OPTIMIZING WEB APPLICATIONS USING DECLARATIVE SOFTWARE REWRITING

SATYAJIT GOKHALE

Doctor of Philosophy in Computer Science
Khoury College of Computer Sciences
Northeastern University

August 2025

Northeastern University
Khoury College of
Computer Sciences

PhD Thesis Approval

Thesis Title: Optimizing Web Applications using Declarative Software Rewriting

Author: Satyajit Gokhale

PhD Program: x Computer Science ____ Cybersecurity ____ Personal Health Informatics

PhD Thesis Approval to complete all degree requirements for the above PhD program.

Thesis Advisor	Date: 7/29/25
Thesis Reader	Date: July 29 2025
Thesis Reader	Date: 07.30.2025
Thesis Reader	Date: July 30, 2025
Thesis Reader	Date

KHOURY COLLEGE APPROVAL:

Associate Dean for Graduate Programs	Date: 7/30/2025

COPY RECEIVED BY GRADUATE STUDENT SERVICES:

Recipient's Signature	Date: 7/30/2025

Distribution: Once completed, this form should be attached as page 2, immediately following the title page of the dissertation document. An electronic version of the document can then be uploaded to the Northeastern University-UMI Website.

ABSTRACT

Performance and security are among the most desirable properties of Web Applications. The success of a Web Application depends on user satisfaction and is strongly correlated with the perceived responsiveness of the application and performance. Additionally, security is of utmost importance to retain user trust in a world ridden with malicious actors. However, drawing upon the full potential of a web application can involve the introduction of non-linear control flow, framework-specific changes and optimizations, or extensive refactoring for adoption of newer constructs in an ever-evolving ecosystem. Similarly, mitigating security vulnerabilities is also not trivial. These changes are often intrusive and can span multiple files. In this thesis, we explore how a declarative approach to specifying project-spanning program transformations can address these challenges. We present three major themes to improve the responsiveness and performance of web applications, and one theme to improve security: introducing asynchrony, introducing laziness, reducing superfluous computation, and introducing memory segmentation. We demonstrate the application of a declarative re-writing approach to these themes in the following contributions: (i) automatic migration from synchronous to asynchronous JavaScript APIs, (ii) increasing the responsiveness of web applications by introducing lazy loading, (iii) remediating superfluous re-rendering in React applications, and (iv) rewriting WebAssembly binaries to mitigate security vulnerabilities.

ACKNOWLEDGMENTS

As a kid who mostly hated studying but sort of enjoyed writing code, I would probably have laughed at the thought of doing a Ph.D. one day. Yet here I am today, on the brink of completing a Ph.D. in Computer Science. This would not have been possible without the right set of circumstances and, more importantly, the right set of wonderful people and their contributions.

I express my heartfelt gratitude to my advisor Frank, who pulled me into the world of research as an apprentice and sowing the idea of doing a Ph.D.. You have been a wonderful advisor who has been supportive and encouraging throughout and believed in me every time I cut things too close to the wire. I look forward to working together on many more projects in the future. I wish to thank my thesis committee, Engin Kirda, Arjun Guha, Ben Liblit, for their support, valuable feedback, and questions. I have enjoyed the conversations I had with each one of you and it has helped me grow.

I wish to thank Omer Tripp and the wonderful folks at CodeGuru/Amazon Q for hosting me as an intern every summer and letting me work on cool new things that were a welcome break from my thesis-related research. It has been fun working with all of you.

I would like to thank all the faculty, staff, and students of the Programming Research Lab at Northeastern University for making my Ph.D. a fun experience and for your insightful comments and feedback about my research and presentations. I have thoroughly enjoyed all our conversations (albeit not fully understood) about a plethora of topics ranging from AI (Artificial Intelligence) to AI (Abstract Interpretation), and everything in between. Thank you Alexi Turcotte, Ellen Arteca, Michelle Thalakottur, Farideh Khalili, Harshit Garg, Max Bernstein, Artem Pelenitsyn, Julia Belyakova, Aviral Goel, Ben Chung, Ming-Ho Yee, Katie Hough, John Gouwar, James Perretta, Cameron Moy, Sam Stites, Vadym Matviichuk, Brianna Marshall, Francesca Lucchetti, Yangtian Zi, Olek Gierczak, Nate Yazdani, Andrew Wagner, Lucy Menon, Donald Pinckney, John Li, Minsung Cho, Gwen Lincroft, Liam DeVoe, Shubh Agrawal, Conrad Zimmerman, Luis Garcia, Sam Caldwell, Michael Ballantyne, and Ryan Doenges. Thank you Jon Bell, for funding me and giving me the opportunity to work on flaky tests. Thank you Sarah Gale, Laura Adrien, and Pete Morency for taking care of all the administrative stuff.

I have worked closely with Alexi throughout my PhD, and prior to that as a research apprentice. I consider myself lucky to have an exemplary researcher as a collaborator and friend. I hope to work with you for many more years.

I would like to thank the Khoury Graduate Research Apprenticeship program, specifically Jose Sierra for nominating me and Kim Gubelman for managing the program, for giving me the opportunity to work with Frank. I would not have done a PhD without it.

Finally, I would like to thank my parents and grandparents for their endless love, support, and encouragement, and for pushing me to finish on time. Thank you, Shreyas and Nandini, for always having my back and all the fun discussions; having you guys in Boston has been great.

CONTENTS

1 INTRODUCTION 1
- 1.1 Anatomy of Source Code Transformations . . . 1
- 1.2 Declarative Rewrite Rules . . . 3
- 1.3 Thesis Overview . . . 5

2 BACKGROUND 7
- 2.1 Asynchronous Programming in JavaScript . . . 7
  - 2.1.1 Synchronous I/O . . . 7
  - 2.1.2 Event-Driven Asynchronous I/O . . . 9
  - 2.1.3 Brief Review of Promises and Async/Await . . . 9
  - 2.1.4 Promise-Based Asynchronous I/O . . . 11
- 2.2 Importing Packages In JavaScript . . . 12
  - 2.2.1 Require . . . 12
  - 2.2.2 Static Imports . . . 12
  - 2.2.3 Dynamic Imports . . . 13
- 2.3 React Basics . . . 13
  - 2.3.1 Components . . . 15
  - 2.3.2 React syntax . . . 15
  - 2.3.3 State and Props . . . 15
  - 2.3.4 Rendering . . . 16
  - 2.3.5 State Management . . . 16
  - 2.3.6 Hooks . . . 17
- 2.4 WebAssembly Primer . . . 18
  - 2.4.1 Modules . . . 19
  - 2.4.2 Types . . . 19
  - 2.4.3 Globals . . . 19
  - 2.4.4 Imports/Exports . . . 20
  - 2.4.5 Functions . . . 20
  - 2.4.6 Linear Memory . . . 21
  - 2.4.7 Indirect Function Calls . . . 21

3 INTRODUCING ASYNCHRONY IN JAVASCRIPT APPLICATIONS 22
- 3.1 Introduction . . . 22
- 3.2 Motivation . . . 24
  - 3.2.1 Performance Benefits of Asynchronous APIs . . . 24
  - 3.2.2 Migrating from Synchronous to Asynchronous APIs . . . 26
- 3.3 Approach . . . 29

3.3.1 Identify Transformation Candidates . . . 30
3.3.2 Discard Unsupported Transformations . . . 32
3.3.3 Transform Source Code . . . 33
3.3.4 Call Graph Construction . . . 36
3.4 Implementation . . . 36
3.5 Evaluation . . . 37
3.5.1 Subject Applications . . . 37
3.5.2 Experimental Design . . . 38
3.5.3 Experimental Results . . . 39
3.6 Threats to Validity . . . 42
3.7 Conclusion . . . 43
3.8 Publication Status . . . 43
4 INTRODUCING LAZY LOADING IN JAVASCRIPT APPLICATIONS 44
4.1 Introduction . . . 44
4.2 Motivation . . . 45
4.3 Approach . . . 47
4.3.1 Identify Candidate Packages for Lazy Loading . . . 48
4.3.2 Validate and Determine Transformations Required . . . 49
4.3.3 Code Transformations . . . 51
4.4 Implementation . . . 53
4.5 Evaluation . . . 53
4.6 Threats to Validity . . . 59
4.7 Conclusion . . . 60
4.8 Publication Status . . . 61
5 PREVENTING SUPERFLUOUS RE-RENDERING IN REACT APPLICATIONS 62
5.1 Introduction . . . 62
5.2 Motivating Example . . . 64
5.3 Anti-Patterns . . . 66
5.3.1 Anti-pattern detection Methodology . . . 66
5.3.2 Anti-Pattern 1: Controlled Component . . . 67
5.3.3 Anti-Pattern 2: State Variables not Affecting the UI . . . 68
5.3.4 Anti-Pattern 3: Propless Child Component . . . 69
5.3.5 Anti-Pattern 4: Child Component with Object or Array Element Prop 70
5.3.6 Anti-Pattern 5: Child Component receiving Function Prop . . . 71
5.4 Detecting and Remediating Anti-Patterns . . . 71
5.4.1 Controlled Component . . . 71
5.4.2 State Variables not Affecting the UI . . . 73
5.4.3 Propless Child Component . . . 73
5.4.4 Child Component with Object or Array Element Prop . . . 73

5.4.5 Child Component receiving Function Prop . . . 74
5.5 Implementation . . . 74
5.6 Evaluation . . . 75
5.6.1 Experimental Methodology . . . 75
5.6.2 RQ1: How prevalent are the anti-patterns? . . . 77
5.6.3 RQ2: How often do the transformations cause behavioral differences? 79
5.6.4 RQ3: How do the transformations impact the number of re-rendering operations . . . 79
5.6.5 RQ4: What is the impact of the transformations on performance? . . 80
5.6.6 RQ5: What is the impact of the transformations on code complexity? 83
5.6.7 RQ6: What is the cost of the analysis? . . . 83
5.7 Threats to Validity . . . 83
5.8 Generalizability . . . 84
5.9 Conclusion . . . 85
6 REWRITING WEBASSEMBLY BINARIES TO MITIGATE SECURITY VULNERABILITIES 86
6.1 Introduction . . . 86
6.2 Motivation . . . 87
6.2.1 Traditional Buffer Overflow Vulnerabilities . . . 88
6.2.2 Persistence of Traditional Buffer Overflows in WebAssembly . . . 89
6.2.3 Preventing Buffer Overflows . . . 92
6.2.4 Hardening buffer.wat . . . 94
6.3 Approach . . . 96
6.3.1 Assumptions for Behavior Preservation . . . 97
6.3.2 Collect Execution Traces and Static Metadata . . . 98
6.3.3 Identify Transformation Candidates and Compute Bounds . . . 99
6.3.4 Rewriting the Binary . . . 100
6.4 Implementation . . . 104
6.5 Evaluation . . . 104
6.5.1 Testset and Experimental Setup . . . 105
6.5.2 RQ1: Runtime Trace Collection Effectiveness . . . 106
6.5.3 RQ2: Hardening Effectiveness . . . 108
6.5.4 RQ3: Correctness and Preservation of Benign Behavior . . . 109
6.5.5 RQ4: Runtime Overhead . . . 110
6.6 Discussion . . . 111
6.7 Conclusion . . . 112
7 RELATED WORK 113
7.1 Declarative Software Rewriting . . . 113
7.2 Imperative Software Rewriting . . . 113

7.3 Example guided Software Rewriting . . . 114
7.4 Introducing asynchrony in synchronous code . . . 114
7.5 Translation within Asynchronous Programming Paradigms . . . 115
7.6 Understanding Asynchronous Programming . . . 115
7.7 Debloating and Lazy Loading . . . 116
7.8 Framework-Specific Studies . . . 117
7.9 Analyzing React and its Semantics . . . 117
7.10 UI Optimization . . . 118
7.11 JavaScript Performance Optimization . . . 118
7.12 The React Compiler . . . 119
7.13 Binary Instrumentation . . . 119
7.14 WebAssembly Security . . . 120
8 CONCLUSION 121
8.1 Introducing Asynchrony in JavaScript Applications . . . 121
8.2 Introducing Lazy Loading in JavaScript Applications . . . 121
8.3 Preventing Superfluous Re-rendering in React Applications . . . 122
8.4 Introducing Memory Segmentation to WebAssembly . . . 122
8.5 Discussion . . . 123
8.5.1 Unsoundness in Analysis . . . 123
8.5.2 Towards a Unified Framework for Declarative Software Rewriting . . . 124
8.5.3 Scalability of Transformations . . . 126
8.6 Closing Thoughts . . . 127

BIBLIOGRAPHY 128

LIST OF FIGURES

Figure 1 Example transformation of string to int parser: (a) original code, (b) naive transformation, (c) correct transformation. 2

Figure 2 Example usage of `gzip` using synchronous and asynchronous APIs: (a) uses a synchronous API, (b) uses an event-driven asynchronous API, (c) uses a promise-based asynchronous API. 8

Figure 3 **(a)** Example React application. **(b)** Optimized version where components are re-rendered less often. 14

Figure 4 Excerpt of a server application for compressing a list of files: (a) synchronous version using `gzipSync` (b) promise-based asynchronous version using `gzip` 24

Figure 5 Gzip performance comparison (sync vs async). 25

Figure 6 Example server application that uses synchronous APIs. 26

Figure 7 Refactored version of the server application that uses asynchronous APIs. 27

Figure 8 Performance of server example, comparing response time for synchronous and asynchronous implementations. 29

Figure 9 Example snippet to demonstrate the construction of a transformation set. 31

Figure 10 A rejected transformation involving a function that already returned a promise. 33

Figure 11 Rewrite rules describing the sync-to-async transformation. 34

Figure 12 Excerpt of a client-side application which uses xlsx: (a) version with static import (b) version with dynamic import 46

Figure 13 Transformation rules for introducing lazy loading and necessary code changes to support newly introduced asynchrony. 52

Figure 14 Code showing the before and after of applying select rewrite rules: **(a)-(b)** shows ForEach-ForOf, **(c)-(d)** shows ForEach-Map, **(e)-(f)** shows Await-Map, and **(g)-(h)** shows Getter. 54

Figure 15 Load times for each subject application are depicted in this plot, with a set of three columns for each application. In each set, three times are presented: first, the time taken pre-refactoring (before), then after refactoring (after), and finally the time taken to dynamically load all packages (dynamic). These are averages over 10 runs, and error bars indicate +/- one standard deviation. 57

