywords, which allow developers to write asynchronous programs in an imperative style. The `async` function tells the interpreter that a function is asynchronous, and will not terminate immediately. The `await` keyword halts the current function execution, and waits until an asynchronous `Promise`

has finished before resuming. These keywords rely on the `Promise` interface. A `Promise` is a standard interface for sequential asynchronous behaviour which allows for chaining and error handling. When executing an asynchronous method, statements are executed sequentially, but other blocks may execute while waiting for asynchronous methods to complete. Under the hood, the `async` keyword rewrites a method into a series of asynchronous calls, passing the remainder of the function to be run into a callback each time the `await` keyword is used.

2.2 Program Analysis

Automated program analysis encompasses all techniques that automatically deduce properties of computer programs. In some cases it is sufficient to only reason about simple properties of a program, such as checking for specific feature usage, while in other cases, like full program verification, an exhaustive analysis of the possible routes through a program is required. The field has developed a wide variety of approaches to deal with these varying demands.

Program Analysis Approaches Program analysis approaches fall into one of two categories, static or dynamic. A static analysis approach operates on program code, reasoning about the entire program. In general, full program verification through static analysis is undecidable [72] which limits its use when analyzing real-world software. A static analysis will approximate the behaviour of complex portions of a program in order to keep reasoning feasible, leading to over-approximation. In an over-approximate analysis false-positives can occur, manifesting as reports of impossible errors or infeasible control flows. There are a wide variety of different approaches to static analysis, ranging from program linting and

type rule enforcing all the way to full program verification.

A dynamic analysis instead checks program properties through behaviour observed during execution. Such analysis will not produce false positives since the analysis does not approximate behaviour. But a dynamic analysis can only report on observed errors, and would require the exploration of all feasible control flows to prove properties of a program. Non-trivial programs have an unbounded number of unique control flows to explore, so full enumeration is generally infeasible. As such, in non-trivial software dynamic analysis approaches are under-approximate, as unobserved behaviours will not be reported upon.

White-box and Black-box analysis The terms white-box and black-box are used frequently in program analysis and testing [93]. A white-box analysis refers to one where the internal program workings are examined during analysis, treating the box containing the program as transparent. A black-box analysis assumes the program is opaque to the analyzer and uses only observed environment behaviour to make decisions about program correctness. Any static analysis is white-box since tools require a complete representation of the program in order to make predictions about program behaviour and conformance with a specification. A dynamic analysis can both be white-box or black-box depending on design. A white-box dynamic analysis uses knowledge of program internals to check that the program is executed in a specific way, and can replace components in the program to reduce external factors. A black-box dynamic analysis relies on only observed behaviours, such as program output and file modifications, to make decisions about program correctness and the program itself is executed unmodified.

Unit Testing, Integration Testing, Conformance Testing, and Test Suites Current JavaScript programs are tested through a combination of white-

box, and black-box dynamic test suites. In these approaches, we execute a portion of the program with a series of fixed test inputs and check that the program does not violate our expectations during test execution. In unit testing, the test cases are white-box, and developers use a special test harness to construct the test environment [55]. The test harness extracts the tested portion of the software and stubs or mocks replace the rest of the code using a false implementation with pre-set behaviours. Splitting up the program allows `units` of the software to be tested independently, which can make it easier to develop specific test cases. It also reduces the impact of stateful behaviour, such as file system interaction, on the test environment. Since the testing is white-box, the test harness can use the state of variables and check if the program called functions to decide if the program passes the test. Integration testing is usually the next step in program testing. An integration test is black-box, executing the program unmodified. Before execution, the test creates a test environment to reduce the impact that external side effects can have on behaviour (e.g., program databases are reset to a default state). After testing, the harness will use the program output and environmental changes (e.g., checking that a new entry is created in the database) to decide if the program is correct. When developing a dynamic symbolic execution engine, we can re-use tooling and paradigms designed for unit and integration testing. We can also use the tests created for a program to guide the DSE engine toward exciting portions of a program, but in general, we cannot use the same test assertions since we do not know how DSE will impact the postconditions of testing [98].

Implementation Conformance Tests Implementation conformance tests are a particular case of integration tests. They are used to test if an implementation adheres to a specification in environments where multiple vendors need to implement the same software. In JavaScript, implementation

conformance suites are used to test that an interpreter is conformant with a specification. The ECMA standards body publishes Test262, an official JavaScript conformance suite which is used by many vendors to demonstrate implementation conformance [39].

Program Fuzzing Fuzzing is a brute force approach to property violation detection in software [121]. A fuzzer repeatedly re-executes a program with randomly generated test cases. As the program executes, embedded assertions in the programs source code are run. If the program terminates with a failure then the input is provided to the operator so that they can fix the error. Since the program is being executed concretely, fuzzing cannot produce false positives. Modern fuzzing tools, such as AFL [5], use a variety of strategies to generate new random inputs since a fully random strategy is not likely to explore much of a program. The approach is effective when testing program interfaces, but the fuzzing engine does use information about the program trace to guide inputs and will not explore deeply into complex programs. Modern dynamic symbolic execution engines extend fuzzing but use a solver to guarantee that each new test case explores a unique route through the program [49].

2.2.1 JavaScript Analysis Tools

Mechanised analysis of JavaScript is challenging because of the complex dynamic type system and prototype based inheritance system, and natural language specification. Previous work toward mechanised specifications of JavaScript have modelled subsets of the language [110, 17, 46, 99], but there is no complete mechanised semantic for the current standard. Verification of JavaScript programs is limited by the same constraints, and there is no engine that can run on the majority of real-world JavaScript programs [108, 45, 103]. Here, researchers cite the complex type system and

dynamic nature of the language as the bottleneck for practical analysis.

There are other approaches toward improving the reliability and security of JavaScript programs without verification. JavaScript has mature unit testing and fuzzing frameworks [65, 87, 117]. Unit testing is capable of finding complex errors in a program, but requires manual creation of test cases. Unit testing JavaScript programs has additional challenge, as developers need to add test cases for type coercions in addition

challenge as developers need to add test cases for type coercions in addition to the tests for behaviour. Fuzzing automatically generates new test cases by guided random test generation, it is good at testing earlier portions of the program but is rarely able to explore deep inside an implementation. Fuzzing is more limited in JavaScript, where inputs to a method can be strings, arrays, or objects, making less likely that the random input generator creates a valuable input.

Hedin et al. [57] develop a custom JavaScript interpreter to track information flow through a program execution. Information flow tracking is an effective approach for finding security vulnerabilities, such as key misuse, through runtime behaviour. While this is effective at errors, it is a dynamic analysis, so it will only be effective if the test suite includes a case for violating control flow. This limitation could be mitigated by usage in the production environment, where the software would terminate upon property violation, but the performance overhead would be too high for most environments.

Chugh, Herman, and Jhala [31], Vekris, Cosman, and Jhala [131], Rastogi et al. [104], and Swamy et al. [122] add additional type information to JavaScript or TypeScript programs in order to improve program safety. These additions mitigate some common mistakes, but they cannot be used to test every property of a program. They also increase developer overheads, since the developer now has to deal with the dual concerns of building their type annotations and implementing the program.

2.2.2 Symbolic Execution

Symbolic execution is a program analysis approach which systematically explores all control flows through a program contingent on specified inputs [69, 100, 9, 124, 81, 119]. First, some inputs to a program are replaced with symbols and then the program is run with a special interpreter. As program execution progresses symbolic expressions will be developed rather than concrete values whenever an operation involves a symbolic value. When treating a conditional operation contingent on a symbolic expression, the interpreter will attempt to fork program execution if there is a feasible input that will take each branch. The fork operation on each conditional causes the symbolic interpreter to explore all control flows through the program contingent on the symbolic input values. In order to fork, the program trace up until the branching condition is represented as a constraint problem and a solver will be used to decide if each branch is satisfiable. If a branch is satisfiable, then it is included in the fork and tracked, if it is not satisfiable then that control flow through the program is impossible. While symbolic execution can be used for program verification [62, 14, 63, 68], the complete exploration of all paths through a program is generally infeasible, which limits the use of symbolic execution for program verification and proofs. However an incomplete result can still be useful, since we know that any policy violation reached is feasible. This makes symbolic execution suited to program testing, where the incomplete results can still find bugs and policy violations in a program.

2.2.3 Example: A Simple Symbolic Execution

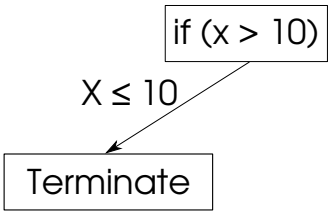
To demonstrate symbolic execution, we present analyze the following program:

```
1  let x = X;
2  if (x > 10) {
```

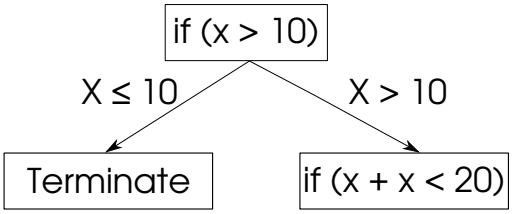
```
3     if (x + x < 20) {
4       console.log("Hello");
5     }
6   }
```

This program initialises a new symbolic input `X`, and assigns it to the variable `x`. Then, it prints `"Hello"` if `x` is greater than `10`, but `x + x` is less than `20`.

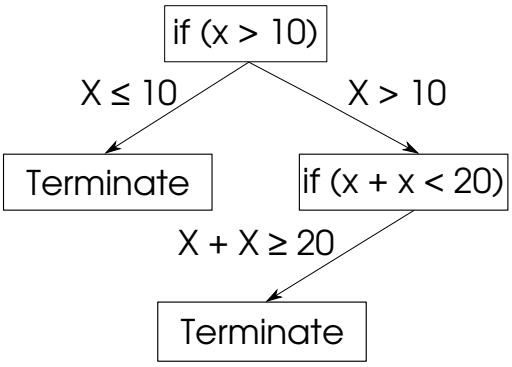
To symbolically execute the program, we begin by constructing the new symbol `X` and then exploring the conditional operation `if (x > 10)`. Since this conditional operation is contigent on our symbol `X`, we query our SMT solver to see which of the branches can be explored. Our first query confirms that the branch `X <= 10` is feasible, leading to the following control flow:



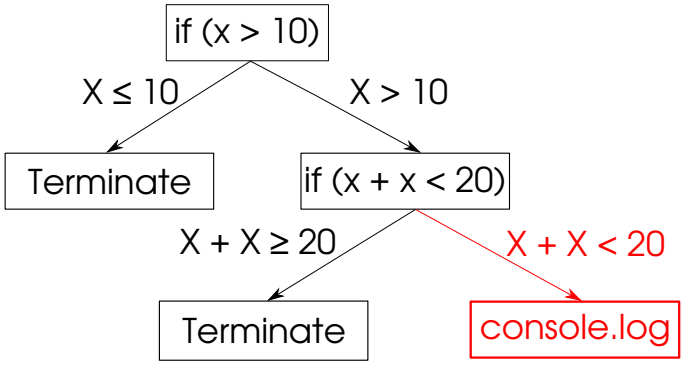
Once we have explored that branch, we ask the constraint solver if `X > 10` is feasible, and we find that it is, proceeding onto the second if condition `if (x + x < 20)`, having explored the following control flows:



As before, this conditional operation has symbolic operands so we use our constraint solver to find our which branches are feasible. To begin, we find that `X + X >= 20` is feasible, leading to program termination:



We now use the constraint solver to decide if `X + X < 20` is feasible. Here, the constraint solver decides that there are no feasible assignments for `X + X < 20`, given the previous assertion that `X > 10`. As such, this branch is infeasible as there is no possible assignment for `X` that will explore it. There are no more branching conditions to explore, so we now terminate symbolic execution with the final control flow graph:



2.2.4 Dynamic Symbolic Execution

In this thesis we focus on the use of dynamic symbolic execution (DSE) for JavaScript analysis. DSE (or concolic execution) is an offline approach to symbolic execution where a concrete state is maintained alongside a sym-

bolic symbolic state and both the symbolic and concrete state are developed concurrently. The symbolic interpreter performs both the standard (concrete) execution, and also develops a symbolic state. Instead of forking immediately on conditional operations, a DSE engine generates new test cases on program termination [49]. To achieve this a *path condition* encodes the conditions under which the current path will be taken. The path condition is a conjunction of all the predicates of conditional statements with predicates negated where the `else` branch was taken. After a program terminates, the *path condition* is sent to a constraint solver, which finds new test cases by negating conditional constraints along the path. If a query to the constraint solver is unsatisfiable, the path corresponding to the new path condition is infeasible; if the solver returns a satisfying assignment then we query the solver for an assignment of our symbolic program inputs that satisfy the constraints and use these as the concrete inputs to our next test case. This process is repeated until all paths have been covered or a stopping condition is reached. Note that since dynamic symbolic execution requires programs to terminate reactive programs like servers will require a test harness that produces a single self-contained run. By maintaining a concrete state, mistakes in the modelling of the program symbolically do not impact interpreter execution. DSE has been used in a variety of different domains, and has been demonstrated as an effective approach to program testing [49, 50, 51, 52].

A dynamic symbolic execution engine can choose to execute parts of the program concretely and still remain sound with respect to under-approximation. This removes the need to model the complete environment, a common cause of errors in other analysis techniques. It also allows for analysis of code which cannot easily be reasoned about statically, such as reflection or dynamically generated code.

Concolic Values We maintain dual concrete and symbolic state through concolic values, a data structure with a concrete and symbolic component. In a concolic value the concrete component is a JavaScript value used by the DSE engine to decide on the behaviour along this path, while the symbolic component is the set of constraints applied to the symbol as the program executes. A concolic representation of values in a program allows us to link the symbolic and concrete representations tightly [114]. We use concolic values to propagate the concrete representation of a value for this test case and the symbolic expression which encodes the constraints applied to that value concurrently. When we introduce a symbolic input, it is provided with an initial concrete value and represented as a structure. For example, we would represent a new integer symbol X with an initial value of 0 as `{symbolic: X, concrete: 0}`. When executing a program, the symbolic interpreter will check if operands are in this form while processing each instruction, and if so, it will treat them symbolically rather than concretely. The use of concolic values makes maintaining dual concrete and symbolic states concurrently more straightforward.

Prior JavaScript DSE Engines

Previous symbolic execution engines for JavaScript were effective at bug finding in web applications and browser extensions [109, 111, 73, 115, 7]. Some of these engines include a limited support for strings, but no engine has a complete support for regular expressions [111, 7]. One common limitation we found with these engines was the limited compatibility with real JavaScript programs. Saxena et al. [111] and Li, Andreasen, and Ghosh [73] developed custom versions of JavaScript interpreters to facilitate symbolic execution. These engines were not maintained and the complexity of instrumenting a modern JavaScript engine made it difficult to merge changes to JavaScript into the project. As a result, these engines

are incompatible with current JavaScript. Other works used instrumentation based approaches, which were more compatible, but did not factor the JavaScript event loop into their design, which could cause incorrect executions [115]. At the onset of this work, we found there were no up-to-date symbolic execution engines for JavaScript capable of executing real-world programs, and while there has been some further development, ExpoSE, the tool we develop in this thesis, is currently the only DSE tool for JavaScript capable of analyzing real-world Node.js applications and webpages. This motivated our decision to focus on developing an architecture compatible with real-world JavaScript, maintainable, and able to run in the browser and on the server-side.

2.2.5 Applications of Symbolic Execution

Symbolic execution can be used to find specification violations. During execution, the symbolic analyser can use the collected path constraints and the SMT solver to attempt to find path-specific specification violations. For example, given a specification which demands that no reference should ever be set to a null value, the symbolic engine can test the specification by querying whether it is feasible for the value being assigned to be null for each assignment encountered along the program trace, and if it is possible the symbolic execution will be able to generate a test input that will violate the specification. Specification testing through DSE automates detection of non-obvious security flaws, such as the detection of weak TLS certificate validation [27], or verifying non-interference [85].

Chandra, Fink, and Sridharan [25] propose the use of backwards symbolic execution, beginning at an interesting point within the program and working back to the program entry point. This approach allows for analysis to only consider paths that will drive execution toward a specific part of the program, but is under-approximate along paths which consider loops

or recursion. One advantage of this work is the potential to act as a verifier for an analysis that can produce false positives. By setting the potential error point as the starting point for analysis, backwards symbolic execution can decide whether there is any feasible input that really triggers the bug without the overhead of standard symbolic execution or input fuzzing.

Symbolic execution and model checking can be combined to produce faster program verifiers [137]. Here, the symbolic execution is used to produce path level models of a program, which a model checker can use to prune redundant paths. The combined approach notably reduces verification times when compared to a purely symbolic approach, enabling verification of more complex programs.

Differential symbolic execution allows a symbolic execution engine to decide if two versions of a program have equivalent behaviour [101] through the generation of comparable program summaries. This enables automatic regression detection after program refactors.

Yadegari and Debray [136] demonstrate that DSE can expose obfuscated malicious behaviours in programs. Often, malicious behaviours are craftily hidden within obfuscated programs, and do not execute until specific conditions are met. Such behaviours are often out of reach of manual or static analysis, as they often rely on code patching or dynamic code injection. By making inputs and environment variables symbolic, a symbolic analysis can identify these obfuscated behaviours. Similarly, symbolic analysis can find the set of URLs that an application will contact [142], allowing developers to examine undocumented web APIs and find potentially malicious or insecure behaviours. It has also been used to analyze deep neural networks, with particular success in the detection of adversarial inputs [53]. A neural network can be symbolically executed by converting that network into an imperative program. From there, the symbolic execution can be used to detect edge cases that could be exploited to craft malicious inputs, such as finding single pixels that can decide the

outcome of a classification.

Guo et al. [54] show that symbolic execution can identify differences in program execution due to speculative execution. This is useful when testing implementations of cryptographic primitives, where information may be leaked by processor specific side channel attacks due to speculative execution.

Performance profiles and probabilistic average, best, and worst case performance of a program can be discovered through symbolic execution [28]. This is useful both in evaluating practical program bottlenecks, but also as a tool for education, where it can be used to automatically identify input-specific flaws that hurt the worst-case performance in an otherwise correct implementation.

Test generation through symbolic execution is often unable to scale when analyzing large programs. Compositional symbolic execution [48] provides an alternate, bottom up, approach to symbolic execution which can further overall analysis. Here, portions of the program are summarized into a set of pre and post conditions, reducing the complexity of constraints considered. Later improvements to this work introduce demand-driven compositional symbolic execution [8], which allows for incomplete summaries, performing expensive analysis only if the summary is incomplete for a given path.

2.2.6 Unique Challenges for Symbolic Execution of JavaScript

Analysis of JavaScript programs comes with some unique challenges. The highly dynamic nature of the language makes it expensive to accurately represent values in the program symbolically in a manner suitable for practical analysis. Modelling implicit type coercion rules using current SMT solvers is often impractical due to a lack of support for conversion between symbols of different theories or solving times too long for practi-

cal DSE. For example, when using the builtin support for string to integer conversion in Z3, the simple constraint problem below tries to reinterpret the string "65" as an integer, selecting the character "6" from the string "66" and the character "5" from the string "55", fails to solve within the default timeout set by the Z3 tutorial website [1]:

```
1  str.to.int (str.++ (str.at "55" 1) (str.at "66" 0))
```

The language features many subtle nuances, such as `typeof(null)== "object"`, which need to be modeled in order to maintain backwards compatibility. Symbolic representation of these features requires a mechanised encoding and carries significant development effort. Further, JavaScript software relies heavily upon asynchronous event callbacks for general operation since there is no parallelism within the standard language. Callbacks are frequently used so it is essential that a symbolic execution framework correctly handles asynchronous events. Additionally, programs also make heavy use of string and regular expression operations. Effective reasoning about programs which perform heavy string processing is often limited, as the cost of symbolically executing string methods and regular expression matchers prohibits meaningful analysis. Recent constraint solvers have drastically improved support for string processing, and often have a limited support for regular expression constraints, but this support is insufficient for JavaScript, where regular expressions include capture groups and backreferences.

2.3 Satisfiability Modulo Theories and Program Analysis

SMT solvers are widely researched and utilized in academia and industry [97, 18, 12, 88, 105]. Satisfiability Modulo Theories (SMT) are decision

[1] https://rise4fun.com/Z3/

problems expressed in first-order logic. An SMT solver is a tool which can take these SMT problems and decide if there is an input which satisfies all of the constraints. Many state of the art SMT solvers use DPLL(T), a solving architecture where DPLL boolean satisfaction is combined with a set of theories to reduce the time it takes to take SMT problems. In this architecture SMT solvers reduce constraint problems expressed in first order logic into boolean satisfaction problems which can then be solved by a SAT solver [35, 34]. SAT solving comes at heavy cost, so solvers reduce the complexity of the underlying SAT problems through the use of theories [44, 94]. In DPLL(T), the complexity of problems is reduced through the use of theory solvers, specialized solvers which can only reason about conjunctions of formulas specific to that theory. In this architecture a SAT solver generates a series of assertions. These assertions are then forwarded to the relevant theory solver. The theory solver will be used to check and refine refine the problem. This process of refinement continues across multiple iterations until a solution to the problem is found or the problem is found to be unsatisfiable.

Current state of the art SMT solvers support many theories which are useful for program analysis, such as bitvectors, strings, or floats, and are often necessary to make problems which use these feasible [88, 37, 19, 139]. In DSE we use an SMT solver to find alternate test cases [49]. SMT can be used to represent program control flow by representing the steps taken during a programs execution as a series of constraints on the inputs to the program. We can use this to find new inputs to a program by taking the constraints applied at the point of a conditional operation and testing whether there is an assignment of inputs to the program that would have produced the opposite outcome for that conditional operation given the collected constraints. Through this we can find new inputs for DSE. In addition to finding new inputs, the constraints collected along a control flow can be used to test assumptions about a program. For example, if a pro-

gram has a rule that a property of an object can never be set to `null`, this can be tested for all observed paths by collecting the constraint problems and using an SMT solver to query for the counter case, an instance where the property is set to a value other than `null`. If this problem is unsatisfiable, then the assumption is never violated across the observed executions of the program.

The DSE tool introduced in this thesis is driven by the Z3 SMT solver [88]. Z3 is a state-of-the-art SMT solver with theory for strings, regular expressions, and arrays. The solver includes language bindings for C, Python and several other languages however it does not support JavaScript out of the box.

Strings in Constraint Solvers Strings and regular expressions are ubiquitous in JavaScript programs. Current SMT solvers include a degree of support for strings and regular expressions [43, 2, 3, 1, 141, 139, 130, 77], but this support is limited to regular languages. In JavaScript, regular expressions are non-regular, supporting features like capture groups and backreferences, and so cannot be directly encoded into SMT by existing solvers.

Prior Analysis Tools with String Support Prior work has used the support in solvers to analyze software [111, 7, 4, 67, 128]. In these works, it was shown that strings are highly utilized and support improves the effectiveness of analysis approaches. Previous work symbolically executing JavaScript has included a degree of support for strings [111, 7], but no prior work included a full support for JavaScript regular expressions.

2.4 Source Code Instrumentation

In DSE we monitor each instruction processed to develop a symbolic trace. Typically this is achieved via a custom interpreter or rewriting the program [111, 73]. A custom interpreter provides a high level of granularity, as the entire state of the program is known, but it has maintenance overheads since the developer must now keep their custom language interpreter up to date. JavaScript has seen rapid development in the last 10 years with a new language specification released once per year on average. With such a rapid rate of change the development effort of maintaining a custom JavaScript interpreter and keeping it compatible with real world software is infeasible for most research groups. A custom interpreter also locks the symbolic execution engine to a single JavaScript vendor, but it would be ideal if the same DSE framework can be used to test software on different interpreters. The alternative is to rewrite the program so that it records the program trace as it executes [115, 86, 120, 138]. Here, we can use existing open source software to assist us, since there are already open-source libraries for parsing and rewriting JavaScript programs such as `babel` and `acorn`. By using these tools we can inject the program tracing component of a DSE engine directly into the program source code. Since our DSE engine is embedded into the program under test, it can then be executed on any interpreter.

2.4.1 Instrumenting JavaScript

Jalangi2 is a framework for the instrumentation of Node.js applications, designed to enable the construction of dynamic analysis [115]. The tool takes a program and instruments it so that every operation triggers a callback, tracking operations through a minimal interface by rewriting complex operations into simpler ones. Callbacks are provided with all

```
1
2  let x = input;
3  x = -x;
4  let y = x + 5;
5  if (y > 5) {
6    throw "Bad Input";
7  }
```

Uninstrumented

```
1  process.on("exit", () => { done(); });
2  let x = input;
3  x = unary("-", x);
4  let y = binary("+", x, 5);
5  if (conditional(binary(">", y, 5))) {
6    throw "Bad Input";
7  }
```

Instrumented

Figure 2.1: An illustration of source code instrumentation.

operands to the instruction, allowing for granular dynamic analysis. From this, the analyser is presented with a monitor of the program from which they can perform analysis. Jalangi2 resolves dynamic code injection, such as `eval` and `require`, transparently by instrumenting any new code added to the program on demand. The tool supports JavaScript ES5 through a custom language rewriter based on the `acorn` parsing framework. It generates JavaScript output files which can then be directly executed by a JavaScript interpreter.

2.5 Example: A Prototype DSE Engine By Instrumentation

We now build our own simple DSE engine through source code instrumentation. In this example, we will draw on our description of DSE engines (Section 2.2.4), and program instrumentation (Section 2.4), to develop a simple symbolic execution engine which can track unary and binary operations. Our DSE engine will use concolic values to propagate symbolic expressions through the execution (Section 2.2.4). First, we build up a set of callbacks. Then, we instrument the program we want to test, adding callbacks for unary, and binary operations as well as conditional

branching. We also instrument the program to call a method once execution terminates.

Our prototype DSE engine is presented in Figure 2.2. This implementation expects for the unary, binary, and conditional callbacks to be called during execution and for `done` to be called when the program terminates. In our prototype, the unary and binary callbacks develop symbolic expressions through their respective symbolicUnary and symbolicBinary methods. When we reach a conditional operation, our callback will check if the branching value is symbolic. If symbolic, we add to the path condition so that new test inputs can be generated upon program termination. When `done` is called, we use an SMT solver to find any feasible alternate routes for the branching conditions explored during this execution. If feasible, new inputs will be executed with the `Replay` method. Note that, in a real implementation a guard would be added to prevent the creation of repeat cases.

We now move on to the program we are going to execute with our DSE engine. Figure 2.1 shows the original and instrumented code for the program we will analyze. The program begins by assigning `x = input`, which is our symbolic input. We then negate it with a unary operation. Next, the variable `y` is set to `x + 5`. Finally, the program uses an **`if`** condition to crash on positive values of `x`, since `y > 5` implies that `x > 0`. When instrumenting the unary operation on line 3, we replace `-x` with a function call to the unary callback method with the operation and operands. The callback allows the analysis analysis to observe, and potentially modify the result of the operation. Similarly, on lines 4 and 5, we replace the binary operation and conditional operation with callbacks so that the analysis can track and potentially modify the result.

To execute our program, we replace our input for `x` with a concolic value (Section 2.2.4), an object with a concrete and symbolic portion. For our initial test case we use the input `{symbolic: X, concrete: 0}`. Upon

reaching line 3, we execute our unary callback. This calls the symbolicUnary method and generates the return value `{symbolic: -X, concrete : -0}`. Next, we execute line 4, triggering the binary callback. This, in turn, calls the symbolicBinary method and assigns y the concolic value `{symbolic: X + 5, 5}`. Executing line 5, the binary operation produces the symbolic value `{symbolic: X + 5 > 5, concrete: false}`, since 5 is not greater than 5, and our conditional callback uses this result to assert $\neg(X + 5 > 5)$ onto the path condition, since $X + 5$ is equal to 5 in this test case. The program now terminates, and our `done` callback is triggered. Here, the solver checks if there is any feasible assignment for X such that $X + 5 > 5$, and generates a new input $X = 1$. The replay method is then called with this new input, triggering the process to begin again, exploring the new path. During the next execution, we explore the alternate branch of the if condition, since our input of 1 will cause $X + 5 > 5$ to be true. Here, we trigger the program throw, showing that it is possible to trigger this assertion.

```
1  function unary(op, operand) {
2    if (isSymbolic(operand)) {
3      return symbolicUnary(op, operand);
4    } else {
5      return concreteUnary(op, operand);
6    }
7  }
8
9  function binary(op, left, right) {
10   if (isSymbolic(left) || isSymbolic(right)) {
11     return symbolicBinary(op, left, right);
12   } else {
13     return concreteBinary(op, left, right);
14   }
15 }
16
17 function conditional(op, branchingValue) {
18   if (isSymbolic(branchingValue)) {
19     if (branchingValue.concrete) {
20       assertPathCondition(branchingValue.symbolic);
21     } else {
22       assertPathCondition(¬branchingValue.symbolic);
23     }
24   }
25   return branchingValue.concrete;
26 }
27
28 function done() {
29   for (conditional in pathCondition) {
30     if (Solve(¬conditional)) {
31       Replay(NewInput());
32     }
33     SolverAddConstraint(conditional);
34   }
35 }
```

Figure 2.2: A prototype DSE engine by instrumented callbacks.

ExpoSE: Practical Symbolic Execution of JavaScript

3

In this chapter, we introduce ExpoSE, our new dynamic symbolic execution tool for JavaScript programs, describing the novelties of our design. We then demonstrate that compatibility with modern JavaScript standards notably impacts the performance of a DSE engine and that our default test case selection strategy improves coverage in time-limited scenarios and decreases coverage plateaus.

Prior Appearances The paper introducing ExpoSE was published at the SPIN workshop in 2017. It is a joint work with Duncan Mitchell of Royal Holloway, University of London, and Johannes Kinder of Research Institute CODE, Bundeswehr University Munich. This chapter shares the name of that work, but focuses on the design and technical novelties of ExpoSE, as well as adding support for ES6 and ES7.

3.1 Introduction

In this chapter we detail ExpoSE, our new DSE engine for JavaScript, highlighting the areas of the design that enable analysis of real-world JavaScript. Our design under-approximates the language when treating type-coercions and non-modeled methods, but our support is sufficient for meaningful analysis of most real-world JavaScript programs. We include a versatile function modeling framework that allows for a developer to specify custom models and concretization rules, enabling extension of ExpoSE when the default support is insufficient. We address the issue of language compatibility by leveraging existing source code instrumentation and transpiler tools. These tools are widely relied on in the community for maintaining compatibility with older browsers, and are integrated into ExpoSE in such a way that they can be quickly upgraded or replaced with alternative tools upon the release of new JavaScript standards.