Figure 16 Re-render comparison before vs. after memoization when the "2" button is clicked with "2*" previously inputted, red indicating components that were re-rendered. 64
Figure 17 Example: unnecessary re-rendering in Calculadora. 65
Figure 18 pattern *State Variables not Affecting the UI* example: **(a)** Uses state. **(b)** Uses reference. 68
Figure 19 pattern *Propless Child Component* example: **(a)** Non-memoized. **(b)** Memoized. 69
Figure 20 pattern *Child Component with Object or Array Element Prop* example: **(a)** Non-memoized. **(b)** Memoized. 70
Figure 21 Rewrite rules describing the remedial code transformations. . . . 72
Figure 22 Illustration of a buffer overflow vulnerability in C. The `setX` function invokes the memory unsafe `strcpy` function that can overflow the character array x in Line 549. Overflowing x can overwrite the pointers on Line 550, or the instruction pointer in traditional binaries. 88
Figure 23 Illustration of a buffer overflow vulnerability in WASM where the `call_indirect` function can be redirected: (a) shows the function `baz` (func 5) using a `call_indirect` instruction to read a function table index at address 1040 in linear memory to invoke `foo()`, (b) shows the exploit code in JavaScript to overflow the buffer at address 1024 in linear memory with " 02" through `setX` (func 4), redirecting control flow from `foo()` to `bar()`. 90
Figure 24 Memory layout of buffer.wasm: (a) Memory Layout of buffer.wasm showing the string `"hello"` at address 1024, the function table index (01) to `foo` at address 1040, and the function table index (02) to `bar` at address 1056. (b) Memory Layout of buffer.wasm after exploiting the vulnerability in `setX` and overflowing the string `"hello"` to overwrite the function table index `01` with `2`. (c) Memory Layout of buffer.wasm after hardening. The read-only function table index `01` is now moved to the immutable segment at address 65536. 91
Figure 25 Illustration of helper functions: **(a)** shows the function readonly_bound_check() that prevents writing to the read-only segment, **(b)** shows the function upper_bound_check() that prevents writing beyond a given address in linear memory. The functions are inserted in **buffer.wat** in Figure 26 on Line 650 and Line 651 respectively. 92

Figure 26 Illustration of security measures implemented to mitigate the buffer overflow in **buffer.wat**: (i) The function readonly_bound_check() from Figure 25(a) is inserted in the binary on Line 650. (ii) The function `upper_bound_check()` from Figure 25(b) is inserted in the binary on Line 651. (iii) The hardened binary uses the functions readonly_bound_check() and upper_bound_check() along with read-only address remapping on Line 646 to mitigate buffer overflows and control flow hijacking. 95
Figure 27 High level architecture of Wassy 97
Figure 28 Rewrite rules describing the transformations for memory segmentation. 101
Figure 29 Rewrite rules describing the transformations for per-instruction bounds enforcement. 103

LIST OF TABLES

Table 1 Subject Applications for evaluation. The first row reads: *the first application used in the evaluation is called* ***deepforge****;* ***deepforge*** *contains* ***34,541*** *lines of code across* ***314*** *files and has* ***7,169*** *functions. This project contains 44 synchronous API calls.* 38

Table 2 Evaluation Results. The first row reads: *For* ***deepforge****, it took* ***222*** *seconds to build a callgraph of size* ***1,020kB****, and it took* ***0.9*** *seconds to transform the project. The analysis detected* ***14*** *synchronous APIs and transformed* ***13****, i.e,* ***93%*** *of them; these transformations required* ***5*** *other changes. The run time of the test suite was* ***7.6*** *seconds before and after applying the transformations.* 39

Table 3 Coverage Results. The first row reads: *The test suite for the* ***deepforge*** *application has* ***49.53%*** *statement coverage,* ***73.41%*** *branch coverage,* ***24.60%*** *function coverage, and* ***49.52%*** *line coverage. The test suite covered all* ***13 (100%)*** *of the* ***transformed synchronous APIs****, and covers all* ***14 (100%) synchronous APIs detected*** *in* ***deepforge****.* 40

Table 4 Information about subject applications. The first row reads: *the first application is called* ***upoint-query-builder*** *from* ***Harinathlee****, and commit hash* `f9aa0f1` *was used for the evaluation;* ***upoint-query-builder*** *has 10,341 lines of code. The initial size of the application is 0.84mb, reduced to 0.61mb after loading modules lazily, corresponding to a 27.4% size reduction. The size of the application once modules are loaded dynamically is 0.84mb. It took 201s to run* Lazifier *on this project, which required an additional 28s to build the CodeQL database.* 55

Table 5 Information about code transformations. The first row reads: *in* **upoint-query-builder***, 2 packages were loaded dynamically instead of statically; 3 dynamic import statements were added, and 12 applications of other rewrite rules were required to support the transition.* 60

Table 6 Overview of results. The first row of the table reads: in 16-feb-react-grupo-1, 0 instances of anti-pattern P1, 1 of P2, 11 of P3, 0 of P4, and 1 of P5 were detected. The number of component renders in the scenario we studied was reduced by 16% after refactoring. Rendering operations took on average 4.83ms before, and 4.45ms after refactoring; an improvement of 7.9%. It took 6s to build the CodeQL database, and 688s to run all queries for this application. The cyclomatic complexity changed from 325 to 338 after refactoring. Bold numbers in % Time Saved indicate a statistically significant difference between performance before and after refactoring. We omit render reductions and time spend rendering for TwitchKillMe as behavioral differences were observed in this application. 78

Table 7 Breakdown of discarded WASM Modules and Memory Trace Collection Results by CWE Category. The first row reads: *The **CWE-121** category contains **2074** WASM modules, **627** of which had to be discarded and **1447** were transformed. During dynamic analysis, **1,445,483** execution traces were captured across the 2074 modules with **696.95** linear memory accesses recorded for each module on average.* 107

Table 8 Breakdown of Hardening effectiveness and Runtime Overhead by CWE Category. The first row reads: *The **CWE-121** category contains **1447** that were transformed. After hardening, **859 (59.36%)** modules presented benign behavior under malicious inputs and a buffer overflow was detected in **10 (00.69%)** modules, both of these cases are considered mitigated. The average execution time for the WASM modules before hardening was **0.3144** seconds and after hardening was **0.3243** seconds indicating an overhead of **3.15%**.* 109

ACRONYMS

API	Application Programming Interface
AST	Abstract Syntax Tree
CWE	Common Weakness Enumeration
SPA	Single Page Application
WASM	WebAsembly
WASI	WebAssembly System Interface
WAT	WebAsembly Text Format
NPM	Node Package Manager
LLM	Large Language Model
DOM	Document Object Model
VDOM	Virtual Document Object Model
IIFE	Immediately Invoked Function Expression
ESM	ECMAScript Module System
JSON	JavaScript Object Notation
HTML	Hypertext Markup Language
XML	Extensible Markup Language
ASLR	Address Space Layout Randomization
CFI	Control Flow Integrity
SFI	Software-Based Fault Isolation
CFG	Control Flow Graph
JSX	JavaScript XML

1 INTRODUCTION

Performance and security are among the most desirable properties of Web Applications. The success of a Web Application depends on user satisfaction and is strongly correlated with the perceived responsiveness of the application and performance. Additionally, security is of utmost importance to retain user trust in a world ridden with malicious actors. However, drawing upon the full potential of a web application can involve the introduction of non-linear control flow, framework-specific changes and optimizations, or extensive refactoring for adoption of newer constructs in an ever-evolving ecosystem. Similarly, mitigating security vulnerabilities is also not trivial. These changes are often intrusive and can span multiple files. Identifying the source locations and the corresponding changes to be applied at those locations often requires complex semantic and syntactic analysis. Thus, there is a strong need to capture the semantics of such transformations effectively so that they may be applied reliably. A declarative approach can decompose these complex transformations into a set of rewrite rules that can succinctly capture a set of semantic predicates and map them to syntactic transformations. In this dissertation, we explore the application of *Declarative Rewrite Rules*, a declarative approach to specify transformations to source code.

1.1 ANATOMY OF SOURCE CODE TRANSFORMATIONS

Real-world software often goes through extensive source code transformations during its lifetime. These transformations may be triggered for a variety of reasons, such as refactoring to improve maintainability, migrating deprecated Application Programming Interfaces (APIs), upgrading dependencies, or addressing performance and security concerns. Regardless of the reason for making code changes, transformations typically follow the same pattern. Based on the reason for the transformation, there exists a primary syntactic code change, or a set of primary syntactic code changes that serve as the focal point of the transformation. These pieces of code may be semantically related to other locations in the program through control or data flow. These secondary locations may or may not require a set of changes, thereby triggering changes to more semantically related locations. Thus, a code change in the program can trigger a set of code changes on a set of transitive dependencies. Interestingly, although the code changes to transitive dependencies are semantically related, they are syntactically independent of each other. This enables us to express code changes as a set of declarative rewrite rules, a set of transformation rules

```
1  function parseDecimalStrToInt(str) {
2
3    if(!isValidNumericString(str)) {
4      throw new Error('NaN: ', str);
5    }
6
7    return parseInt(str);
8
9  }
10
11 parseDecimalStrToInt('10')
12 // produces 10
13
14 const listOfStrings = ['1', '2', '3'];
15 const listOfNumbers = listOfStrings
16   .map(parseDecimalStrToInt);
17 // produces [1, 2, 3]
18
```

(a)

```
19 function parseStrToInt(str, radix=10) {
20
21   if(!isValidNumericString(str)) {
22     throw new Error('NaN: ', str);
23   }
24
25   return parseInt(str, radix);
26
27 }
28
29 parseStrToInt('10');
30 // produces 10
31
32 const listOfStrings = ['1', '2', '3'];
33 const listOfNumbers = listOfStrings
34   .map(parseStrToInt);
35 // produces [1, NaN, NaN]
36
```

(b)

```
37 function parseStrToInt(str, radix=10) {
38
39   if(!isValidNumericString(str)) {
40     throw new Error('NaN: ', str);
41   }
42
43   return parseInt(str, radix);
44
45 }
46
47 parseStrToInt('10');
48 // produces 10
49
50 const listOfStrings = ['1', '2', '3'];
51 const listOfNumbers = listOfStrings
52   .map((v) => parseStrToInt(v));
53 // produces [1, 2, 3]
54
```

(c)

Figure 1: Example transformation of string to int parser: (a) original code, (b) naive transformation, (c) correct transformation.

that capture the semantic relations as the pre-conditions and match on the syntactic type to perform source code transformations.

Figure 1(a) contains an example of a function `parseDecimalStrToInt` defined on Line 1 that accepts a string, performs some validations and returns a number by invoking `parseInt` on Line 7 on the string. If a radix is not provided, the `parseInt` function uses the default radix 10. Line 11 invokes this function on the string `'10'` which produces the result `10`, and Line 16 maps over a list containing `['1', '2', '3']` which produces the result `[1, 2, 3]`.

Let us assume that the function was modified so that it can convert to numbers with a user-provided radix. To achieve this, the following changes are made to the definition of the function `parseDecimalStrToInt`: (i) the function is renamed from `parseDecimalStrToInt` to `parseStrToInt`, (ii) the function accepts a second argument called `radix` with the default value `10`, and (iii) the new argument is passed to `parseInt` as the radix. Figure 1(b) shows the updated function. Since the name and signature of the function have changed, all invocations of this function need to be updated to refer to the new function definition. Naively, since the default radix for `parseInt` is `10`, the second argument can be omitted. Thus, in simple cases, as shown on Line 29, simply renaming the function `parseStrToInt` would suffice. However, performing the same change on Line 34 produces an incorrect result. This is caused by JavaScript implicitly passing the index of the element as the second argument to `parseStrToInt`. When JavaScript maps over the second element in the list `['1', '2', '3']`, it attempts to parse the string `"2"` with the radix `1`. Since the only valid digit in base `1` is `0`, the number is invalid and produces an `NaN` instead. Thus, to correctly transform the code to preserve behavior, we need to introduce an arrow function as shown on Line 52 in Figure 1(c).

Thus, in Figure 1, the change in the definition of function `parseDecimalStrToInt` is the focal point of the transformation. There are two semantically related entities in the form of function calls that resolve to `parseDecimalStrToInt`. The two semantically related entities are present in two syntactic forms, namely, a call expression and an argument to an iterable. Finally, for each syntactic form of the related entity, there is an associated transformation. This set of transformations can be succinctly captured in the form of declarative rewrite rules, as shown in the following section.

1.2 DECLARATIVE REWRITE RULES

The general format for a rewrite rule is as follows:

$$\frac{\textit{preconditions}}{\textit{old code} \;\rightarrow\; \textit{new code}} \quad \text{(Rule-Name)}$$

The consequent of the rule describes the transformation from `old code` into `new code`, found below the bar. The premise of the rule, found above the bar, includes the *preconditions* that must be met for the transformation to take place. The rewrite rules map a set of semantic constraints in the premise to a syntactic transformation in the consequent. Typically, these preconditions represent some facts obtained using static data flow analysis or analysis of the Abstract Syntax Tree (AST). In formulating the rules, we will rely on auxiliary functions of the form **aux_func**(A, e, e'), $e =$ **aux_func**(P), or $\mathrm{P}' =$ **aux_func**(P). The function **aux_func**(A, e, e') takes some AST node A and replaces the occurrences of e with e', where e and e' represent expressions. The function $exprs =$ **aux_func**(P) takes some program body P and returns a set of expressions e. The function $\mathrm{P}' =$ **aux_func**(P) takes some program body P and returns a modified program body P'.

As a concrete example, consider the simple rules Rename-References and Transform-Iterable for the transformations in Figure 1 shown below:

$$\frac{\begin{array}{c}\texttt{f} \text{ is the function definition of } \textit{parseDecimalStrToInt} \\ \texttt{g} \text{ may point to } \texttt{f}\end{array}}{\texttt{parseDecimalStrToInt} \longrightarrow \texttt{parseStrToInt}} \quad \text{(Rename-References)}$$

$$\frac{\begin{array}{c}\texttt{f} \text{ is the function definition of } \textit{parseDecimalStrToInt} \\ \texttt{g} \text{ may point to } \texttt{f} \\ \texttt{iterable} \in \{\text{map, forEach}\}\end{array}}{\texttt{arr.iterable(g)} \longrightarrow \texttt{arr.iterable((v)} \Rightarrow \texttt{g(v))}} \quad \text{(Transform-Iterable)}$$

The rules refer to `f`, which represents the definition of the function `parseDecimalStrToInt`. The rule Rename-References states that for any AST node g that resolves to f, if the identifier of g is *parseDecimalStrToInt* then update the identifier to *parseStrToInt*. This will rename both references in Figure 1(a) on Line 11 and Line 16. The rule Transform-Iterable defines `map` and `forEach` as iterables of interest. The rule Transform-Iterable states that if an AST node g can be resolved to f, such that g is an

argument to an iterable, then replace `arr.iterable(g)` with `arr.iterable((v) => g(v))` where `v` is a new variable. Thus, the rule RENAME-REFERENCES will match all references to the old function and rename them, and the rule TRANSFORM-ITERABLE will handle the edge case of iterables.

1.3 THESIS OVERVIEW

Web applications present many opportunities to improve performance and security. However, the techniques required for performing these optimizations are different based on various factors such as whether the software is running on the server or in the client browser, or based on the framework being used. Although techniques and code transformations may be different for each application, declarative rewrite rules provide a structured approach to describe these techniques. With this research, the aim is to explore declarative software rewriting to optimize web applications.

We explore the following thesis statement:

Declarative software rewriting can be used to automatically improve the performance, responsiveness, and security of Web applications.

Declarative software rewriting is a way to describe the necessary code transformations for some optimization technique as a set of rewrite rules. We develop tools and frameworks that can *automatically* perform the necessary program analysis and transformations described by the rewrite rules. We develop techniques to optimize *performance* of server-side JavaScript applications, *responsiveness* of client-side JavaScript applications, and *security* of WebAssembly binaries and investigate the application of automated tools based on declarative software rewriting to these techniques.

The thesis is organized as follows:

- Chapter 2 presents the necessary background to understand this thesis. This chapter provides an introduction to asynchronous programming in JavaScript and various mechanisms for synchronization. Next, we introduce the various mechanisms for importing external dependencies in JavaScript, followed by an overview of the React framework. Finally, we present a primer on WebAssembly.

- Chapter 3 describes how concurrency can be introduced in server-side JavaScript applications. JavaScript relies on the event loop model, which uses a single thread for the execution of code which may be blocked by synchronous I/O operations. However, asynchronous counterparts of such I/O operations are delegated to a pool of background threads, thereby unblocking the main thread for execution. This work describes a technique for automatically migrating synchronous APIs to their asyn-

chronous counterparts, thereby improving the performance of JavaScript applications.

- Chapter 4 describes the introduction of lazy-loading in client-side JavaScript applications. Page load times are an important metric in web applications that is strongly correlated with the size of the application. Interestingly, many applications contain external dependencies that are not required on the initial load of the application. Loading such dependencies on demand can significantly improve the initial load of the application with negligible downsides. This work describes a technique for introducing lazy-loading in client-side JavaScript applications.

- Chapter 5 describes a set of anti-patterns that lead to superfluous re-rendering in React components, and an automated approach for their remediation. React is a modular, component-based front-end framework that aims to update components only when meaningful changes occur. However, certain coding patterns undermine this goal and cause avoidable computation that can have a negative impact on the responsiveness of the application. This work presents a set of anti-patterns that occur in React applications and describes a technique to remediate them automatically.

- Chapter 6 describes the introduction of memory segmentation and memory safety enforcement to WebAssembly binaries. WebAssembly is a low-level bytecode format that is intended to run in the web browser. However, due to its design and peculiar memory model, WebAssembly binaries can present a set of buffer overflow vulnerabilities. Introducing memory segmentation and enforcing per-instruction bounds on store instructions serve as the first step towards mitigating such vulnerabilities. This works presents a binary rewriting technique to mitigate buffer overflows in WebAssembly binaries.

- Chapter 7 provides an overview of the literature broadly related to this thesis.

- Finally, Chapter 8 concludes this manuscript with a detailed retrospective and a discussion on future directions.

2 BACKGROUND

This chapter provides an introduction to concepts that are relevant to the problems discussed in this thesis. Section 2.1 contains an overview of asynchronous programming constructs in JavaScript and Section 2.2 describes the mechanisms for importing external dependencies. Sec 2.3 and Section 2.4 contain a primer for the React framework and WebAssembly, respectively. Readers familiar with these topics are encouraged to skip this chapter.

2.1 ASYNCHRONOUS PROGRAMMING IN JAVASCRIPT

Nearly all software relies on functions in I/O libraries to interact with file systems, databases, and servers. In JavaScript, library I/O functions can be synchronous or asynchronous. In *synchronous I/O library functions*, execution of the application's source code is suspended until the I/O operation has completed. At this time, the results of the I/O operation are returned as the function's return value. On the other hand, a call to an *asynchronous I/O library function* returns immediately. At a later time, when the I/O operation completes, the application receives notification that the results of the I/O operation have become available. Two mechanisms for providing such notifications are in widespread use:

- In *event-based asynchronous APIs*, a callback function that was passed to the I/O library function is invoked with the result of the I/O operation.
- In *promise-based asynchronous APIs*, the promise [54, Section 25.6] that was returned by the I/O library function is resolved or rejected with the value that represents the results of the I/O operation.

Below, we briefly discuss synchronous and asynchronous I/O libraries using examples, and discuss the tradeoffs of the two mechanisms.

2.1.1 *Synchronous I/O*

Figure 2(a) shows a function `compress` that uses the `gzipSync` function provided by the built-in `zlib` package of Node.js to compress its argument `input`.

```
let zlib = require('zlib');

function compress(input) {
  try {
    let output = zlib.gzipSync(input);
    console.log(output);
  } catch (err) {
    console.log("Error: " + err);
  }
}

compress("example");
```

(a)

```
let zlib = require('zlib');

function compress(input) {
  zlib.gzip(input, (err, output) => {
    if (err) {
      console.log("Error: " + err);
    } else {
      console.log(output);
    }
  })
}

compress("example");
```

(b)

```
let util = require('util');
let gzip = util.promisify(
  require('zlib').gzip
);

async function compress(input) {
  try {
    let output = await gzip(input);
    console.log(output);
  } catch (err) {
    console.log("Error: " + err);
  }
}

compress("example");
```

(c)

Figure 2: Example usage of `gzip` using synchronous and asynchronous APIs: (a) uses a synchronous API, (b) uses an event-driven asynchronous API, (c) uses a promise-based asynchronous API.

On line 66, `compress` is invoked on the string value `"example"`, causing `gzipSync` to be invoked on line 59 to compress this value. Execution of the program will be suspended until the compression operation has completed, at which time a buffer containing the compressed string will be assigned to variable `output`, which is then printed. If an error occurs, the handler on line 62 will execute.

As can be seen in the figure, the use of synchronous APIs such as `gzipSync` leads to code with straightforward control flow that is easy to understand. However, use of such *blocking I/O prevents any event handler from executing*, which, e.g., leads to applications becoming unresponsive while I/O is taking place, which can be problematic.

2.1.2 *Event-Driven Asynchronous I/O*

Figure 2(b) shows a variation on the example of (a) in which an equivalent event-based asynchronous API is used. Here, the function `gzip` is invoked on line 71, to start the compression operation. The second argument is a callback function that is *invoked asynchronously when the I/O operation has completed*. This callback takes two arguments, `err` and output. If the operation was successful, `err` has the value `null` and output contains the result of compressing the input value, which is then printed on line 75. If the operation fails, an error message is printed on line 73.

The use of an asynchronous API such as `gzip` has the advantage that it is *non-blocking*, thus preventing the responsiveness issues triggered by the use of synchronous APIs. However, the use of event-based asynchronous APIs leads to convoluted and error-prone control flow [104] sometimes referred to as "callback hell".

An alternative asynchronous method of performing I/O uses promises and **`async`**/**`await`**, and is depicted in Figure 2(c)—before describing that code, however, we will review JavaScript's promises and **`async`**/**`await`** features.

2.1.3 *Brief Review of Promises and Async/Await*

In recent years, the JavaScript community has rapidly adopted promises as a mechanism for asynchronous programming that is more convenient and less error-prone than event-based programming. Here, we present a very brief summary of promises. For complete details, the reader is referred to the ECMAScript specification [54, Section 25.6], and a formal account of the semantics of promises can be found in [102].

A *promise* represents the value computed by an asynchronous computation, and is one of three states: *pending*, *fulfilled*, or *rejected*. Upon creation, a promise is in the pending state until it is settled (i.e., fulfilled or rejected).

CREATING PROMISES A promise is created by invoking the **`Promise`** constructor, e.g.:

```
let p = new Promise((resolve, reject) => {...})
```

Here, `resolve` and `reject` are functions that must be invoked to resolve or reject the newly created promise, respectively. For example, the following code

```
let p1 = new Promise((resolve, reject) => resolve(17))
```

creates a promise that is fulfilled immediately with the value 17.

REGISTERING REACTIONS ON PROMISES Programmers can register *reactions* on promises, i.e., functions that are invoked asynchronously when a promise is fulfilled or rejected. This is accomplished using the `then` method, e.g., extending the previous example with

```
p1.then( (v) =>  v + 1 )
```

registers a reaction on `p1` that is invoked with its argument `v` bound to the value that `p1` was resolved with (i.e., 17). This reaction will be executed asynchronously and return `v+1`, i.e., 18.

creating promise chains. The `then` method returns a promise that is resolved with its reaction's return value, enabling the construction of *promise chains*. In the following example:

```
let p = new Promise((resolve, reject) => resolve(17));
p.then( (v) => v + 1 )
 .then( (v) => console.log(v) );
```

a promise is created that is fulfilled with the value 17. The reaction registered on this promise returns 18, causing the first invocation of `then` to return a promise that is resolved with that value. On the last line, a reaction is registered on the latter promise, causing `18` to be printed.

Promises also support a method `catch` to facilitate error handling, which is typically used at the end of a promise chain. For example, in the following example

```
let p = new Promise( ... );
p.then( ... )
 .then( ... )
 .catch( (err) => console.log("An error occurred") );
```

the reaction passed to `catch` will be executed if any of the previous promises in the chain are rejected (e.g., if an uncaught exception occurs).

PROMISE LINKING A special case arises when a reaction returns a value that is a promise p. In such cases, the promise p′ returned by `then` or `catch` becomes *linked* with p. Concretely, if p is resolved with a value v, then p′ is resolved with v as well, if p is rejected with value e, then p′ is rejected with e as well, and if p remains pending, so does p′. For example, the following code fragment:

```
let p = Promise.resolve(17);
let q = new Promise((resolve, reject) =>
    setTimeout( () => resolve(18) , 1000)
);
p.then( () => q ) // the promise returned by p.then becomes linked with q
 .then( console.log ); // prints 18 after 1 second
```

creates promises `p` and `q`. When `p` is fulfilled, its reaction is executed and returns `q`, so `q` and the promise returned by `p.then()` become linked. After 1 second, `q` resolves to 18, so the promise returned by `p.then()` resolves to 18 as well, causing the second reaction to execute, which prints this value.

SYNCHRONIZATION USING PROMISE.ALL The `Promise.all` function provides a mechanism for waiting for a set of promises to be fulfilled in no particular order. It takes as an argument an array of promises $p_1, \cdots, p_n$ and returns a promise p′. If each p_i is fulfilled with a value v_i, then p′ is resolved with an array $[v_1, \cdots, v_n]$. If any p_i is rejected, then p′ is immediately rejected with that value, regardless of the disposition of the promises p_j ($j \neq i$).

INTEROPERABILITY WITH ASYNC/AWAIT The `async`/`await` feature provides syntactic sugar on top of promises. A function declared as `async` returns a promise that is fulfilled with the function's return value. Inside an `async` function, `await`-expressions may be used to wait for a promise to be settled. If an awaited promise p is fulfilled with value v, then an expression `await` p evaluates to v; if it is rejected with a value `err`, `err` is thrown as an exception that can be caught using standard `try`/`catch` error handling.

2.1.4 *Promise-Based Asynchronous I/O*

Figure 2(c) shows another version of the example, in which a promise-based version of the asynchronous `gzip` API is used. Here, the `promisify` function from the standard package `util` is used to convert the callback-based `gzip` API into an equivalent promise-based API. *Promisification* is a technique for converting an event-based API into an equivalent promise-based API, and involves the creation of a promise that is fulfilled or rejected when the callback is invoked, depending on whether an error occurred.

The use of a promise-based API enables us to turn `compress` into an `async` function, in which the `await`-expression on line 89 will suspend execution of `compress` until the promise returned by `gzip` is resolved or rejected. Standard `try/catch` syntax can be used for error handling, leading to code that is very similar to that in Figure 2(a). Crucially, the use of the asynchronous `gzip` function does *not* block program execution, allowing other event handlers to execute until the promise returned by `gzip` has been settled.

In summary, promise-based asynchronous APIs share the convenient syntax and support for error handling using `try/catch` with synchronous APIs, while providing the responsiveness benefits of event-based asynchronous APIs.

2.2 IMPORTING PACKAGES IN JAVASCRIPT

Since Chapter 4 is concerned with lazily loading packages, we overview JavaScript's mechanisms for importing packages.

2.2.1 *Require*

The traditional method of including external code in JavaScript is to use `require`, a function that dynamically *and synchronously* loads and executes the package matching the supplied name. Consider:

```
const xlsx = require("xlsx");
function importXLSXData(data) {
  const contents = xlsx.read(data, {...});
  // do stuff with the contents.
}
```

First, the `"xlsx"` package is imported at runtime and saved in the `xlsx` global variable. `"xlsx"` exports a `read` function to convert raw spreadsheet data, and so inside `importXLSXData` the exported function is referenced as a property on the `xlsx` object (`xlsx.read`). Notably, `xlsx` contains *the entire* package code.

2.2.2 *Static Imports*

ECMAScript 6 introduced the static `import` declaration as an alternative to the dynamic `require`. These `import` statements must be at the top level, all bindings must be identifiers, and the package name must be a string literal (this makes them easier to analyze statically); e.g., the statement `import * as xlsx from "xlsx"` imports the entire `"xlsx"` package. A major advantage of static `import` statements is that a developer can specify which parts of

a package they want to import; e.g., in the following snippet, the `read` function exported by `"xlsx"` is imported directly:

```
import { read } from "xlsx";
function importXLSXData(data) {
  const contents = read(data, {...});
  // do stuff with the contents.
}
```

The strict nature of these static import statements allows static analyzers to more effectively determine the extent to which an application exercises the code it imports, which can sometimes lead to smaller distributions—this is called *tree-shaking* [132, 201]. Unfortunately, JavaScript's high degree of dynamism limits the power of these static analyses [84, 95, 128], preventing tree-shaking from removing much code.

2.2.3 *Dynamic Imports*

Static imports are syntactically rigid by design, and so ECMAScript 2020 introduced a dynamic, *asynchronous* `import` function. In a sense, the code transformation proposed in Chapter 4 assists programmers in migrating applications that use static imports to use dynamic imports instead.