With this approach we maintain a high level of compatibility with current JavaScript standards while avoiding a significant maintenance burden. To address the maintainability and compatibility problems we observed in prior work we split the engine into components. Our component based design enables support for multiple different analysis backends, which we demonstrate through our support for both Node.js and web applications out of the box. To decrease analysis times we execute test cases concurrently, directed by a scheduler. Concurrent execution of test cases allows better scaling when analyzing large applications, which, when combined with a practical focus for function models, lets us analyze all but the most expansive JavaScript applications.

We evaluate the utility of our new engine through two studies. Here, we show that ES7 support is critical for any engine targeting modern JavaScript programs, increasing compatibility with the most popular packages on NPM by 24%. In the same study we see that ExpoSE is compatible with 83% of the 1000 most depended packages on NPM, demonstrating that our instrumentation approach is widely compatible with real-world code. We also demonstrate that our test case selection strategy prioritises test cases which explore new portions of the program, increasing the overall amount of a codebase which DSE explores.

To summarize, in this chapter we present our first contribution, ExpoSE, to our knowledge the first DSE engine for JavaScript which supports both web and Node.js applications. We address the practical issues that need to be overcome when trying to support real world JavaScript. In particular, we:

- Present a scalable, distributed, instrumentation based approach to JavaScript DSE.
- Detail a practical incremental modeling approach for the type system and built-in methods.

- Detail our coverage calculation and test case selection approaches.
- Evaluate the compatibility of our approach on popular Node.js libraries and justify our choice of test selection strategy.

3.2 Example: Symbolic Execution With ExpoSE

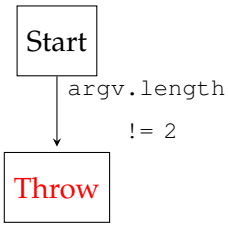
To illustrate how ExpoSE analyzes a program we now run through the symbolic execution of a small argument preprocessor. ExpoSE is a concolic execution engine, so test execution involves both a concrete component, to handle the current test case, and a symbolic component which tracks the symbolic state. Our example program uses several complex JavaScript features, including dynamic type coercions and a regular expression in order to process arguments for the method `execute(...)`:

```
1  process.argv = S$.symbol("argv", [""]);
2  if (process.argv.length != 2) {
3    throw "Incorrect number of arguments";
4  }
5  const target = process.argv[process.argv.length - 1];
6  const parsed = /name=(.+)?/.exec(target);
7  if (!parsed) {
8    throw "Incorrect argument format";
9  }
10 execute(parsed[1].toLowerCase());
```

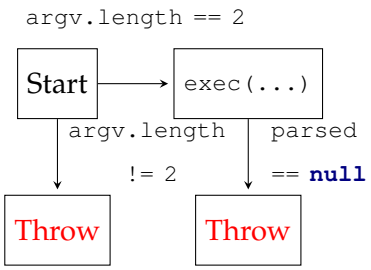
One line 2 the program checks that it has been called with the correct number of arguments. If it has, then the last argument is extracted (line 5), and a regular expression is used to parse it (line 6). In order to symbolically execute this program we need support for symbolic arrays, strings, and regular expressions. The goal of our analysis is to identify a subtle bug trigerred by lines 6 and 10.

To begin we replace the concrete arguments with a symbol. The `S$` library is the interface between ExpoSE and the program-under-test. The

`symbol` method returns a fresh symbol from a symbolic name and initial test case input. Next the program checks if it has been called with the expected number of arguments. As the input to our first test input, `[""]`, has a concrete length of zero we explore the false case and trigger the throw.

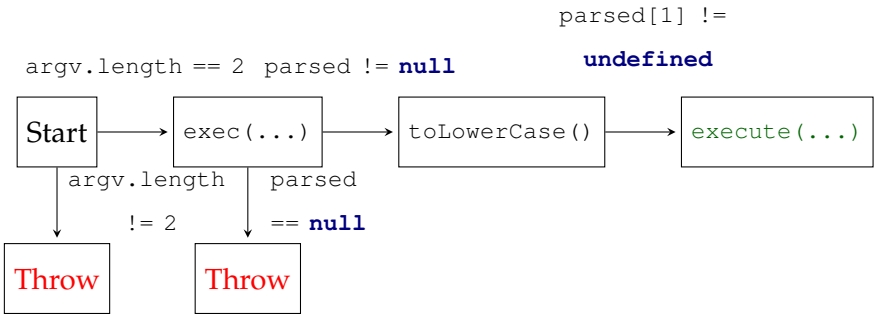


The symbolic state is now used to generate an alternate test case which would satisfy the `argv.length == 2` constraint. We find the input `["", ""]`, an array with two empty strings. Executing this new test case we now reach line 5, which selects the last argument from `argv` and assigns it to a variable `target`. This does not result in any alternate paths as we have already checked that the length of argv is not zero. After this the regular expression operation `/name=(.+)?/.exec(target)` extracts the program input from the `target` string. Operation success is checked with the following if condition. For our current input `["", ""]` the string does not match the regular expression, triggering the throw.

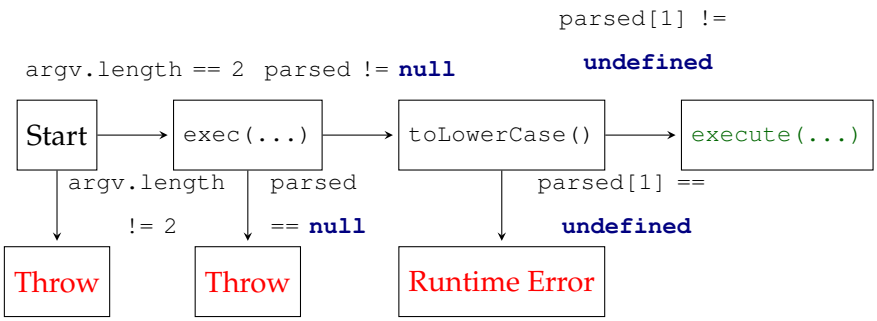


The symbolic state is now used to generate a new input which satisfies the regular constraint, `["","name=xyz"]`. This new test case reaches line 10,

and calls `toLowerCase()` on the first capture group of the regular expression. Capture groups return all characters matched between complementary `(` and `)` characters in a regular expression and are widely used. In this case the capture group should contain whatever is matched by the `(.+)` in our regular expression. In our current test case the capture group will be the string `xyz` which is then passed to the method `execute(...)`.



It is here we find our subtle bug in the program. In our regular expression we have written `name=(.+)?` rather than `name=(.*)`. Here the optional (`?`) operator outside of the capture group allows for the capture group to be **`undefined`** in a match. To explore this this ExpoSE will now generate a new test case, `["", "name="]`, where the regular expression will be satisfied but the first capture group will be **`undefined`**. As we execute this test case we will call **`undefined`**`.toLowerCase()`. This will trigger a runtime error.



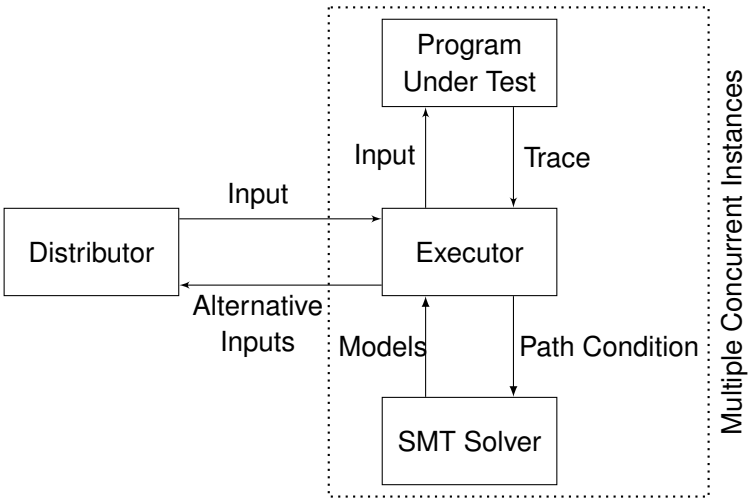


Figure 3.1: The ExpoSE architecture.

With this final test case complete ExpoSE has exhausted all of the unique routes in which the program could be executed, and has found all of the exceptions which could be triggered by and incorrect input to the program.

3.3 General Architecture

ExpoSE is a instrumentation based concolic execution framework for JavaScript. ExpoSE takes a JavaScript program and a symbolic unit test (the test harness) and generates test cases until it has explored all feasible paths or exceeds a time bound. Multiple test cases can be executed concurrently. ExpoSE consists of three main components, the *test executor*, *test distributor*, and the *SMT Solver*, as shown in the overview in Figure 3.1.

The distributor manages the global state of the exploration, aggregates statistics, and schedules test cases for symbolic execution. The test execu-

tor handles program instrumentation, execution and fault reporting for a single test case. Program coverage is tracked inside the executor through instrumented callbacks. A symbolic interpreter inside the test executor keeps track of symbolic state during execution, including symbolic function models. The SMT solver component is used by the executor to extract alternate test cases given a complete symbolic program trace. This component converts the abstract symbolic state to an SMT program for the Z3 solver and also houses our regular expression parser and refinement strategy. As a test executor instance terminates it transfers all relevant state to the distributor. This architecture allows for multiple concurrent test executors instances, each processing a unique test case. ExpoSE includes symbolic exploration strategies for all JavaScript types except functions. In the case of objects and non-homogeneous arrays a type-aware enumeration strategy is applied to facility exploration.

3.4 Test Case Parallelism

ExpoSE is designed to execute many test cases concurrently through its decoupled architecture. The distributor is responsible for prioritisation and scheduling of test cases, as well as presenting analysis information to the operator. The test case executor is responsible for test case instrumentation, execution, and alternative test case generation. It does not produce any output directly and instead relays all information through the distributor. Since test cases can be executed in parallel, individually tricky test cases do not block the overall analysis from making progress.

There is only a small amount of communication between the distributor and executor instances. When a new executor instance is created the distributor issues it with a concrete input for each symbol in the program, as well as a bound expansion variable which is used to prevent repeating

previously explored test cases [50]. The executor then executes entirely independently from the distributor until either analysis is completed or it is terminated due to an execution timeout. Upon termination the executor transfers its state to the distributor. This state includes any runtime exceptions, a coverage map, and any alternative test cases which should be explore. The distributor uses this information to schedule new test cases, as well as produce reports.

3.5 Test Case Isolation

JavaScript programs execute in a single thread, but rely on asynchronous operations to achieve pseudo parallelism. Callbacks from triggered asynchronous operations are not executed preemptively, and will only process when the current execution finishes. In order to avoid issues from lingering asynchronous events we create a separate process for each test case and do not reuse test execution instances. This prevents spill-over effects from one execution to the next where the program affects the global state or dynamically modifies parts of the standard library.

3.6 Source Code Instrumentation

We rewrite programs into a language subset which preserves the semantics of the program, while using fewer distinct operators. This language subset allows for straightforward symbolic execution by reducing the number of language features which require modelling. The source code instrumentation in ExpoSE is performed by a modified version of the Jalangi2 [115] instrumentation framework. Jalangi2 supports dynamic code injection through methods like `eval` and `require` by instrumenting new code on-demand. Our modifications to Jalangi2 include support for

Table 3.1: Instrumented language operations.

Feature	Description
`getField`	Read a specified field from a JavaScript value. In ExpoSE we track these to develop symbolic object models.
`setField`	Set the specified field on a JavaScript value. In ExpoSE we track these to develop symbolic object models.
`invokeFun`	Invoke a function with the specified thisValue and arguments. In ExpoSE we track these to exercise function models.
`binary`	Perform a binary operation with a specified operation and two values. In ExpoSE we track these to develop the symbolic state.
`unary`	Perform a unary operation with a specified operation and value. In ExpoSE we track these to develop the symbolic state.
`conditional`	Perform a conditional check with a specified JavaScript value. In ExpoSE we use track these to develop the path condition.

ES6 and ES7 and improved support for instrumentation of webpages, required for Browser support (Section 3.9).

Jalangi2 reduces all operations which require instrumentation into the operations listed in Table 3.1. ExpoSE uses these hooks to develop symbolic execution. The source code instrumentation adds a monitor to the program through callbacks which execute before and after each rewritten statement. When instrumenting a program it is important to only add code where essential to avoid slowing down program execution. In ExpoSE we add custom logic for these six behaviours in order to implement

symbolic execution. At each of these steps, operands to the operation are checked, and if none are symbolic then we default to the native interpreter behaviour. In the case that one of the operands is symbolic we use our symbolic interpreter to model the behaviour of the program.

3.7 Symbolic Values and the Type System

ExpoSE can symbolically represent JavaScript values for the data types: null, undefined, boolean, number, string, array, and object. This is achieved through a translation of each operation on these variables into SMT via a set of custom encodings. Support for non-homogeneous arrays and objects is achieved through a intermediate encoding layer. In this section we describe the SMT encoding for JavaScript values in ExpoSE (Section 3.7.2, Section 3.7.3). We then cover our support for symbolic arrays and objects (Section 3.7.4). Finally, we describe how we handle type coercions in ExpoSE (Section 3.7.6).

3.7.1 Example: Concolic Values in JavaScript

We will now illustrate how ExpoSE propagates symbolic values through an example trace of a program execution. Take the following program:

```
1 let a = S$.symbol("A", 10);
2 let b = a > 5;
3 if (b) {
4   ...
5 }
```

In this program, we first create a symbolic value by calling a library function `S$.symbol(name, initial)`. The method symbol takes a symbol name, an identifier for the symbol in SMT, and an initial value. The initial value is used as the seed value for the first test case. In ExpoSE, the initial

value also specifies the symbolic type of the value, so in this case, ExpoSE will only generate new numeric values for A.

In ExpoSE, symbols are propagated using concolic values. A concolic value is a object with a concrete and symbolic component. The concrete component is the value for this path, and the symbolic portion is a SMT expression encoding all constraints applied to that value. As the analyzer executes a program test case, it will unwrap these values whenever they are used. The concrete portion is used to decide how the program should behave, while the symbolic portion is modified to represent the new constraints. After processing each operation, a new concolic value is returned, representing the result of the action. In the execution of line 1 in the program above the value of a will be equal to the concolic value `Concolic (10,A)`. Then, on line 2, when we apply the binary operation `a > 5`, this value will be unwrapped. The concrete value, 10, will be used to compute the concrete result, true, and the symbolic portion will be updated to represent this constraint. Hence, this binary operation will yield the value `Concolic(true, A > 5)` for the variable b.

When executing a conditional operation with a symbolic value, we again unwrap the concolic value. In this case, however, the symbolic representation is used to develop the symbolic path condition. In our example, this first occurs on line 3. Here, as we take apart the concolic value stored in b, we use the concrete value true to decide which side of the condition to take for this test case. We also add the symbolic representation of this value, `A > 5`, as an assertion to our path condition, representing the impact that this value has had on program control flow. This design lets us propagate symbolic state through a real JavaScript interpreter.

3.7.2 Overview

Primitive JavaScript types can be directly translated into SMT formulas supported by many off-the-shelf SMT solvers. Values such as null and undefined do not require complex translation as they can only be a single value. This leaves booleans, numbers and strings to be encoded. In this section we describe the framework ExpoSE uses to convert values into SMT constraints at runtime.

Symbols are propagated through test executor instances by the use of concolic values. A concolic value replaces the JavaScript value for each of the symbolic inputs to the program and is composed of a concrete JavaScript value and a symbolic portion which represents it in SMT. The initial test case seeds the concrete portion of symbolic inputs with values provided by the test harness. All later test cases use values provided by the SMT solver.

As each operation occurs in the program under test the test executor is informed through instrumented callbacks (Section 3.6). If any operand is concolic then it is treated by a symbolic interpreter in the test executor rather than the native JavaScript interpreter. In the case of binary and unary operations the symbolic interpreter first computes the concrete result. Next, it generates a symbolic representation of the operation using the ExpoSE SMT bindings. If any operand is not concolic then it is upgraded, turned into a concolic value with a constant symbolic portion. The result of such operations is itself concolic, and will be returned to the program under test where it can continue to propagate through the program. For operations such as `getField` and `setField` the symbolic interpreter may choose to apply some custom reasoning depending on the field requested and the operand types. This is further detailed in our support for strings (Section 3.7.3), arrays, and objects (Section 3.7.4).

The development of the symbolic path condition is driven by treat-

ment of conditional operations. As a conditional operation is executed the operand is checked and, if symbolic, the symbolic interpreter is used to update the symbolic path condition. First non-boolean conditions are coerced by symbolic type coercion. Next, the concrete value of the operand is evaluated to decide which branch of the conditional operation this test case will take. This information is used to update the symbolic path condition of the program so that it is consistent with the concrete path taken by the test case.

For performance reasons ExpoSE uses real values to represent numeric JavaScript values symbolically. Real values differ from JavaScript numeric values in subtle ways. A real value has a different range, and no concept of `Infinity` or `NaN`. These differences do not present issues in the majority of applications, but may lead to divergence between the concrete and symbolic state.

Concretization During execution ExpoSE may be unable to encode an operation symbolically, either because the operation is black-box (i.e., a native library), or too complex. In these cases, ExpoSE can concretize the variable, deleting the symbolic component and proceeding with just the concrete value for that path. When ExpoSE concretizes a value, the symbolic component is lost, so analysis is under-approximate.

3.7.3 Strings

String solving support is a recent addition to many SMT solvers [141]. The support allows for SMT solvers to reason about programs containing complex string constraints, but representation of JavaScript strings in SMT is not as straightforward as other primitive values. One concern is that string concatenation is represented with a distinct operator, where previously the `+` operator always evaluated to the same SMT expression regardless of

value type. To treat this the symbolic interpreter was made type-aware. Whenever an instruction is interpreted symbolically the concrete value of operands is now used to select which SMT operator should be used when encoding. Unlike other primitives, the length field of strings is dependent on the value of the string itself. As such, the symbolic interpreter has a custom treatment for `getField` operations which request the `length` field, returning a concolic length value composed of the concrete length and the constraints on the string length, rather than using the concrete value. This is important when analysing programs with checks on string lengths. For example, when exploring the program:

```
1 const input = S$.symbol("input", "");
2
3 if (input.length == 5) {
4   throw "Error";
5 }
```

We would not be able to generate a test case which exercises the throw operation without instrumenting the length field lookup operation.

Regular Expressions Regular expressions in JavaScript support non-regular features such as backreferences and capture groups, but existing SMT solvers only include support for language membership queries on purely regular languages. ExpoSE includes a rewriting for regular expressions, allowing for symbolic representation of membership language queries including backreferences. It also supports symbolic capture groups. The rewriting operates by splitting each regular expression up into a set of regular language membership queries and string constraints. Operator matching precedence is handled through a CEGAR refinement loop. This support is fully detailed in Chapter 4.

3.7.4 Arrays and Objects

Objects are maps from string-based keys to values with no predefined structure. Arrays and objects are not required to be homogeneously typed. For example, `[1, true, "Hello"]` is a valid array and each element is a different type. This design is a challenge when encoding the program in SMT because solvers do not support these constructs, but support is essential for detailed symbolic execution.

In ExpoSE, we encode arrays and objects through a layer inside the symbolic interpreter. This layer simulates the behavior of symbolic objects while keeping all underlying values homogenously typed. For homogeneously typed arrays (i.e., all values are of the same primitive type), we use direct SMT encoding and in other cases, we explore objects by field enumeration. Each `putField` operation handled by the symbolic interpreter is recorded and updates the known state. If a `getField` operation requests an unknown field in the object, then a new untyped symbol is created and returned for that field.

We describe the ExpoSE encoding for objects and arrays in more detail in Chapter 5, where we illustrate the encoding and show how it is useful when testing JavaScript language polyfills.

3.7.5 Untyped Symbols

Some JavaScript programs behave differently depending upon the type of inputs. Here, we allow untyped symbolic inputs in order to explore the program. ExpoSE supports this through untyped symbols. If a symbol is created without a specified initial input then it is marked as untyped by the symbolic interpreter. ExpoSE will explore operations on the symbol as if the symbol was any of the supported symbolic types. The following example expects a numeric input for `x`:

```
1  const x = S$.symbol('x');
```

```
2 if (typeof x === "number") {
3   console.log(2 * x);
4 } else {
5   console.log("Input error: " + x);
6 }
```

If `x` is numeric then `2*x` is printed to the console, otherwise an error is printed. If the type of the symbol `x` is fixed then the symbol execution would either explore the program logic or the error handler. Here the use of an untyped symbol allows for complete exploration of this program without prior knowledge of expected typing.

Untyped symbolic inputs are supported through base type enumeration. Whenever a fresh untyped symbol is created one new test case is created for an assignment of that symbol in each of the ExpoSE base types. The use of untyped symbols should be limited in practice since interactions between untyped symbols can cause path explosion.

3.7.6 Type Coercions

Table 3.2 details each coercion rule and its symbolic representation. JavaScript has complex type coercion rules and direct representation of these rules is too costly for current SMT solvers. ExpoSE supports simplified type coercion rules and concretizes inputs in other cases, to avoid hindering DSE through infeasible SMT queries.

3.8 Modelling Functions

The JavaScript specification defines a set of built in methods to be implemented by each vendor. These are black-box so ExpoSE cannot instrument them for symbolic execution. We concretize the inputs to these calls because passing concolic values to these methods may break the concrete

Table 3.2: Type Coercion Rules in ExpoSE.

Rule	Type	Rewriting
ToBool	`boolean`	this
	`number`	**if** this! = 0 **then** true **else** false
	`string`	**if** this! = "" **then** true **else** false
	`object`	concretize
ToNumber	`boolean`	**if** this **then** 1 **else** 0
	`number`	this
	`string`	concretize
	`object`	concretize
ToString	`boolean`	**if** this **then** "true" **else** "false"
	`number`	concretize
	`string`	this
	`object`	concretize

execution, since the black-box implementation does not know how to process concolic values. Concretization makes analysis under-approximate, so ExpoSE includes a set of symbolic function models for common black-box methods. Whenever a black box method is executed the symbolic interpreter first executes the method with concrete inputs. If a function model exists for the method we use it to simulate the method call symbolically. The concrete result from the function call and the symbolic result of the function model are then combined into a result value. If there is no function model for that method then only the concrete result is returned.

3.8.1 Modelled APIs

The default behaviour when executing black-box methods is concretization so that concolic values do not break test execution. However, concretization causes under-approximation and some methods are called frequently, so ExpoSE models a subset of the JavaScript standard library covering the most common method calls. A function model simulates a

method behaviour while maintaining the symbolic state. When building models for black-box methods we have two options, direct SMT encoding, or reimplementation (i.e., we implement the method ourselves using operations we can directly encode into SMT). We now discuss the APIs for which ExpoSE includes built-in models.

Math In ExpoSE, we translate math functions directly into SMT. SMT solvers include support for all methods, so a one-to-one translation from a built-in method call to SMT is straightforward.

Strings We support the most commonly called string methods. Modelling for string methods often is not straightforward since there may not be a one-to-one translation into SMT. In these cases, we reimplement the function using instructions that we can directly translate to SMT.

Regex We support all regular expression application methods. In the case of straightforward methods, like `test` and `exec`, the call can be directly translated to SMT after the encoding, though a CEGAR refinement loop built into ExpoSE may be triggered by the solver (Chapter 4). For more complex methods, such as `split` and `matchAll`, we reimplement the method in terms of `exec`. Modelling these methods requires a looped exec operation, and so can generate an infinite number of paths when treating an unconstrained symbolic input.

Array We include two encodings for arrays, mixed-types through an intermediate encoder and homogenously typed through direct translation to SMT (Chapter 5). The dual encoding requires that each function has two models. For most array methods function models are by reimplementation, the intermediate encoding makes calls to the symbolic object encoder

to simulate the function model. In contrast, where possible, the homogeneously typed arrays reimplement the methods through operations that can be translated directly into SMT.

JSON We support serialization of objects with symbolic components to JSON. Symbolic serialization works through a reimplementation of the `JSON.stringify` method to form a symbolic concatenation of the symbolic and concrete components with the JSON syntax. For example, the following program performs a symbolic serialization:

```
1 var x = X;
2 var y = { x: X };
3 var j = JSON.stringify(y);
```

After execution, the variable `j` will be the symbolic string `"{\"x\":"++ X ++ "}"`. Symbolic serialization takes the current concrete list of fields, and is under-approximate. We do not directly support symbolic deserialization, since the complex string constraints required to symbolically reason about parsing of JSON strings is too expensive for practical DSE of most real-world programs. If symbolic reasoning of JSON parsing is desired, a user of ExpoSE can replace `JSON.parse` with a re-implementation in JavaScript. ExpoSE will then symbolically execute the parsing process using its string support, exploring each of the different feasible resulting objects, though this will increase analysis time.

3.9 Supporting Web Analysis

ExpoSE provides support for web applications through instrumentation of a custom web browser. When adding web support for ExpoSE, the design goal was to leverage as much of the existing design as possible. We use our existing distributed design and a new custom web browser to support web applications with few changes. The purpose of the custom

web browser is twofold. It allows us to instrument all loaded JavaScript into the program, and it enables communication with the distributor and path generation components of ExpoSE, escaping the browser sandbox.

We redirect every request from the custom browser through an internal rewriter. Using this rewriter, we scan each request for JavaScript to instrument. We also inject a complete version of the ExpoSE analyzer every JavaScript environment before execution. This deployment of ExpoSE contains all dependencies in a single file. From here, DSE is identical to Node.js, with the distributor creating a new instance of the web browser for each test case, but the browser requires more resources than Node.js, and so often tuning of the distributor is required.

Web applications have no clear termination point. By default, ExpoSE applies a 30s timeout to test case execution for web applications. We selected this timeout after observing page load times when loading common websites in ExpoSE, where it was enough time for the JavaScript that executes as a page loads to finish executing. For more complex applications, a developer will have to tailor ExpoSE behavior and their test harness to their specific use-case. For example, in Chapter 6, we terminate execution when the page `onload` event is triggered.

We further elaborate on our support for web applications in Chapter 6, where we use our support for web applications to reduce performance overhead and improve the privacy of resource dependency resolution proxies through a constraint-based caching scheme.

3.10 Test Case Selection

In general the number of unique control flows through a program is too large to fully explore. Instead, symbolic execution usually terminates after a time limit or goal is reached. A symbolic execution framework can

improve its performance by prioritising test cases which are more likely to drive execution toward a desired goal. For example, by choosing test cases which are created from infrequently explored fork points, we are likely to improve coverage.

ExpoSE supports the prioritisation of test cases through its Strategies API. A strategy can be used by the test distributor in order to reorder the queue of waiting test cases when new alternatives are added. Each time a new set of test cases is sent to the distributor by a terminating test executor the active strategy is queried. A strategy has access to detailed information about each test case, including which conditional operation in the program triggered it's creation. It also has information about the test execution which generated this new test case, including the program trace and which methods and language features were used. Finally, it has access to the entire global state of the distributor, including current coverage information. ExpoSE includes four search strategies and uses the random fork point strategy by default.

FIFO A simple, deterministic exploration strategy. Test cases are queued in the order that they are created by test executor instances.

Random A completely random strategy. Whenever a new set of test cases are added to the queue the queue is randomly shuffled. This strategy is useful in programs where the fork point is not a useful selection criteria but the FIFO strategy gets stuck early treating expensive test case.

Least Common Fork Point The goal of this strategy is to always prioritize test cases from the least explored conditional within the program. Such as strategy should encourage test cases which explore new parts of a program, leading to increased program coverage during analysis. Whenever a test case is created its fork point, a unique identifier for the condi-

tional operation which triggered its creation, is recorded. Prior to selecting a test case for execution the queue of waiting test cases is sorted by the number of times that their fork point has been selected during the current analysis and the least selected fork point is placed at the front of the queue. As a test is selected, a counter for the number of times its fork point has been selected is incremented.

Random Fork Point Like least common fork, the goal of random fork point selection is to improve program coverage of symbolic execution by avoiding sets of test cases generated at the same fork point in the program. This strategy is similar to a strategy successfully employed in the symbolic execution of language interpreters [20] . Here, the test case queue is periodically reordered. Prior to reordering a random fork point is selected from the list of fork points used by test cases in the queue. The queue of waiting test cases is then sorted so that test cases created by that fork point will be selected first. Depending on preference the list can be reordered after each test case is selected, after a set number of test cases have been selected, or once all test cases for that fork point have been exhausted. By selecting a random fork point to explore we can prioritise exploration of new parts of the program while also employing a randomized strategy that can avoid falling into local traps.

3.11 Coverage Calculation

Program coverage metrics are often used to determine how effective analysis of a program has been. Whilst there are drawbacks with such metrics, they often provide a reasonable intuition as to whether analysis has been sufficient. ExpoSE supports three different code coverage metrics:

Line Coverage Line coverage is the most straightforward coverage metric supported by ExpoSE. As each line in a program is executed it is recorded. After analysis is complete the proportion of lines covered across all test cases is computed.

Term Coverage In some programs a single line may contain a large amount of functionality. One example of this is chained method callbacks, a commonly employed pattern in JavaScript programs. Term coverage provides is a more useful metric in such cases. The metric splits each line in a program up into individual terms, the literals and expressions which make up each line. For example, given the following program:

```
1 let result = input().extract().else((error) => "default");
```

Here a line coverage metric would return 100% coverage, regardless of whether the error handling code is ever executed. A term metric coverage instead would give the proportion of terms that are executed. As such it would return only 75% coverage if the error handler is not triggered during analysis.

Decision Coverage Decision coverage is the proportion of the conditional branches in the program control flow are taken at least once during analysis. With such a metric, each conditional operation is split into two possible branches, the true branch and the false branch. As a conditional operation is executed the branch taken is recorded and added to the coverage metrics. For example, given the program:

```
1 var x = ...;
2
3 if (x) {
4   ...
5 } else {
6   ...
7 }
```

If either the true or false condition is taken during analysis, then the decision coverage would be 50%, however if test cases explore both branches then decision coverage would be 100%.

Coverage Calculation ExpoSE generates a source map to create a relationship between the simplified terms in the instrumented program and the original source code. Each simplified term in the instrumented program is issued a unique identifier. Whenever an analysis callback is triggered during test-case execution this identifier is recorded, along with the branch taken if the operation was conditional. Upon test case termination this information is sent back to the Distributor where the coverage calculator merges it into the global coverage state. The source map is then used to compute the coverage of the original program from the map of reached instrumented terms.