The `import` function accepts a string containing the name or path of a package as an argument and returns a promise. That promise can either resolve with an object containing all the exported functions and objects, or be rejected if the package cannot be found. This syntax is especially useful for importing large or rarely used external packages, since they will not be bundled with the rest of the application. This can often result in smaller initial application sizes and potentially faster load times. The following code snippet illustrates how to dynamically import `"xlsx"` only in the context of `importXLSXData`:

```
async function importXLSXData() {
    const xlsx = await import("xlsx");
    const data = xlsx.read(...);
}
```

Note that if a dynamic import for a particular package is encountered more than once, the package is loaded only once, and all subsequent invocations resolve to the same cached instance. Thus, even if `import("xlsx")` or `importXLSXData` is invoked multiple times, the `"xlsx"` package will be loaded only once and served to all subsequent invocations.

2.3 REACT BASICS

React [108] is a powerful JavaScript library for building interactive User Interfaces (UIs) that has been embraced enthusiastically by the web development community. As of Jan-

```jsx
function CounterButton({ message, onClick }) {
  return (
    <div>
      <h1>{message}</h1>
      <button onClick={onClick}>Click me!</button>
    </div>
  );}
function Counter() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState('Guest');
  const handleClick = () => {
    setCount(count + 1);
  };
  const handleNameChange = (e) => {
    setName(e.target.value); };
  return (
    <div>
      <input
        type="text"
        value={name}
        onChange={handleNameChange}
        placeholder="Enter your name" />
      <CounterButton
         message={`Hi ${name}, you clicked ${count} times.`}
        onClick={handleClick} />
    </div>
  );}
```

(a)

```jsx
const CounterButton=React.memo(({message, onClick})=>{
  return (
    <div>
      <h1>{message}</h1>
      <button onClick={onClick}>Click me!</button>
    </div>
  );}
function Counter() {
  const [count, setCount] = useState(0);
  const nameRef = useRef('Guest');
  const handleClick = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, []);


  return (
    <div>
      <input
        type="text"
        ref={nameRef}

        placeholder="Enter your name" />
      <CounterButton
        message={`Hi ${nameRef.current.value}, you clicked ${count} times.`}
        onClick={handleClick} />
    </div>
  );}
```

(b)

Figure 3: **(a)** Example React application. **(b)** Optimized version where components are re-rendered less often.

uary 2025, React is the most popular front-end UI framework, with a usage rate of 64.8% among developers globally [67] and `w3techs.com` reports that 4.7% of all websites use React [162]. Since Chapter 5 is concerned with identifying and remediaitng anti-patterns in React applications, we overview the React framework.

2.3.1 *Components*

A React application comprises a collection of ***components*** corresponding to elements of an application's UI. By structuring UIs as a collection of components, developers can build complex UIs from simple, modular pieces, facilitating reuse. React components can be functions or classes. We only consider functional components, which is currently the preferred mechanism.

2.3.2 *React syntax*

React applications are typically written in JSX[1], a syntactic extension of JavaScript in which XML values may occur as literals[2]. JSX literals can be turned into ***template literals*** that contain ***placeholders*** consisting of a JavaScript expression surrounded by curly braces. At runtime, such embedded expressions are executed to obtain concrete values with which the template is instantiated, resulting in concrete Hypertext Markup Language (HTML) values that a browser can render. An embedded expression that evaluates to an array is turned into a sequence of HTML elements. Syntactically, functional React components are simply JavaScript functions that return a JavaScript XML (JSX) literal. Figure 3(a) shows an example React application consisting of two components, `CounterButton` and `Counter`. The `CounterButton` component consists of a header containing a text message followed by a simple button, which is expressed by the JSX literal on Lines 136-139. A React component may include another React component by simply referring to the latter's name between angular brackets, a mechanism that is referred to as ***JSX tags***. In the example, the `Counter` component defines a UI element that contains a text form in which users can enter their name, followed by a `CounterButton`'s JSX tag, as can be seen in the JSX literal on Lines 150-159.

2.3.3 *State and Props*

React components may have ***state***, i.e., values that persist until explicitly changed or the component is destroyed. State elements are declared by invoking React's ***useState*** function, which takes the initial value as an input and returns the state and a ***state-setter*** function that must be used to change the state. E.g., the `Counter` component maintains `counter` and `name` as its state. On Lines 142-143, these are initialized to the values `0` and `'Guest '`, respectively. The state-setter functions, `setCount` and `setName`, are invoked on lines 145, and 148, respectively.

1 JSX is an acronym for "JavaScript Extensible Markup Language (XML)".
2 React applications can also use TSX, an equivalent extension of TypeScript.

Components may pass values—commonly referred to as ***properties*** or ***props*** in React parlance—to their sub-components (also known as their *child components*) by including them as key-value pairs when referencing the sub-component. For example, `CounterButton` is a child of `Counter`, and on lines 184–186, the `Counter` component passes values to `CounterButton`, binding them to the props `message` and `onClick`.

2.3.4 *Rendering*

Rendering a React component involves executing its code to produce HTML. A component re-renders if its state, or one of its ancestor's state, changes. One of the key strengths of React is its support for incremental re-rendering of an application's UI. The Virtual Document Object Model (VDOM) is central to this efficiency. The VDOM is a simplified, in-memory representation of the actual Document Object Model (DOM). Unlike the DOM, a complex, live structure rendered by the browser, the VDOM is a lightweight abstraction that allows React to perform updates efficiently. When a state change occurs, React creates a new version of the VDOM. This new version is then compared to the last snapshot of the VDOM, a process known as ***diffing***. During diffing, React identifies the exact differences between the old and new VDOMs and computes the minimal set of changes needed to update the actual DOM in a process referred to as ***reconciliation***. This selective update process ensures the browser re-renders only the changed parts of the DOM rather than the entire DOM. This targeted updating enhances performance significantly by reducing the amount of DOM manipulation required, as browser re-rendering is time-consuming.

2.3.5 *State Management*

React applications adopt a centralized state management approach where the nearest common ancestor component maintains the state needed by several sub-components. This ancestor makes the state accessible to its children, passing it as props in their JSX tags. This may include event handlers, allowing child components to update the state also accessed by its parent and/or sibling sub-components. Consequently, any change in the ancestor's state prompts re-rendering of the ancestor and its descendants, ensuring UI consistency across the component hierarchy.

This is demonstrated in Figure 3, where the `Counter` component passes `handleClick` as the `onClick` prop to its child (Figure 3(a), line 158). The `handleClick` function uses `setCount` to update the state of the `Counter` component. When the button is clicked, `onClick` handler of the `<button>` element invokes the `handleClick` function from the `Counter` component. Inside `handleClick`, the `setCount` state-setter is called, causing the `Counter` component to be

re-rendered. As a descendant, the `CounterButton` component also re-renders, ensuring the UI reflects the updated state.

2.3.6 *Hooks*

React's diffing and reconciliation mechanisms aim to ensure efficient updates to the DOM. However, as we shall see in Chapter 5, React may occasionally re-render components unnecessarily, i.e., without causing changes in the DOM. Fortunately, React provides a number of mechanisms for functional components that provide more fine-grained control over the rendering process. These mechanisms are collectively known as ***hooks*** because they enable one to "hook into" React's state and life-cycle management. The `useState` function we saw previously is an example of React hooks and provides access to a component's state. We now discuss a number of other hooks.

- ***useRef*** enables value persistence across rendering operations without triggering a re-render when the stored mutable value is updated. This hook is commonly used to keep a reference to a DOM element or to keep track of mutable data whose change does not require a re-render of the component. It returns a reference object with a *.current* property where the stored value can be accessed or modified.
- ***Memoization*** is a performance optimization technique for preventing unnecessary re-rendering by performing a shallow comparison between the current and new props, and only re-rendering the component if there is a difference. A functional component can be memoized by wrapping it in the `React.memo` function[3]. Since React maintains a copy of the VDOM for comparison, memoization does not incur a significant overhead.
- ***useCallback*** memoizes functions within components: it returns a version of a callback function that only updates when its dependencies (specific states or props that, if altered, necessitate an update) change. This is useful when passing callbacks to child components, specifically those wrapped in `React.memo`, which compares props by reference. By keeping the function reference unchanged across re-renders, `useCallback` helps prevent child components from redundant re-rendering.

Figure 3(b) shows a variation of the example of 3(a) that uses these `useRef`, `useCallback`, and `React.memo` functions. On line 171, `useRef` is used to initialize a variable `nameRef` to "Guest". This reference tracks the user's name from the input element (line 181), updating

3 React.memo is a higher-order component that enhances a functional component by wrapping it. Wrapping in this context means that React.memo takes the original component as an argument and returns a new component with added functionality.

as the input value changes. These updates do not trigger a component re-render on every key press. Instead, the updated value becomes visible only after the `Counter` component re-renders, which occurs upon a button click. Figure 3(b) also illustrates memoization: the child `CounterButton` is wrapped in `React.memo` (line 162), signaling React to bypass re-rendering it if the current and new props are the same. Also, the `handleClick` function is wrapped in `useCallback` (line 172) to ensure that the reference to the `onClick` prop remains the same when its parent component `Counter` re-renders.

Having these memoization steps applied, the `CounterButton` component will not re-render if `Counter` updates without altering `message` or `handleClick`. This example highlights the two different ways of handling form data in React. Figure 3(a) presents an example of a ***controlled component***, where the React component's state handles form data. On the other hand, Figure 3(b) presents an instance of an ***uncontrolled component*** where the DOM handles form data. Section 5.3.2 describes this anti-pattern and the steps involved in remediating it.

2.4 WEBASSEMBLY PRIMER

Since Chapter 6 involves introducing memory segmentation to WebAssembly, this section provides an introduction to the WebAssembly language and its associated ecosystem. Unless otherwise specified, we only refer to WebAssembly 1.0, which is the core language specification for which most WebAssembly compilers produce, and is supported by all major browsers.

WebAsembly (WASM) is a low-level bytecode format designed for computationally-intensive tasks in the web browser. It was developed by a consortium of engineers representing all major browser vendors and first introduced in 2017. WASM aims to provide a secure, fast, portable, and compact alternative to JavaScript as an implementation language for the web. Due to the low-level nature of the language, WASM is typically used as a compilation target as opposed to being written by hand. WASM compilers exist for a variety of source languages such as C/C++, Rust, and Go [171]. The compiled WASM binaries can be used on the client or server by simply linking the binary to JavaScript code [73], or in standalone runtimes such as Wasmtime [169], Wasmer [167], or WasmEdge [166]. Unlike other binaries such as x86, static disassembly for WASM binaries is simple and reliable. WASM binaries can be translated to a human-readable format called WebAsembly Text Format (WAT) and compiled back without any loss. All WebAssembly code examples shown in this dissertation use the WAT syntax.

2.4.1 *Modules*

A WebAssembly binary is organized into modules, and are defined using the *module* instruction as shown below:

```
(module
  (...) ;; types/imports/exports/funcs/table/memory
)
```

WebAssembly modules contain types, functions, and global variables and have an associated linear memory and table section. Additionally, every WebAssembly module can declare a set of imports and exports that facilitate inter-operation with the host environment.

2.4.2 *Types*

WebAssembly is a statically-typed language that supports only four low-level types: 32-bit integers (`i32`), 64-bit integers (`i64`), 32-bit floating point values (`f32`), and 64-bit floating point values (`f64`). All other types present in the source language are expressed using these four types. Types in WebAssembly are defined using the `type` instruction and are identified using numbers. As an example, a type declaration with the identifier 4, for a function that accepts two arguments of `type i32` and produces a result of `type i32` is shown below:

```
(type (;4;) (func (param i32) (param i32) (result i32)))
```

The arguments `param` and `result` are optional and may be omitted if the function has no arguments or does not produce a result, respectively.

WebAssembly bytecode is executed on a stack-based virtual machine (VM) where instructions pop inputs from the stack and push the result to the top of the stack, similar to other stack-based systems. Thus, a function of the above-mentioned `type 4` will consume the first two elements on the execution stack, and upon completion, push one element on the execution stack. The execution stack, global variables, and local variables are all managed by the VM.

2.4.3 *Globals*

WebAssembly modules can declare global variables that can be accessed by all functions. Global variables can be mutable or immutable and can store 32-bit or 64-bit integers and floating point values. They are defined using the `global` instruction and have a numeric identifier similar to types as shown below:

```
(global (;0;) (mut i32) (i32.const 1234))
```

The instruction `global.get` can be used to read the variable and `global.set` can be used to write to the global variable.

2.4.4 *Imports/Exports*

Import statements allow WebAssembly binaries to access and invoke functions implemented in the host environment such as JavaScript. Conversely, export statements allow the host environment to invoke functions implemented in WebAssembly binaries. The import statement maps a named function in the host environment to a function with a numeric identifier and type in WebAssembly that can be invoked by other functions in the binary. The export statements map the numeric identifier from WebAssembly to a string that can be used to access the function in the host environment. The code below contains an import and export statement:

```
(import "env" "print" (func (;0;) (type 1)))
(export "foo" (func 1))
```

A function called `print` in the host environment is being imported into WebAssembly with the identifier `0` and type `1`. The print function can be invoked from WebAssembly using the `call` instruction as `call 0`. Similarly, the function with the identifier *1* is being exported as "foo" to the host environment.

2.4.5 *Functions*

Every WebAssembly module contains a vector of functions, defined using the `func` instruction. Each function has a statically defined type and contains a sequence of instructions that upon completion must produce an execution stack that complies with result defined in the function type. Similarly to types, functions are also labeled with numeric identifiers that can be used to invoke them. The code below shows an example of a function:

```
(func (;8;) (type 3) (param i32) (result i32)
  (...)  ;; sequence of instructions
)
```

Additionally, a function can declare a set of mutable local variables. The function parameters and local variables can be accessed using the `local.get` and `local.set` instructions.

2.4.6 *Linear Memory*

WebAssembly instructions may refer to local or global variables, and they may load values from and store values into *linear memory*, which is a contiguous, mutable array of raw bytes, that is zero-initialized by default, consisting of up to 64K pages. The memory is addressed by 32-bit pointers, and `i32` serves as the pointer type. A WebAssembly program can also request that the VM increase the linear memory via the `memory.grow` instruction. The *memory* instruction is used to define the initial size of the linear memory (as a multiple of 64K) along with the maximum size to which it can grow. The linear memory is zero initialized and can be initialized to non-zero values using the `data` instruction. An example of linear memory with size 256 and the string "hello" at location 1024 is shown below:

```
(memory (;0;) 256 256)
(data (;0;) (i32.const 1024) "hello")
```

The `load` instruction can be used to read data from linear memory whereas the `store` instruction can be used to write to it.

2.4.7 *Indirect Function Calls*

As there are no true function pointers in WebAssembly, we instead have `indirect calls`. The `call_indirect` instruction pops a value from the stack, which is used as an index to look up in the `table` section. The entries in the table map indices to functions, which are subsequently called. Due to management by the VM, we are unable to directly access function memory addresses, instead we must call via table index value. Functions can be referenced multiple times in the table, and not every entry is required to have a corresponding value. Before executing the call, the VM first checks that the target function signature matches the declared function signature in the indirect call instruction, otherwise it raises a function signature error, and halts execution. A function table of size 4 containing references to 3 functions is shown below:

```
(table (;0;) 4 4 funcref)
(elem (;0;) (i32.const 1) func 2 3 1)
```

The function table references begin at index 1, and thus in the above example, index 1 points to function 2, index 2 points to function 3, and index 3 points to function 1.

3 INTRODUCING ASYNCHRONY IN JAVASCRIPT APPLICATIONS

3.1 INTRODUCTION

The JavaScript ecosystem provides equivalent synchronous and asynchronous Application Programming Interfaces (APIs) for many commonly used I/O operations. When faced with a choice, programmers often favor the synchronous APIs because they enable solutions with simple sequential control flow and avoid the complexities of asynchronous programming. However, this greater ease of use comes at a price, as synchronous solutions prevent applications from making progress while the I/O operation is pending, given that JavaScript does not support concurrency at the language level. As a result, the use of synchronous I/O in client-side JavaScript applications may cause user-interfaces to become non-responsive, while on the server-side the use of such APIs may negatively affect both responsiveness and throughput. This chapter explores the development of automated tool support for assisting programmers with the migration from synchronous APIs to equivalent asynchronous APIs.

As an example, consider the function `fs.readFileSync` in the `fs` (file system) package that is part of the Node.js distribution [125]. This function, when given the name of a file, will return a string value representing the file's contents. The operation is *synchronous* in the sense that, when this function is invoked, program execution will block until the read operation has completed. At that time, the function returns a string containing the file contents or throws an exception if an I/O error occurred. The use of blocking synchronous APIs such as `fs.readFileSync` prevents any other event handler from executing, and is considered undesirable because it reduces the application's responsiveness.

To avoid blocking, programmers can use *asynchronous* APIs to perform I/O operations. Two types of asynchronous APIs are commonly used:

- In *event-based asynchronous APIs*, a callback function must be provided that will be invoked upon completion of the I/O operation. For example, the function `fs.readFile` takes a callback as an argument that is invoked when the read operation completes. The callback is invoked asynchronously with two arguments: an error object (or null if no error occurred), and a string containing the file contents if no error occurred.

- In *promise-based asynchronous APIs*, a promise [54, Section 25.6] is returned that is settled upon completion of the I/O operation. For example, the function `fs.promises.`

`readFile` returns a promise that is eventually fulfilled with a string value representing the file contents, or rejected with an error message if an I/O error occurred.

These two approaches are functionally equivalent, but promise-based APIs are rapidly gaining in popularity due to the lower syntactic burden and improved facilities for error handling, especially when used in combination with JavaScript's recent **`async`**/**`await`** feature [54, Section 25.7].

Despite the advantages of asynchronous APIs in terms of improved responsiveness and scalability, many JavaScript applications continue to rely on synchronous APIs, presumably because of their greater ease of use. This work presents a semi-automatic refactoring to help users migrate from synchronous APIs to promise-based asynchronous APIs, in combination with the **`async`**/**`await`** feature. The refactoring replaces synchronous calls with asynchronous calls, and automatically determines when functions need to become **`async`** so that asynchronous calls within their body can be awaited using **`await`**. When functions become **`async`**, functions that invoke them may have to become **`async`** as well, which is determined by inspecting the program's call graph. The suggested refactorings may fail to preserve behavior for several reasons. Most significantly, the call graph may be unsound—nodes and edges may be missing because the analysis is unable to reason precisely about JavaScript's dynamic features. Unsoundness may also arise because, after the refactoring, interleavings will be possible where event handlers are scheduled at times when an introduced asynchronous call is being awaited, and we do not attempt to determine how this may impact program behavior. For these reasons, the proposed transformations are presented as *suggestions* that the programmer needs to inspect carefully and confirm, by running tests and inspecting the code.

We implemented the refactoring in a tool named *Desynchronizer*, which supports 51 different synchronous APIs. In an empirical evaluation, we applied *Desynchronizer* to 12 subject applications containing 316 synchronous API calls. *Desynchronizer* identified 256 of these calls as candidates for refactoring. Of these candidates, 244 were transformed successfully, and only 12 resulted in behavioral changes. Further inspection of these problems revealed that the majority of them can be attributed to unsoundness in the call graph.

In summary, our work makes the following contributions.

1. We present a technique to generate refactoring suggestions to help users migrate their use of synchronous APIs to asynchronous equivalents.

2. We implemented this approach in *Desynchronizer*.

3. We present an empirical study where we apply *Desynchronizer* to 12 applications. In these applications, *Desynchronizer* successfully transformed 244 of the 256 synchronous API calls that were identified as candidates for refactoring. The majority

```
204 const gzipSync = require('zlib').
        gzipSync;
205
206 function compressFiles(files) {
207   const compressedFiles = [];
208   files.forEach(file => {
209     compressedFiles.push(gzipSync(file)
        );
210   });
211   return compressedFiles;
212 }
```

(a)

```
213 let util = require('util');
214 let gzip =
215   util.promisify(require('zlib').gzip);
216
217 async function compressFiles(files) {
218   const cFileProms =
219     files.map(f => gzipPromise(f));
220
221   return await Promise.all(cFileProms);
222 }
```

(b)

Figure 4: Excerpt of a server application for compressing a list of files: (a) synchronous version using `gzipSync` (b) promise-based asynchronous version using `gzip`

of the 12 cases in which *Desynchronizer* did not preserve behavior can be attributed to unsoundness in the static call graphs upon which *Desynchronizer* relies.

A code artifact including *Desynchronizer* is available [68].

This chapter is organized as follows. Section 3.2 presents a motivating example that demonstrates the performance benefits of asynchrony. Section 3.3 describes our static analysis and code transformation techniques and Section 3.4 describes the implementation of *Desynchronizer*. Section 3.5 presents our evaluation. We discuss potential limitations in Section 3.6 before concluding with a summary of our findings in Section 3.7.

3.2 MOTIVATION

3.2.1 *Performance Benefits of Asynchronous APIs*

Asynchronous APIs (both event-based and promise-based) may enable multiple I/O requests to be processed concurrently. To illustrate this point, consider Figure 4, which shows the relevant code fragments of two versions of a server application that performs a compression operation on elements of an array of files. Compression is widely used by server-side web applications, in order to reduce the size (and hence the time needed to transfer) of the responses it sends.

Figure 4(a) shows a version that uses the synchronous API discussed in Chapter 2, Section 2.1.1. Here, the `forEach`-loop on line 208 is used to invoke `gzipSync` on each file in an input array named `files`, storing the result in a newly created array named `compressedFiles`.

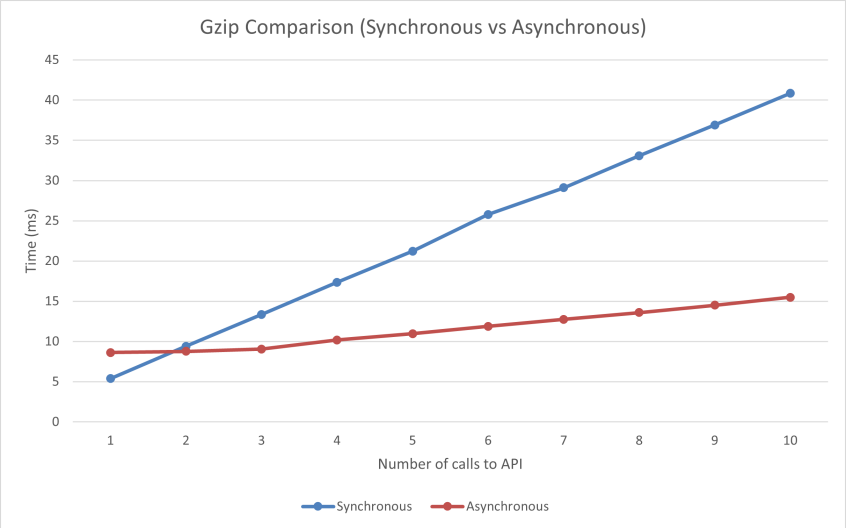


Figure 5: Gzip performance comparison (sync vs async).

Figure 4(b) shows an alternative solution that uses the promise-based asynchronous API discussed in Section 2.1.4. In this version, lines 218–219 create an array of promises that will eventually resolve to the gzipped files. After creating all of these promises, the synchronization operation `Promise.all` is used to wait for all of these promises to be fulfilled.

These two versions of the applications have very different performance characteristics. In version (a), the gzip operations occur in strictly sequential order, with each gzip operation starting after the previous one has finished. In version (b), all `gzip` operations are started at once, without waiting for the previous operation to terminate. Although JavaScript does not have concurrency at the language level, it relies on I/O libraries that take advantage of the fact that modern operating systems are capable of handling multiple I/O requests concurrently.

While this suggests that asynchronous solutions might scale better than sequential ones, we are not aware of published research in which this difference in performance is confirmed experimentally. Therefore, we constructed a set of synthetic benchmarks in which the performance of synchronous APIs such as the one in Figure 4(a) is compared against that of a corresponding asynchronous API such as the one in Figure 4(b). Figure 5 visualizes the results of such an experiment for the `gzipSync` and promisified `gzip` APIs[1]. The figure shows that, if there is only one call to the `gzipSync`/`gzip` API, the synchronous solution requires 5 msec and the asynchronous solution 8 msec. However, as the number of

1 The results for similar experiments involving 10 APIs can be found in the supplemental materials.

```
223  const express = require('express');
224  const fs = require('fs');
225  const zlib = require('zlib');
226
227  const processRequest = (basePath) => {
228    const variants = fs.readdirSync(basePath);
229    return variants.map(variant =>
230      JSON.parse(fs.readFileSync(basePath + variant, 'utf8'))
231    );
232  }
233
234  const app = express();
235  app.get('/product/details', (request, response) => {
236      const data = processRequest(__dirname + '/variants/');
237      response.send(zlib.gzipSync(JSON.stringify(data)));
238  });
239
240  var port = process.env.PORT || 5000;
241  app.listen(port);
```

Figure 6: Example server application that uses synchronous APIs.

API calls increases, the asynchronous solution quickly outperforms the synchronous one (e.g., 40 msec for the synchronous solution and only 15 msec for the asynchronous solution if there are 10 API calls). This experiment was performed on a Windows 10 machine with an i7 processor and 16GB of RAM.

To obtain such performance benefits, programmers should refactor their code to migrate from synchronous to asynchronous APIs. The next section presents such a refactoring.

3.2.2 *Migrating from Synchronous to Asynchronous APIs*

This section explores the challenges associated with refactoring applications so that uses of synchronous APIs are replaced with their promise-based asynchronous counterparts. Here, we will illustrate the issues and review the code transformations involved in the refactoring using a motivating example that represents a small server application[2]. Section 3.3 will present our approach for automating this refactoring.

Figure 6 shows a simple server application that was built using the `express` framework [56] on the Node.js platform [125]. The application responds to requests for product details by invoking the `processRequest` function on line 236. This function obtains a list of all

2 This application is available as part of supplemental materials.

```
244 const express = require('express');
245 const fs = require('fs');
246 const zlib = require('zlib');
247 const util = require('util');
248 const readdir = util.promisify(fs.readdir);
249 const readFile = util.promisify(fs.readFile);
250 const gzip = util.promisify(zlib.gzip);
251
252 const processRequest = async (basePath) => {
253   const variants = await readdir(basePath);
254   return Promise.all(variants.map(async (variant) =>
255     JSON.parse(await readFile(basePath + variant, 'utf8'))
256   ));
257 };
258
259 const app = express();
260 app.get('/product/details', async (request, response) => {
261   const data = await processRequest(__dirname + '/variants/');
262   response.send(await gzip(JSON.stringify(data)));
263 });
264
265 var port = process.env.PORT || 5000;
266 app.listen(port);
```

Figure 7: Refactored version of the server application that uses asynchronous APIs.

files containing descriptions of different variants of the product by calling `readdirSync` on line 228. Then, it relies on the `Array.map` function to read each file by invoking `readFileSync` and calls `JSON.parse` to parse it and create its JavaScript Object Notation (JSON) representation on line 230. The resulting array is converted to a string and then compressed by invoking `gzipSync` on line 237 before being returned to the user.

Note that the use of three synchronous APIs (`readdirSync` on line 228, `readFileSync` on line 230, and `gzipSync` on line 237) will render the server unresponsive while it waits, e.g., for a file to be read.

To improve scalability, the application can be refactored using our *Desynchronizer* tool to use the asynchronous counterparts for these APIs resulting in the code that is shown in Figure 7[3]. The key steps in this refactoring include: replacing calls to synchronous API functions with `await`-expressions that refer to the corresponding asynchronous API function, and changing functions containing migrated API functions into `async` functions,

3 The code shown here has been modified slightly for readability, by changing variable names and eliminating common subexpressions. The original version produced by our tool is included as part of supplemental materials.

as `await`-expressions can only be used inside `async` functions [54, Section 25.7]. In addition, the callers of functions that have become `async` may have to become `async` functions as well, depending on if and how they use the return value of the invoked function; for example, if a function needs to `await` a call to an asynchronous function, it will need to become `async` as well. In cases where API functions are invoked inside loops or in invocations of callbacks passed to `Array.forEach`, we perform a simple analysis to determine whether the loop iterations or callback invocations are independent (i.e., if they operate on different state in each iteration or invocation). If so, we rely on `Promise.all` to enable each iteration or invocation to proceed concurrently. If not, the loop order is preserved and the promises computed in each iteration are awaited in succession.

Concretely, in the case of our motivating example, the source code changes can be summarized as follows:

- The calls to `readdirSync`, `readFileSync`, and `gzipSync` are replaced with `await`ed calls to promisified versions of the library functions `readdir` (line 253), `readFile` (line 255), and `gzip` (line 262), respectively,
- `processRequest` has become an `async` function (lines 252–257), to enable the use of `await`-expressions inside its body.
- Similarly, the callback functions passed to `variants.map` (lines 254–256) and `app.get` (lines 260–263) are made `async` as well, to enable the use of `await` in their bodies.
- Finally, the synchronization function `Promise.all` is used on line 254 to wait until each element of the array of promises returned by the call to `variants.map` has settled.