3.12 Evaluation

We now set out to evaluate the practicality of ExpoSE on real-world JavaScript programs. In particular, we aim to find out if the steps taken in design successfully mitigate the limitations of prior work, focusing on application compatibility, and whether the default search strategy selected is the correct choice. We set out to answer the following research questions:

(**RQ1**) Does compatibility with ES7 really matter?

(**RQ2**) Does test case selection strategy impact analysis performance?

We answer RQ1 through an evaluation comparing the ExpoSE instrumentation to Jalangi2 (Section 3.12.1). RQ2 is evaluated through an evaluation of several test case selection strategies on real world JavaScript libraries (Section 3.12.2).

3.12.1 Application Support

A principle goal in our design for ExpoSE was to maintain a high degree of compatibility with modern JavaScript. To improve our compatibility, we implemented an extra step in instrumentation which allows us to analyze software written with the ES7 specification. To measure how much of an impact ES7 support has, we now symbolically execute the 1,000 most depended on JavaScript packages on NPM with support for ES7 enabled and disabled. Here, we answer RQ1, showing that ES7 support increases supported libraries by 24%.

Methodology

We selected the latest version of the 1,000 most depended NPM libraries. We execute each library with ExpoSE twice, using an automatically generated harness. One execution includes our modified Jalangi2 instrumentor, whilst the other uses the latest stock version of Jalangi2. The number of JavaScript source files successfully instrumented by each execution is collected. If the number of files instrumented is more than zero then it is counted as successful. While we cannot test if a package is fully supported since we do not know if the published package has errors or if the package is designed to be imported, a number greater than zero means that the library is at least partially supported by ExpoSE. We do not need to re-execute tests since instrumentation of programs is deterministic, so while coverage may change between executions due to test case selection, it will not if instrumentation failed.

Results

Table 3.3 gives the result of our compatibility analysis. We find that our custom instrumentation increases compatibility with the top 1000 Node.js

Table 3.3: Compatibility with the top 1,000 most depended libraries on NPM.

# Libraries	# Supported (ES5)	%	# Supported (ES7)	%
1,000	591	59%	883	83%

applications by 24%. We do not achieve complete compatibility with NPM packages but not all NPM packages are importable, as some are CLI tools or meta packages which will be marked as unsupported. Further, our improvements to Jalangi2 are incomplete so some files will fail to instrument. As such, we answer RQ1 in the affirmative, ES7 support significantly improves compatibility with real-world JavaScript programs.

Conclusions

Addressing RQ1, we see a 24% increase in library compatibility in the wild, from 59% to 83%, a notable increase in library compatibility. The analysis was done using an automatic harness using unknown libraries, so true library support is likely to be even higher. Overall, ExpoSE is highly compatible with modern JavaScript, and while further refinement may improve compatibility in complex applications, ExpoSE will is able to treat the majority of real-world JavaScript problems without issue.

3.12.2 Test Case Selection

ExpoSE has several test case selection strategies, and uses the random fork point strategy by default. To justify this choice, we evaluate multiple packages to see how our default selection strategy can impact coverage. We test each of the strategies described in Section 3.10 on several widely depended on libraries and collect the coverage over time for each test strategy and use this to show that our random fork point strategy consistently produces

Table 3.4: Total coverage for each search strategy.

Library	Deterministic	Random	Least Common Fork	Random Fork
`minimist`	56%	62%	68%	68%
`validator`	67%	66%	72%	73%
`body-parser`	33%	34%	32%	34%
`uuid`	65%	63%	65%	65%
`q`	37%	37%	48%	49%
`less`	21.6%	21.7%	21.5%	21.5%

high coverage. Answering RQ2, we find that test case selection strategy has a notable impact on the peformance of symbolic execution, and that our default strategy produces the best results.

Methodology

If we tested a random sample of JavaScript programs we would not be able to identify if a change in coverage was the result of bugs or non-determinism in ExpoSE or the selection strategy. To avoid the problem, we selected six libraries for which we known ExpoSE has good support and that produce fairly consistent results across tests in order to test only the impact of the selection strategy on program coverage. Each library was executed with a five minute timeout using an automatically generated test harness. ExpoSE was executed once for each of the four strategies. We collected term coverage and test execution rate across executions. Each test was repeated five times and the median test case was selected, though in practice only the random strategy showed notable variation.

Results

Table 3.4 details coverage of the executions. We find that in general the random fork point search strategy yields the highest coverage, although

gains are often small when compared to the least common fork point strategy. The deterministic and random strategies generally perform poorly over time, plateauing with occasional coverage increases.

To further illustrate how search strategies impact coverage we now present Figure 3.2, Figure 3.3 and Figure 3.4. Each figure show the first hundred seconds of coverage and test execution rate for each strategy on three of our tested libraries. From this we see the frequency of coverage increases. We found that a purely deterministic strategy performed very poorly over time, with few test cases increasing program coverage. This can be attributed the strategy executing many similar test cases, as many subsequent cases will be minor variation on the previous as they were generated by the same path. The random strategy also performed poorly, except in the case of Figure 3.3, where it performed best of the four strategies. Even in this case, however, we would not suggest its use as the default search strategy due to the unpredictability. The random strategy was also the only strategy to saturate the machine with long running test cases, as shown by the drop in test execution rate in Figure 3.2. Both of our fork point base strategies performed well, however the random fork point strategy tended to slightly outperform the deterministic fork point.

Conclusions

In our evaluation we observed that the test case selection strategy impacts both overall coverage and the frequency with which the DSE engine plateaus. As such, we answer RQ2 in the affirmative, in time-limited scenarios prioritising test cases will impact overall performance of DSE. While overall program coverage is not always the metric analyzers will want to optimise for, it is generally a good barometer for DSE engine performance.

Discussing our use of the random fork point selection strategy by de-

(a) Deterministic Search Strategy.

(b) Random Fork Point Strategy.

(c) Least Common Fork Point Strategy.

(d) Random Test Case Search Strategy.

Figure 3.2: ExpoSE search strategy performance on `minimist`.

(a) Deterministic Search Strategy.

(b) Random Fork Point Strategy.

(c) Least Common Fork Point Strategy.

(d) Random Test Case Search Strategy.

Figure 3.3: ExpoSE search strategy performance on `body-parser`.

(a) Deterministic Search Strategy.

(b) Random Fork Point Strategy.

(c) Least Common Fork Point Strategy.

(d) Random Test Case Search Strategy.

Figure 3.4: ExpoSE search strategy performance on `validator`.

fault in ExpoSE, the strategy consistently reached the highest program coverage in the libraries tested. Further, we observed that the strategy hit a coverage plateau less frequently than others in practice, leading to more linearity in coverage increases through symbolic execution. Most users of DSE will prefer maximized program coverage when analyzing their program. As such, the random fork point strategy is a good default strategy for ExpoSE.

Sound Regular Expression Semantics for ExpoSE

4

In this chapter we implement a critical missing feature in ExpoSE, support for modern regular expressions. Our new encoding supports all ES6 JavaScript features, including capture groups, backreferences, assertions, and word boundaries. We scanned NPM, a popular JavaScript package repository, and found that 34% of all packages contain regular expressions, demonstrating that support is crucial. We show that our encoding has a notable impact on DSE performance, increasing coverage by up to 1,338%.

Prior Appearances This work was published at the PLDI conference in 2019. It is a joint work with Duncan Mitchell of Royal Holloway, University of London, and Johannes Kinder of Research Institute CODE, Bundeswehr University Munich. My contributions to this work include the design, CEGAR loop, implementation and evaluation. I also jointly helped develop the formal encoding, particularly focusing on the practical limitations.

4.1 Introduction

Regular expressions are popular with developers for matching and substituting strings and are supported by many programming languages. For instance, in JavaScript, one can write `/goo+d/.test(s)` to test whether the string value of `s` contains `"go"`, followed by one or more occurrences of `"o"` and a final `"d"`. Similarly, `s.replace(/goo+d/,"better")` evaluates to a new string where the first such occurrence in `s` is replaced with the string `"better"`.

Several testing and verification tools include some degree of support for regular expressions because they are so common [75, 130, 111, 128, 79]. SMT solvers support theories for strings and classical regular expressions [77, 76, 3, 2, 128, 139, 141, 16, 140, 42], which allow expressing con-

straints such as `s` $\in \mathcal{L}($`goo+d`$)$ for the `test` example above. Although any general theory of strings is undecidable [15], many string constraints are efficiently solved by modern SMT solvers.

SMT solvers support regular expressions in the language-theoretical sense, but "regular expressions" in programming languages like Perl or JavaScript—often called *regex*, a term we also adopt from now on—are not limited to representing regular languages [6]. For instance, the expression `/<(\w+)>.*?<\/\1>/` parses any pair of matching XML tags, which is a context-sensitive language (because the tag is an arbitrary string that must appear twice). Problematic features that prevent a translation of regexes to the word problem in regular languages include capture groups (the parentheses around `\w+` in the example above), backreferences (the `\1` referring to the capture group), and greedy/non-greedy matching precedence of subexpressions (the `.*?` is non-greedy). In addition, any such expression could also be included in a lookahead `(?=)`, which effectively encodes intersection of context sensitive languages. In tools reasoning about string-manipulating programs, these features are usually ignored or imprecisely approximated. This is a problem, because they are widely used, as we demonstrate in Section 4.7.1.

For ExpoSE this lack of support can lead to loss of coverage or missed bugs where constraints would have to include membership in non-regular languages. The difficulty arises from the typical mixing of constraints in path conditions—simply *generating* a matching word for a standalone regex is easy (without lookaheads). For ES6 regular expressions, matching precedence with capture groups is particularly tricky to soundly encode.

In this chapter, we make our second significant contribution, a novel scheme for supporting ECMAScript regex in ExpoSE and show that it is effective in practice (C2). In particular, we make the following contributions:

- We fully model ES6 regex in terms of classical regular languages and string constraints (Section 4.4) and cover several aspects missing from previous work [111, 112, 79]. We introduce the notion of a *capturing language* to make the problem of matching and capture group assignment self-contained.
- We introduce a counterexample-guided abstraction refinement (CEGAR) scheme to address the effect of greediness on capture groups (Section 4.5), which allows us to deploy our model in DSE without sacrificing soundness for under-approximation.
- We present the first systematic study of JavaScript regexes, examining feature usage across 415,487 packages from the NPM software repository. We show that non-regular features are widely used (Section 4.7.1).

4.2 ECMAScript Regex

We review the ES6 regex specification, focusing on differences to classical regular expressions. We begin with the regex API and its matching behavior (Section 4.2.1) and then explain capture groups (Section 4.2.2), backreferences (Section 4.2.3), and operator precedence (Section 4.2.4). ES6 regexes are comparable to those of other languages but lack Perl's recursion and lookbehind and do not require POSIX-like longest matches.

4.2.1 Methods, Anchors, Flags

A classical regular expression defines a regular language; deciding membership of a word in that language is independent from any implementation details. However, most implementations of regular expression are

Table 4.1: JavaScript regular expression API.

Method	Description	Global Flag
`test`	Returns **`true`** if the regular expression matches its argument and **`false`** otherwise.	Starts at `lastIndex`, updates it to the first index after the matched substring (Section 4.2.1).
`search`	Returns the index of the first matching substring of the given regular expression, or -1 if no match.	No effect.
`exec`	Returns an array of the matching substring and all captures when the regular expression matches the given string; returns **`null`** otherwise.	Starts at `lastIndex` and updates it to the first index after the matched substring.
`match`	Returns an array of the matching substring and all captures when the given regular expression matches the string; returns **`null`** otherwise.	Returns an array with all matching substrings but no captures.
`replace`	Replaces the first substring matching the given regular expression with the given string.	Replaces all substrings.
`split`	Splits a string into an array of substrings using the given regular expression as separator.	No effect.

more expressive and expose internals of the matching process. In this section, we explain parts of the JavaScript API for regular expressions which affect our modeling. Other languages have different APIs, but generally provide similar facilities.

ES6 regexes are `RegExp` objects, created from literals or the `RegExp` constructor. Programs can use regular expressions through the methods shown in Table 4.1. `RegExp` objects have two methods, `test` and `exec`, which expect a string argument; String objects offer the `match`, `split`, `search` and `replace` methods that expect a `RegExp` argument.

A regex accepts a string if any portion of the string matches the expression, i.e., it is implicitly surrounded by wildcards. The relative position in the string can be controlled with *anchors*, with ^ and $ matching the start and end, respectively.

Flags in regexes can modify the behavior of matching operations. The *ignore case* flag `i` ignores character cases when matching. The *multiline* flag `m` redefines anchor characters to match either the start and end of input or newline characters. The *unicode* flag `u` changes how unicode literals are escaped within an expression. The *sticky* flag `y` forces matching to start at `RegExp.lastIndex`, which is updated with the index of the previous match. Therefore, `RegExp` objects become stateful as seen in the following example:

```
1  r = /goo+d/y;
2  r.test("goood"); // true;  r.lastIndex = 6
3  r.test("goood"); // false; r.lastIndex = 0
```

The meaning of the *global* flag `g` varies. It extends the effects of `match` and `replace` to include all matches on the string and it is equivalent to the sticky flag for the `test` and `exec` methods of `RegExp`.

4.2.2 Capture Groups

Parentheses in regexes not only change operator precedence (e.g., `(ab)*` matches any number of repetitions of the string `"ab"` while `ab*` matches

the character `"a"` followed by any number of repetitions of the character `"b"`) but also create *capture groups*. Capture groups are implicitly numbered from left to right by order of the opening parenthesis. For example, `/a|((b)*c)*d/` is numbered as `/a|(`^{1}`(`^{2}`b)*c)*d/`. Where only bracketing is required, a non-capturing group can be created by using the syntax `(?:...)`.

For regexes, capture groups are important because the regex engine will record the *most recent* substring matched against each capture group. Capture groups can be referred to from within the expression using backreferences (see Section 4.2.3). The last matched substring for each capture group is also returned by some of the API methods. In JavaScript, the return values of `match` and `exec` are arrays, with the whole match at index 0 (the implicit capture group 0), and the last matched instance of the i^{th} capture group at index i. In the example above, `"bbbbcbcd".match(/a|((b)*c)*d/)` will evaluate to the array `["bbbbcbcd", "bc", "b"]`.

4.2.3 Backreferences

A *backreference* in a regex refers to a numbered capture group and will match whatever the engine last matched the capture group against. For example, expression `/(a|b)\1/` will match the strings `"aa"` or `"bb"`, with `"a"` or `"b"` stored in the capture group, respectively, but not `"ab"` or `"ba"`. In general, the addition of backreferences to regexes makes the accepted languages non-regular [6].

Backreferences refer to the most recent matched substring of the referenced capture group; Inside quantifiers (Kleene star, Kleene plus, and other repetition operators), the string matched by the backreference can change across multiple matches. For example, the regex `/((a|b)\2)+/` can match the string `"aabb"`, with the backreference `\2` being matched twice: the first time, the capture group contains `"a"`, the second time it contains `"b"`. This logic applies recursively, and it is possible for backreferences to

Table 4.2: Regular expression operators, separated by classes of precedence.

Operator	Name	Rewriting
`(r)`	Capturing parentheses	
`\n`	Backreference	
`(?:r)`	Non-capturing parentheses	
`(?=r)`	Positive lookahead	
`(?!r)`	Negative lookahead	
`\b`	Word boundary	
`\B`	Non-word boundary	
$r*$	Kleene star	
$r*?$	Lazy Kleene star	
$r+$	Kleene plus	$r*r$
$r+?$	Lazy Kleene plus	$r*?r$
$r\{m,n\}$	Repetition	$r^n \mid \ldots \mid r^m$
$r\{m,n\}?$	Lazy repetition	$r^m \mid \ldots \mid r^n$
$r?$	Optional	$r \mid \epsilon$
$r??$	Lazy optional	$\epsilon \mid r$
$r_1 r_2$	Concatenation	
$r_1 \mid r_2$	Alternation	

in turn be part of outer capture groups.

4.2.4 Operator Evaluation

We explain the operators of interest for this chapter in Table 4.2; the implementation described in Section 4.6 supports the full ES6 syntax [38]. Some operators can be rewritten into semantically equivalent expressions to reduce the number of cases to handle (shown in the **Rewriting** column).

Regexes distinguish between *greedy* and *lazy* evaluation. Greedy op-

erators consume as many characters as possible such that the entire regular expression still matches; lazy operators consume as few characters as possible. This distinction—called *matching precedence*—is unnecessary for classical regular languages, but does affect the assignment of capture groups and therefore backreferences.

Zero-length assertions or *lookarounds* do not consume any characters but still restrict the accepted word, enforcing a language intersection. Positive or negative *lookaheads* can contain any regex, including capture groups and backreferences. In ES6, *lookbehind* is only available through `\b` (word boundary), and `\B` (non-word boundary), which are commonly used to only (or never) match whole words in a string.

4.3 Overview

In an overview of our approach, we now define the word problem for regex (Section 4.3.1) and how it arises in DSE (Section 4.3.2). We introduce our model for regex by example (Section 4.3.3) and explain how to eliminate spurious solutions by refinement (Section 4.3.4).

4.3.1 The Word Problem and Capturing Languages

For any given classical regular expression r, we write $w \in \mathcal{L}(r)$ whenever w is a word within the (regular) language generated by r. For a regex R, we also need to record values of capture groups within the regex. To this end, we introduce the following notion:

Definition 1 (Capturing Language)**.** The *capturing language* of a regex R, denoted $\mathcal{L}_c(R)$, is the set of tuples $(w, C_0, \ldots, C_n)$ such that w is a word of the language of R and each $C_0, \ldots, C_n$ is the substring of w matched by the corresponding numbered capture group in R.

```
1  let timeout = '500';
2  for (let i = 0; i < args.length; i++) {
3    let arg = args[i];
4    let parts = /<(\w+)>([0-9]*)<\/\1>/.exec(arg);
5    if (parts) {
6      if (parts[1] === "timeout") {
7        timeout = parts[2];
8      }
9      ...
10   }
11 }
12 assert(/^[0-9]+$/.test(timeout) == true);
```

Listing 1: Code example using regex.

A word w is therefore matched by a regex R $\iff$ $\exists \mathcal{C}_0, \ldots, \mathcal{C}_n$: $(w, \mathcal{C}_0, \ldots, \mathcal{C}_n) \in \mathcal{L}_c(R)$. It is not matched if and only if $\forall \mathcal{C}_0, \ldots, \mathcal{C}_n$: $(w, \mathcal{C}_0, \ldots, \mathcal{C}_n) \notin \mathcal{L}_c(R)$. For readability, we will usually omit quantifiers for capture variables where they are clear from the context.

4.3.2 Regex In Dynamic Symbolic Execution

The code in Listing 1 parses numeric arguments between XML tags from its input variable `args`, an array of strings. The regex in line 4 breaks each argument into two capture groups, the tag and the numeric value (`parts[0]` is the entire match). When the tag is "`timeout`", it sets the `timeout` value accordingly (lines 6–7). On line 12, a runtime assertion checks that the timeout value is truly numeric after the arguments have been processed. The assertion can fail because the program contains a bug: the regex in line 4 uses a Kleene star and therefore also admits the empty string as the number to set, and JavaScript's dynamic type system will allow setting `timeout` to the empty string.

DSE finds such bugs by systematically enumerating paths, including the failure branches of assertions [49]. Without support for regex, a DSE engine will *concretize* `arg` on the call to `exec`, assigning the concrete result to

`parts`. With all subsequent decisions concrete, the path condition becomes $pc = \texttt{true}$ and the engine will be unable to cover more paths and find the bug.

Implementing regex support ensures that `parts` is *symbolic*, i.e., its elements are represented as formulas during symbolic execution. The path condition for the initial path thus becomes $pc = (\texttt{args[0]}, \mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2) \notin \mathcal{L}_c(R)$ where $R = \texttt{<(\textbackslash w+)>([0-9]*)<\textbackslash /\textbackslash 1>}$. Negating the only clause and solving yields, e.g., `args[0]` = `"<a>0</a>"`. DSE then uses this input assignment to cover a second path with $pc = (\texttt{args[0]}, \mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2) \in \mathcal{L}_c(R) \wedge \mathcal{C}_1 \neq \texttt{"timeout"}$. Negating the last clause yields, e.g., "`<timeout>0</timeout>`", entering line 7 and making `timeout` and therefore the assertion symbolic. This leads to $pc = (\texttt{args[0]}, \mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2) \in \mathcal{L}_c(R) \wedge \mathcal{C}_1 = \texttt{"timeout"} \wedge (\mathcal{C}_2, \mathcal{C}_0') \in \mathcal{L}_c(\texttt{\^{}[0-9]+\$})$, which, after negating the last clause, triggers the bug with the input "`<timeout></timeout>`".

4.3.3 Modeling Capturing Language Membership

Capturing language membership constraints in the path condition cannot be directly expressed in SMT. We model these in terms of classical regular language membership and string constraints. For a given ES6 regex R, we first rewrite R (see Table 4.2) in atomic terms only, i.e., `|`, `*`, capture groups, backreferences, lookaheads, and anchors. For consistency with the JavaScript API, we also introduce the outer capture group $\mathcal{C}_0$. Consider the regex $R = \texttt{(?:a|(b))\textbackslash 1}$. After preprocessing, the capturing language membership problem becomes

$$(w, \mathcal{C}_0, \mathcal{C}_1) \in \mathcal{L}_c(\texttt{(?:.|\textbackslash n)*?((?:a|(b))\textbackslash 1)(?:.|\textbackslash n)*?}),$$

a generic rewriting that allows for characters to precede and follow the match in the absence of anchors.

We recursively reduce capturing language membership to regular membership. To begin, we translate the purely regular Kleene stars and the outer capture group to obtain

$$\begin{aligned}(w, \mathcal{C}_0, \mathcal{C}_1) \in &\mathcal{L}_c(R) \implies w = w_1 \mathbin{+\!\!+} w_2 \mathbin{+\!\!+} w_3 \\ &\wedge w_1 \in \mathcal{L}(\texttt{(:?.|\n)*?}) \\ &\wedge (w_2, \mathcal{C}_1) \in \mathcal{L}_c(\texttt{(?:a|(b))\1}) \wedge\ \mathcal{C}_0 = w_2 \\ &\wedge w_3 \in \mathcal{L}(\texttt{(:?.|\n)*?}),\end{aligned}$$

where $\mathbin{+\!\!+}$ is string concatenation. We continue by decomposing the regex until there are only purely regular terms or standard string constraints. Next, we translate the nested capturing language constraint

$$\begin{aligned}(w_2, C_1) \in \mathcal{L}_c(\texttt{(?:a|(b))\1}) \implies \\ w_2 = w_1' \mathbin{+\!\!+} w_2' \wedge (w_1', \mathcal{C}_1) \in \mathcal{L}_c(\texttt{a|(b)}) \wedge (w_2') \in \mathcal{L}_c(\texttt{\1}).\end{aligned}$$

When treating the alternation, either the left is satisfied and the capture group becomes undefined (which we denote as $\varnothing$), or the right is satisfied and the capture is locked to the match, which we model as

$$(w_1' \in \mathcal{L}(a) \wedge \mathcal{C}_1 = \varnothing) \vee (w_1' \in \mathcal{L}(b) \wedge \mathcal{C}_1 = w_1').$$

Finally we model the backreference, which is case dependent on whether the capture group it refers to is defined or not:

$$(\mathcal{C}_1 = \varnothing \implies w_2' = \epsilon) \wedge (\mathcal{C}_1 \neq \varnothing \implies w_2' = \mathcal{C}_1).$$

Putting this together, we obtain a model for R:

$$
\begin{aligned}
&(w, \mathcal{C}_0, \mathcal{C}_1) \in \mathcal{L}_c(\mathrm{R}) \implies w = w_1 \mathbin{+\!\!+} w_1' \mathbin{+\!\!+} w_2' \mathbin{+\!\!+} w_3 \\
&\wedge \mathcal{C}_0 = w_1' \mathbin{+\!\!+} w_2' \\
&\wedge \left((w_1' \in \mathcal{L}(\mathrm{a}) \wedge \mathcal{C}_1 = \varnothing) \vee (w_1' \in \mathcal{L}(\mathrm{b}) \wedge \mathcal{C}_1 = w_1')\right) \\
&\wedge (\mathcal{C}_1 = \varnothing \implies w_2' = \epsilon) \wedge (\mathcal{C}_1 \neq \varnothing \implies w_2' = \mathcal{C}_1) \\
&\wedge w_1 \in \mathcal{L}(\texttt{(:?.|\n)*?}) \wedge w_3 \in \mathcal{L}(\texttt{(:?.|\n)*?}).
\end{aligned}
$$

4.3.4 Refinement

Because of matching precedence (greediness), these models permit assignments to capture groups that are impossible in real executions. For example, we model `/^a*(a)?$/` as

$$
\begin{aligned}
(w, \mathcal{C}_0, \mathcal{C}_1) \in \mathcal{L}_c(\texttt{/\^a*(a)?\$/}) \implies w = w_1 \mathbin{+\!\!+} w_2 \\
\wedge w_1 \in \mathcal{L}(\texttt{a*}) \wedge w_2 \in \mathcal{L}(\texttt{a}|\epsilon) \wedge \mathcal{C}_0 = w \wedge \mathcal{C}_1 = w_2.
\end{aligned}
$$

This allows $\mathcal{C}_1$ to be either `a` or the empty string ϵ, i.e., the tuple `( "aa", "aa", "a")` would be a spurious member of the capturing language under our model. Because `a*` is *greedy*, it will always consume both characters in the string `"aa"`; therefore, `(a)?` can only match ϵ. This problem posed by *greedy* and *lazy* operator semantics remains unaddressed by previous work [111, 128, 79, 112]. To address this, we use a counterexample-guided abstraction refinement scheme that validates candidate assignments with an ES6-compliant matcher. Continuing the example, the candidate element `("aa", "aa", "a")` is validated by running a concrete matcher on the string `"aa"`, which contradicts the candidate captures with $\mathcal{C}_0 =$ `"aa"` and

$C_1 = \epsilon$. The model is refined with the counter-example to the following:

$$\begin{aligned} w = & w_1 +\!\!+ w_2 \\ & \wedge w_1 \in \mathcal{L}(\texttt{a}*) \wedge w_2 \in \mathcal{L}(\texttt{a}|\epsilon) \wedge C_0 = w \wedge C_1 = w_2 \\ & \wedge \big(w = \texttt{"aa"} \implies (C_0 = \texttt{"aa"} \wedge C_1 = \epsilon)\big). \end{aligned}$$

We then generate and validate a new candidate (w, C_0, C_1) and repeat the refinement until a satisfying assignment passes the concrete matcher. This scheme is discussed in detail in Section 4.5, along with a discussion on soundness and possible non-termination.

4.4 Modeling ES6 Regex

We present a general model for ECMAScript 6 regular expressions, extending substantially on previous work on capture groups only [111] and preliminary work on backreferences [79]. We now detail the process of modeling capturing languages. After preprocessing a given ES6 regex R to R′ (Section 4.4.1), we model constraints $(w, C_0, \ldots, C_n) \in \mathcal{L}_c(R')$ by recursively translating terms in the abstract syntax tree (AST) of R′ to classical regular language membership and string constraints (Section 4.4.2–4.4.3). Finally, we show how to model negated constraints $(w, C_0, \ldots, C_n) \notin \mathcal{L}_c(R')$ (Section 4.4.5).

4.4.1 Preprocessing

For illustrative purposes, we make the concatenation R_1R_2 of terms R_1, R_2 explicit as the binary operator $R_1 \cdot R_2$. Any regex can then be split into combinations of atomic elements, capture groups and backreferences (referred to collectively as *terms*, in line with the ES6 specification [38]), joined by explicit operators. Using the rules in Table 4.2, we rewrite any R to an equiv-

alent regex R' containing only alternation, concatenation, Kleene star, capture groups, non-capturing parentheses, lookarounds, and backreferences. We rewrite any remaining lazy quantifiers to their greedy equivalents, as the models are agnostic to matching precedence (this is dealt with in refinement).

Note that the rules for Kleene plus and repetition duplicate capture groups, e.g., rewriting `/(a){1,2}/` to `/(a)(a)|(a)/` adds two capture groups. We therefore explicitly relate capture groups between the original and rewritten expressions. When rewriting a Kleene plus expression $S+$ containing K capture groups, $S*S$ has $2K$ capture groups. For a constraint of the form $(\mathcal{C}_1, \ldots, \mathcal{C}_K) \in \mathcal{L}_c(S+)$, the rewriting yields

$$(\mathcal{C}_0, \mathcal{C}_{1,1}, \ldots, \mathcal{C}_{K,1}, \mathcal{C}_{1,2}, \ldots, \mathcal{C}_{K,2}) \in \mathcal{L}_c(S*S).$$

Since $S*S$ contains two copies of S, $\mathcal{C}_{i,j}$ corresponds to the i^{th} capture in the j^{th} copy of S in $S*S$. We express the direct correspondence between captures as

$$\begin{aligned}(w, \mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_K) \in \mathcal{L}_c(S+) \iff \\ (w, \mathcal{C}_0, \mathcal{C}_{1,1}, \ldots, \mathcal{C}_{K,1}, \mathcal{C}_{1,2}, \ldots, \mathcal{C}_{K,2}) \in \mathcal{L}_c(S*S) \\ \wedge \forall i \in \{1, \ldots, K\},\ \mathcal{C}_i = \mathcal{C}_{i,2}.\end{aligned}$$

For repetition, if $S\{m, n\}$ has K capture groups, then $S' = S^n \mid \ldots \mid S^m$ has $\frac{K}{2}(n+m)(n-m+1)$ captures. In S', suppose we index our captures as $\mathcal{C}_{i,j,k}$ where $i \in \{1, \ldots, K\}$ is the index of the capture group in S, $j \in \{0, \ldots, n-m\}$ denotes which alternate the capture group is in (0 being the rightmost), and $k \in \{0, \ldots, m + j - 1\}$ indexes the copies of S within each alternate. Intuitively, we pick a single $x \in \{0, \ldots, n - m\}$ that corresponds to the first satisfied alternate. Comparing the assignment of captures in $S\{m, n\}$ to S', we know that the value of the capture is the last possible match, so

$\mathcal{C}_i = \mathcal{C}_{i,x,m+x-1}$ for all $i \in \{1, \ldots, K\}$. Formally, this direct correspondence can be expressed as

$$
\begin{aligned}
&(w,\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_K) \in \mathcal{L}_c(S\{m,n\}) \iff \\
&\quad (w, \mathcal{C}_0, \mathcal{C}_{1,0,0}, \ldots, \mathcal{C}_{K,n-m,n}) \in \mathcal{L}_c(S^n \mid \ldots \mid S^m) \\
&\quad \wedge \exists x \in \{0, \ldots, n-m\}: \\
&\quad\quad \big((w, \mathcal{C}_0, \mathcal{C}_{1,x,0}, \ldots, \mathcal{C}_{K,x,m+x-1}) \in \mathcal{L}_c(S^{m+x}) \\
&\quad\quad\quad \wedge \forall x' > x,\ (w, \mathcal{C}_0, \mathcal{C}_{1,x',0}, \ldots, \mathcal{C}_{K,x',m+x'-1}) \notin \mathcal{L}_c(S^{m+x'}) \\
&\quad\quad\quad \wedge \forall i \in \{1, \ldots, K\},\ \mathcal{C}_i = \mathcal{C}_{i,x,m+x-1}\big).
\end{aligned}
$$

4.4.2 Operators and Capture Groups

Table 4.3: Models for regex operators.