We constructed a set of synthetic benchmarks to measure the performance of the synchronous implementation shown in Figure 6 and compared the results against the asynchronous implementation shown in Figure 7. The code in Figure 6 loops over user requests one-by-one, and blocks on a file access needed to process the request, whereas the code in Figure 7 can processes requests *concurrently* using promises and `async`/`await`. We measured the performance on a Windows 10 machine with an i7 processor and 16GB of RAM.

We observed that the synchronous implementation of the server performs better for a low number of clients, whereas the asynchronous implementation performs much better when dealing with a high number of concurrent clients. The performance breakdown can be seen in Figure 8.

The next section will present our approach for inferring these transformations.

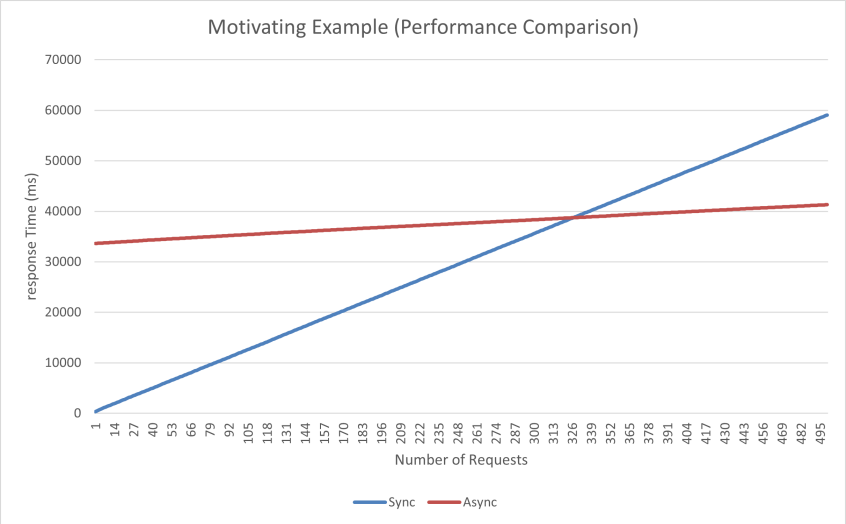


Figure 8: Performance of server example, comparing response time for synchronous and asynchronous implementations.

3.3 APPROACH

We present a three-step approach for automatically determining how to migrate from synchronous APIs to their asynchronous counterparts, consisting of the following steps:

IDENTIFY TRANSFORMATION CANDIDATES The first step of our approach is to identify synchronous API calls that we aim to transform, and determine the extent of the required transformations. Once a synchronous call is identified and made asynchronous, it must be enclosed in an `await` expression to preserve the original semantics of the program: If the original, synchronous call returned a value of type *T*, the now asynchronous call will return a value of *Promise<T>*, and awaiting this call will ensure that the expected value becomes available. In JavaScript, `await` expressions can only appear in `async` functions, so this transformation *propagates* up from callee to caller until the top-level of the program or an already asynchronous function is reached. The result of this step is a *transformation set* of functions that will need to become `async`, and will be discussed in more detail in Section 3.3.1.

DISCARD UNSUPPORTED TRANSFORMATIONS Each transformation set is then checked to ensure that all functions can be transformed, since the transformation to support asynchrony is an all-or-nothing proposition. There are several reasons why such a set may need to be discarded: for example, transformations that would require making a constructor into an `async` function must be abandoned, as construc-

tors cannot be `async` in JavaScript. Section 3.3.2 reviews the preconditions that must hold in order to allow migration to asynchronous APIs.

TRANSFORM SOURCE CODE The last step is to analyze the transformation sets and apply the necessary transformations to the code. These range from simple transformations, such as turning calls to synchronous APIs into `await`-expressions, to more complex transformations, such as changing a map of a (now asynchronous) function into an array of promises that is awaited simultaneously using `Promise.all`. Further details are presented in Section 3.3.3.

3.3.1 *Identify Transformation Candidates*

The first step in the approach is to identify transformation candidates and build a transformation set for each synchronous API call. Algorithm 1 shows how transformation sets are constructed.

Algorithm 1 : BuildTransformationSet

```
   Data : n_sync: a synchronous API call
   Data : CG: the call graph of the program
1  let T := [];
2  let P := [n_sync];
3  while P not empty do
4      let p := pop (P);
5      if p not visited then
6          if p is a reaction or p is argument to promise constructor then
7              continue;
8          let callers := callers of p in CG;
9          for c in callers do
10             if c test runner then
11                 remove c from callers;
12         T := T ∪ callers;
13         P := P ∪ callers;
14         mark p as visited;
15 return T;
```

On line 1, the algorithm initializes a list T, which will contain all functions requiring transformation, and line 2 initializes P, the list of functions that still need processing.

```
267 function inner(listOfFiles) {
268   listOfFiles.map(readFileSync);
269 }
270 function outer(listOfFiles) {
271   inner(listOfFiles);
272 }
273 outer(["a.txt", "b.txt"]);
```

Figure 9: Example snippet to demonstrate the construction of a transformation set.

While there are still functions to process (line 3), a function p is popped from P (line 4). If it has not been visited (line 5), the algorithm identifies all callers of p in the call graph (line 8), and adds them to the list T of functions that would need to be transformed (line 12), and also to the list of functions P that still need to be processed (line 13), as we need to recursively visit their callers. When no additional functions can be found, T is returned.

There are a few situations where the traversal of the call graph can be ended early. The first case we will discuss is depicted on lines 9-11 of the algorithm. If a function c is part of a test harness (like Jest [76] or Mocha [113]), there is no need to transform it or propagate the transformation through it, as they are already equipped to handle asynchrony.

The next case, depicted on lines 6-7, is more subtle. If a function that is flagged for transformation is either an argument to the promise constructor, or is a reaction registered through `then` or `catch`, then the transformation need not be propagated further. Recall that when a function is made `async`, its semantics are altered such that it will now return a promise. In the first case, the promise constructor does not do anything with the return value of the function passed to it, and so no further transformations are necessary. In the second case, reactions registered with `then` and `catch` *already* return promises, and so the promise returned by the newly asynchronous reaction will be *linked* with the promise that is already returned(2.1.3), which will preserve the original semantics.

To illustrate what a transformation set would look like, consider snippet in Figure 9. We draw the reader's attention to the synchronous API call inside of the call to `Array.map` in the body of function `inner`. Since `map` invokes the callback function that is passed to it as an argument[4], our algorithm would add `map` to the transformation set for the call to `readFileSync`. Since `map` is called by `inner`, which is called by `outer`, which is called at the top level of the program, the following transformation set will be computed:

{`fs.readFileSync`, `Array.map`, `inner`, `outer`, program root}.

4 To determine this, we rely on a call graph builder that is equipped with models for external functions such as `map` and `forEach`.

3.3.2 *Discard Unsupported Transformations*

Once transformation sets have been constructed for each synchronous API call, it is necessary to check if the associated transformations are feasible. In general, the transformation of a synchronous API call must be abandoned if any function in its transformation set is called in a context which is flagged as not transformable by Algorithm 2. The cases where transformation sets are abandoned can be found on lines 2-5 and are described below.

Algorithm 2 : IsSetViable

```
   Data : T: a transformation set
1  for function f in T do
2      if f is a constructor or
3      f returns a promise or
4      f unhandled extern
5      f does not support Immediately Invoked Function Expression (IIFE) then
6          return false;
7  return true;
```

CONSTRUCTORS Constructors cannot receive the `async` modifier. This precludes the transformation of synchronous API calls in constructors or in functions called by constructors because `await`-expressions can only be used in the body of `async` functions. While it may be possible to replace constructors with constructs that support asynchrony [25], these would be complex transformations and are beyond the scope for this project.

FUNCTIONS ALREADY RETURNING PROMISES Transformations involving functions that already return promises are also rejected. If these functions were to be made `async`, they would return a promise linked to another promise, and while this is not an issue in and of itself given the behavior of linked promises, the insertion of `await` expressions at call sites to such functions is fraught. To illustrate, consider the code in Figure 10, where the transformation would alter the value stored in `r` (it would no longer be a promise). Determining the return type of a function is difficult in JavaScript due to the dynamism of the language. In the case of this example, our approach infers that the function returns a promise from the fact that reactions are registered on it using `then`, and therefore abandons the transformation.

UNSUPPORTED EXTERNAL FUNCTIONS The technique supports the external functions `Array.map` and `Array.forEach`, as will be discussed in further detail in Section 3.3.3. If

```
let n = "foo.txt";
let q;
function foo() {
  q = fs.readFileSync(n);
  // ...
  return some_promise;
}

let r = foo(); // is a promise
// ...
r.then(() => { ... });
```

(a) Untransformed code.

```
let n = "foo.txt";
let q;
async function foo() {
  q = await fs.promises.readFile(n);
  // ...
  return some_promise;
}

let r = await foo(); // not a promise
// ...
r.then(() => { ... });
```

(b) Transformed code.

Figure 10: A rejected transformation involving a function that already returned a promise.

calls to synchronous APIs occur in (functions invoked by) callbacks passed to other external functions, the transformation is abandoned.

ASYNCHRONOUS CALLS AT THE TOP LEVEL JavaScript only allows the use of `await` inside the body of `async` functions. That said, if a transformation would require an `await`-expression at the top-level of the program, our approach will attempt to introduce an (asynchronous) IIFE. Unfortunately, this is not always possible: ES6 introduced *static* `import` and `export` statements to JavaScript. `import` statements can be safely hoisted outside of the IIFE (as one can only import string literals), but `export` statements cannot be, as they can refer to values computed at run time that will be out-of-scope if removed from the IIFE.

3.3.3 *Transform Source Code*

After eliminating the transformation candidates that do not satisfy the conditions presented in Section 3.3.2, the final step is transformation of the source code. Figure 11 presents these transformations as a set of rewrite rules. Throughout these rules, we make reference to T_f, the set of functions that are to be made asynchronous, obtained by taking the union of all transformation sets that have not been discarded. Note that these rules asbtract away from minor details in JavaScript syntax, e.g., in rule ASYNC-FUNCTION, `fun f(A) {B}` refers to *any* JavaScript function, including arrow functions.

The first two rules, ASYNC-FUNCTION and ASYNC-CALL, are concerned with making functions asynchronous. These transformations simply add the `async` modifier to func-

$$\frac{\texttt{f} \in \mathsf{T_f} \qquad \textit{where f is not async}}{\texttt{fun f(A) \{B\}} \rightarrow \texttt{async fun f(A) \{B\}}} \text{(Async-Function)}$$

$$\frac{\texttt{f} \in \mathsf{T_f} \qquad \textit{where f is not awaited}}{\texttt{f(args)} \rightarrow \texttt{await f(args)}} \text{(Async-Call)}$$

$$\frac{\begin{array}{c}\texttt{f} \in \mathsf{T_f} \qquad \mathrm{B} \text{ body of } \texttt{f} \\ \text{no returns in } B \qquad e = \text{the single argument of } \texttt{f}\end{array}}{\texttt{arr.forEach(f)} \rightarrow \texttt{for([i, e] of arr.entries())\{B\}}} \text{(forEach-forOf)}$$

$$\frac{\begin{array}{c}\texttt{f} \in \mathsf{T_f} \qquad \texttt{f} \text{ equivalent to } \texttt{(v) => e} \\ \texttt{f}_\texttt{A} = \textbf{transform}(\texttt{f})\end{array}}{\texttt{arr.forEach(f)} \rightarrow \texttt{await Promise.all(arr.map(f}_\texttt{A}\texttt{))}} \text{(forEach-map)}$$

$$\frac{\begin{array}{c}\exists\, \texttt{f} \in \mathrm{B} \mid \texttt{f} \in \mathsf{T_f} \qquad \mathrm{B}' = \textbf{transform}(\mathrm{B}) \\ \texttt{f}_\texttt{A} = \texttt{async fun f}_\texttt{A}\texttt{() \{B'\}} \qquad \textit{where p does not return async iife}\end{array}}{\texttt{get p()\{B\}} \rightarrow \texttt{get p()\{return f}_\texttt{A}\texttt{()\}}} \text{(getter)}$$

$$\frac{\begin{array}{c}\texttt{f} \in \mathsf{T_f} \qquad \texttt{f}_\texttt{A} = \textbf{transform}(\texttt{f}) \\ \textit{where map not in Promise.all}\end{array}}{\texttt{arr.map(f)} \rightarrow \texttt{await Promise.all(arr.map(f}_\texttt{A}\texttt{))}} \text{(Promise-All)}$$

$$\frac{\begin{array}{c}\mathrm{P} \in \mathsf{T_f} \qquad \textit{imports} = \textbf{hoistImports}(\mathrm{P}) \\ \mathrm{P}' = \textbf{removeImports}(\mathrm{P})\end{array}}{\mathrm{P} \rightarrow \textit{imports};\ \texttt{(async () => \{P'\})()}} \text{(Top-Level-IIFE)}$$

Figure 11: Rewrite rules describing the sync-to-async transformation.

tions, and turn function calls into `await`-expressions if the function is in T_f. All call sites identified by our call graph as calling a newly asynchronous function are `await`-ed in this way.

Next, ForEach-ForOf defines translation of `Array.forEach` into an `async`-ready `for ... of` loop. If the callback `f` is being made asynchronous, the body B of `f` is found, and the name e of the single argument of `f` is extracted (note: the callback passed to `Array.forEach` must be a single-argument function). Then, a `for ... of` loop is constructed as follows: the loop iterator variables are $[i, e]$ (i for the array index, which is unused, and e, the callback argument name extracted previously), and the loop iterates over the entries of the array. Note that if the callback passed to `Array.forEach` contains a return, we do not perform this transformation; typically, these callbacks do not contain returns as `Array.forEach` does not return anything. The body of the loop is simply the body of the callback, and requires no additional transformation.

Another rule, ForEach-map, depicts a special case of a `forEach` transformation, when the callback passed to `forEach` is an arrow function expression with a single expression as its body. This situation is more elegantly handled by a transformation to `Array.map`. There is nothing in principle preventing a transformation to a `for ... of` loop like above, this is simply more in line with observed language paradigms (programmers often use `Array.forEach` and `Array.map` with arrow functions of this style).