Operation	t	**Overapproximate Model for** $(w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t)$
Alternation	$t_1 \mid t_2$	$\big((w, \mathcal{C}_i, ..., \mathcal{C}_{i+j}) \in \mathcal{L}_c(t_1)$ $\wedge\, \mathcal{C}_{i+j+1} = ... = \mathcal{C}_{i+k} = \varnothing\big)$ $\vee\big((w, \mathcal{C}_{i+j+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_2) \wedge$ $\mathcal{C}_i = ... = \mathcal{C}_{i+j} = \varnothing\big)$
Concatenation	$t_1 \cdot t_2$	$w = w_1 \mathbin{+\!\!+} w_2$ $\wedge (w_1, \mathcal{C}_i, ..., \mathcal{C}_{i+j}) \in \mathcal{L}_c(t_1)$ $\wedge (w_2, \mathcal{C}_{i+j+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_2)$
Backreference-free Quantification	t_1*	$w = w_1 \mathbin{+\!\!+} w_2 \wedge w_1 \in \mathcal{L}(\widehat{t_1*})$ $\wedge\, (w_2, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1 \mid \epsilon)$ $\wedge\, \big(w_2 = \epsilon \implies (w_1 = \epsilon \wedge \mathcal{C}_i = \ldots = \mathcal{C}_{i+k} = \varnothing)\big)$

Positive Lookahead	$(?{=}t_1)\,t_2$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+j}) \in \mathcal{L}_c(t_1 \,.\, *) \wedge (w, \mathcal{C}_{i+j+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_2)$
Negative Lookahead	$(?{!}t_1)\,t_2$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+j}) \notin \mathcal{L}_c(t_1 \,.\, *) \wedge (w, \mathcal{C}_{i+j+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_2)$
Input Start	$t_1\,\hat{}$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}(. * \langle)$
Input Start (Multiline)	$t_1\,\hat{}$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}(. * \langle | \backslash n)$
Input End	$\$t_1$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}(\rangle .*)$
Input End (Multiline)	$\$t_1$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}(\rangle | \backslash n .*)$
Word Boundary	$t_1 \backslash b\ t_2$	$w = w_1 \mathbin{+\!\!+} w_2 \wedge (w_1, \mathcal{C}_i, ..., \mathcal{C}_{i+j}) \in \mathcal{L}_c(t_1) \wedge (w_2, \mathcal{C}_{i+j+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_2) \wedge \Big(\big((w_1 \in \mathcal{L}(. * \backslash W) \vee w_1 = \epsilon) \wedge w_2 \in \mathcal{L}(\backslash w \,.\, *) \big) \vee \big(w_1 \in \mathcal{L}(. * \backslash w) \wedge (w_2 \in \mathcal{L}(\backslash W \,.\, *) \vee w_2 = \epsilon) \big) \Big)$
Non-Word Boundary	$t_1 \backslash B\ t_2$	$w = w_1 \mathbin{+\!\!+} w_2 \wedge (w_1, \mathcal{C}_i, ..., \mathcal{C}_{i+j}) \in \mathcal{L}_c(t_1) \wedge (w_2, \mathcal{C}_{i+j+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_2) \wedge \big((w_1 \notin \mathcal{L}(. * \backslash W) \wedge w_1 \neq \epsilon) \vee w_2 \notin \mathcal{L}(\backslash w \,.\, *) \big) \wedge \big(w_1 \notin \mathcal{L}(. * \backslash w) \vee (w_2 \notin \mathcal{L}(\backslash W \,.\, *) \wedge w_2 \neq \epsilon) \big)$
Capture Group	(t_1)	$(w, \mathcal{C}_{i+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1) \wedge \mathcal{C}_i = w$
Non-Capturing Group	$(?{:}t_1)$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1)$
Base Case	t regular	$w \in \mathcal{L}(t)$

Let t be the next term to process in the AST of R'. If t is capture-free and purely regular, there is nothing to do in this step. If t is non-regular, it contains $k+1$ capture groups (with $k \geq -1$) numbered i through $i+k$. At each recursive step, we express membership of the capturing language $(w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t)$ through a model consisting of string and regular language membership constraints, and a set of remaining capturing language membership constraints for subterms of t. We then recursively model these subterms. Note that we record the locations of capture groups within the regex in the preprocessing step. When splitting t into subterms t_1 and t_2, capture groups $\mathcal{C}_i, \ldots, \mathcal{C}_{i+j}$ are contained in t_1 and $\mathcal{C}_{i+j+1}, \ldots, \mathcal{C}_{i+k}$ are contained in t_2 for some j. Each language constraint is of the form $(w', \mathcal{C}_j, ..., \mathcal{C}_{j+l}) \in \mathcal{L}_c(t')$, where w' is a substring of w, the captures discussed are an incrementally indexed subset of $\{\mathcal{C}_i, ..., \mathcal{C}_{i+k}\}$, and t' is a subterm of t. The models for individual operations are given in Table 4.3; we discuss specifics of the rules below.

When matching an alternation `|`, capture groups on the non-matching side will be undefined, denoted by $\varnothing$, which is distinct from the empty string ϵ.

When modeling quantification $t = t_1*$, we assume t_1 does not contain backreferences (we address this case in Section 4.4.3). In this instance, we model t via the expression $\hat{t_1}*t_1 \mid \epsilon$, where $\hat{t_1}$ is a regular expression corresponding to t_1, except each set of capturing parentheses is rewritten as a set of non-capturing parentheses. In this way, $\hat{t_1}$ is regular (it is backreference-free by assumption). However, $\hat{t_1}*t_1 \mid \epsilon$ is not semantically equivalent to t: if possible, capturing groups must be satisfied, so $\hat{t_1}*$ cannot consume all matches of the expression. We encode this constraint with the implication that $\hat{t_1}*$ must match the empty string whenever $t_1 \mid \epsilon$ does.

Lookahead constrains the word to be a member of the languages of both the assertion expression and t_2. The word boundary `\b` is effectively a single-character lookaround for word and non-word characters. Because

the boundary can occur both ways, the model uses disjunction for the end of w_1 and the start of w_2 being word and non-word, or non-word and word characters, respectively. The non-word boundary `\B` is defined as the dual of `\b`.

For capture groups, we bind the next capture variable $\mathcal{C}_i$ to the string matched by t_1. The i^{th} capture group must be the outer capture and the remaining captures $\mathcal{C}_{i+1}, \ldots, \mathcal{C}_{i+k}$ must therefore be contained within t_1. There is nothing to be done for non-capturing groups and recursion continues on the contained subexpression.

Anchors assert the start (`^`) and end (`$`) of input; we represent the beginning and end of a word via the meta-characters $\langle$ and $\rangle$, respectively. In most instances when handling these operations, t_1 will be ϵ; this is because it is rare to have regex operators prior to those marking the start of input (or after marking the end of input, respectively). In both these cases, we assert that the language defines the start or end of input—and that as a result of this, the language of t_1 must be an empty word, though the capture groups may be defined (say through t_1 containing assertions with nested captures). We give separate rules for matching a regular expression with the multiline flag set, which modify the behavior of anchors to accept either our meta-characters or a line break.

4.4.3 Backreferences

Table 4.4 lists our models for different cases of backreferences in the AST of regex R; `\k` is a backreference to the k^{th} capture group of R. Intuitively, each instance of a backreference is a variable that refers to a capture group and has a type that depends on the structure of R.

We call a backreference *immutable* if it can only evaluate to a single value when matching; it is *mutable* if it can take on multiple values, which is a rare but particularly tricky case. For example, consider `/((a|b)\2)+\1\2/`.

Table 4.4: Modeling backreferences.

Type of $\backslash k$	**Capturing Language Encoding**	**Approximation**
Empty	$(w) \in \mathcal{L}_c(\backslash k)$ $w = \epsilon$	Exact
Immutable	$(w) \in \mathcal{L}_c(\backslash k)$ $(\mathcal{C}_k = \varnothing \implies w = \epsilon) \wedge (\mathcal{C}_k \neq \varnothing \implies w = \mathcal{C}_k)$	Over
Immutable	$(w) \in \mathcal{L}_c(\backslash k*)$ $(\mathcal{C}_k = \varnothing \implies w = \epsilon) \wedge (\mathcal{C}_k \neq \varnothing \implies \exists m \geq 0 : w = {+\!+}_{i=0}^{m} \mathcal{C}_k)$	Over
Mutable	$(w, \mathcal{C}_k) \in \mathcal{L}_c(\,(?:(t_1)\backslash k)*)$ t_1 is capture group-free $\big(w = \epsilon \wedge \mathcal{C}_k = \varnothing\big) \vee \big(\exists m \geq 1 : w = {+\!+}_{i=1}^{m}(\sigma_{i,1} {+\!+} \sigma_{i,2})$ $\wedge \forall i > 1, \big((\sigma_{i,1}, \mathcal{C}_{k,i}) \in \mathcal{L}_c(t_1) \wedge \sigma_{i,2} = \mathcal{C}_{k,i}\big) \wedge \mathcal{C}_k = \mathcal{C}_{k,m}\big)$	Over
Mutable	$(w,\mathcal{C}_k) \in \mathcal{L}_c(\,(?:(t_1)\backslash k)*)$ t_1 is capture group-free $\big(w = \epsilon \wedge \mathcal{C}_k = \varnothing\big) \vee \big(\exists m \geq 1 : w = {+\!+}_{i=1}^{m}(\sigma_{i,1} {+\!+} \sigma_{i,2})$ $\wedge (\sigma_{i,1}, \mathcal{C}_k) \in \mathcal{L}_c(t_1) \wedge \forall i \geq 1, (\sigma_{i,1} = \sigma_{1,1} \wedge \sigma_{i,2} = \sigma_{1,1})\big)$	Unsound

Here, the backreference `\1` and the second instance of `\2` are immutable. However, the first instance of `\2` is mutable: each repetition of the outer capture group under the Kleene plus can change the value of the second (inner) capture group, in turn changing the value of the backreference inside this quantification. For example, the string `"aabbbbb"` satisfies this regex, but `"aababaa"` does not. To fully characterize these distinctions, we introduce the following definition:

Definition 2 (Backreference Type)**.** Let t be the k^{th} capture group of a regex R. Then

1. $\backslash k$ is *empty* if either k is greater than the number of capture groups in R, or $\backslash k$ is encountered before t in a post-order traversal of the AST of R;

2. $\backslash k$ is *mutable* if $\backslash k$ is not empty, and both t and $\backslash k$ are subterms of some quantified term Q in R;

3. otherwise, $\backslash k$ is *immutable*.

When a backreference is empty, it is defined as ϵ, because it refers to a capture group that either is a superterm, e.g., `/(a\1)*/`, or appears later in the term, e.g., `/\1(a)/`.

There are two cases for immutable backreferences. In the first case, the backreference is not quantified. In our model for R, $\mathcal{C}_k$ has already been modeled with an equality constraint, so we can bind the backreference to it. In the second case, the backreference occurs within a quantification; here, the matched word is a finite concatenation of identical copies of the referenced capture group. Both models also incorporate the corner case where the capture group is $\varnothing$ due to alternation or an empty Kleene star. Following the ES6 standard, the backreference evaluates to ϵ in this case.

Mutable backreferences appear in the form $(...t_1...\backslash k...)*$ where t_1 is the k^{th} capture group; ES6 does not support forward referencing of backreferences, so in $(...\backslash k...t_1...)*$, $\backslash k$ is empty. For illustration purposes, the fourth entry of Table 4.4 describes the simplest case for mutable backreferences, other patterns are straightforward generalizations. In this case, we assume t_1 is the k^{th} capture group but is otherwise capture group-free. We can treat the entirety of this term at once: as such, any word in the language is either ϵ, or for some number of iterations, we have the concatenation of a word in the language of t_1 followed by a copy of it. We introduce new variables $\mathcal{C}_{k,i}$ referring to the values of the capture group in each iteration, which encodes the repeated matching on the string until settling on the final value for $\mathcal{C}_k$. In this instance, we need not deal with the possibility that any $\mathcal{C}_{k,i}$ is $\varnothing$, since the quantification ends as soon as t_1 does not match.

Unfortunately, constraints generated from this model are hard to solve and not feasible for current SMT solvers, because they require "guessing" a partition of the matched string variable into individual and varying components. To make solving such queries practical, we introduce an alternative to the previous rule where we treat quantified backreferences as immutable. The resulting model is shown in the last row of Table 4.4. E.g., returning to `/((a|b)\2)+\1\2/`, we accept (`"aaaaaaaaa"`, `"aaaaaaaaa"`, `"aaaa"`, `"a"`), but not (`"aabbaabbb"`, `"aabbaabbb"`, `"aabb"`, `"b"`). We discuss the soundness implications in Section 4.5.4. Quantified backreferences are rare (see Section 4.7.1), so the effect is limited in practice.

4.4.4 Backreferences and Overlapping Capture Groups

We now consider the two cases for overlapping capture groups, nested capture groups contained completely inside a parent, and overlapping

capture groups caused by lookahead. Nested backreferences in JavaScript are always fully enclosed by their parent. In the case of nested immutable backreferences, we bind the final assignment of the capture group to $\mathcal{C}_k$ so that it may later be referenced. Recall that we rewrite any quantified capture group to provide a single assignment for the capture group during preprocessing(Section 4.4). When treating an immutable backreference k, the assignment for the capture group it references has already been bound to a globally named $\mathcal{C}_k$, which is supplied to the encoder during treatment of the backreference. This also functions for nested capture groups, which map to a substring in their parent. For example, given the regular expression `/(aa(a))+\2/`, which includes an immutable backreference to a capture group defined inside another, we rewrite this regular expression to `/(?:aa(?:a))*(aa(a))\2/` during preprocessing to capture the final loop iteration so that $\mathcal{C}_1$ and $\mathcal{C}_2$ are not quantified (Section 4.4.1). After treatment we are left with $w = w_1 +\!+ w_2 +\!+ w_3 +\!+ w_4$, where $w_1 \in \mathcal{L}(\texttt{aaa}*)$, $w_2 \in \mathcal{L}(\texttt{aa})$, $w_3 \in \mathcal{L}(\texttt{a})$, and $w_4 = \mathcal{C}_2$. We also have the capture group assignments $\mathcal{C}_1 = w_2 +\!+ w_3$, $\mathcal{C}_2 = w_3$. Here, we have split the regular expression so that the quantification is capture group and backreference free, and the final loop iteration acts as a single capture group assignment for `\2`. This final iteration is also used to construct the assignment for $\mathcal{C}_1$ and the nested $\mathcal{C}_2$.

If a backreference is mutable then our encoding is under-approximate, locking all repeats of the backreference to a single string. The regular expression `/((.)\2)+/` includes a quantified mutable backreference. After pre-processing this regular expression is rewritten to `/(?:.\2)*((.)\2)/`. In this special case we permit a forward reference to $\mathcal{C}_2$ so that the rewritten quantification can reference the final assignment for `\2`. After encoding this regular expression we are left with $w = w_1 +\!+ w_2 +\!+ w_3$ where $w_1 = +\!+_{i=0}^{n} w_{i,1} +\!+ \mathcal{C}_2$, $w_{i,1} \in \mathcal{L}(\texttt{.})$, allowing unlimited repeats of our quantification, $w_2 \in \mathcal{L}(\texttt{.})$, which forms the assignment for our second capture

group, and $w_3 = \mathcal{C}_2$ which enforces the final backreference. We additionally have the capture group constraints $\mathcal{C}_1 = w_2 \mathbin{+\!\!+} w_3$, and $\mathcal{C}_2 = w_3$.

When treating capture groups that overlap due to lookahead we do not have the guarantee that capture groups will be wholly contained by each other, there may be a partial overlap. For example, `/(?=(.{4})).+\1/` will accept any string where the first four characters are the same as the last four, and in the case of the accepted strings less than 8 characters long there will be some overlap between the captured string in the lookahead and the string used by the backreference. Here, matching the string `"aaaaa"` would bind the first four instances of `"a"`, to $\mathcal{C}_1$, and use the last four instances of `"a"` to match the backreference, with the middle three characters being used to both assign $\mathcal{C}_1$ and to satisfy `\1`. Our encoding satisfies lookahead by generating a string that matches both the lookahead regular expression and the base regular expression and shares capture groups between both. Assignment for capture groups in lookahead follows the same rules as with nested capture groups, binding to a globally defined $\mathcal{C}_k$. In this case the backreference is immutable, since it references a capture group that is not quantified. We would encode this expression by first forming the constraints for the lookahead, $w_1 \mathbin{+\!\!+} w_2$ where $w_1 \in \mathcal{L}(\texttt{.\{4\}})$ and $w_2 \in \mathcal{L}(\texttt{.*})$. Here, we accept any string (since there is no \$ the lookahead implicitly accepts characters after the match), and we bind the first four characters matched to w_1. Next we encode the base regular expression by $w = w_3 \mathbin{+\!\!+} w_4$, where $w_3 \in \mathcal{L}(\texttt{.+})$ and $w_4 = \mathcal{C}_1$. After that, we enforce that our lookahead is satisfied by asserting that $w_3 \mathbin{+\!\!+} w_4 = w_1 \mathbin{+\!\!+} w_2$, and we bind our capture group with the assertion that $\mathcal{C}_1 = w_1$.

Lookahead can also be used inside quantified portions of a regular expression, such as `(?:(?=(.)).)+`. This is encoded in the same way as a mutable backreference, and results in an under-approximate encoding accepting only a single assignment for any quantified capture groups, even if they appear in a quantified lookahead.

4.4.5 Modeling Non-Membership

The model described so far overapproximates membership of a capturing language. We define an analogous model for non-membership of the form $\forall \mathcal{C}_0, \ldots, \mathcal{C}_n : (w, \mathcal{C}_0, \ldots, \mathcal{C}_n) \notin \mathcal{L}_c(R)$. Intuitively, non-membership models assert that for all capture group assignments there exists some partition of the word such that one of the individual constraints is violated. Most models are simply negated. In concatenation and quantification, only language and emptiness constraints are negated, so the models take the form

$$\begin{aligned} w = w_1 \mathbin{+\!\!+} w_2 \\ \wedge\big(\ldots \notin \mathcal{L}_c(\ldots) \vee \ldots \notin \mathcal{L}_c(\ldots) \\ \vee (w_2 = \epsilon \wedge \neg(w_1 = \epsilon \ldots))\big). \end{aligned}$$

In the same manner, the model for capture groups is

$$(w, \mathcal{C}_{i+1}, ..., \mathcal{C}_{i+k}) \notin \mathcal{L}_c(t_1) \wedge \mathcal{C}_i = w.$$

Returning to the example of Section 4.3.3, the negated model for $\forall \mathcal{C}_0, \mathcal{C}_1 : (w, \mathcal{C}_0, \mathcal{C}_1) \notin \mathcal{L}_c(\texttt{(?:a|(b))\1})$ becomes

$$\begin{aligned} &\forall \mathcal{C}_0, \mathcal{C}_1 : w = w_1 \mathbin{+\!\!+} w_1' \mathbin{+\!\!+} w_2' \mathbin{+\!\!+} w_3 \\ &\wedge \mathcal{C}_0 = w_1' \mathbin{+\!\!+} w_2' \\ &\wedge \big(\neg\big((w_1' \in \mathcal{L}(a) \wedge \mathcal{C}_1 = \varnothing) \vee (w_1' \in \mathcal{L}(b) \wedge \mathcal{C}_1 = w_1')\big) \\ &\quad \vee \neg(\mathcal{C}_1 = \varnothing \implies w_2' = \epsilon) \vee \neg(\mathcal{C}_1 \neq \varnothing \implies w_2' = \mathcal{C}_1) \\ &\quad \vee w_1 \notin \mathcal{L}(\texttt{(:?.|\n)*?}) \vee w_3 \notin \mathcal{L}(\texttt{(:?.|\n)*?})\big). \end{aligned}$$

4.5 Matching Precedence Refinement

We now explain the issue of matching precedence (Section 4.5.1) and introduce a counterexample-guided abstraction refinement scheme (Section 4.5.2) to address it. We discuss termination (Section 4.5.3) and the overall soundness of our approach (Section 4.5.4).

4.5.1 Matching Precedence

The model in Tables 4.3 and 4.4 does not account for matching precedence (see Section 4.3.4). A standards-compliant ES6 regex matcher will derive a unique set of capture group assignments when matching a string w, because matching precedence dictates that greedy (non-greedy) expressions match as many (as few) characters as possible before moving on to the next [38]. These requirements are not part of our model, as encoding them directly into SMT would require nesting of quantifiers for each operator, making them impractical for automated solving.

4.5.2 CEGAR for ES6 Regular Expression Models

We eliminate infeasible elements of the capturing language admitted by our model through counter example-guided abstraction refinement (CEGAR).

Algorithm 3 is a CEGAR-based satisfiability checker for constraints modeled from ES6 regexes, which relies on an external SMT solver with classical regular expression and string support and an ES6-compliant regex matcher. The algorithm takes an SMT problem P (derived from the DSE path condition) as a conjunction of constraints, some of which model the $m \geq 0$ original capturing language membership constraints. We number the original capturing language constraints $0 \leq j < m$ so that we can

Algorithm 3: Counterexample-guided abstraction refinement scheme for matching precedence.

Input : Constraint problem P including models for $\mathfrak{m}$ constraints $(w_j, \mathcal{C}_{0,j}, \ldots, \mathcal{C}_{n_j,j}) \boxdot_j \mathcal{L}_c(\mathsf{R}_j)$.

Output: `null` if P is unsatisfiable, or a satisfying assignment for P otherwise

```
1  M := null;
2  Failed := false;
3  do
4  |  M := Solve(P);
5  |  if M = null then
6  |  |  return null;
7  |  Failed := false;
8  |  for j := 0 to m − 1 do
9  |  |  (C♮_{0,j}, ..., C♮_{n_j,j}) := ConcreteMatch(M[w_j], R_j) ;
10 |  |  if (C♮_{0,j}, ..., C♮_{n_j,j}) then
11 |  |  |  if ⊡_j = ∈ then
12 |  |  |  |  for i := 0 to n_j do
13 |  |  |  |  |  if C♮_{i,j} ≠ M[C_{i,j}] then
14 |  |  |  |  |  |  Failed := true;
15 |  |  |  |  |  |  P := P ∧ (w_j = M[w_j] ⟹ ⋀_{0≤i≤n_j} C_{i,j} = C♮_{i,j}) ;
16 |  |  |  else                                  // Non-membership query
17 |  |  |  |  Failed := true;
18 |  |  |  |  P := P ∧ (w_j ≠ M[w_j]);
19 |  |  else                                     // No concrete match
20 |  |  |  if ⊡_j = ∈ then
21 |  |  |  |  Failed := true;
22 |  |  |  |  P := P ∧ (w_j ≠ M[w_j]);
23 while Failed;
24 return M;
```

refer to them as $(w_j, \mathcal{C}_{0,j}, \ldots, \mathcal{C}_{n_j,j}) \boxdot_j \mathcal{L}_c(R_j)$, where $\boxdot \in \{\in, \notin\}$. The algorithm returns `null` if P is unsatisfiable, or a satisfying assignment with correct matching precedence.

In a loop, we first pass the problem P to an external SMT solver. The solver returns a satisfying assignment M or `null` if the problem is unsatisfiable, in which case we are done (lines 4–6). If M is not null, the algorithm uses a concrete regular expression matcher (e.g., Node.js's built-in matcher) to populate concrete capture variables $\mathcal{C}^{\natural}_{i,j}$ corresponding to the words w_j in M.

Lines 8–22 describe how the assignments of capture groups are checked for each regular expression R_j in the original problem P. We first check whether the concrete matcher returned a list of valid capture group assignments, i.e., whether the word $M[w_j]$ from the satisfying assignment matches concretely. If it did, then w_j is a member of the language generated by R_j. If $\boxdot_j = \in$, i.e., the membership constraint was positive, then we must check if the capture group assignments are consistent with those from M (line 13). If they are, we move on to the next regex, otherwise we refine the constraint problem by fixing capture group assignments to their concrete values for the matched word (line 15). Dually, if a modeled non-membership constraint was satisfiable but the word from the current satisfying assignment $M[w_j]$ did match concretely, we refine the problem by asserting that w must not equal that word (line 18). We do the same if $M[w_j]$ did not match concretely but came from a satisfied positive membership constraint (line 22).

If no refinement was necessary we have confirmed the overall assignment satisfies P and return M (line 24). Otherwise, the loop continues with solving the refined problem.

4.5.3 Termination

Unsurprisingly, CEGAR may require arbitrarily many refinements on pathological formulas and never terminate. This is unavoidable due to undecidability [15]. In practice, we therefore impose a limit on the number of refinements, leading to *unknown* as a possible third result. SMT solvers already may timeout or report *unknown* for complex string formulas, so this does not lead to additional problems in practice.

4.5.4 Soundness

Section 4.3.4 demonstrated through example approximation introduced by using our model; in this section we clarify the relationship between this approximation and comment on the soundness of using this model in the analysis of programs. When constructing the rules in Tables 4.3 and 4.4, we followed the semantics of regular expressions as laid out in the ES6 standards document [38]. The ES6 standard is written in a semi-formal fashion, so we are confident that our translation into logic is accurate, but cannot have formal proof. Existing attempts to encode ECMAScript semantics into logic such as JSIL [17] or KJS [99] do not include regexes.

With the exception of the optimized rule for mutable backreferences, our models are overapproximate, because they ignore matching precedence. When the CEGAR loop terminates, any spurious solutions from overapproximation are eliminated. As a result, we have an *exact* procedure to decide (non)-membership for capturing languages of ES6 regexes without quantified backreferences.

In the presence of quantified backreferences, the model after CEGAR termination becomes *underapproximate*. Since DSE itself is an underapproximate program analysis (due to concretization, solver timeouts, and partial exploration), our model and refinement strategy are *sound for DSE*.

4.6 Implementation

We now describe an implementation of our approach in ExpoSE. We explain how we implement our encoding in SMT, so that they can be solved by Z3 (Section 4.6.1). We explain how to model the regex API with capturing language membership (Section 4.6.2).

4.6.1 Using our Encoding in ExpoSE

In order to implement our encoding in ExpoSE we need to express it in SMT. To do this, we have to relax some constraints so that the encoding remains practical in Z3, the SMT solver used by ExpoSE. We encode a regular expression into SMT on-demand during test case execution once per regular match operation. As with our formal encoding, we split the total match into a concatenation of smaller, purely regular, matches so that we can uniquely reference portions of the match and add constraints. For each regular operation we encounter across a test case we generate a set of symbolic strings to act as our partial match strings w_i. The strings are created by a custom regular expression parser, with each string matching a portion of the regular expression. For example, the regular expression `/^.*(aa).*$/` would be parsed into three symbolic strings, S_1, S_2, S_3, where $S_1 \in \mathcal{L}(\texttt{.*})$, $S_2 \in \mathcal{L}(\texttt{aa})$, $S_3 \in \mathcal{L}(\texttt{.*})$. Here, the total match and $\mathcal{C}_0$ are equal to $S_1 +\!+ S_2 +\!+ S_3$ and $\mathcal{C}_1$ is equal to S_2. This concatenation of regular matches will match our JavaScript regular expression. We bind our set of strings to the string being matched S by asserting $S \in \mathcal{L}(\texttt{^.*(aa).*}) \implies S = S_1 +\!+ S_2 +\!+ S_3$.

Quantified segments that are bound to capture groups are rewritten to have a single instance of the expression in the loop repeated immediately after the loop so that it can be directly bound to a string. For example, `(a)*\1` will be rewritten to `a*(a)?\1` and encoded to S_1, S_2, S_3 where $S_1 \in$

$\mathcal{L}(\texttt{a}*)$, $S_2 \in \mathcal{L}(\texttt{a?})$, $S_3 \in \mathcal{L}(\texttt{a?})$. Here, we add the additional constraints $S_2 = S_3$, to enforce the backreference, and $S_2 = \epsilon \implies S_1 = \epsilon$, since we need prevent the solver choosing an assignment in which S_1 is satisfied but S_2 is empty.

To enforce backreferences the parser will generate a series of constraints which must be satisfied in order for the query to hold. For example, a match on the regular expression `/(aa).*\1/` will be rewritten into the strings S_1, S_2, S_3, where $S_1 \in \mathcal{L}(\texttt{aa})$, $S_2 \in \mathcal{L}(\texttt{.}*)$, and $S_3 \in \mathcal{L}(\texttt{aa})$, with the additional constraint that $S_1 = S_3$. ExpoSE will add these constraints to the SMT problems presented to Z3, but will not try to flip these constraints to generate new test cases as it would with other assertions along the path condition. When attempting to generate a string that does not match a regular expression, we query for an assignment where the string is not matched or any of our additional constraints is violated. For example, to find a non-matching string for `/(aa).*\1/` we would look query for S where $S \neq S_1 +\!\!+ S_3 +\!\!+ S_3 \vee S_1 \neq S_3$.

The principle divergence between the implementation in ExpoSE and our encoding is the removal of quantification when treating mutable backreferences. Z3 does not perform well when constraint problems contain quantifiers, so a direct implementation of our encoding would not be practical in cases where a backreference appears in a loop. Instead, any quantified block that contains a backreference is under-approximated to a single repeated string. This restriction is unlikely to impact the real-world performance of the model as we found that only 0.01% of JavaScript regular expressions use quantified backreferences (Table 4.6).

Additionally, current SMT solvers do not allow for non-concrete strings to be used as the regular language in a constraint problem and most solvers do not perform well when string problems and existential quantification are combined. This makes it difficult to encode backreferences that occur inside quantified blocks using the encoding described in Ta-

ble 4.4, since we need to assert that our match includes the backreference and is valid in the regular language. We split this problem into two cases, one where the sub-expression has a constant value, for example, `/(..)\1*/`, and one where some of the matched string can vary between repetitions, for example, `/(..)(.\1)*/`. In the case that the sub-expression cannot change between repeats we use a model similar to the `String.repeat` method, creating a new integer $0 \leq r$ and then asserting that the match for the repetition of sub-expressions is equal to r repeats of the sub-expression match string. The case where the matching string can vary between refinements is more complex. Here, we relax our constraint problem and rely on our refinement scheme in order to find a valid solution. First, we rewrite the regular language to allow any string in the backreference. In our example above, we would replace `/(.\1)*/` with `/(.(:?.*))*/`, which will allow the SMT solver to use any string as the backreference. We then query the SMT solver for a string that would satisfy this relaxed expression and refine it using our CEGAR based refinement strategy. The effectiveness of the refinement strategy can be improved by enumerating these sub-expressions during refinement, removing the quantification entirely and instead generating SMT problems that contain a fixed number of repeats of the sub-expression. The refinement strategy will then change the number of repetitions until it finds a satisfying assignment or the refinement limit is reached.

4.6.2 Modeling the Regex API

The ES6 standard specifies several methods that evaluate regexes [38]. We follow its specified pseudocode for `RegExp.exec(s)` to implement matching and capture group assignment in terms of capturing language membership in Algorithm 4. Notably, our algorithm implements support for all flags and operators specified for ES6.

`RegExp.test(s)` is precisely equivalent to the expression `RegExp.exec(s) !==undefined` (Algorithm 5). In the same manner, one can construct models for other regex functions defined for ES6. Our implementation includes partial models for the remaining functions that allow effective test generation in practice but are not semantically complete.

Algorithm 4: RegExp.exec(input)

```
1  input′ := '⟨' + input + '⟩';
2  if sticky or global then
3  |   offset := lastIndex > 0 ? lastIndex + 1 : 0;
4  |   input′ := input′.substring(offset);
5  source′ := '(:?.|\n)*?(' + source + ')(:?.|\n)*?';
6  if caseIgnore then
7  |   source′ := rewriteForIgnoreCase(source′);
8  if (input′, C0, ..., Cn) ∈ Lc(source′) then
9  |   Remove ⟨ and ⟩ from (input′, C0, ..., Cn);
10 |   lastIndex := lastIndex + C0.startIndex + C0.length;
11 |   result := [C0, ..., Cn];
12 |   result.input := input;
13 |   result.index := C0.startIndex;
14 |   return result;
15 else
16 |   lastIndex := 0;
17 |   return undefined;
```

Algorithm 5: RegExp.test(input)

```
1  if this.exec(realInput) !== undefined then
2  |   return true;
3  else
4  |   return false;
```

Algorithm 4 first processes flags to begin from the end of the previous

match for sticky or global flags, and it rewrites the regex to accept lower and upper case variants of characters for the ignore case flag.

We introduce the ⟨ and ⟩ meta-characters to *input* which act as markers for the start and end of a string during matching. Next, if the sticky or global flags are set we slice *input* at `lastIndex` so that the new match begins from the end of the previous. Due to the introduction of our meta-characters `lastIndex` needs to be offset by 1 if it is greater than zero. We then rewrite the regex source to allow for characters to precede and succeed the match. Note that we use `(?:.|\n)*?` rather than `.*?` because the wildcard . consumes all characters except line breaks in ECMAScript regexes. To avoid adding these characters to the final match we place the original regex source inside a capture group. This forms $\mathcal{C}_0$, which is defined to be the whole matched string [38]. Once preprocessing is complete we test whether the input string and fresh string for each capture group are within the capturing language for the expression. If they are then a results object is created which returns the correctly mapped capture groups, the input string, and the start of the match in the string with the meta-characters removed. Otherwise `lastIndex` is reset and **`undefined`** is returned.

4.7 Evaluation

We now empirically answer the following research questions:

(**RQ1**) Are non-classical regexes an important problem in JavaScript?

(**RQ2**) Does accurate modeling of ES6 regexes make DSE-based test generation more effective?

(**RQ3**) Does the performance of the model and the refinement strategy enable practical analysis?

We answer the first question with a survey of regex usage in the wild (Section 4.7.1). We address RQ2 by comparing our approach against an existing partial implementation of regex support in ExpoSE [79] on a set of widely used libraries (Section 4.7.2). We then measure the contribution of each aspect of our approach on over 1,000 JavaScript packages (Section 4.7.3). We answer RQ3 by analyzing solver and refinement statistics per query (Section 4.7.4).

4.7.1 Surveying Regex Usage

We focus on code written for Node.js, a popular framework for standalone JavaScript. Node.js is used for both server and desktop applications, including popular tools *Slack* and *Skype*. We analyzed 415,487 packages from the NPM repository, the primary software repository for open source Node.js code. Nearly 35% of NPM packages contain a regex, 20% contain a capture group and 4% contain a backreference.

Methodology We developed a lightweight static analysis that parses all source files in a package and identifies regex literals and function calls. We do not detect expressions of the form `new RegExp(...)`, as they would generally require a more expensive static analysis. Our numbers therefore provide a lower bound for regex usage.

Results We found regex usage in JavaScript to be widespread, with 145,100 packages containing at least one regex out of a total 415,487 scanned packages. Table 4.5 lists the number of NPM packages containing regexes, capture groups, backreferences, and backreferences appearing within quantification. Note that a significant number of packages make use of capture groups and backreferences, confirming the importance of supporting them.

Table 4.5: Regex usage by NPM package.

Feature	**Count**	%
Packages on NPM	415,487	100.0%
. . . with source files	381,730	91.9%
. . . with regular expressions	145,100	34.9%
. . . with capture groups	84,972	20.5%
. . . with backreferences	15,968	3.8%
. . . with quantified backreferences	503	0.1%

Table 4.6: Feature usage by unique regex.

Feature	**Total**	%	**Unique**	%
Total Regex	9,552,546	100%	305,691	100%
Capture Groups	2,360,178	24.71%	119,051	38.94%
Global Flag	2,620,755	27.44%	90,356	29.56%
Character Class	2,671,565	27.97%	71,040	23.24%
Kleene+	1,541,336	16.14%	67,508	22.08%
Kleene*	1,713,713	17.94%	66,526	21.76%
Ignore Case Flag	1,364,526	14.28%	58,831	19.25%
Ranges	1,273,726	13.33%	52,155	17.06%
Non-capturing	1,236,533	12.94%	25,946	8.49%
Repetition	360,578	3.7%	17,068	5.58%
Kleene* (Lazy)	230,060	2.41%	13,250	4.33%
Multiline Flag	137,366	1.44%	10,604	3.47%
Word Boundary	336,821	3.53%	9,677	3.17%
Kleene+ (Lazy)	148,604	1.56%	6,072	1.99%
Lookaheads	176,786	1.85%	3,123	1.02%
Backreferences	64,408	0.67%	2,437	0.80%
Repetition (Lazy)	2,412	0.03%	221	0.07%
Quantified BRefs	1,346	0.01%	109	0.04%
Sticky Flag	98	<0.01%	60	0.02%
Unicode Flag	73	<0.01%	48	0.02%

Table 4.6 reports statistics for all 9M regexes collected, giving for each feature the fraction of expressions including it. Many regexes in NPM packages are not unique; this appears to be due to repeated inclusion of the same literal (instead of introduction of a constant), the use of online solutions to common problems, and the inclusion of dependencies (foregoing proper dependency management). To adjust for this, we provide data for both all expressions encountered and for just unique expressions. In both cases, there are significant numbers of capture groups, backreferences, and other non-classical features. As the occurrence rate of quantified backreferences is low, we do not differentiate between mutable and immutable backreferences.

Conclusions Our findings confirm that regexes are widely used and often contain complex features. Of particular importance is a faithful treatment of capture groups, which appear in 20.45% of the packages examined. On the flip side, since quantified backreferences make up just 0.01% of regexes, the optimization introduced in Section 4.4.3 will rarely lead to additional underapproximation during DSE.

4.7.2 Improvement Over State of the Art

We compare our approach against the original ExpoSE [79], which is, to our knowledge, the only available and functional implementation of regex support in JavaScript.

Methodology We evaluated statement coverage achieved by both versions of ExpoSE on a set of libraries, which we chose for their popularity (with up to 20M weekly downloads) and use of regex. This includes the three libraries minimist, semver, and validator, which the first version of ExpoSE was evaluated on [79]. To fairly compare original ExpoSE

Table 4.7: Statement coverage with our approach (**New**) vs. [79] (**Old**) and the relative increase (**+**) on popular NPM packages (**Weekly** downloads). **LOC** are lines loaded and **RegEx** are regular expression functions symbolically executed.

Library	Weekly	LOC	RegEx	Old(%)	New(%)	+(%)
babel-eslint	2500k	23047	902	21.0	26.8	27.6
fast-xml-parser	20k	706	562	3.1	44.6	1338.7
js-yaml	8000k	6768	78	4.4	23.7	438.6
minimist	20000k	229	72530	65.9	66.4	0.8
moment	4500k	2572	21	0.0	52.6	∞
query-string	3000k	303	50	0.0	42.6	∞
semver	1800k	757	616	51.7	46.2	-10.6
url-parse	1400k	322	448	60.9	71.8	17.9
validator	1400k	2155	94	67.5	72.2	7.0
xml	500k	276	1022	60.2	77.5	28.7
yn	700k	157	260	0.0	54.0	∞

against our extension, we use the original automated library harness for both. Therefore we do not take advantage of other improvements for test generation, such as symbolic array support, which we have added in the course of our work. We re-executed each package six times for one hour each on both versions, using 32-core machines with 256GB of RAM, and averaged the results. We limited the refinement scheme to 20 iterations, which we identified as effective in preliminary testing (see Section 4.7.4).

Results Table 4.7 contains the results of our comparison. To provide an indication of program size, we use the number of lines of code loaded at runtime (JavaScript's dynamic method of loading dependencies makes it hard to determine a meaningful LOC count statically).

The results demonstrate that ExpoSE extended with our model and refinement strategy can improve coverage more than tenfold on our sample

of widely-used libraries. In the cases of moment, query-string, and yn, the lack of ES6 support in the original ExpoSE prohibited meaningful analysis, leading to 0% coverage. In the case of semver, we see a decrease in coverage if stopped after one hour. This is due to the modeling of regex increasing solving time (see also Section 4.7.4). The coverage deficit disappears when executing both versions of ExpoSE with a timeout of two hours.

Conclusions We find that our modifications to ExpoSE make test generation more effective in widely used libraries using regex. This suggests that the new method of solving regex queries presented in this chapter has a substantial impact on practical problems in DSE. We also see that other improvements to ExpoSE, such as ES6 support, have affected coverage. Therefore, we continue with an evaluation of the individual aspects of our model.

4.7.3 Breakdown of Contributions

We now drill down into how the individual improvements in regex support are contributing to increases in coverage.

Methodology From the packages with regexes from our survey Section 4.7.1, we developed a test suite of 1,131 NPM libraries for which ExpoSE is able to automatically generate a meaningful test harness. In each of the libraries selected, ExpoSE executed at least one regex operation on a symbolic string, which ensures that the library contains some behavior relevant to the scope of this chapter. The test suite constructed in this manner contains numerous libraries that are dependencies of packages widely

used in industry, including Express and Lodash.[1]

Automatic test generation typically requires a bespoke test harness or set of parameterized unit tests [125] to achieve high coverage in code that does not have a simple command line interface, including libraries. ExpoSE's harness explores libraries fully automatically by executing all exported methods with symbolic arguments for the supported types `string`, **`boolean`**, `number`, **`null`** and **`undefined`**. Returned objects or functions are also subsequently explored in the same manner.

We executed each package for one hour, which typically allowed to reach a (potentially initial) coverage plateau, at which additional test cases do not increase coverage further. We break down our regex support into four levels and measure the contribution and cost of each one to line coverage and test execution rate (Table 4.8). As baseline, we first execute all regex methods concretely, concretizing the arguments and results. In the second configuration, we add the model for ES6 regex and their methods, including support for word boundaries and lookaheads, but remove capture groups and concretize any accesses to them, including backreferences. Third, we also enable full support for capture groups and backreferences. Fourth, we finally also add the refinement scheme to address overapproximation.

Results Table 4.8 shows, for each level of support, the number and percentage of target packages where coverage improved; the geometric mean of the relative increase in coverage; and the mean test execution rate. The final row shows the effect of enabling full support compared to the baseline. Note that the number of packages improved is less than the sum of the rows above, since the coverage of a package can be improved by multiple features.

[1] Raw data for the experiments, including all package names, is available at `https://github.com/ExpoSEJS/PLDI19-Raw-Data`.

Table 4.8: Breakdown of how different components contribute to testing 1,131 NPM packages, showing number (#) and fraction (%) of packages with coverage improvements, the geometric mean of the relative coverage increase from the feature (**Cov**), and test execution rate.

Regex Support Level	**Improved** #	**Improved** %	**Cov +(%)**	$\frac{\text{Tests}}{\text{min}}$
Concrete Regular Expressions	-	-	-	11.46
+ Modeling RegEx	528	46.68%	+6.16%	10.14
+ Captures & Backreferences	194	17.15%	+4.18%	9.42
+ Refinement	63	5.57%	+4.17%	8.70
All Features vs. Concrete	617	54.55%	+6.74%	

In a dataset of this size that includes many libraries that make only little use of regex, average coverage increases are expected to be small. Nevertheless, we see that dedicated support improves the coverage of more than half of packages that symbolically executed at least one regex function. As expected, the biggest improvement comes from supporting basic symbolic execution of regular expressions, even without capture groups or regard for matching precedence. However, we see further improvements when adding capture groups, which shows that they indeed affect program semantics. Refinement affects fewer packages, although it significantly contributes to coverage where it is required. This is because a lucky solver may generate correct inputs on the first attempt, even in ambiguous settings.

On some libraries in the dataset, the approach is highly effective. For example, in the manifest parser *n4mf-parser*, full support improves coverage by 29% over concrete; in the format conversion library *sbxml2json*, by 14%; and in the browser detection library *mario*, by 16%. In each of these packages the refinement scheme contributed to the improvement in cov-

erage. In general, the largest increases are seen in packages that include regular expression-based parsers.

Each additional feature causes a small decrease in average test execution rate. Although a small fraction (~1%) of queries can take longer than 300s to solve, concurrent test execution prevents DSE from stalling on a single query.

Conclusions Full support for ES6 regex improves performance of DSE of JavaScript in practice at a cost of a 16% increase in execution time (RQ2). An increase in coverage at lower execution rate in a fixed time window suggests that full regular expression support increases the quality of individual test cases.

4.7.4 Effectiveness on Real-World Queries

We now investigate the performance of the model and refinement scheme to answer RQ3. Finally, we also discuss the refinement limit and how it affects analysis.

Methodology We collected data on queries during the NPM experiments (Section 4.7.3) to provide details on SMT query success rates and execution times, as well as on the usage of the refinement scheme.

Results We found that 753 (66%) of the 1,131 packages tested executed at least one query containing a capture group or backreference. Of these packages, 653 (58% overall) contained at least one query to the SMT solver requiring refinement, and 134 (12%) contained a query that reached the refinement limit.

In total, our experiments executed 58,390,184 SMT queries to generate test cases. As expected, the majority do not involve regexes, but they form

Table 4.9: Solver times per package and query.

	Constraint Solver Time		
Packages/Queries	**Minimum**	**Maximum**	**Mean**
All packages	0.04s	12h 15m	2h 34m
With capture groups	0.20s	12h 15m	2h 40m
With refinement	0.46s	12h 15m	2h 48m
Where refinement limit is hit	3.49s	11h 07m	3h 17m
All queries	0.001s	22m 26s	0.15s
With capture groups	0.001s	22m 26s	5.53s
With refinement	0.005s	18m 51s	22.69s
Where refinement limit is hit	0.120s	18m 51s	58.85s

a significant part: 4,489,581 (7.6%) queries modeled a regex, 645,295 (1.1%) modeled a capture group or backreference, 74,076 (0.1%) required use of the refinement scheme and 2,079 (0.003%) hit the refinement limit. The refinement scheme was overwhelmingly effective: only 2.8% of queries with at least one refinement also reached the refinement limit (0.003% of all queries where a capture group was modeled). Of the refined SMT queries, the mean number of refinements required to produce a valid satisfying assignment was 2.9; the majority of queries required only a single refinement.

Table 4.9 details time spent processing SMT problems per-package and per-query. We provide the data over the four key aspects of the problem: we report the time spent in the constraint solver both per package and per query in total, as well as the time in the constraint solver for the particularly challenging parts of our strategy. We found that the use of refinements increased the average per-query solving time by a factor of four; however, this is dominated by SMT queries that hit the refinement limit, which took ten times longer to run on average. The low minimum time spent in the solver in some packages can be attributed to packages where

a regular expression was encountered early in execution but limitations in the test harness or function models (unrelated to regular expressions) prevented further exploration.

Conclusions We find the refinement scheme is highly effective, as it is able to solve 97.2% of encountered constraint problems containing regexes. It is also necessary, as 10% of queries containing a capture group had led to a spurious satisfying assignment and required refinement.

Usually, only a small number of refinements are required to produce a correct satisfying assignment. Therefore, even refinement limits of five or fewer are feasible and may improve performance with low impact on coverage.

4.7.5 Threats to Validity

We now look at potential issues affecting the validity of our results, in particular soundness, package selection, and scalability.

Soundness In addition to soundness of the model (see Section 4.5.4), one must consider the soundness of the implementation. In the absence of a mechanized specification for ES6 regex, our code cannot be proven correct, so we use an extensive test suite for validation. However, assuming the concrete matcher is specification-compliant, Algorithm 3 will, if it terminates, return a specification-compliant model of the constraint formula even if the implementation of Section 4.4 contains bugs. In the worst case, the algorithm would not terminate, leading to timeouts and loss of coverage. Bugs could therefore only have lowered the reported coverage improvements.

Package Selection and Harness In Section 4.7.3, we chose packages identified in our survey (Section 4.7.1) where our generic harness encountered a regular expression within one hour of DSE. This allowed us to focus the evaluation on regex support as opposed to evaluating the quality of the harness (and having to deal with unreachable code in packages). Use of this harness may have limited package selection to simpler, unrepresentative libraries. However, we found that simple APIs do not imply simple code: the final dataset contains several complex packages, such as language parsers, and the types of regexes encountered were in line with the survey results. On simple code we found that ExpoSE would often reach 100% coverage; failure to do so was either due to the complexity of the code or the lack of support for language features unrelated to regex and APIs that would require additional modeling (e.g., the file system).

Scalability Scalability is a challenge for DSE in general, and is not specific to our model for regex. Empirically, execution time for a single test (instrumentation, execution, and constraint generation) grows linearly with program size, as does the average size of solver queries. The impact of query length on solving time varies, but does not appear to be exacerbated by our regex model. In principle, our model is compatible with compositional approaches [47, 8] and state merging [71, 10], which can help DSE scale to large programs.

The scalability of our approach suffices for Node.js, however: JavaScript has smaller LOC counts than, e.g., C++, and code on NPM is very modular. For instance, among the top 25 most depended-upon NPM libraries, the largest is 30 KLOC (but contains no regex). Several packages selected for our evaluation, such as babel-eslint, had between 20-30 KLOC and were meaningfully explored with the generic harness.

4.8 Related Work

In theory, regex engines can be symbolically executed themselves through the interpreter [20]. While this removes the need for modeling, in practice the symbolic execution of the entire interpreter and regex engine quickly becomes infeasible due to path explosion.

There have been several other approaches for symbolic execution of JavaScript; most include some limited support for classical regular expressions. Li et al. [75] presented an automated test generation scheme for programs with regular expressions by on-line generation of a matching function for each regular expression encountered, exacerbating path explosion. Saxena et al. [111] proposed the first scheme to encode capture groups through string constraints. Li and Ghosh [74] and Li, Andreasen, and Ghosh [73] describe a custom browser and symbolic execution engine for JavaScript and the browser DOM, and a string constraint solver *PASS* with support for most JavaScript string operations. Although all of these approaches feature some support for ECMAScript regex (such as limited support for capture groups), they ignore matching precedence and do not support backreferences or lookaheads.

Thomé et al. [123] propose a heuristic approach for solving constraints involving unsupported string operations. We choose to model operations unsupported by the solver and employ a CEGAR scheme to ensure correctness. Abdulla et al. [2] propose the use of a refinement scheme to solve complex constraint problems, including support for context-free languages. The language of regular expressions with backreferences is not context-free [23] and, as such, their scheme does not suffice for encoding all regexes; however, their approach could serve as richer base theory than classic regular expressions. Scott, Flener, and Pearson [112] suggest backreferences can be eliminated via concatenation constraints, however they do not present a method for doing so.

Further innovations from the string solving community, such as work on the decidability of string constraints involving complex functions [29, 58] or support for recursive string operations [127, 126], are likely to improve the performance of our approach in future. We incorporate our techniques at the level of the DSE engine rather than the constraint solver, which allows our tool to leverage advances in string solving techniques; at the same time, we can take advantage of the native regular expression matcher and can avoid having to integrate implementation language-specific details for regular expressions into the solver.

A previous survey of regex usage across 4,000 Python applications [26] also provides a strong motivation for modeling regex. Our survey extends this work to JavaScript on a significantly larger sample size.

Systematic Generation of Conformance Tests

5

In this chapter, we introduce a novel approach to generate new conformance tests for JavaScript standard library implementations through symbolic execution of polyfills. We use this new approach to find 96,470 new test cases, discovering bugs in a popular built-in implementation and improving the overall coverage of Test262, the JavaScript conformance test suite.

Attribution This work is currently in submission. It is a joint work with Johannes Kinder at Research Institute CODE, Bundeswehr University Munich. I contributed to all aspects of this work.

5.1 Introduction

Smooth interoperability between interpreters by different JavaScript vendors is ensured by the common ECMAScript standard and its test suite. While there is a formally verified reference interpreter for the core language, which closely follows the natural language specification [17], all fully-fledged implementations in browsers and other systems rely on test suites to ensure conformance. The main mechanism for validating conformance to the ECMAScript standard is Test262 [39], a manually curated test suite with the goal of covering all observable behavior of the ECMAScript specification.

Because Test262 is created manually, it is likely that it does not exhaustively test implementation behaviour. This can lead to interpreters which contain legal behavior that is not exercised by Test262. When corner cases remain untested, there is a potential for hidden divergences from the specification. DSE seems ideally suited to fill this gap and exercise hidden behavior. In principle, DSE allows generation of test cases for *implementations* of ECMAScript language semantics.

In this chapter, we use ExpoSE to analyze straightforward implementations of JavaScript language features to generate new test cases. These are then executed on a portfolio of JavaScript interpreters, using a *majority vote* to decide the correct behavior. We find that polyfills, JavaScript implementations of built-in language features, are ideally suited for this task: polyfills are directly executable and provide more detail than the ECMAScript specification; at the same time, they are much more compact than implementations in an interpreter. Since these polyfill implementations are written in JavaScript we can symbolically analyze them directly using ExpoSE. Polyfills also have the advantage of providing a clear entry point for each supported feature, which makes directed testing possible. In interpreters, the implementation of language semantics is hidden behind parsing and translation layers, far removed from any external entry point that could be controlled by a test generation tool.

Entry points of polyfill code can require structured input such as objects and arrays, so we extend ExpoSE to support a dynamic encoding approach for objects and arrays that does not require specific fields or typing. We intercept accesses to object fields and array elements by the JavaScript program at runtime and generate test cases for each meaningful outcome, handling possible name aliasing, different field types, and the special quirks of JavaScript arrays.

We evaluate our approach using ExpoSE. We automatically generate a rich suite of tests from the Mozilla Developer Network polyfills (`mdn-polyfills`) and `core-js` that we run against SpiderMonkey, Node.js, and Quickjs. In summary, in this chapter we develop a new generic encoding for objects and arrays (C3)) and use this new support to systematically explore polyfill implementations, finding inputs uncovered by existing conformance suites (C4). In particular, we make the following contributions:

- We present a methodology for automated generation of conformance tests from polyfills. We employ differential testing across multiple implementations to compensate the lack of testing oracles (Section 5.2).
- We define a model for symbolic objects and symbolic arrays that dynamically synthesizes test inputs in untyped JavaScript code (Section 5.3).
- We improve the state of the art in conformance testing of ECMAScript implementations through our methodology. Using our new tests, we found 17 bugs in polyfill implementations and were able to augment the coverage of Test262 in JavaScript interpreters by up to 15% (Section 5.4).

5.2 Conformance Testing using PolyFills

We generate new implementation conformance tests for JavaScript interpreters through symbolic execution of polyfills; implementations of built-in methods in JavaScript. Existing supplementary test suites like Test262, the official ECMA test suite [39], are created by exploring conditions in the specification. Since they are manually curated, bugs may be missed – particularly when treating edge cases. Here, we use a DSE engine to automatically find subtleties of built-in implementations through symbolic exploration of polyfills, and then apply the generated tests to other implementations, since they should all behave identically.

5.2.1 Polyfills

With each evolution of JavaScript there is a period of time where new feature support will not be ubiquitous, since each vendor will take time to

update their implementation. To remedy this, polyfills, short programs which implement built-in methods, have become common. A polyfill will inject a built-in into the standard library at runtime if it is not already supported by the host interpreter.

In this chapter we use two polyfill packages, `core-js` [102] and `mdn-polyfills` [64], to test our approach. These libraries contain polyfill implementations of standard library methods added in the ES6 standard. `core-js` is the de-facto standard for polyfills with 78,000,000 monthly downloads. `mdn-polyfills` is less highly depended on, with 72,000 monthly downloads on NPM, the largest JavaScript package repository.

5.2.2 Architecture

We generate new test cases by dynamic symbolic execution of polyfills. Analysis of these polyfills will generate inputs that explore the intricacies of built-in specifications, but we do not have a ground-truth for the correct behavior of a test case. To solve this problem we use a suite of interpreters and have them vote on the correct answer. This acts an oracle to identify when an implementation is incorrect, and only requires manual intervention when two or more implementations diverge.

We split our implementation into two components, the test case generator, and the test case executor. The test case generator uses ExpoSE to generate new test cases. The test case executor executes a test suite extracted from the symbolic executions and checks that each of our selected interpreters is implemented correctly.

5.2.3 Test Case Generation

We generate new test cases by symbolically executing polyfills using ExpoSE. Figure 5.1 provides an overview of the architecture. We begin by

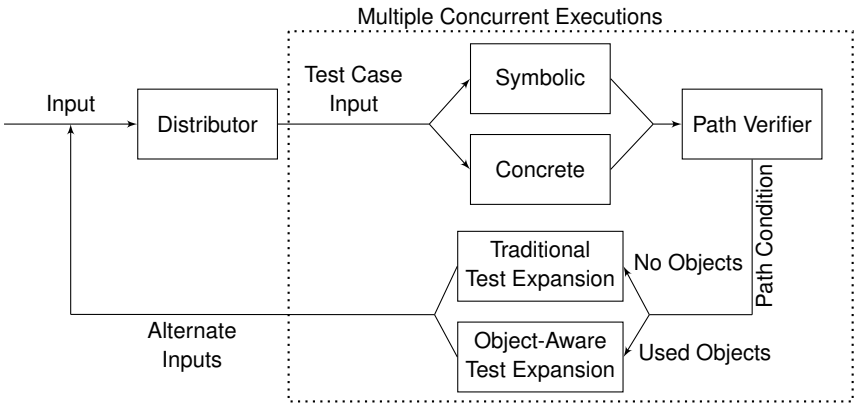


Figure 5.1: Test Case Generator Overview.

supplying the test apparatus with a target built-in and the number of arguments the method expects. The apparatus then constructs a series of symbolic inputs to use as arguments. ExpoSE then analyzes the generated test harness and begins to output a series of test cases. We also execute each new test case in `Node.js` to remove spurious outputs triggered by errors inside ExpoSE. We forward the result of the concrete and symbolic executions to the path verifier, a tool that double-checks that the concrete and the symbolic result are identical. If they are not, then the test case is discarded. Otherwise, we add the test case to the generated test case suite, and the symbolic path condition is used to generate new test cases. We use an object-aware type encoding when finding alternate test cases to explore more of our target polyfills (Section 5.3).

Modifications to ExpoSE In addition to adding support for symbolic objects (Section 5.3.2), we made additions to ExpoSE so that it can treat type-coercions we observed in existing polyfills. In JavaScript, numeric values may be either integers or floating point values and there is no idiomatic

way to ensure that a value is an integer. If a developer wants to force a number to be an integer they often use a bitwise operation to force the coercion, since bitwise logic truncates operands to integers. To illustrate this, the `targetLength = targetLength >> 0` ensures that the length is an integer with a bitshift by 0. ExpoSE did not accurately model bitwise operations and other esoteric behaviours of the type system, but these are used often in built-in implementations, so we modified the engine to support them.

5.2.4 Test Case Executor

The second component in our design is the test case executor. Our automatically generated test cases do not have a predetermined expected result because the result found during symbolic execution may be from a flawed implementation. Instead of using predetermined test case results, we use a consensus-based approach to detect incorrect implementations, illustrated in Figure 5.2. We execute each test case in several different interpreters. Each interpreter has a different interface so we generate a compatible test through a test translator that takes a test input and returns a program compatible with a specific engine. For polyfills, we inject the target method into a Node.js instance, replacing any existing implementation. We then execute each of these programs and collect the output.

Once the test case has been executed by each implementation, we pass the results to a voting mechanism. The voting mechanism looks for implementations where behaviour diverges from the others. If the outcome of a built-in call diverges either in exception type or result then we say that the interpreters disagree and raise an error. Specifically, we say that an implementation disagrees if either of the following two conditions are violated:

1. If a test case throws an exception and others do not, or the exception

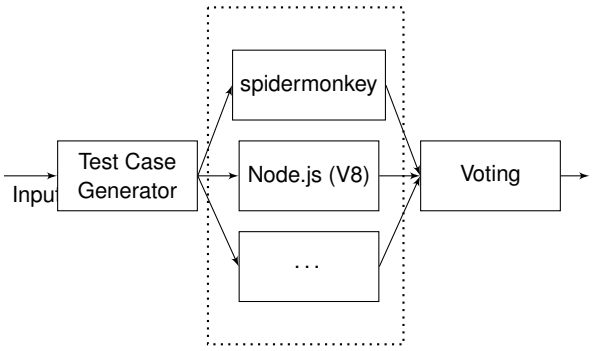


Figure 5.2: Test Case Executor Overview.

type differs from other implementations.