Rule Promise-All depicts a case where `Array.map` is translated into an equivalent construct that properly handles asynchronous function calls. If the callback `f` is being made asynchronous, the required transformation (depicted here with $f_A = \textbf{transform}(f)$) is applied, and the new call to `arr.map`(f_A) is wrapped in an awaited `Promise.all`. This has the effect of waiting until the (now asynchronous) applications of f_A for each of `arr`'s elements completes, semantics similar to that of the original map. The reasoning behind this transformation is that mapping an asynchronous function on an array would result in an array of promises, and awaiting a call to `Promise.all` will wait for all of the promises to settle.

Getter defines the translation of a special case of function definitions. Here, if the body B of some getter `p` contains a function to transform, the body is transformed into B′ and wrapped in a new asynchronous function f_A, and the body of the getter is changed to immediately return a call to that function. Getters in JavaScript are forbidden from being made asynchronous, and so the body of the getter is translated into an anonymous async function which is immediately called and returned. This causes the getter to return a promise, and so calls to the getter can be awaited.

Finally, Top-Level-IIFE depicts the transformation for introducing immediately invoked asynchronous function expressions to the top level of the program. The transformation relies on an external function, **hoistImports**, that simply identifies and collects all of the `import` statements from the program body, P. P′ is the transformed program body with

the `import` statements removed. The transformed body is surrounded by an asynchronous IIFE, which in JavaScript looks like `(async () => {...})()`, which can accommodate `await` expressions in P′.

The motivating example in Figures 6 and 7 shows code examples of transformation rules ASYNC-FUNCTION, ASYNC-CALL, and FOREACH-FOROF.

3.3.4 *Call Graph Construction*

Our technique relies on call graphs to determine functions that transitively invoke methods containing synchronous calls and that may need to be changed into `async` functions. In principle, any call graph construction algorithm can be used for this purpose, allowing for a range of tradeoffs in terms of cost and precision. In practice, the high level of dynamicity in the JavaScript language makes it extremely challenging to compute call graphs soundly while at the same time computing results that are sufficiently precise and scalable to be useful. Very few practical implementations of call graph construction algorithms for JavaScript exist, and we are not aware of any that can easily be applied to the Node.js applications that we consider.

In this project, we aim to use call graphs in an interactive tool that generates refactoring suggestions that need to be reviewed and confirmed by the programmer. In this setting, some unsoundness can be tolerated, but the analysis must complete in a reasonable amount of time. We opted to design and implement a simple underapproximate algorithm, which is based on the approximate call graph construction algorithm by Feldthaus et al. [57]. The algorithm is a constraint-based analysis that simultaneously constructs a data flow graph and a call graph. The data flow graph shows how functions and classes flow from their declaration to expressions that may hold a reference to them. The call graph shows, for each call-expression, what functions may be invoked from that call site. Similar to Feldthaus et al., only the flow of functions and classes is tracked (i.e., the flow of primitive values and objects is ignored). However, our algorithm extends the algorithm by Feldthaus et al. in several signficant ways: it accommodates all features in ECMAScript 2020 [54], such as classes, the ECMAScript Module System (ESM) and many other features that did not exist in JavaScript at the time of the work by Feldthaus et al.

The algorithm is specified as a set of constraint generation rules and implemented using GitHub's CodeQL language [18].

3.4 IMPLEMENTATION

The technique described in this section was implemented in a tool named *Desynchronizer*. The tool is implemented in JavaScript (running on NodeJS v14.15.4), and uses the Ba-

bel [20] parser to build the program AST. The static call graph builder is implemented in version of CodeQL v2.2.4 wherein we modified a library, as by default CodeQL ignores externally defined functions such as the synchronous APIs targeted by our approach, and has little support for externs such as `Array.map` and `Array.forEach`. We modified CodeQL's definitions for external library functions to reflect how they invoke callback functions. The transformations themselves are carried out by replacing subtrees in the AST and using Babel to regenerate the code afterwards. Our code and experiments are available as an artifact [5].

3.5 EVALUATION

In the previous two sections, we have described the approach behind *Desynchronizer*, as well as the implementation of the tool itself. This section reports on an evaluation of *Desynchronizer*, in which we aim to answer the following research questions:

- **RQ1 (Applicability)** How many transformations are proposed by *Desynchronizer*?
- **RQ2 (Soundness)** How often can the unsoundness of proposed transformations be detected by running application test suites?
- **RQ3 (Performance)** What is the impact of refactoring on the performance of applications?
- **RQ4 (Processing Time)** What is the performance of *Desynchronizer*?

We answer these through a series of experiments conducted on 12 subject applications from Node Package Manager (NPM).

3.5.1 *Subject Applications*

Table 1 lists some general information about the projects selected for our evaluation. The first row of the table reads: the `deepforge` project has 34,541 lines of code across 314 files, contains 7,169 functions, and 44 synchronous API calls.

To find these projects, we ran a QL query over some 50k projects that highlighted projects which had calls to synchronous API functions[6] supported by our tool. We designed our query to look for calls to these functions in the main project sources (excluding

5 Link to artifact: https://doi.org/10.5281/zenodo.5502210

6 `readFileSync`, `writeFileSync`, `readdirSync`, `accessSync`, `appendFileSync`, `chmodSync`, `fchmodSync`, `lchmodSync`, `chownSync`, `fchownSync`, `lchownSync`, `mkdirSync`, `mkdtempSync`, `statSync`, `lstatSync`, `fstatSync`, `linkSync`, `symlinkSync`, `readlinkSync`, `realpathSync`, `unlinkSync`, `rmdirSync`, `renameSync`, `openSync`, `closeSync`, `existsSync`, `copyFileSync`, `truncateSync`, `ftruncateSync`, `utimesSync`, `futimesSync`, `fsyncSync`, `writeSync`, `readSync`, `fdatasyncSync`, `gzipSync`, `gunzipSync`, `brotliCompressSync`, `brotliDecompressSync`,

Table 1: Subject Applications for evaluation. The first row reads: *the first application used in the evaluation is called **deepforge**; **deepforge** contains **34,541** lines of code across **314** files and has **7,169** functions. This project contains 44 synchronous API calls.*

Project	LOC	Files	Functions	Sync API Calls
`deepforge`	34,541	314	7,169	44
`meteor-desktop`	13,017	120	1,172	228
`apps`	2,349	26	179	46
`switchBoard`	14,476	268	426	70
`flatsheet`	28,701	32	3,129	20
`bonescript`	12,832	56	410	139
`FiltersCompiler`	3,115	25	174	49
`a11y-metrics`	1,589	28	56	14
`ember-watson`	2,445	142	227	97
`adapt_authoring`	45,354	665	16,573	31
`TurboScript`	34,264	23	2,307	51
`useragent`	8,295	12	95	6

test code, examples, config files, etc.). To determine which of these projects to use in our evaluation, we cloned and built the projects from GitHub and kept those which had tests that ran and passed (note: very few JavaScript projects have test suites, fewer still have test suites where all tests pass). Concretely, the QL query identified 14k of the original 50k projects containing > 5 synchronous API calls, of which several hundred built and had running tests—from these, we selected projects at random.

3.5.2 *Experimental Design*

We ran the call graph builder on each of the projects, choosing the application tests as the entry point for our analysis. In a few cases (`switchBoard` and `flatsheet`), the test code invoked the application code dynamically, which prevented the call graph builder from recognizing application code as being reachable. In these cases, we selected all `.js` files as entry points.

We then ran *Desynchronizer* on each project, applied every transformation that *Desynchronizer* suggested, and reran the application test suite on the transformed application. *Desynchronizer* was configured to log when it abandoned transformations; this, together with the number of synchronous API calls from Table 1, enables us to answer **RQ1**. To

`deflateSync`, `inflateSync`, `deflateRawSync`, `inflateRawSync`, `unzipSync`, `execSync`, `spawnSync`, `execFileSync`, `pbkdf2Sync`, `generateKeyPairSync`, `randomFillSync`, `scryptSync`

answer **RQ2**, instances where *Desynchronizer* introduced failing tests were recorded. We timed the test execution time pre- and post-transformation to help answer **RQ3**. Finally, the entire process was timed, from the beginning of call graph construction to the end of the transformation, to answer **RQ4**.

Experiment Infrastructure

All of our experiments were performed on a server running CentOS Linux release 7.8.2003 (Core), with 2x 32-core processors, running at 2.35GHz. The server is equipped with 128GB of RAM. Timing tests were run on a quiet machine, with no other processes running.

3.5.3 *Experimental Results*

Table 2 summarizes the experimental results and Table 3 summarizes the coverage metrics for synchronous APIs.

Table 2: Evaluation Results. The first row reads: *For **deepforge**, it took **222** seconds to build a call-graph of size **1,020kB**, and it took **0.9** seconds to transform the project. The analysis detected **14** synchronous APIs and transformed **13**, i.e, **93%** of them; these transformations required **5** other changes. The run time of the test suite was **7.6** seconds before and after applying the transformations.*

	Transformation Stats			**Sync Calls**			**Related Fns**	**Test Run Time**	
Project	Build (s)	CG Size (kB)	Trans (s)	Detected	Trans	Trans(%)	Trans.	before	after
`deepforge`	222	1,020	0.9	14	13	93	5	7.6	7.6
`meteor-desktop`	380	1,400	25.1	22	18	82	38	11.6	11.6
`apps`	75	52	0.4	13	8	62	13	10.8	10.5
`switchBoard`	3208	5,400	371.3	65	56	86	28	2.2	2.2
`flatsheet`	236	257	0.5	16	16	100	8	0.5	0.5
`bonescript`	83	160	0.3	11	11	100	6	1.3	1.3
`FiltersCompiler`	114	1,004	15.9	49	22	45	1	19.5	19.4
`a11y-metrics`	73	99	0.2	5	3	60	2	1.1	1.1
`ember-watson`	89	397	0.8	96	96	100	52	1.6	1.6
`adapt_authoring`	1120	1,900	61.4	19	8	42	5	5.8	5.7
`TurboScript`	141	296	0.1	4	3	75	0	50.6	50.8
`useragent`	486	551	0.2	2	2	100	0	0.4	0.4
Total/Average	518.9	1044.6	39.7	316	256	81	158	9.4	9.3

RQ1: How many transformations are proposed by Desynchronizer*?*

Consider Table 2, particularly the columns under **Sync Calls**, which represent the total number of synchronous API calls identified, and the total number that were successfully transformed by *Desynchronizer*. In all, *Desynchronizer* suggested transformations for 256 of 316 synchronous API calls. To make sense of the landscape of abandoned transformations, we broke down the reasons for abandonment by type. In total, 60 transformations were abandoned: 4 were due to a constructor definition being in the transformation set, 5 were due to functions already returning promises, and the remaining 51 were due to calls to unsupported external functions in the transformation set (calls to `readFile`, `writeFile`, `http.server.timeout` , `Array.reduce` and `Array.filter`, and test related functions such as `describe.each` and `describe.skip`).

Answer to RQ1: *Desynchronizer* identifies the majority of synchronous API calls (256 of 316) as candidates for refactoring.

Table 3: Coverage Results. The first row reads: *The test suite for the **deepforge** application has **49.53%** statement coverage, **73.41%** branch coverage, **24.60%** function coverage, and **49.52%** line coverage. The test suite covered all **13 (100%)** of the **transformed synchronous** APIs, and covers all **14 (100%) synchronous** APIs **detected** in **deepforge**.*

	Test Coverage				Sync Function coverage					
Project	Statement	Branch	Function	Line	Trans covered	Trans Total	Trans %	Total covered	Total	Total %
`deepforge`	49.52	73.41	24.60	49.52	13	13	100.0	14	14	100.0
`meteor-desktop`	58.21	50.56	62.41	58.30	05	08	27.77	6	22	27.27
`apps`	50.02	42.86	20.83	50.02	06	08	75.00	10	13	76.92
`switchBoard`	65.15	75.74	52.48	65.15	48	56	85.71	51	65	78.46
`flatsheet`	00.60	00.40	00.90	00.64	00	16	00.00	00	16	00.00
`bonescript`	88.85	71.24	66.67	88.85	11	11	100.0	11	11	100.0
`FiltersCompiler`	90.99	78.44	94.63	90.83	22	22	100.0	47	49	95.91
`a11y-metrics`	56.06	51.47	28.21	55.90	00	03	00.00	00	05	00.00
`ember-watson`	93.76	96.08	90.23	93.76	95	96	98.95	95	96	98.95
`adapt_authoring`	51.82	66.55	55.67	51.82	08	08	100.0	18	19	94.73
`TurboScript`	49.86	34.93	50.81	50.40	00	03	00.00	01	04	25.00
`useragent`	92.32	68.87	66.67	92.32	01	02	50.00	01	02	50.00
Total/Average	62.26	59.21	51.17	62.29	209	256	81.64	254	316	80.37

RQ2: How often can the unsoundness of proposed transformations be detected by running application test suites?

Desynchronizer relies on an unsound static call graph analysis, and the refactorings produced by the tool may not preserve program behavior. To determine how often incorrect refactoring suggestions are offered to the user, we applied every suggested transformation to the code, and ran the application's test suite after transformation to observe if test failures occurred.

We found that in 12 cases, *Desynchronizer* suggested transformations that resulted in behavioral changes. In 9 of these cases, we determined that the problem could be attributed directly to unsoundness in the call graph. In 3 cases, the problem related to the test infrastructure (in particular, two cases because `async`/`await` cannot be used in conjunction with the `done` function in the test framework, and one case where a mock function needed updating). While this is promising, we must emphasize that, in general, running an application's tests may not reveal all such incorrect transformations. To provide a fuller picture, we have included the coverage of the application test suites in Table 3, which quantify how much of an application's code is exercised by the test suite—e.g., 80% statement coverage means that 80% of an application's statements are exercised by the tests. On average, we found 62.26% statement coverage, 59.21% branch coverage, 51.17% function coverage, and 62.29% line coverage. Further, we found that 254 (80.37%) of the identified 316 synchronous API calls were covered. Overall, we think this is reasonable coverage.

Answer to RQ2: Only 12 out of 256 transformations resulted in behavioral changes detectable by running an application's tests.

RQ3: What is the impact of refactoring on the performance of applications?

Translation to asynchronous APIs may result in improved application performance by enabling the application to take advantage of the underlying operating system to process multiple I/O requests concurrently. As is suggested in Figure 5, asynchronous APIs outperform their synchronous counterparts if many simultaneous requests are processed, though a slowdown may occur if there is no concurrency. To measure the effect of our transformation on the performance of applications, we collected the run times of the test suites before and after transformation. Table 2 summarizes these results in the **Test Run Time** columns labeled 'before' and 'after'. In each case, the original and transformed code had similar execution times. While the code transformations do not improve test suite run times in a meaningful way, we note that none of these tests exercised the application *at scale* by processing many concurrent requests.