2. If a test case has output different from the others.

Through this mechanism we can compare the behaviour of methods as long as they do not return functions, since there is no standard way of comparing function implementation details in JavaScript. We do not compare the exact text of exceptions because it is not specified by the ECMAScript specification. If a single implementation disagrees then it is marked as incorrect. When multiple implementations disagree, we cannot make any conclusion about correct behavior and mark the test case for manual review.

5.3 Representing Symbolic Data Structures in JavaScript

To allow automated generation of structued test inputs for built-in methods, we require a method for maintaining symbolic objects and arrays. We developed new encodings for untyped symbolic objects, i.e., symbolic ob-

jects with no pre-specified property names or types (Section 5.3.2), arrays of mixed types (Section 5.3.3), and for homogeneously typed arrays (Section 5.3.4).

5.3.1 Motivation

Support for symbolic objects is key to the exploration of built-ins because it allows thorough exploration of object and array centric built-ins. More subtly, support allows the DSE engine to consider esoteric type-checking in built-in methods. The specification includes precise but unintuitive rules on how input values are to be interpreted and when type contract violations should raise an error. To highlight how an object encoding can improve coverage of these edge cases, we now consider `Array.prototype.find`.

Usually, this method is given an array as its base argument and a predicate. The array is then searched, left to right, until a value satisfying the predicate is found. If no values satisfy the predicate then **`undefined`** is returned. For example, `[11,23,20].find((x)=> x % 2 == 0)` would yield `20`, the first even number in the array. If we look at the method specification, we see that there is a quirk to this method contract. The method accepts any object which looks like an array (i.e., any object with a length property). Because of this `Array.prototype.find.call({0: 11, 1: 23, 2: 20, length: 3}, (x)=> x % 2 == 0)` behaves equivalently to the previous example, but `Array.prototype.find.call({0: 11, 1: 23, 2: 20}, (x)=> x % 2 == 0)` will yield **`undefined`**, since the object does not specify a length.

One further quirk is the coercion of `length` to an integer. The specification does not reject non-integer length properties, leading to a coercion that resolves `Array.prototype.find.call({0: 20, length: true}, (x)=> x % 2 == 0)` to `20`, but `Array.prototype.find.call({0: 20, length`

`: false}, (x)=> x % 2 == 0)` to `undefined`, as `true` is coerced to `1`.

In `mdn-polyfills` these checks are implemented by `var o = Object (this)`, which ensures the value is either an array or an object, followed by `var len = o.length >>> 0`, which selects the length of the object and ensures it is an integer using type coercion [92]. In this case, if the length property is not an integer then it is first coerced to a number and subsequently truncated to an integer. Through our encoding of objects, we can synthesize useful test cases for such behavior.

5.3.2 Symbolic Objects

Representing JavaScript objects in DSE engines is challenging due to the dynamic type system. Existing SMT solvers do not support a "theory of objects." Recreating a dynamic datatype in SMT and implementing the required reasoning would complicate solver-side logic and effectively move much language-specific reasoning into the SMT solver, which is designed to be language-agnostic. So instead we opt to translate the reasoning about symbolic objects into a form that can be represented as an SMT problem over primitive types. We develop an intermediate encoder that outputs typed SMT problems directly in the DSE engine. The intermediate encoder does not require solver extensions, instead simulating symbolic objects by following every object operation along a program trace and exploring feasible alternatives.

We model symbolic objects by tracking property lookups and updates on objects. For this, we rewrite all property lookups to use the common interface `getProperty(object, propertyName)`, and all property updates to use the common interface `setProperty(object, propertyName, value)`. We supply `getProperty` with `object`, the object operated on, and `propertyName`, a string indicating which property is being accessed. `setProperty` is additionally supplied with

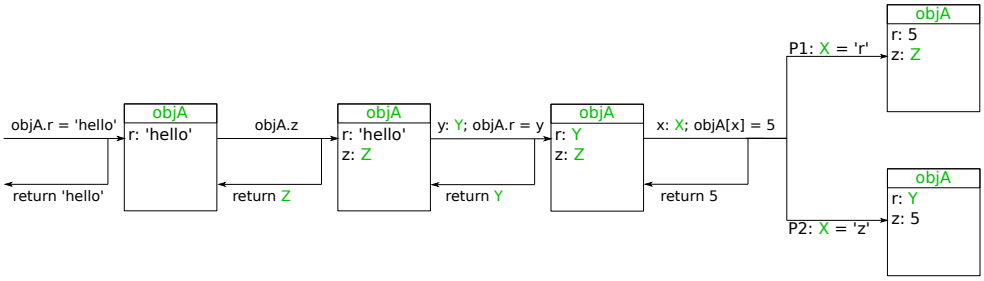


Figure 5.3: Illustration of symbolic object modeling.

`value`, which is the new value for the given property. With this instrumentation, we can keep track of all object operations during execution, updating the symbolic state when appropriate. We instrument arrays similarly, with `getProperty` and `setProperty` interfaces for all property lookups. They differ in the typing of property names, where they also accept integer values, since arrays can contain integer and string property names.

The root of our encoding is the creation of new symbolic values for properties we have not seen before while returning the value stored in a state for properties that we have previously set. Our encoding for objects is illustrated in Figure 5.3. Here we see how a symbolic object behaves under various typical operations.

The first step in Figure 5.3 shows how symbolic objects support fully concrete operations. Here, we record the concrete value supplied to be returned on subsequent lookups. When we perform a lookup for a property that we have not encountered before, we introduce a new symbolic value to the program and set it to the appropriate property. The created symbol does not have a fixed type, and instead uses existing support in the DSE engine to explore the program as if it were any of the supported symbolic types. In the case of ExpoSE, the DSE engine we use in this chapter, the symbolic types supported are undefined, null, boolean, number, string

and through our encoding also objects and arrays. The second operation illustrates this process on `objA` in Figure 5.3, where the new symbol Z is introduced and assigned to the property z.

Next, we want to set a property with a concrete property name but a symbolic value. As with a fully concrete set property, we record the supplied property value to the object state; here, it makes no difference if the supplied properties are concrete or symbolic.

The last matter that we address in this example is how we approach setting and getting of properties with symbolic property names. Here, we attempt to create new test cases for each of the previously recorded properties of an object - even if they are subsequently deleted. The final operation illustrates this in the figure, where we write a concrete value with a symbolic property name, leading to two new paths. One where the property `r` is replaced with 5, and another where the property `z` is replaced with `5`. This final step causes under-approximation in our encoding: We do not enumerate on properties that we have not seen previously. We could, in principle, support this through the enumeration of all possible property names, but this would lead to an infeasible number of paths to explore.

To implement our encoding we instrument the `getProperty` and `setProperty` operations executed by a program with our object encoder, detailed in Algorithm 6 and Algorithm 7. We send any portions of the program trace involving symbolic objects to these intermediate encoders. The distinction between known and unknown properties is core to our symbolic object encoding, with the symbolic object keeping track of any properties that it has encountered before. Each symbolic object is created with an initial map of known properties. A `setProperty` operation with a concrete property name marks that property as known, and it will then on return the supplied value to preserve JavaScript semantics. The complementary `getProperty` operation on a fixed property has normal behav-

Algorithm 6: Symbolic object encoder – getProperty(base, property).

```
1  if property is symbolic then
2      for knownProp in base do
           // Attempt to generate a test case for each
              known property
3          if Concrete(property) = knownProp then
4              PC ← PC ∧ property = knownProp;
5          else
6              PC ← PC ∧ property ≠ knownProp;
7      return base[property];
8  else
9      if property not in base then
10         base[property] =fresh symbol;
11     return base[property];
```

ior, returning the (potentially symbolic) known value from the object. So far, this encoding is straightforward and preserves standard semantics, returning known property values for an object. However, in order to explore the program symbolically, we need a special approach to treating unknown field lookups. Whenever a program performs a `getProperty` on an unknown field we return a new, untyped, symbolic variable rather than **`undefined`** (the standard behavior). The specified property of the symbolic object is then marked as known and fixed to this new symbolic value. When a test case terminates, new tests will be created to explore the program for each supported symbolic type.

There are a number of advanced features which can change the behavior of `getProperty` and `setProperty` operations, such as `defineProperty`, which can trigger the execution of a function instead of map lookup. Methods can also be used to change the enumerability of properties within an object. We concertize the symbolic objects when

Algorithm 7: Symbolic object encoder – setProperty(base, property, value).

```
1 if property is symbolic then
2     for knownProp in base do
          // Attempt to generate a test case for each
             known property
3         if Concrete(property) = knownProp then
4             PC ← PC ∧ property = knownProp;
5         else
6             PC ← PC ∧ property ≠ knownProp;
7     return base[property] = value;
8 else
      // Record the new value for property
9     return base[property] = value;
```

handling these cases, and so our encoding is under-approximate when modeling these behaviors.

5.3.3 Mixed Type Arrays

We have described an approach to model objects, which are in essence maps between string property names and values of any type. We now show the same approach can be applied to arrays as well. Conceptually, arrays are very similar to objects, mapping integer or string property names to values. The most significant differences between arrays and objects are the custom behaviors of the length property, enumeration, and accompanying methods (e.g., `push` and `pop`). In JavaScript, it is valid to also write to non-integer properties to arrays, with the array acting as a object in these cases. For example, `let arr = [1,2,3]; arr['dst'] = '/home';` would yield `[0: 1, 1: 2, 2: 3, length: 3, dst: 'home']`.

We intercept reads and writes to array length, which is a reserved prop-

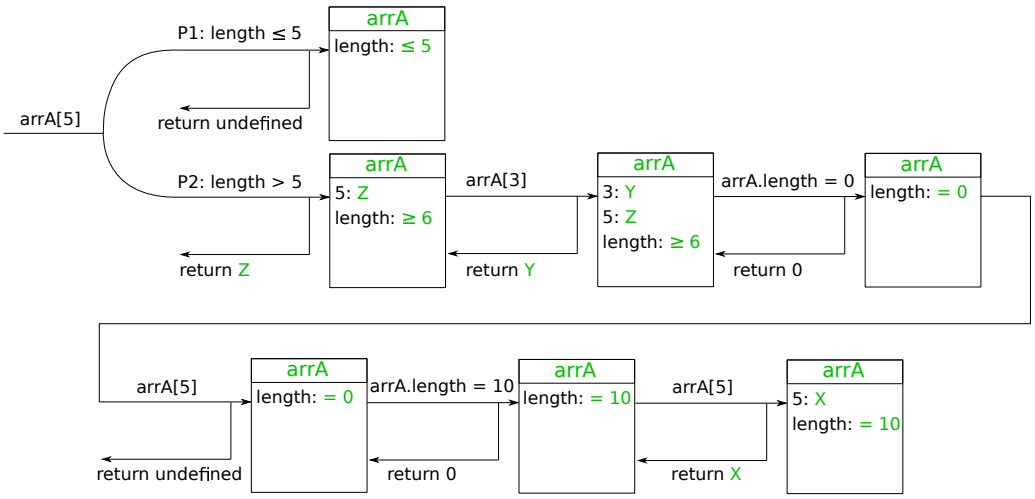


Figure 5.4: Illustration of symbolic array modeling

erty name in arrays. The array length property will always be one higher than the largest element index in the array. This point is important because arrays do not need to be contiguous (i.e., there may be gaps between two indices). This design choice has an impact on enumeration, where looping on the array length will include all indices `0 <= index < arrayLength`, but using the `of` or `in` operators will only include those which have been set, since these operators will only include properties which are marked as enumerable. For example, examine the following program:

```
1  let arr = []
2  arr[0] = 1;
3  arr[4] = 2;
```

Here, the interpreter will yield the array `[0: 1, 4: 2, length: 5]`. If we enumerate using the `of` or `in` operators we would see `1` and `2` enumerated upon, however if we enumerate and print all properties through the array length then we would see `1, undefined, undefined, 2` printed.

When a program writes to the array length property, the array will be truncated or expanded to the new length. If the value is less than the current array length, any values in the indices `newLength <= index < oldLength` will be deleted from the array. If the value is greater than the current array length, then the array will be extended with `undefined` values. To illustrate this, see the following program:

```
1 let x1 = [1,2,3]
2 let x2 = [1,2,3]
3 x1.length = 100;
4 x2.length = 0;
```

In this example the variable x1 would have a length of 100 with all values after 3 being **`undefined`**, while x2 will be empty.

We illustrate these changes in behavior through Figure 5.4. To ensure we accurately model array length, we create a separate symbolic integer to represent it. This value is initially unbounded and has constraints applied as the program executes. As we fetch property 5, we explore two paths, one where the existing length property is large enough to accommodate the new value and one where it is not. In the case where it is not the value of the property will be undefined, and in the other case it will return a new symbol using the same approach as our symbolic objects. The second step illustrates what happens when an array lookup occurs on an array that is longer than our property index. Here, a second path is infeasible because the array length cannot be less than six. Direct writes to an array fix the symbolic length; writing a length of zero to the array truncates it, removing all properties. Subsequent property lookups will all return undefined. A write of length 100 expands the array to a fixed length but does not fix any properties. Here, a property lookup creates a fresh symbol because the previous one was erased. The new symbol is given a unique name in the path condition, and can interact with the symbol that used to occupy this property.

5.3.4 Optimized Support for Homogeneous Typed Arrays

The final component of our encoding is a direct translation to SMT for homogeneously typed arrays. This encoding enables symbolic property names in homogeneously typed arrays. As motivated previously, directly encoding JavaScript arrays in SMT is too expensive for DSE, since we would need to encode potentially recursive values in SMT. Our generic array and object encoding overcome this by on-demand symbol generation, but this strategy cannot reason about property indices symbolically. For example, in the following program, we will not exercise the error:

```
1 let i = I with initial value 0;
2 let arr = A with initial value [];
3  if (arr[0] == 5 && arr[i] != 5) {
4  throw 'Error';
5 }
```

In this program, we do not exercise the error because we concretize symbolic property names. Thus, `arr[i]` will resolve to `arr[0]`, leading to an infeasible constraint of $arr[0] = 5 \wedge arr[0] \neq 5$, and will not consider any paths where i is not 0 due to concretization. If we set `i` to `1`, then this error would be found. We provide an encoding for homogeneously typed arrays directly in SMT to explore portions of a program where property name concretization is limiting analysis. Since the encoding is directly in SMT, we no longer need to concretize property names, allowing us to reason about property names symbolically.

Our encoding uses existing SMT solver support to represent arrays. A typed array has two symbolic components, the array data and array length. The array base is a symbolic mapping of integer property names to symbolic values of the array's type. The symbolic length property is used to represent the current constraints on array length, which is necessary to test out-of-bounds array element access. A symbolic `getProperty` can explore two paths, one where the array is shorter than the index resulting

Algorithm 8: Homogeneous Array – getProperty(base, index).

```
1 if 0 ≤ index < base.length then
2 |   PC ← PC ∧ 0 ≤ index < base.length;
3 |   return select(base, index);
4 else
5 |   PC ← PC ∧ (index < 0 ∨ index > base.length);
6 |   return undefined;
```

Algorithm 9: Homogeneous Array – setProperty(base, index, value).

```
1 if index > 0 then
2 |   base.length = index + 1 if index ≥ base.length otherwise
  |     base.length;
3 |   base = store(base, index, value);
4 return value
```

in `undefined`, the second where the array includes the index, resulting in a value of the array type. `setProperty` operations update the symbolic length to accommodate the new value and then inserts it into the base. This is illustrated in Algorithm 8 and Algorithm 9. In these algorithms, the methods `select` and `store` map directly to SMT. We downgrade when a `setProperty` is given a value that is not the array base type.

The process for downgrading a homogeneously typed array to a mixed-type array is detailed in Algorithm 10. Array downgrading converts a homogeneously typed array into a generic array to allow mixed types. We do this by using the concrete array length to derive the initial mapping for the mixed-type array. We copy the homogeneously typed array's length into the new array so that we respect existing length constraints.

Algorithm 10: Homogeneous Array – downgrade(array)

```
1 knownValues = [];
2 for i in Concrete(base.length) do
3 |   knownValues[i] = select(base, i);
4 return GenericArray(knownValues, base.length)
```

5.4 Evaluation

We now set out to show the effectiveness of our approach on a subset of JavaScript built-in functions introduced with the ES6 specification. Here, we set out to answer the following research questions:

RQ1: Can a standard DSE engine be modified to usefully analyze standard library method implementations?

RQ2: Can our approach find any bugs in built-in methods?

RQ3: Does the addition of our test cases improve coverage of Test262?

We answer these research questions through three experiments on selected functions introduced with the ES6 specification. In the first experiment we evaluate the effectiveness of our conformance test case generation strategy using two polyfill packages. Here, we show that ExpoSE achieves high coverage of many method implementations. For our second experiment we use our generated conformance test suite and voting mechanism to search for errors in existing implementations of the ES6 standard, finding 17 bugs in a widely depended on built-in implementation. Finally, we evaluate the coverage of our test suite against Test262 under the quickjs interpreter. In this study we see that, while Test262 generally covers more branches of tested methods overall, our test cases explore parts of the built-in implementations which are not covered by Test262.

5.4.1 Test Case Generation

In our first experiment we answer RQ1 through an evaluation on two popular ES6 built-in method implementations found on NPM. We extracted our surrogate implementations from `core-js` and `mdn-polyfills` packages. Overall, we collected 96,470 new unique test cases. We show that we achieve high coverage of the built-in implementations during symbolic execution, suggesting that a large portion of the implementation is covered.

Methodology

Our test harness loads the portions of the library we wish to test and selects a target method. The method is then executed with symbolic arguments for both the `this` argument and each of the method arguments. We analyze this harness with ExpoSE. Each method is tested in isolation, through a single analysis using ExpoSE with a timeout of one hour on a 64 core machine. After analysis, the generated test cases are combined and duplicates are removed.

Results

We generated 129,960 new test cases overall, which was reduced to 96,470 after removal of duplicate tests. Table 5.1 presents the results of our evaluation, providing coverage information from the analysis of the corejs and `mdn-polyfills` variant if the method was supported by that library. Overall, we found that our prototype is more capable of generating test cases for string methods than array methods. These results are inline with our expectations, as the string support in ExpoSE is mature. Further improvements in ExpoSE modeling and SMT solvers could improve this support even further. In particular, our encoding currently does not include

Function	Test Cases	Coverage	
		corejs	mdn-polyfills
Array.from	13122	90%	84%
Array.of	162	84%	82%
Array.fill	2645	89%	85%
Array.filter	81	88%	N/A
Array.findIndex	162	72%	51%
Array.forEach	81	78%	N/A
Array.reduce	729	76%	N/A
Array.some	162	84%	35%
String.endsWith	64179	86%	82%
String.includes	15957	93%	91%
String.padStart	13220	94%	94%
String.padEnd	13220	94%	94%
String.repeat	2066	96%	83%
String.startsWith	4215	91%	88%
String.trim	2025	95%	83%

Table 5.1: Automatically generated test cases by built-in method.

symbolic models for array methods other than `push`, `pop`, `includes`, `indexOf`, which may lower overall performance.

5.4.2 Executing Our Test Cases

We have now generated a suite of test cases for our selected methods and are ready to test built-ins. In this section we set out to answer RQ2 by executing our tests on five JavaScript built-in implementations. We execute each of our generated test cases on three interpreters and two polyfill implementations. To analyze the output of these test cases, we construct the voting mechanism outlined in Section 5.2.4 from our selected interpreters. Each test case is executed once per interpreter, and after they finish they vote on the correct output. We found 17 unique bugs in `mdn-polyfills`,

Implementation	Unique Exceptions	Test Case Failure	Bugs
mdn-polyfills [64]	34	200	17
corejs [102]	63	125	0
spidermonkey [89]	72	66	0
Node.js [32]	56	122	0
quickjs [13]	24	141	0

Table 5.2: Test case summaries for 5 built-in implementations.

showing that our approach is effective in generating useful test cases for conformance testing. We did not find bugs in any other implementations but this was expected as the methods tested are from a mature standard. We found zero cases which required manual intervention during voting.

Methodology

We selected quickjs, spidermonkey (through the standalone interpreter), Node.js, corejs and `mdn-polyfills` for testing. We tested each of the test-cases identified in Section 5.4.1. We executed each test case once with each competing implementation and stored the output. Next, we examined the result of each test case for divergence between the tested implementations. If there is any divergence then we used the outlined voting mechanism to resolve the failing case. Test cases were each executed with a maximum time of 10 minutes on each interpreter, though no test cases hit this boundary. Tests which crashed or exceeded the timeout are terminated with a failure.

Results

Table 5.2 presents a summary of test case executions for the 5 built-in implementations. `Unique Exceptions` gives the number of unique excep-

tions identified across the executions of all test cases (i.e, where an exception text has not been seen before after test specific details are removed). `Test Case Failure` details the total number of test cases where the interpreter failed to give a result due to crash or timeout. The final column, `Bugs` gives the number of bugs found in each implementation.

Our test case executor found 17 bugs automatically, all within the implementation `mdn-polyfills`. These bugs were confirmed with manual analysis. For example, in one test case we observed that `String.prototype.includes.apply([0,0], [[]])` should yield **`true`**, but in mdn-polyfills the built-in returns **`false`**. We found this divergence occurs because the implementation does not coerce the `[0, 0]` to a string. The identified bugs show that a consensus based test executor can be used to verify the correct behavior of built-in JavaScript methods. Manual analysis found that the bugs identified were all triggered by unconsidered type coercions in string and array methods. In some cases, this led to the method producing output when it should have thrown an error. In others, the method produced an incorrect output, such as `Array.includes`, which would return **`true`** when it should have returned **`false`** on some inputs.

In addition to finding some bugs, we exercised many unique exceptions in interpreters. The high number of unique exceptions suggests that our test suite is exploring many interesting corner cases of implementation. Interestingly, we do not see the same number of unique exceptions across interpreters. We found that some implementations have much more verbose error messages for built-ins than others. While the exception messages are not standardized, and so this is not an implementation error, the lack of verbosity could make errors harder to debug.

We experienced some test case failure for each of the implementations tested. We observed zero cases of failure due to test timeouts or interpreter error; instead, all observed failures were due to interpreter memory limits. Most of these errors occur in `String.repeat`, where many of the

test inputs are large values which hit interpreter memory limits. We examined our surrogates to understand why the DSE engine is generating such extreme cases. We find that in one of our surrogate implementations there is an upper limit on string size through a boundary condition `if (str.length * count >= 1 << 28)`. The condition drives ExpoSE to generate a series of test cases supplying large arrays or strings as input. The specification does not specify interpreter memory limits, so the different number of failing cases is not an error. In particular, we observed that spidermonkey avoids test case failure in these cases by having stricter limits on bounds for `repeat`. As an example, at the time of writing, Node.js will execute `'h'.repeat(1 << 28)` but spidermonkey will not. In the specification, ECMAScript does not add any constraints to the range of strings, so long as they are positive integers. In practice, the reason we see these memory errors in quickjs and Node, but not spidermonkey, is because string boundaries are explicit in spidermonkey method implementations. So these errors manifest as exceptions without crashing the interpreter.

Our study has shown that we can detect faults in a real built-in implementation with 35,000 weekly downloads at time of writing. The ability to detect real bugs using our approach shows that a consensus based approach for test case evaluation can be effective. In addition, our approach generated a large number of unique exceptions in the tested cases and covered an obscure difference in string length constraints between interpreters, demonstrating that our test cases explore interesting paths through the implementations.

5.4.3 Test Suite Coverage

To ensure that our new approach generates novel test cases, we now compare the branch coverage of the new test cases to Test262. We show that the addition of our test cases leads to an increases in overall branch cov-

erage in quickjs, demonstrating that our approach is generating novel test cases.

To test our approach we built a version of the quickjs interpreter with support for gcov so we could collect internal code coverage metrics. quickjs is a complete ES6 implementation of JavaScript [13]. We selected this interpreter because it executes the code in a purely interpreted manner, without JIT or other runtime optimization, and it has built-ins implemented directly in its source code. This is important as many prominent engines, including Node.js [32] and spidermonkey [89], do not implement language built-ins directly in source code. Instead, these engines implement a small subset of JavaScript in their native language and then implement the remaining built-ins in JavaScript. Implementing built-ins in JavaScript allows these engines to take advantage of JIT optimizations and reduce engine development time, but, this makes it challenging to collect coverage metrics as built-in functions do not have a clear instrumentation point.

In our study, we found that our automatically generated conformance test suite improves branch coverage by up to 15%. We see coverage improvements in almost every tested function, demonstrating that the approach is versatile. Our results show that we can use automatically generated test cases to supplement the Test262 suite to provide greater over-all coverage of JavaScript interpreters.

Methodology

We modified the quickjs build process to include support for branch coverage output via gcov, a tool which collects coverage information through compile time instrumentation. For each built-in method, we then executed all of our generated test cases and the relevant portion of the Test262 suite. Once each had finished, we extracted the covered branches, using a

manual analysis to identify the appropriate function names in the quickjs source code.

When evaluating the coverage of a function within a program, we present both shallow and deep metrics for combined coverage increases, and follow calls to a depth of 3 when presenting absolute branch coverage. If we only present shallow coverage metrics (i.e., we do not follow function calls), then we may under-represent coverage improvements as logic for built-ins is often spread across many methods. Conversely, including all reachable functions may make our results less insightful by including large amounts of indirectly related code, such as utility methods, which may also be called by other methods during execution. By presenting our combined coverage improvements at different call depths, the reader can see how branch coverage changes as we follow an implementation deeper into the methods it calls.

In our coverage metrics we only include methods defined in the core quickjs implementation and do not include library calls.

Results

Table 5.3 details total number of branches, branches covered by our systematically generated conformance tests (ExpoSE), and branches covered by Test262. We selected a call depth of 3 as following calls further included many utility methods, making results less insightful. Here, tests generated by our approach achieve reasonable branch coverage, but do not exceed the coverage of Test262 which is already very high for every method. When we combine the branches covered by automatically generated conformance tests and Test262 we see an overall coverage improvement over Test262 for every tested function, demonstrating that generated conformance tests are exploring new routes through the implementation.

Table 5.4 shows the results of our coverage study at various call

depths. The function names in the table are the internal function names in quickjs. quickjs sometimes implements optimized methods for typed arrays, which is why there may be two methods for the same feature. We see branch coverage improvements in many of the methods we test, in some cases seeing a 15% improvement overall. Our results demonstrate that our automatically generated test cases do explore further into built-in method behavior than Test262 answering RQ3. In most functions, we see notable coverage increases, even at a call depth of 0 (i.e., not including the coverage impact of any called methods). These results highlight that our approach is exploring untraveled paths through built-in function implementations login, and not just expanding coverage in utility methods. The coverage increases at low call depths show that built-in specific edge cases are being exercised, as these expressed near the surface of the call-tree. A full set of results including absolute branch numbers is available publicly online [1]

Our study of interpreter coverage between automatically generated conformance tests and Test262 shows that supplementing Test262 with automatically generated test cases will improve the test suite. We found that our approach would improve branch coverage of the test suite by up to 15% in a complete ES6 JavaScript engine. These improvements demonstrate that our method can improve conformance testing for JavaScript interpreters using only automatically generated test cases. Such coverage improvements in the testing suite raise the likelihood that implementation errors will be detected before they cause problems in the wild.

[1] https://anonymous.4open.science/r/fef322ae-fbad-4037-9b1a-c535ffacb8be/

Function	Total Branches	ExpoSE	%	Test262	%	Combined	%
array_from	1075	640	60%	802	75%	957	89%
typed_array_from	897	572	64%	696	78%	829	92%
array_of	875	509	58%	640	73%	754	86%
typed_array_of	512	285	56%	405	79%	457	89%
array_fill	740	436	59%	586	79%	663	90%
typed_array_fill	222	165	74%	180	81%	206	93%
array_every	953	564	59%	748	78%	873	92%
array_find	527	294	56%	422	80%	471	89%
typed_array_find	1472	869	59%	1103	75%	1302	88%
array_reduce	702	404	58%	556	79%	636	91%
array_includes	734	430	59%	589	80%	667	91%
string_includes	220	148	67%	183	83%	194	88%
string_pad	139	85	61%	112	81%	121	87%
string_trim	54	34	63%	46	85%	47	87%
string_repeat	139	89	64%	112	81%	116	83%

Table 5.3: Branch coverage for systematically generated conformance tests and Test262 at a call depth of 3.

Function	+Branches% (Depth)		
	0	3	5
array_from	+4.76%	+14.42%	+14.68%
typed_array_from	+6.14%	+14.82%	+13.48%
array_of	+0%	+13.03%	+12.83%
typed_array_of	+41.67%	+10.16%	+14.19%
array_fill	+0%	+10.41%	+10.57%
typed_array_fill	+8.33%	+11.70%	+14.97%
array_every	+8.33%	+13.12%	+12.64%
array_find	+8.51%	+9.3%	+11.41%
typed_array_find	+11.43%	+13.52%	+15.32%
array_reduce	+3.13%	+11.4%	+12.36%
array_includes	+1.67%	+10.63%	+11.76%
string_includes	+0%	+5%	+9.69%
string_pad	+0%	+6.47%	+10.68%
string_trim	+0%	+1.85%	+8.37%
string_repeat	+0%	+2.88%	+10.52%

Table 5.4: Coverage improvements of automatically generated tests at various call depths by built-in method implementation.

5.5 Related Work

Dhok, Ramanathan, and Sinha [36] develop extensions to Jalangi [115] which reduce the frequency of redundant test cases when exploring JavaScript programs with unfixed types. Here, the authors use a type aware encoding strategy to reduce the frequency of redundant test cases by collecting type expectations during test case execution. This encoding strategy can be combined with our encoding scheme to improve the performance of symbolic execution further. In particular, this would allow for better automated selection of our typed array support, where we can know that an array will not be subsequently downgraded.

Mayhem [24] is a dynamic symbolic execution engine for compiled programs that represents a 32-bit address space symbolically to model program memory. In this work a symbolic memory model improved the effectiveness of DSE by 40%, showing that supporting symbolic memory is crucial. To make their solution feasible, they limit the symbolic representation to reads and do not consider writes symbolically. EXE [22] supports a single-object model, where pointers are concretized and only a single address is considered. The approaches are similar to our own when treating symbolic field names, where ExpoSE concretizes the field name to avoid exploring an unbounded number of inputs.

S2E [30] models system memory symbolically in a symbolic machine emulation, achieved through instrumentation of memory reads and writes. Modeling memory interactions in low level applications is very different from JavaScript, since memory is fixed type and the DSE engine does not need specific encodings for language structures.

The DSE engine KLEE [21] supports multiple memory models, including a forking approach where one path is created to explore each symbolic memory region, and a flat approach which reasons about memory as a single continuous block. Recent approaches split memory regions into segments to allow more efficient analysis [66]. These approaches are highly tailored to reasoning about systems memory with C style pointers and are not directly applicable to JavaScript object modelling.

There has been work to enable automated testing for Java [56, 106, 9]. Symbolic encodings for Java classes are insufficient for JavaScript as they rely upon a known structures and typing [68, 132, 135]. Our approach is similar to previous symbolic representations of maps, but does not require fixed type fields.

Kristensen and Møller [70] use TypeScript type specifications and feedback directed random fuzzing to identify mismatches between type specifications and observed behaviours. Through this approach the authors

identify many inconsistencies, motivating the use of dynamic analysis for specification testing.

Marinescu and Cadar [84] symbolically execute test suites to find bugs. A symbolic execution runs on the existing harnesses used by a program for unit testing, replacing concrete values with symbolic ones in order take advantage of interesting test conditions. Unlike our approach, only simple error conditions are considered because the tool cannot deduce the expected output after a charge in input.

Palikareva, Kuchta, and Cadar [98] use DSE to automatically discover differences in behaviour between program versions. The authors test versions of the same software, while our approach tests differences between many implementations of the same specification. As versions of the same software are tested but program specifications are not static, it is difficult to decide whether changes in behaviour between two versions are desired. This differs from our approach, where the behaviour of compliant implementations is fixed and divergence is an error.

Bae, Park, and Ryu [11] use type expectations provided by analysis tools to prioritise likely useful types when selecting new test cases. This approach can be applied to our exploration strategy, but requires a pre-analysis of the program and is unlikely to provide notable benefit when dealing with complex objects and arrays, areas where JavaScript type analysis tools are unreliable.

Selakovic et al. [113] develop `LambdaTester`, a testing tool for JavaScript methods which accept functions as inputs. This tool explores these methods by inserting generated callbacks as function arguments during dynamic testing. A similar extension to ExpoSE may improve the performance of conformance test generation, since some standard library methods are high-order, but the lambda generation approaches presented in the paper would also require significantly more test cases, increasing overall test case generation time.

Accelerating the Web with Symbolic Execution

6

In this chapter, we use ExpoSE to develop a new web accelerator, Oblique, which uses a symbolic offline analysis of a webpage to facilitate pre-fetching of page dependencies, even if they have nondeterministic URLs. Our new proxy, Oblique, uses dependencies computed by ExpoSE alongside a lightweight solver and the concrete request headers to quickly decide ahead of time what resources that need to be pre-fetched for a client.

Prior Appearances This work will appear in NSDI 2021. It is a joint work with Ronny Ko and James Mickens at Havard University and Ravi Netravali at UCLA. My primary contributions to this work are the extension of ExpoSE to support symbolic execution of web applications, and developing the apparatus required to find dependencies.

6.1 Introduction

Speculative loading (SL) describes a system for web acceleration by triggering to the browser to load a resource before it is needed (pre-fetching). By sending webpages before they are requested, a speculative loading approach shifts the bottleneck to the device's CPU, removing the impact of network latency. Existing solutions include resource dependency resolution (RDR), which tasks resource discovery to a proxy with a high-speed internet connection [90, 78, 91, 118, 133]. RDR proxies use an online approach to resource discovery; a browser instance which executes the webpage inside the proxy. The proxy tracks network requests made by the browser and begins transmitting these back to the client, even though they have not yet been requested. A component in the client caches these incoming dependencies until the browser requests them. Through this scheme, all of a devices bandwidth is used, reducing overall page loading time. Such proxies are expensive to run, since they require a full browser

instance.

Other approaches, such as Vroom [107], perform offline analysis to speculatively load webpages. Here, a browser instance is used to identify dependencies offline and cached so that they are available for pre-fetching on the next request. But approaches like Vroom are not able to explore JavaScript control-flow to find all dynamically driven dependencies. Nor are they able to pre-fetch non-deterministic URLs, leading to missed opportunities for pre-fetching and increase page-load times. The lack of non-deterministic URL support in prior work can also increase the number of incorrectly pre-fetched URLs, which consume network resources.

Existing speculative loading approaches pre-fetch based on observed network requests during dynamic analysis, or static analysis of the webpage source code. These analyses miss dependencies since they are unable to explore network requests driven by JavaScript. To improve this we use ExpoSE to explore the JavaScript on a webpage. We explore all control flows necessary for pre-fetching by replacing JavaScript values dependent on request headers with symbols. Since ExpoSE already builds a symbolic path condition, we can use it as the trigger condition for a dependency. The work has one notable advantage over previous work; it can support non-deterministic dependency URLs. For example, our approach can prefetch the URL loaded by the following JavaScript: `load(window.userAgent + 'hello.js')`. The support for non-deterministic URLs makes the dependencies detected by DSE more reliable than other SL approaches, reducing the number of incorrectly pre-fetched pages in non-deterministic dependencies. Additionally, only one analysis is necessary to pre-fetch for all variations of request headers.

In this chapter we introduce Oblique, a constraint-based SL proxy for web acceleration (C5). First, we use ExpoSE to execute web pages symbolically, with the request headers made symbolic, and all other inputs to the webpage are fixed. We use the results of the analysis to build a

constraint-based resource dependency table. When a browser requests a webpage the request headers and page dependency table can be used to decide which resources will be requested. In this chapter we present the following contributions:

- Oblique, our new web acceleration system.
- To enable symbolic execution of webpages we extend ExpoSE with support for symbolic execution of webpages based on symbolic request headers.

6.2 Proxies and Speculative Loading

A web proxy is a program which makes HTTP requests on a clients behalf. A proxy tool waits for HTTP requests to be sent by a client (usually, a web browser), and then returns a response by itself requesting the original target. The proxy has visibility on the request and any subsequent response, enabling analysis and caching. Proxy tools have a variety of use cases across the web, including load balancing, caching, and security.

Speculative loading approaches reduce page-load times by routing traffic through proxies which can begin pushing the files a webpage is expected to load before the client browser has made the request. These approaches reduce the network bottleneck in page load times. Speculative loading can decide what resources will be fetched through online or offline analysis. Resource dependency resolution is an online approach which loads the webpage in a browser running on the proxy to pre-fetch files [78, 90]. Vroom [107] is a hybrid approach which uses a cache of expected dependencies, but combines the result with statically identifiable dependencies in pages as they pass through the proxy.

Resource dependency resolution is an established approach for reducing browser loading times. In these systems, a RDR proxy leverages its

own internet connection to begin transmitting the dependencies of a webpage before the client has requested them. Resources are pre-transmitted by using a pool of real browser instances inside the proxy to directly load the website and execute any JavaScript to find dependencies. The approach is expensive for the proxy provider, as typically browsers consume large amounts of system resources when executing webpages. Proxies usually serve many clients concurrently, so in high-load situations, this extra overhead may end up slowing page-load times as the proxy struggles to keep up. RDR proxies are vulnerable to denial-of-service attacks. By requesting several instances of a page with bad JavaScript, such as an infinite loop, a malicious user can consume all available resources on the system.

Ruamviboonsuk et al. [107] use a hybrid offline and online analysis to reduce the cost of speculative loading. Vroom uses online analysis to prefetch all directly observed dependencies of a webpage using a lightweight static analysis. It also performs an offline analysis using RDR to prefetch dependencies that can not be identified statically, such as those driven by JavaScript, or triggered by later dependencies. This approach is effective in practice, reducing the cost of RDR by removing the need for a browser instance per served client. The drawback is the lack of complete coverage of the application; Vroom will work best if the client is using the same, or suitably similar request headers, and will perform poorly when the client is using a different browser to the offline analyser or headers are specific to the client (e.g., the cookie flag). Vroom also cannot support non-deterministic URLs, which can cause missed pre-fetches and lead to incorrectly pre-fetched resources.

6.3 Overview

Figure 6.1 describes the architecture of Oblique. Whenever a request is processed, a dependency database will be used to check dependencies for this page have been precomputed. If we do not have dependencies, then we act like a normal proxy and do not pre-fetch. Once the same page has been seen a sufficient number of times by several different users, ExpoSE is scheduled to perform a symbolic analysis of the page in the background. Once ExpoSE has analyzed the webpage discovered dependencies are added to the dependency database with a time-to-live. While a page is in the dependency database, requests for the website can be prefetched. On request, a client's request headers are supplied as arguments for the dependency table trigger conditions which are then used to decide what dependencies will be requested. Here, non-deterministic dependency URLs are turned into concrete URLs by substituting in the request headers. Once the time-to-live expires, we remove the dependencies; this step is necessary to mitigate the impact of webpages changing after deployment. Note that, since pages are only cached after they have been requested multiple times by different users, we avoid the problem of ExpoSE symbolically executing many client-specific URLs.

6.4 Analysis of Client-Side JavaScript

We seek to analyze client-side JavaScript and identify resource dependencies triggered by changes in request headers. We use ExpoSE to identify these resources and the corresponding dependency trigger conditions. We extend the ExpoSE analysis framework, adding support for web applications and resource dependency detection. To begin, we provide a description of our modifications to ExpoSE for web applications (Section 6.4.1).

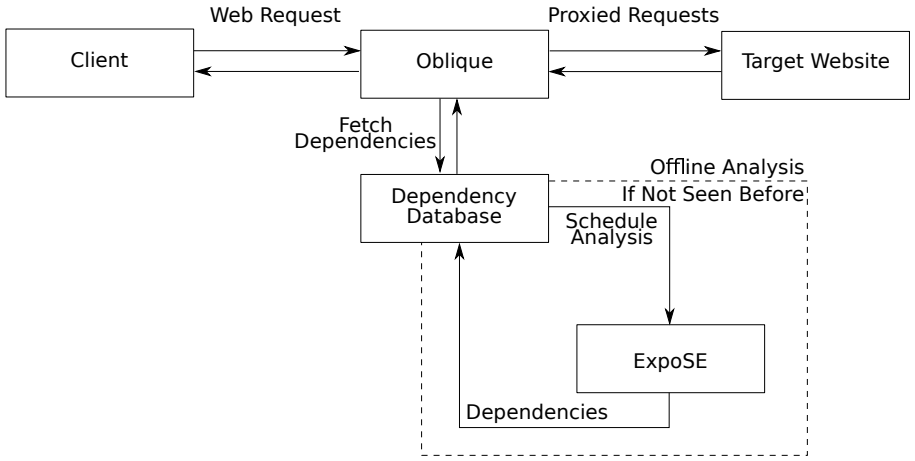


Figure 6.1: An overview of the approach architecture.

Next, we present a scheme to represent resource dependencies and the corresponding trigger conditions (Section 6.4.2). Finally, we illustrate how this approach works by example (Section 6.4.3).

6.4.1 Symbolic Execution of Web Applications

No publicly released DSE engines for JavaScript supported web applications. This led us to modify our JavaScript DSE framework to add support for symbolic execution in the browser. In order to add support for web-based JavaScript to ExpoSE we implement a custom browser based on Electron [40]. This browser has two responsibilities. First, it facilitates the instrumentation of incoming JavaScript, rewriting it with the existing instrumentation infrastructure. Second, it exposes the Z3 solver and native libraries to the instrumented JavaScript. This is required because web browsers are sandboxed, and interaction with a native library is prohibited. In order to detect resource dependencies we developed a novel test

harness for web applications. This test harness automatically generates test cases for alternate control flows through the program by replacing the HTTP request headers with symbolic values. The custom browser layer uses these symbols in the network requests it makes in order to avoid cases where servers deploy different code based on request headers.

Figure 6.2 gives a high level overview of our revised architecture for ExpoSE. A set of request headers are sent to a test-case executor. The executor propagates these request headers to a browser instance, which loads the given webpage and instruments all source code. The instrumented program is then executed by the browser's interpreter. After the page finishes loading, the program trace is then sent to the executor which symbolically encodes the trace. Next, an SMT solver is used to generate alternate assignments for the request headers.

This approach allows us to instrument and trace executing JavaScript inside the browser using the same symbolic interpreter that we use for our Node.js analysis. The hosted browser uses its own interpreter to perform concrete execution, but defers symbolic execution to our existing `Executor`. Through this architecture we can reuse the components of ExpoSE that make it practical, such as the distributed architecture, string and regular expression models, and faithful handling of asynchronous events.

6.4.2 Detecting Resource Dependencies

In order for our proxy to function, the triggering condition and symbolic URLs must be stored. We split our dependency detector into two components, a static HTML analyzer, and dynamic dependency detector. The purpose of the first component, the static resource dependency detector, is to identify all resource dependencies which are not driven by JavaScript, such as images embedded directly into the webpage source. In our framework, we treat these as constant dependencies, assuming that a client will

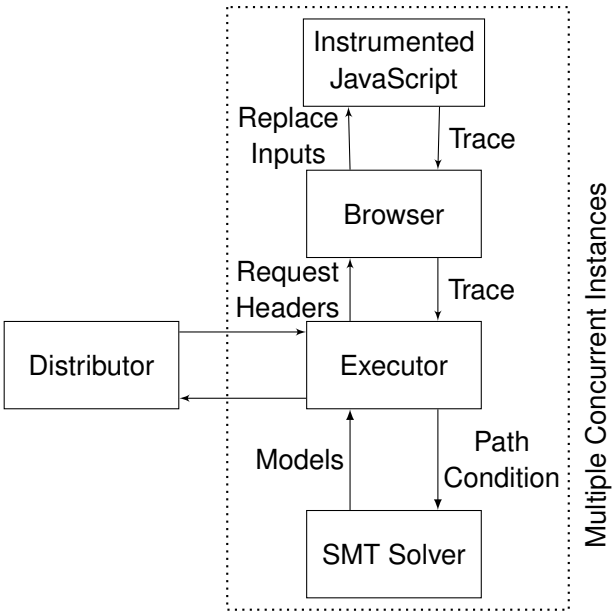


Figure 6.2: Our new ExpoSE architecture.

always request them when loading a given page. The second component uses ExpoSE to find the dynamically driven requests within a webpage. Here, the trigger condition and resulting URLs can contain symbolic components which depend on the real client's request headers.

ExpoSE builds the path condition, a symbolic representation of the current program trace, during execution. We use the path condition and request URL as our trigger conditions and symbolic URL. We configured ExpoSE to log the path condition and resource URL upon resource load. After all test cases have finished executing these outputs are collected and a resource dependency table is developed. The server front end then uses the stored path conditions as trigger conditions, substituting in a client's request headers to decide what resource dependencies will be requested following an initial page load. We do this by substituting the symbols for each request header value in the SMT path constraint with the real clients request headers. If the SMT problem is satisfiable then the corresponding URL is marked as dependency. URLs in the dependency table may contain symbolic portions, such as `'http://example.com/?' + userAgent`. These symbolic portions are made concrete during this step using the request headers from the real HTTP request.

6.4.3 Example: Find Resource Dependencies with ExpoSE

To illustrate how DSE finds resource dependencies in a JavaScript program we will now look at the following example.

```
1 navigator.userAgent = UA;
2 document.cookie = COOKIE;
3 if (/Chromium/.exec(navigator.userAgent)) {
4  loadScript("chrome.js");
5 }
6 if (/ads=on/.exec(document.cookie)) {
7  loadScript(navigator.userAgent + "ads.js");
8 }
```

The program loads the `chrome.js` JavaScript program only if the word `Chromium` appears in the user agent, and then loads `userAgent + ads.js` if the string `ads=on` appears in the request cookies. To analyze the program we first set the `userAgent` and `cookie` to be symbolic, given the symbolic names `UA` and `COOKIE`.

Test Case 1 We then begin our concolic execution with the seed test case, inputs `userAgent = ""`, and `cookie = ""`. Executing line 1 we take the else condition, as `/Chromium/.exec("") == null`. This adds the logical constraint $UA \notin /Chromium/$ to the path condition. Next, on line 4, we also find that `/ads/.exec("") == null` and add $COOKIE \notin /ads = on/$ to the path condition. With our first test execution complete, we now turn to the SMT solver to find alternative test cases. First, we query whether $UA \in /Chromium/$ is feasible and from this generate the new the test case input `userAgent = "Chromium", cookie = ""`. Next, we query whether $UA \notin /Chromium/ \wedge COOKIE \in /ads = on/$ is feasible and generate the new input `userAgent = "", cookie="ads=on"`.

Test Case 2 Executing our second test, `userAgent = "Chromium", cookie = ""`. On line 1, we take the true branch of the if condition, and load the script `chrome.js`. On line 4 we take the else condition, since our cookie is empty. After execution terminates, we query whether $UA \in /Chromium/ \wedge COOKIE \in /ads = on/$ is feasible, and generate the new test case `userAgent = "Chromium", cookie = "ads=on"`

Test Case 3 After that we execute the test `userAgent = "", cookie="ads=on"`. Here, we take the else branch on line 1, and the true branch on line 4. This causes us to load a script with the symbolic URL

Table 6.1: A resource dependency table after symbolic execution.

Trigger Condition	Symbolic URL
$UA \in /Chromium/ \wedge COOKIE \notin /ads = on/$	`chrome.js`
$UA \notin /Chromium/ \wedge COOKIE \in /ads = on/$	`userAgent + ads.js`
$UA \in /Chromium/ \wedge COOKIE \in /ads = on/$	`chrome.js`, `userAgent + ads.js`

`userAgent + ads.js`. Upon termination, the DSE engine decides that there are no feasible alternate test cases for this path.

Test Case 4 We now execute our final test case, `userAgent = "Chromium", cookie = "ads=on"`, which was generated by test case 2. During execution of this test, we take the true branch on lines 1, and 4. As such, the browser during this test loads both loads both `chrome.js` and the symbolic URL `userAgent + ads.js`.

After Execution Our symbolic execution has now exhausted all test cases, and so terminates. We now collect all of the loaded URLs across paths and construct a dependency table for this website. We do this by scanning the output for URL load messages and taking the symbolic path condition (PC) at the point where URL was loaded to be the trigger condition. This leads to the resource dependencies in Table 6.1. Our proxy can use this table to decide which JavaScript driven resource dependencies will be requested for the given page.

6.5 Proxy Implementation

To test this scheme in practice we developed Oblique, a resource dependency resolution proxy that can take a dependency table, developed by ExpoSE, and use it to send webpage dependencies to a client before they are requested. Oblique has two components, a local cache, and a SL proxy.

Oblique needs to be able to send dependencies to the client device and hold onto them before they are requested. We achieve this by the use of a local cache. The local cache is responsible for caching incoming files until they are requested. Modern browsers have these caches built in, and the HTTP/2 push standard allows for a web server to reuse an existing connection to push expected dependencies to a browser cache [61]. In other deployments, a custom proxy running on the local machine can be used to facilitate pre-fetching and caching.

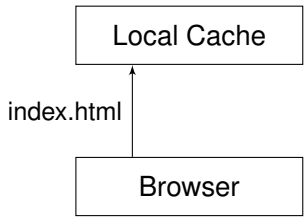
When the proxy receives a request, it examines the dependency table for any resource dependencies for the given URL. Each resource dependency stored comes with two elements, `Trigger Condition` and `Symbolic URL`. The trigger condition is an SMT program which, if satisfiable, means that the resource will be loaded. The Symbolic URL is an SMT expression that can be resolved to the concrete URL. To resolve a resource dependency, we use a custom solver capable of quickly solving SMT programs with concrete inputs (i.e., we use the syntax of SMT solvers but expect only concrete problems). We provide this solver with the concrete request headers of the web request, making the `Trigger Condition` and `Symbolic URL` wholly concrete problems, since every symbol now has a concrete value. We first test if the `Trigger Condition` is satisfied. If it is, then we say that the result of evaluating the `Symbolic URL` is a dependency of the webpage. Any identified dependencies are then pushed to the client cache.

As a webpage executes, network requests will trigger. Requests will first

check the cache, to see if the request has been pre-fetched. If it has, then the pre-fetched resource is selected and can be used immediately. If the request is not pre-fetched, then a network request is made for the resource.

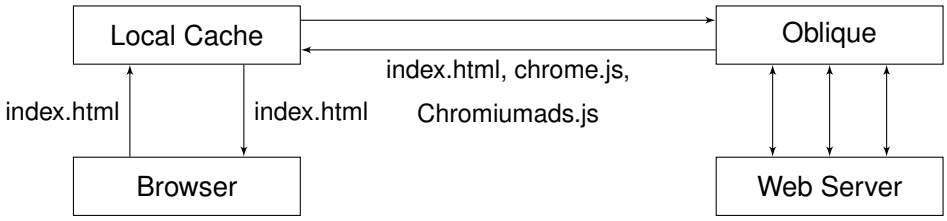
6.6 Example: Oblique in Action

We now illustrate this how Oblique works in action by executing it on the JavaScript analyzed in our previous example (Section 6.4.3). We examine Oblique when resolving a request to this website by a browser with the user agent `Chromium` and the cookie value `ads=on`, which according to our previous analysis has the dependencies `chrome.js` and `userAgent + ads.js`. To begin the browser makes a request for `index.html` on the target web server via the local cache:

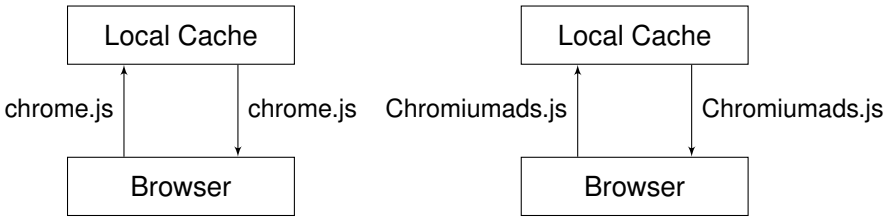


Next, our local cache initiates a connection with Oblique and requests the file `index.html` from the specified web server. We now need to decide which resources to pre-fetch. Upon receiving this request our proxy checks it's database and finds the existing resource dependency conditions for the website. The request user agent and cookie values are used to test whether any of the resource dependency conditions are met, and in this case, we identify that `chrome.js` and `userAgent + ads.js` will be requested later. Since one of our symbols is non-deterministic, we now substitute in the userAgent for that client, resulting in the URL `Chromiumads.js`. As such, Oblique identifies `chrome.js` and `Chromiumads.js` as resource dependencies.

We have now found the two resource dependencies for this website and concretized them, so Oblique begins the process of pre-fetching. Oblique now requests `index.html`, `chrome.js`, and `Chromiumads.js` from the target webserver and begins transmitting them both back to the clients cache, even though the client has not yet requested `chrome.js` or `Chromiumads.js`. Upon receipt, the local cache returns `index.html` but holds on to `chrome.js` and `Chromiumads.js`:



Once `index.html` is returned to the browser it is processed. As the JavaScript on the webpage executes, the browser then requests `chrome.js` and `Chromiumads.js` from the local cache. Since we pre-fetched this files they are returned immediately:



6.7 Evaluation

To evaluate Oblique and the web browsing component of ExpoSE, we compare the effectiveness of our pre-fetching approach to Vroom and

RDR, state of the art speculative loading approaches in offline and online analysis. We selected 200 websites from the Majestic Million website rankings so that we can compare the performance across an array of real-world webpages and JavaScript [82]. For our evaluation, we pre-analyzed the websites with ExpoSE and Vroom so that we can evaluate their performance directly. We focused our analysis on evaluating the page load performance of Oblique, as well as the effectiveness of ExpoSE in detecting dynamic URLs across domains. In particular, we set out to answer the following research questions:

(**RQ1**) What impact does a DSE driven speculative loading approach have on page load times when compared to a standard browser and existing web acceleration schemes?

(**RQ2**) How successful is ExpoSE at identifying dynamic resource dependencies?

6.7.1 Methodology

For all experiments throughout this section, we used Galaxy S10e (Android 9), running Chromium 78 on Linux on Dex. We selected top 200 landing webpages in the Majestic Million whose average round trip time (RTT) between our phone and the real web server is not greater than that between our phone and our proxy in the cloud (47 ms). ExpoSE was executed on each website for up to 30 minutes, with a maximum individual test time of 10 minutes. We implemented evaluation scripts that measure various performance metrics (the PLT, speed index, cached URL hit rates, unused pre-fetched URLs) based on the Browsertime library. For the web proxy, we used a Digital Ocean virtual machine with 8 virtual CPU cores, 16GB RAM, 640GB SSD, and 2GBits/sec network bandwidth. We implemented the RDR proxy in C and used headless chromium. We imple-

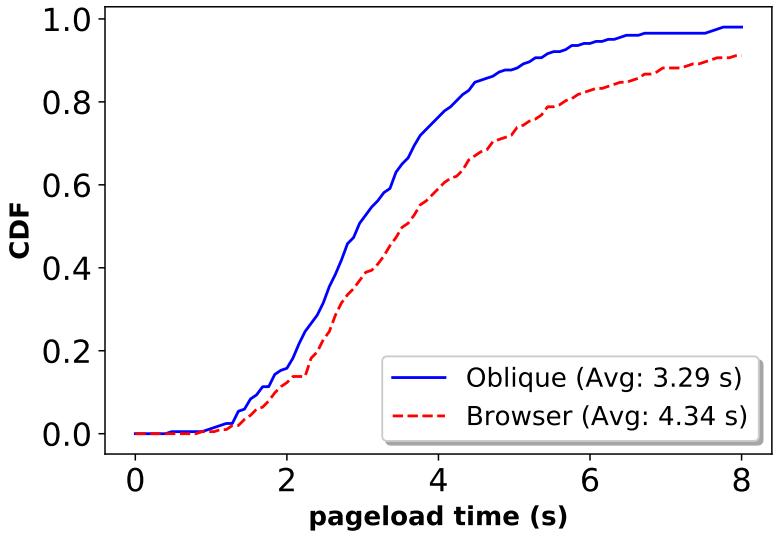


Figure 6.3: The page load times of our evaluated webpages when compared to a standard browser.

mented the Vroom proxy based on the source code of nghttp2, myhtml, and katana. For measuring each performance metric we tested the each webpage 5 times and computed the average. For Oblique and Vroom, we tested each webpage 1 hour after each of their dependency analysis.

6.7.2 Browser Performance

We evaluated the performance of Oblique by measuring the average page load times of Oblique, Vroom, RDR, and a standard browser using the 200 websites selected from the Majestic Million list. We found that Oblique's average page load time (3.29s) was 31.9% faster than a vanilla browser (4.34s). This speedup was 17.3% higher than RDR (3.79s) and 5.4% higher than Vroom (3.43s). Overall Oblique improved page loading times by 1.1s on average in our evaluation.

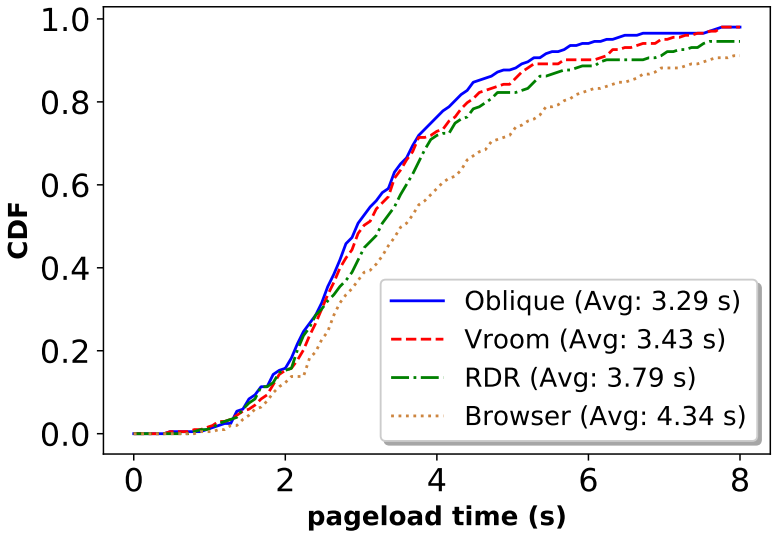


Figure 6.4: The page load times of our evaluated webpages when compared to a standard browser, Vroom, and RDR.

Figure 6.3 gives a cumulative distribution of time until page load for our evaluated webpages comparing Oblique and the standard browser. Oblique provides little advantage in pages which load quickly, indicating these pages tend to load quickly due to a lack of dependencies. In pages which take a long time to load, Oblique has the largest impact, shifting the distribution of page-load times for latent pages toward the 4 second mark, where the distribution is evenly spread between 4 to 8 seconds in the vanilla browser.

Figure 6.4 presents a cumulative distribution of Oblique, a standard browser, Vroom, and RDR. Oblique and Vroom both show more speedup than RDR as the depth of dependencies in the program execution increased because the offline analysis allows for these approaches to prefetch URLs immediately, but RDR needs to execute the JavaScript first.

Oblique's average loadtime was shorter than Vroom due to its higher hit rate for pre-fetched resources, in particular for dynamic URLs that involve non determinism.

On average, 2.06% of the URLs Oblique pre-cached went unused, and 2.24% went unused in Vroom. 22.8% of the URLs fetched by our RDR proxy were never requested, a much higher proportion of URLs. Each unused dependency consumes bandwidth, so maintaining a low number of incorrectly fetched files is key to maintaining performance.

For Oblique and Vroom, we also measured the page load time at various periods after the offline analysis of the website. After a 7 hour gap Oblique and Vroom's average page load time was 3.56s and 3.67s, respectively; 12 hours after analysis, their average was 3.58s and 3.88s. All of these were still faster than the average page load time of RDR (3.98s) and a vanilla browser (4.34s). Here, Oblique shows the smallest gap due to its support for pre-fetching of non-deterministic URLs.

6.7.3 Performance of ExpoSE

We now evaluate the performance of ExpoSE at finding resource dependencies. In our evaluation, the pre-fetched and missed dynamic dependencies (JavaScript driven dependencies discovered by ExpoSE) of a webpage were measured, so we know what proportion of page dependencies driven by the JavaScript engine were pre-fetched. Since Oblique's pre-fetch behaviour for dynamic URLs is dependent on ExpoSE having discovered the URL during symbolic execution we can directly use it to compare the performance of ExpoSE. Overall, we pre-fetched an average of 77.8% of the dynamically driven dependencies on a website. Figure 6.5 plots a cumulative distribution of dynamic URL hit rate across the websites in our evaluation. Overall ExpoSE performs well, identifying 9.6% more dynamic resources than Vroom, the other offline analysis in our eval-

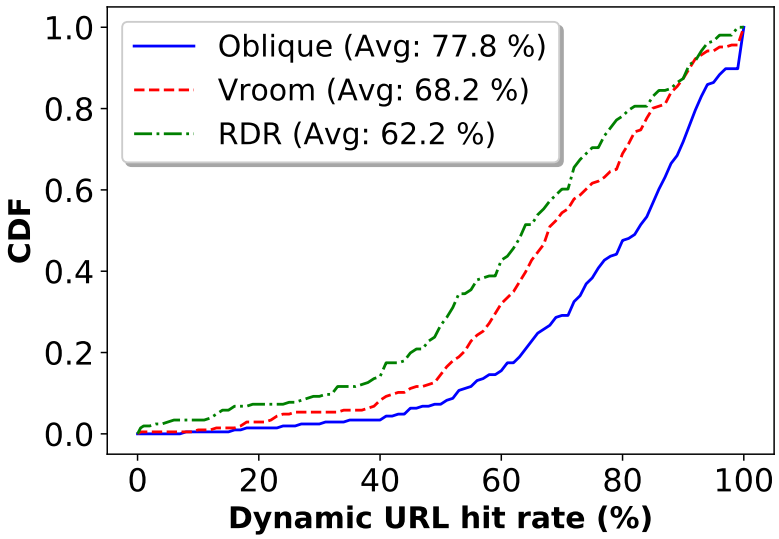


Figure 6.5: A cumulative distribution of Oblique, Vroom and RDR.

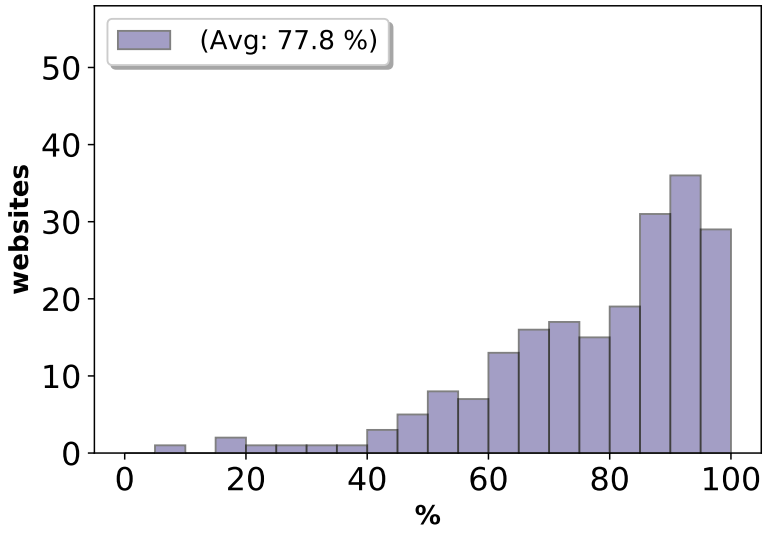


Figure 6.6: The distribution of cache misses across websites in our evaluation.

uation. Here, the lack of a need for a concrete browser shows a distinct advantage; The RDR implementation pre-fetched 15% less dynamic URLs than Oblique, since the browser did not load and execute the JavaScript fast enough to pre-fetch the resource for the client.

Digging further into the performance of dynamic URL component of Oblique further we now present Figure 6.6, looking at the distribution of websites and percentage of correctly identified URLs. We see that ExpoSE performs well, analyzing the majority of websites, since the weight of the distribution is between the 60% and 90% for dynamically identified URLs. In some cases, ExpoSE performs very poorly, identifying less than 40% of the total dynamic URLs on a webpage for pre-fetching, and in cases is as low as 10%. This is due to webpages with complex JavaScript, where the cost of executing an instrumented program, or solving path conditions with an SMT solver is infeasible. In these cases, ExpoSE fails to meaningfully analyze the page, but subsequent improvements in DSE for JavaScript may alleviate these issues.

6.7.4 Conclusions

Answering RQ1, we find that Oblique outperforms existing speculative loading technologies at reducing page load times, with a 5.4% improvement over Vroom, 17.3% improvement over RDR, and a 32% improvement over an unmodified browser. While the offline analysis component of ExpoSE takes longer than prior work, the results are more stable, since Oblique can support non-deterministic URLs and so analysis only needs to be run when the website source code changes. As such, we conclude that a speculative loading engine driven by DSE discovered resource dependencies can decrease page load times notably, outperforming previous approaches.

Addressing, RQ2, Oblique pre-fetches 77.8% of the dynamically loaded

URLs for a webpage. Since Oblique only uses the dependencies detected by ExpoSE to pre-fetch dynamic URLs, we can use this to directly evaluate the performance of ExpoSE on the tested websites. We see that ExpoSE is good at finding dynamic URLs in websites, even in those with long dependency chains. While we do not discover all of the dynamically trigerred URLs, DSE is rarely exhaustive and our models for the DOM are incomplete, so this was expected. These results show that ExpoSE is a good choice when analyzing web applications, since it is able to explore the majority of dynamically driven URLs on a page.

Overall, while there is room for improvement in the missed pre-fetches by ExpoSE, we see that DSE is a good option for offline analysis of webpages when performing speculative loading.

7 Conclusions

In this thesis we have furthered the state of the art in dynamic JavaSript program analysis through our new tool ExpoSE, which improves upon prior symbolic execution tools for JavaScript in a number of key areas. ExpoSE uses an instrumentation based approach to perform symbolic execution in a standard interpreter. The engine isolates individual test cases and does not reuse executions to ensure the correct execution of asynchronous code and avoid testing artefacts. Concurrent test case execution speeds up symbolic execution, and makes ExpoSE highly scalable. The modular design allows tailoring to specific use cases, with ExpoSE supporting the execution of Node.js and web applications out of the box. ExpoSE can symbolically encode all JavaScript base types, including strings, arrays, and objects, but may under-approximate to keep SMT problems feasible. We use search strategies to decrease coverage plateus, reducing the time required to achieve useful symbolic execution. The resulting symbolic execution framework is highly compatible with current JavaScript programs, scalable, and performs well when exploring popular real JavaScript libraries.

Our complete support for ES6 regular expressions allows ExpoSE to explore deeper into programs. Regular expression modeling in JavaScript is challenging because JavaScript regular expressions are non-regular, supporting features such as capture groups and backreferences. With a complete encoding, our analysis is still over-approximate due to matching precedence, which can lead to us generating non-useful test cases. We address the problem of matching precedence through a CEGAR refinement strategy which guides the SMT solver toward a correct assignment for capture groups in accepted matches. This encoding is particularly important for ExpoSE because 35% of all NPM packages use regular expressions, and 21% of all NPM packages contain capture groups. With this support ExpoSE is able to explore programs which use these features more accurately, increasing coverage and making bug detection more likely.

Our support for symbolic objects and arrays also allows us to explore further into programs. Objects and arrays are hard to model because they do not have strict structure or typing, but current SMT solvers cannot reason about untyped maps and arrays. Our dynamic approach to object encoding uses field operations observed at runtime to guide the generation of new test cases. With this approach we support symbolic inputs but keep the SMT problems that drive test case generation feasible for current solvers.

We demonstrate that ExpoSE is capable of executing the intricacies of JavaScript by generating supplementary conformance tests for JavaScript native library methods. We generate these new test cases through the symbolic execution of JavaScript polyfill implementations. Each test case is then used to check a set of known JavaScript implementations for correctness, using a battery of interpreters and differential testing to decide if an interpreter fails a test. Using this approach, we were able to find several errors in a widely depended polyfill implementation and extend coverage of the Test262 suite, showing that our approach to symbolic execution is capable of reasoning about intricacies of low-level JavaScript code.

By applying ExpoSE to webpage dependency detection we show that our engine can explore the majority of code on current websites. The resource dependency constraints we identify during symbolic execution are used to generate a set of concrete URLs to speculatively load. The advantage of this approach is support for speculative loading of non-deterministic URLs, since we use the symbolic representation built up during analysis with ExpoSE. In evaluation we showed that ExpoSE performs well when symbolically executing webpages, finding the 77.8% of dynamic dependencies. We also demonstrate that symbolic execution improves the performance over speculative loading when compared to previous approaches.

Overall, in this thesis, we have shown that the symbolic execution can be

usefully applied to modern JavaScript. We have developed a highly compatible dynamic symbolic execution engine for JavaScript which scales to real world programs by keeping constraint problems feasible for current solvers. The development of automated program analysis tools for JavaScript opens up new opportunities, not only for improving the stability and security of programs but also in other domains, such as improvement web browser performance. We have shown that this new engine is useful in two domains, conformance test case generation and speculative loading, showing that a JavaScript analysis engine is not limited to program bug finding and demonstrating the utility of our new tool. The work presented in this thesis unlocks dynamic symbolic execution for real-world JavaScript, increasing confidence in software we use every day while providing a versatile test-bed with which to carry out future research.

Bibliography

[1] P. A. Abdulla et al. “Trau: SMT solver for string constraints”. In: *2018 Formal Methods in Computer Aided Design (FMCAD)*. 2018, pp. 1–5.

[2] Parosh Aziz Abdulla et al. “Flatten and Conquer: A Framework for Efficient Analysis of String Constraints”. In: *ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI. 2017.

[3] Parosh Aziz Abdulla et al. “Norn: An SMT Solver for String Constraints”. In: *Computer Aided Verification (CAV)*. 2015.

[4] Parosh Aziz Abdulla et al. “String Constraints for Verification”. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 150–166. DOI: 10.1007/978-3-319-08867-9_10.

[5] *AFL*. https://lcamtuf.coredump.cx/afl/.

[6] Alfred V. Aho. “Algorithms for Finding Patterns in Strings”. In: *Handbook of Theoretical Computer Science (Vol. A)*. Ed. by Jan van Leeuwen. Cambridge, MA, USA: MIT Press, 1990, pp. 255–300.

[7] Roberto Amadini et al. "Constraint Programming for Dynamic Symbolic Execution of JavaScript". In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2019, pp. 1–19.

[8] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. "Demand-driven Compositional Symbolic Execution". In: *14th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*. TACAS. 2008.

[9] Saswat Anand, Corina S Păsăreanu, and Willem Visser. "JPF–SE: A symbolic execution extension to java pathfinder". In: *International conference on tools and algorithms for the construction and analysis of systems*. Springer. 2007, pp. 134–138.

[10] Thanassis Avgerinos et al. "Enhancing symbolic execution with Veritesting". In: *36th Int. Conf. Software Engineering (ICSE)*. 2014, pp. 1083–1094. DOI: 10.1145/2568225.2568293.

[11] Sora Bae, Joonyoung Park, and Sukyoung Ryu. "Partition-Based Coverage Metrics and Type-Guided Search in Concolic Testing for JavaScript Applications". In: *Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering*. FormaliSE '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 72–78. ISBN: 9781538604229.

[12] Clark Barrett et al. "Cvc4". In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 171–177.

[13] Fabrice Bellard. *QuickJS*. https://bellard.org/quickjs/. 2017.

[14] Josh Berdine, Cristiano Calcagno, and Peter W O'hearn. "Symbolic execution with separation logic". In: *Asian Symposium on Programming Languages and Systems*. Springer. 2005, pp. 52–68.

[15] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. "Path Feasibility Analysis for String-Manipulating Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2009.

[16] Nikolaj Bjørner et al. "SMT-LIB Sequences and Regular Expressions". In: *Int. Workshop on Satisfiability Modulo Theories (SMT)*. 2012.

[17] Martin Bodin et al. "A Trusted Mechanised JavaScript Specification". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: ACM, 2014, pp. 87–100.

[18] Robert Brummayer and Armin Biere. "Boolector: An efficient SMT solver for bit-vectors and arrays". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2009, pp. 174–177.

[19] Roberto Bruttomesso et al. "The mathsat 4 smt solver". In: *International Conference on Computer Aided Verification*. Springer. 2008, pp. 299–303.

[20] Stefan Bucur, Johannes Kinder, and George Candea. "Prototyping Symbolic Execution Engines for Interpreted Languages". In: *Proc. 19th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2014, pp. 239–254.

[21] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Proc. 8th Symp. Operating Systems Design and Implementation (OSDI 2008)*. USENIX, 2008, pp. 209–224.

[22] Cristian Cadar et al. “EXE: automatically generating inputs of death”. In: *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*. ACM, 2006, pp. 322–335.

[23] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. “A Formal Study of Practical Regular Expressions”. In: *Int. J. Foundations of Computer Science* 14.06 (2003).

[24] Sang Kil Cha et al. “Unleashing Mayhem on Binary Code”. In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. 2012.

[25] Satish Chandra, Stephen J. Fink, and Manu Sridharan. “Snugglebug: A Powerful Approach to Weakest Preconditions”. In: (2009).

[26] Carl Chapman and Kathryn T. Stolee. “Exploring Regular Expression Usage and Context in Python”. In: *Int. Symp. on Software Testing and Analysis*. ISSTA. 2016.

[27] S. Y. Chau et al. “SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 503–520. DOI: 10.1109/SP.2017.40.

[28] Bihuan Chen, Yang Liu, and Wei Le. “Generating Performance Distributions via Probabilistic Symbolic Execution”. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Association for Computing Machinery, 2016. ISBN: 9781450339001. DOI: 10.1145/2884781.2884794. URL: https://doi.org/10.1145/2884781.2884794.

[29] Taolue Chen et al. “What is decidable about string constraints with the ReplaceAll function”. In: *PACMPL* 2.POPL (2018), 3:1–3:29.

[30] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, 2011, pp. 265–278.

[31] Ravi Chugh, David Herman, and Ranjit Jhala. "Dependent types for JavaScript". In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 2012, pp. 587–606.

[32] Ryan Dahl. *Node.js*. `https://github.com/nodejs/node`. 2009.

[33] *Dart*. `https://dart.dev/guides/language/specifications/DartLangSpec-v2.2.pdf`.

[34] Martin Davis, George Logemann, and Donald Loveland. "A Machine Program for Theorem-Proving". In: *Commun. ACM* 5.7 (July 1962), pp. 394–397.

[35] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (July 1960), pp. 201–215.

[36] Monika Dhok, Murali Krishna Ramanathan, and Nishant Sinha. "Type-Aware Concolic Testing of JavaScript Programs". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 168–179. ISBN: 9781450339001. DOI: `10.1145/2884781.2884859`. URL: `https://doi.org/10.1145/2884781.2884859`.

[37] Bruno Dutertre and Leonardo De Moura. "A fast linear-arithmetic solver for DPLL (T)". In: *International Conference on Computer Aided Verification*. Springer. 2006, pp. 81–94.

[38] ECMA International. *ECMAScript 2015 Language Specification*. ECMA International. 2015.

[39] ECMA International. *Test262*. `https://github.com/tc39/test262`. 2017.

[40] *Electron*. `https://www.electronjs.org/`.

[41] *Flow*. `http://flowtype.org/`.

[42] Xiang Fu et al. "Simple linear string constraints". In: *Formal Asp. Comput.* 25.6 (2013).

[43] Vijay Ganesh et al. "HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection". In: *Computer Aided Verification*. Springer Berlin Heidelberg, 2011, pp. 1–19.

[44] Harald Ganzinger et al. "DPLL (T): Fast decision procedures". In: *International Conference on Computer Aided Verification*. Springer. 2004, pp. 175–188.

[45] Philippa Gardner, Daiva Naudziuniene, and Gareth Smith. "JuS: Squeezing the sense out of JavaScript programs". In: *JSTools@ ECOOP* (2013).

[46] Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. "Towards a program logic for JavaScript". In: *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2012, pp. 31–44.

[47] Patrice Godefroid. "Compositional Dynamic Test Generation". In: *ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*. POPL. 2007.

[48] Patrice Godefroid. "Compositional dynamic test generation". In: *Proc. 34th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2007)*. 2007, pp. 47–54.

[49] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *Proc. ACM SIGPLAN 2005 Conf. Programming Language Design and Implementation (PLDI 2005)*. ACM, 2005, pp. 213–223.

[50] Patrice Godefroid, Michael Levin, and David Molnar. "Automated Whitebox Fuzz Testing". In: *Network and Distributed System Security Symp. (NDSS)*. 2008.

[51] Patrice Godefroid, Michael Y Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing". In: *Queue* 10.1 (2012), pp. 20–27.

[52] Patrice Godefroid and Daniel Luchaup. "Automatic partial loop summarization in dynamic test generation". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 2011, pp. 23–33.

[53] D. Gopinath et al. "Symbolic Execution for Attribution and Attack Synthesis in Neural Networks". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2019, pp. 282–283. DOI: 10.1109/ICSE-Companion.2019.00115.

[54] Shengjian Guo et al. "SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1235–1247. ISBN: 9781450371216. DOI: 10.1145/3377811.3380428. URL: https://doi.org/10.1145/3377811.3380428.

[55] Paul Hamill. *Unit test frameworks: tools for high-quality software development*. " O'Reilly Media, Inc.", 2004.

[56] Klaus Havelund and Thomas Pressburger. "Model Checking Java Programs using Java PathFinder". In: *Int. J. Softw. Tools Technol. Transfer (STTT)* 2.4 (2000), pp. 366–381.

[57] Daniel Hedin et al. "JSFlow: Tracking Information Flow in JavaScript and Its APIs". In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC '14. Gyeongju, Republic of Korea: ACM, 2014, pp. 1663–1671.

[58] Lukás Holík et al. "String constraints with concatenation and transducers solved efficiently". In: *PACMPL* 2.POPL (2018), 4:1–4:32.

[59] *How are 10 Global Companies Using Node.js in Production?* `https://www.tothenew.com/blog/how-are-10-global-companies-using-node-js-in-production/`.

[60] *How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript*. `https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/`.

[61] *HTTP2*. `https://tools.ietf.org/html/rfc7540`.

[62] Joxan Jaffar, Jorge A Navas, and Andrew E Santosa. "Unbounded symbolic execution for program verification". In: *International Conference on Runtime Verification*. Springer. 2011, pp. 396–411.

[63] Joxan Jaffar et al. "TRACER: A symbolic execution tool for verification". In: *International Conference on Computer Aided Verification*. Springer. 2012, pp. 758–766.

[64] Michał Jezierski. *mdn-polyfills*. `https://github.com/msn0/mdn-polyfills`. 2016.

[65] *Jsfuzz*. `https://github.com/fuzzitdev/jsfuzz`.

[66] Timotej Kapus and Cristian Cadar. "A segmented memory model for symbolic execution". In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by Marlon Dumas et al. 2019, pp. 774–784. DOI: 10.1145/3338906.3338936. URL: https://doi.org/10.1145/3338906.3338936.

[67] Scott Kausler and Elena Sherman. "Evaluation of String Constraint Solvers in the Context of Symbolic Execution". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 259–270. DOI: 10.1145/2642937.2643003. URL: https://doi.org/10.1145/2642937.2643003.

[68] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. "Generalized Symbolic Execution for Model Checking and Testing". In: *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003*. Springer, 2003, pp. 553–568. DOI: 10.1007/3-540-36577-X_40.

[69] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[70] Erik Krogh Kristensen and Anders Møller. "Type test scripts for TypeScript testing". In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 90:1–90:25. DOI: 10.1145/3133914. URL: https://doi.org/10.1145/3133914.

[71] Volodymyr Kuznetsov et al. "Efficient state merging in symbolic execution". In: *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*. ACM, 2012, pp. 193–204.

[72] William Landi. "Undecidability of static analysis". In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.4 (1992), pp. 323–337.

[73] Guodong Li, Esben Andreasen, and Indradeep Ghosh. "SymJS: automatic symbolic testing of JavaScript web applications". In: *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE)*. 2014, pp. 449–459. DOI: 10.1145/2635868.2635913.

[74] Guodong Li and Indradeep Ghosh. "PASS: String solving with parameterized array and interval automaton". In: *Haifa Verification Conference (HVC)*. 2013.

[75] Nuo Li et al. "Reggae: Automated test generation for programs using complex regular expressions". In: *Automated Software Engineering (ASE)*. 2009.

[76] Tianyi Liang et al. "A Decision Procedure for Regular Membership and Length Constraints over Unbounded Strings". In: *Int. Symp. on Frontiers of Combining Systems*. FroCoS. 2015.

[77] Tianyi Liang et al. "A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions". In: *Computer Aided Verification (CAV)*. 2014.

[78] Xuanzhe Liu et al. "SWAROVsky: Optimizing resource loading for mobile web browsing". In: *IEEE Transactions on Mobile Computing* 16.10 (2016), pp. 2941–2954.

[79] Blake Loring, Duncan Mitchell, and Johannes Kinder. "ExpoSE: Practical Symbolic Execution of Standalone JavaScript". In: *Proc. Int. SPIN Symp. Model Checking of Software (SPIN)*. ACM, 2017, pp. 196–199.

[80] Blake Loring, Duncan Mitchell, and Johannes Kinder. "Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 425–438.

[81] Kin-Keung Ma et al. "Directed symbolic execution". In: *International Static Analysis Symposium*. Springer. 2011, pp. 95–111.

[82] *Majestic Million*. `https://majestic.com/reports/majestic-million`.

[83] *Malicious code found in npm package event-stream downloaded 8 million times in the past 2.5 months*. `https://snyk.io/blog/malicious-code-found-in-npm-package-event-stream/`.

[84] Paul Dan Marinescu and Cristian Cadar. "make test-zesti: A symbolic execution solution for improving regression testing". In: *34th Int. Conf. Software Engineering (ICSE)*. 2012, pp. 716–726. DOI: `10.1109/ICSE.2012.6227146`.

[85] Dimiter Milushev, Wim Beck, and Dave Clarke. "Noninterference via Symbolic Execution". In: *Formal Techniques for Distributed Systems*. Ed. by Holger Giese and Grigore Rosu. Springer Berlin Heidelberg, 2012.

[86] Shabnam Mirshokraie and Ali Mesbah. "JSART: JavaScript assertion-based regression testing". In: *International Conference on Web Engineering*. Springer. 2012, pp. 238–252.

[87] *MochaJS*. `https://mochajs.org/`.

[88] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2008.

[89] Mozilla Foundation. *SpiderMonkey*. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`. 1996.

[90] Ravi Netravali et al. "Polaris: Faster Page Loads Using Fine-grained Dependency Tracking". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 2016.

[91] Ravi Netravali et al. "Watchtower: Fast, secure mobile page loads using remote dependency resolution". In: *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. 2019, pp. 430–443.

[92] Mozilla Developer Network. *Array.prototype.findIndex Polyfill*. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex`. 2020.

[93] Srinivas Nidhra and Jagruthi Dondeti. "Black box and white box testing techniques-a literature review". In: *International Journal of Embedded Systems and Applications (IJESA)* 2.2 (2012), pp. 29–50.

[94] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)". In: *J. ACM* 53.6 (Nov. 2006), pp. 937–977.

[95] *npm bans terminal ads*. `https://www.zdnet.com/article/npm-bans-terminal-ads/`.

[96] *npm Pulls Malicious Package that Stole Login Passwords*. https://www.bleepingcomputer.com/news/security/npm-pulls-malicious-package-that-stole-login-passwords/.

[97] Hristina Palikareva and Cristian Cadar. "Multi-solver support in symbolic execution". In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 53–68.

[98] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. "Shadow of a doubt: testing for divergences between software versions". In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 2016. DOI: 10.1145/2884781.2884845.

[99] Daejun Park, Andrei Stefănescu, and Grigore Roşu. "KJS: A complete formal semantics of JavaScript". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2015, pp. 346–356.

[100] Corina S. Pasareanu and Neha Rungta. "Symbolic PathFinder: symbolic execution of Java bytecode". In: *Int. Conf. on Automated Software Eng. (ASE)*. 2010, pp. 179–180.

[101] Suzette Person et al. "Differential Symbolic Execution". In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '08/FSE-16. New York, NY, USA: Association for Computing Machinery, 2008. ISBN: 9781595939951.

[102] Denis Pushkarev. *core-js*. https://www.npmjs.com/package/core-js. 2014.

[103] Shengchao Qin et al. "Towards an axiomatic verification system for javascript". In: *2011 Fifth International Conference on Theoretical Aspects of Software Engineering*. IEEE. 2011, pp. 133–141.

[104] Aseem Rastogi et al. "Safe & efficient gradual typing for TypeScript". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2015, pp. 167–180.

[105] Heinz Riener et al. "MetaSMT: Focus on Your Application and Not on Solver Integration". In: vol. 19. 5. Berlin, Heidelberg: Springer-Verlag, 2017, pp. 605–621.

[106] Robby, Matthew B. Dwyer, and John Hatcliff. "Bogor: an extensible and highly-modular software model checking framework". In: *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*. ACM, 2003, pp. 267–276. DOI: 10.1145/940071.940107.

[107] Vaspol Ruamviboonsuk et al. "Vroom: Accelerating the mobile web with server-aided dependency resolution". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 2017, pp. 390–403.

[108] José Fragoso Santos et al. "JaVerT: JavaScript verification toolchain". In: *PACMPL* 2.POPL (2018), 50:1–50:33.

[109] José Fragoso Santos et al. "Symbolic execution for javascript". In: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. 2018, pp. 1–14.

[110] José Fragoso Santos et al. "Towards Logic-Based Verification of JavaScript Programs". In: *International Conference on Automated Deduction*. Springer. 2017, pp. 8–25.

[111] Prateek Saxena et al. "A Symbolic Execution Framework for JavaScript". In: *IEEE Symp. Sec. and Privacy (S&P)*. 2010.

[112] Joseph D. Scott, Pierre Flener, and Justin Pearson. "Constraint Solving on Bounded String Variables". In: *Integration of AI and OR Tech. in Constraint Prog. (CPAIOR)*. 2015.

[113] Marija Selakovic et al. "Test generation for higher-order functions in dynamic languages". In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 161:1–161:27. DOI: 10.1145/3276531. URL: https://doi.org/10.1145/3276531.

[114] Koushik Sen. "Concolic Testing". In: *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 571–572.

[115] Koushik Sen et al. "Jalangi: a selective record-replay and dynamic analysis framework for JavaScript". In: *ACM SIGSOFT Symp. Foundations of Software Engineering (ESEC/FSE 2013)*. 2013, pp. 488–498. DOI: 10.1145/2491411.2491447.

[116] Yan Shoshitaishvili et al. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *Proc. IEEE Symp. Security and Privacy (S&P)*. 2016, pp. 138–157.

[117] *Sinon*. https://sinonjs.org/.

[118] Ashiwan Sivakumar et al. "Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction". In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 2014, pp. 325–336.

[119] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[120] Haiyang Sun et al. "Efficient dynamic analysis for Node. js". In: *Proceedings of the 27th International Conference on Compiler Construction*. 2018, pp. 196–206.

[121] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[122] Nikhil Swamy et al. "Gradual typing embedded securely in JavaScript". In: *ACM SIGPLAN Notices* 49.1 (2014), pp. 425–437.

[123] Julian Thomé et al. "Search-driven string constraint solving for vulnerability detection". In: *Int. Conf. on Software Engineering, (ICSE)*. 2017.

[124] Nikolai Tillmann and Jonathan De Halleux. "Pex–white box test generation for. net". In: *International conference on tests and proofs*. Springer. 2008, pp. 134–153.

[125] Nikolai Tillmann and Wolfram Schulte. "Parameterized unit tests". In: *Foundations of Software Engineering (FSE)*. 2005.

[126] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. "Model Counting for Recursively-Defined Strings". In: *Computer Aided Verification (CAV)*. 2017.

[127] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. "Progressive Reasoning over Recursively-Defined Strings". In: *Computer Aided Verification (CAV)*. 2016.

[128] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. "S3: A Symbolic String Solver for Vulnerability Detection in Web Applications". In: *Conf. Computer and Commun. Sec. (CCS)*. 2014.

[129] *TypeScript*. `https://www.typescriptlang.org/`.

[130] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. "Rex: Symbolic regular expression explorer". In: *Software Testing, Verification and Validation (ICST)*. 2010.

[131] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. "Refinement types for TypeScript". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 2016, pp. 310–325.

[132] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. "Test input generation with Java PathFinder". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*. ACM, 2004, pp. 97–107. DOI: 10.1145/1007512.1007526.

[133] Le Wang and Jukka Manner. "Energy-efficient mobile web in a bundle". In: *Computer Networks* 57.17 (2013), pp. 3581–3600.

[134] *What happens when the maintainer of a JS library downloaded 26m times a week goes to prison for killing someone with a motorbike? Core-js just found out*. https://www.theregister.co.uk/2020/03/26/corejs_maintainer_jailed_code_release/.

[135] Tao Xie et al. "Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution". In: *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Springer, 2005, pp. 365–381. DOI: 10.1007/978-3-540-31980-1_24.

[136] Babak Yadegari and Saumya Debray. "Symbolic Execution of Obfuscated Code". In: *Proceedings of the 22nd ACM SIGSAC Con-*

ference on Computer and Communications Security. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015. ISBN: 9781450338325.

[137] Hengbiao Yu. "Combining Symbolic Execution and Model Checking to Verify MPI Programs". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018.

[138] Chuan Yue and Haining Wang. "Characterizing insecure javascript practices on the web". In: *Proceedings of the 18th international conference on World wide web*. 2009, pp. 961–970.

[139] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. "Z3-str: A Z3-based String Solver for Web Application Analysis". In: *Foundations of Software Engineering (FSE)*. Saint Petersburg, Russia, 2013.

[140] Yunhui Zheng et al. "Effective Search-Space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints". In: *Computer Aided Verification (CAV)*. 2015.

[141] Yunhui Zheng et al. "Z3str2: an efficient solver for strings, regular expressions, and length constraints". In: *Formal Methods in System Design* 50.2-3 (2017).

[142] Chaoshun Zuo and Zhiqiang Lin. "SMARTGEN: Exposing Server URLs of Mobile Apps With Selective Symbolic Execution". In: *Proceedings of the 26th International Conference on World Wide Web*. WWW '17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 867–876. ISBN: 9781450349130. DOI: 10.1145/3038912.3052609. URL: https://doi.org/10.1145/3038912.3052609.