Answer to RQ3: The transformations suggested by *Desynchronizer* do not negatively impact the run time of an application's tests.

RQ4: What is the performance of Desynchronizer*?*

To assess the usability of *Desynchronizer*, we measured how long it took to run it on the applications we selected for our evaluation. The results can again be found in Table 2, under the **Transformation Stats** columns (see the 'Build' and 'Refactoring' subcolumns). The build time includes the time it takes to build the QL database, which is required by QL to execute its queries.

We see that *Desynchronizer* generates the call graph in < 2 minutes for over half of the projects, while in one case the call graph is generated in over 50 minutes. The performance varies highly from one application to another, due to the nature of our call graph building algorithm. `adapt_authoring` is a reasonably large project with 665 files and over 16,000 functions to be analyzed. `switchBoard` on the other hand had poor performance due to the choice of entry point resulting in an exceedingly large number of relevant files.

The time to actually build transformation chains and refactor a project is low on all projects save for `switchBoard`. Longer refactoring times correlate with the larger number of related functions that need to be transformed.

Answer to RQ4: The performance of *Desynchronizer* is mostly determined by the cost of call graph construction, which is reasonable in most cases.

3.6 THREATS TO VALIDITY

The main threat to validity is the fact that the transformations proposed by our technique may not preserve behavior. Loss of soundness may occur for various reasons, chiefly (i) the call graphs that our technique relies on may be unsound, and (ii) replacing synchronous calls with asynchronous calls may enable arbitrary event handlers to be interleaved when the de-synchronized call is awaited. We view such unsoundness as inevitable, given the highly dynamic nature of JavaScript. We have therefore expressly opted to focus on the development of a technique that is practical, and our evaluation reports that refactorings that are not behavior-preserving are detected only in a small fraction of all cases.

Beyond concerns about soundness, it is possible that the set of subject applications is not representative of JavaScript projects at large. However, the projects were chosen at random from a pool of thousands of JavaScript projects that had synchronous API calls. This pool was pruned to include only projects with a test suite in which every test passed.

We believe that we have mitigated the risk of bias in our selection process through our random project selection.

To start call graph generation, we manually determined entry points for each application, and in nearly all cases this consisted of the application's tests. We assume that the tests are well written and cover the majority of the application functionality. A sub-optimal test suite could result in the failure of *Desynchronizer* to identify some synchronous API or potential semantics changes caused by the refactoring. This has the potential of negatively impacting our evaluation results, though as seen in Table 2, *Desynchronizer* transforms the majority of synchronous functions, and in Table 3 we see that most of our subject applications have good test coverage.

3.7 CONCLUSION

The JavaScript libraries provide synchronous and asynchronous APIs for many commonly used I/O operations. While the asynchronous mechanisms are preferable because they enable better responsiveness and performance in applications, programmers often opt for the synchronous APIs because of their greater ease of use. We have presented a technique that analyzes an application that uses synchronous APIs and automatically infers how it can be refactored to use asynchronous APIs instead. The technique relies on a static call graph analysis that is unsound, so the inferred refactorings are presented as suggestions that should be reviewed by the programmer.

We implemented the refactoring in a tool called *Desynchronizer*, which we evaluated on 12 subject applications containing 316 synchronous API calls. *Desynchronizer* identified 256 of these as candidates for refactoring. Of these candidates, 244 were transformed successfully, and only 12 resulted in behavioral changes. Further inspection of these cases revealed that the majority of these issues can be attributed to unsoundness in the call graph.

3.8 PUBLICATION STATUS

This work was published in *Proceedings of the ACM on Programming Languages, volume 5, issue OOPSLA* [69]. It was presented at OOPSLA 2021. The artifact for this work was evaluated and awarded the "Available", "Functional", and "Reusable" badges and is available at `https://doi.org/10.5281/zenodo.5502210`.

4 INTRODUCING LAZY LOADING IN JAVASCRIPT APPLICATIONS

4.1 INTRODUCTION

In web application development, one of the key objectives is reducing the time it takes for an application to load and become responsive [34, 65, 98, 100]. To achieve this, developers typically strive to minimize the size of their application bundles, using tools such as bundlers, minifiers, and tree-shakers [49, 132, 196, 201]. Unfortunately, these tools have limited effectiveness when the application includes functionality that may be needed later but not at application startup. In such cases, performance can be enhanced by asynchronously loading the relevant code only when it is first required.

In this work, we present a method for automatically refactoring applications to introduce lazy loading. Our approach focuses on a specific scenario in which the functionality to be lazily loaded is the part of a third-party library imported by the application. Using static analysis, we first identify packages that are exclusively used within event-handling code, as these are likely not needed at startup and can be loaded conditionally. For each identified package, a second static analysis determines the scope of code changes necessary to support asynchronous lazy loading. The final transformation is guided by a set of declarative rewrite rules that define the required code changes.

We implemented this approach in a tool called *Lazifier* that targets the JavaScript programming language (ECMAScript 2021). Similar to recent other refactoring tools [16, 69, 152], *Lazifier* employs unsound static analysis, so the proposed code transformations are presented as *suggestions* that programmers should review and test carefully before applying. In an experimental evaluation on 10 open-source client-side JavaScript applications, the code transformations proposed by *Lazifier* resulted in an average initial application size reduction of 36.2%, which caused applications to speed up initial load time by 29.7% on average. Furthermore, we find that loading the packages that have been removed from the bundle lazily when required incurs little overhead. Finally, despite the potential for unsoundness in the static analysis, we found that no unwanted behavioral differences were caused by the transformations proposed by *Lazifier* for the 10 subject applications in out evaluation.

In summary, our work makes the following contributions:

- We present an automated technique for identifying packages that are potential lazy loading candidates, and a set of rewrite rules specifying the changes required to load those packages lazily;
- We implemented this approach in a tool called *Lazifier*, that targets the JavaScript programming language;
- We present an empirical evaluation where we apply *Lazifier* on 10 applications. In these applications, *Lazifier* reduces the initial application size (36.2%, on average) and load-time significantly (29.7%, on average) with little overhead associated with dynamic loading.

A code artifact including *Lazifier* is available [155].

This chapter is organized as follows. The problem is motivated in Section 4.2, the approach is described in-depth in Section 4.3 followed by an overview of the implementation of our tool, *Lazifier*, in Section 4.4. Section 4.5 presents the evaluation of *Lazifier*, and threats to validity are discussed in Section 4.6, before concluding in Section 4.7.

4.2 MOTIVATION

To illustrate our approach, consider an open-source JavaScript application that displays a list of recent movies to users, complete with information about them (**Movies-web-ui** [7]). The application lets users filter the list of movies and, optionally, export their filtered selection along with information about the movies they are interested as a spreadsheet. The code snippet in Fig 12**(a)** is taken directly from **Movies-web-ui**, and shows the implementation of an "export" button and its functionality. Note that this application relies on a few external packages: `React`, an extremely popular UI framework for JavaScript, `file-saver` [184] a package used for saving files, and `xlsx` [139] a library for handling spreadsheet-like data. The file exports a function `exportCSV` that creates a JSX [1] button component (lines 315-319). The `"click"` event handler associated with this button (lines 316-317) eventually calls the `exportToCSV` function (lines 305-313), which leverages the `xlsx` package to convert a JSON object containing the user's selection to a sheet (line 308), and saves the selection to a file using `file-saver` (line 312).

Crucially, in this example, the `xlsx` and `file-saver` packages are not required if a user only wishes to browse the list of movies. These packages are only needed to implement the export functionality. It should also be noted that the references to these packages on lines 308, 310, and 312 are the only references to these packages in the entire application.

1 JSX is a type provided by React that closely matches HTML, allowing programmers to easily construct HTML-like objects in their JavaScript code.

```jsx
import React from 'react';
import * as fileSaver from 'file-saver';
import * as xlsx from 'xlsx';

export const exportCSV = ({csvData,
    fileName}) => {
  const fileType = '...';
  const fileExtension = '.xlsx';
  const exportToCSV = (csvData, fileName
    ) => {


    const ws = xlsx.utils.json_to_sheet(
    csvData);
    const wb = {Sheets: {...},
    SheetNames: [...]};
    const buffer = xlsx.write(wb, {...})
    ;
    const data = new Blob([buffer], {
    type: fileType});
    fileSaver.saveAs(data, fileName +
    fileExtension);
  }
  return (
    <button className="export"
      onClick={(e) =>
        exportToCSV(csvData,fileName)}>
      Export
    </button>
  )
}
```

(a)

```jsx
import React from 'react';
// this import was removed
// this import was removed

export const exportCSV = ({csvData,
    fileName}) => {
  const fileType = '...';
  const fileExtension = '.xlsx';
  const exportToCSV = async (csvData,
    fileName) => {
    const fileSaver = await
    import('file-saver');
    const xlsx = await import('xlsx');
    const ws = xlsx.utils.json_to_sheet(
    csvData);
    const wb = {Sheets: {...},
    SheetNames: [...]};
    const buffer = xlsx.write(wb, {...})
    ;
    const data = new Blob([buffer], {
    type: fileType});
    fileSaver.saveAs(data, fileName +
    fileExtension);
  }
  return (
    <button className="export"
      onClick={async (e) =>
        await exportToCSV(csvData,
    fileName)}>
      Export
    </button>
  )
}
```

(b)

Figure 12: Excerpt of a client-side application which uses xlsx: (a) version with static import (b) version with dynamic import

In such cases, loading the packages lazily ensures that we do not incur the overhead of loading unnecessary packages whose functionality will not be used by the users. The code snippet in Fig 12**(b)** depicts how this can be achieved, and code changes are highlighted. First, note the lack of static imports to `xlsx` and `file-saver`, and the inclusion of dynamic imports to the packages instead (lines 330-331).

The call `import('file-saver')` on line 330 creates a promise. This promise gets resolved with an object representing the `file-saver` package. Once the package has been loaded, the `await` on the same line ensures that this object can be assigned to the local variable `fileSaver`. Recall that `await` expressions are only allowed in the context of `async` functions, so the `exportToCSV` function must gain the `async` keyword (line 329). This changes the return type of `exportToCSV` to $\mathit{Promise}\langle \mathrm{JSX} \rangle$, so all call sites to this function should be `await`-ed to ensure that application behavior remains unchanged. In particular, an `await` is added at the call to `exportToCSV` on line 341. This new `await` requires the surrounding function to be made `async` as well (line 340), at which point we have reached a context that implicitly handles asynchrony: callbacks that serve as event handlers are not expected to return anything, so no further transformations are required once they are made `async`.

Since `xlsx` and `file-saver` are fairly large packages, this simple refactoring results in the reduction of code that is loaded by 30% (from 1.4mb to 0.96mb), and improves the initial load time of the application by nearly 50% (from 517ms to 286ms, averaged over 10 runs). If the user chooses to export their selection, these packages need to be loaded on demand. Packages requested on demand load rather quickly (0.11s), and the total amount of code loaded by the application is 1.4mb, i.e., the same as the original size.

The above example skips over the details of a few additional complexities associated with loading packages lazily. For instance, when making a function `async`, *all* call sites to the function must be `await`-ed no matter where they are, as making a function `async` causes it to return a `Promise`. This can cause a cascade of transformations that may not be localized to a single file. Further, certain code patterns need to be modified to accommodate `async` functions (e.g., the expression `someArray.forEach(f)` is blocking if the callback `f` is synchronous, but non-blocking if `f` is `async`).

In this work, we present a technique to automatically detect third-party dependencies that are only used in the context of event-handlers, and automatically transform the application to load those dependencies lazily. In the next section, we describe this technique in detail, and describe how the aforementioned complexities are handled by our approach.

4.3 APPROACH

Our approach for automatically refactoring applications to introduce lazy loading consists of the following three steps:

1. *Determine packages that are only used in the context of event handlers;*
2. *Confirm which of these can be loaded lazily, and identify the code transformations required;*
3. *Enact the transformation.*

For (1), we propose a fully automated static analysis to detect which packages are *only* used in the context of event handling code and that therefore are not initially needed by the application. For (2), another static analysis determines the transitive dependencies of the lazy loading candidate, i.e., all of the functions containing references to a given lazy loading candidate. Since web browsers only support asynchronous loading of code and assets, each lazy loading candidate will require a dynamic, *asynchronous* import of the package. This asynchronous import requires several other code transformations to support the newly introduced asynchrony. If any of these transformations are not possible, the lazy loading candidate is discarded. Finally, for (3) we propose a set of declarative rewrite rules describing the code changes required to refactor the application to lazily load the package. Each of these phases is described in turn.

Soundness. We assume that the static analyses used in steps 1) and 2) are potentially *unsound*. Sound, precise, and scalable static analysis of JavaScript is well beyond the state-of-the-art due to the dynamism inherent to the language [84, 95, 128]. This means that the transformations proposed by the approach may not preserve behavior, and should be carefully reviewed by a programmer, similar to the approach taken by other refactoring tools for JavaScript [16, 69, 152]. Section 4.5 investigates the presence of any unwanted behavioral differences caused by this unsoundness.

4.3.1 *Identify Candidate Packages for Lazy Loading*

We provide a fully-automated analysis to identify packages that should be loaded lazily and are only used in the context of event-handling code. The analysis identifies functions that are supplied to event-handling mechanisms (e.g., registered as "on-click" attributes of HTML elements, or registered as event listeners) based on a call graph for the application and determines all of the functions that are (transitively) called from those handlers. If *all* references to a package are in this list of functions, then it is flagged as being a candidate for lazy loading. This list of event handlers is as follows:

- functions passed to `onClick` or other `on` or `click` events on JSX and HTML components, including functions identified using string representations of their name;
- any code snippets included in an event handler attribute (e.g., code in the `onClick` event of an HTML element);

- functions passed as callback arguments to event handlers (e.g., `reader.on('load', callback)`);
- functions assigned to properties of the `window` object that represents the DOM.

4.3.2 *Validate and Determine Transformations Required*

To successfully load a package p lazily, all static imports to p must be removed, and functions containing references to p must be refactored to load the package dynamically. This involves removing static `import ... from 'p'` statements and inserting dynamic `import('p')` expressions where appropriate. The expression `import('p')` yields a promise that eventually resolves with the content of the package `'p'`. While that promise is pending, the current context that depends on the package should not proceed. This is achieved by `await`-ing that call, which suspends execution until the promise is resolved. Then, we may assign the `await`-ed import to a variable (e.g., `let x = await import('p')`), which results in the the package itself being stored in `x` and execution can resume.

Now, `await` expressions are only allowed inside of functions marked as `async`, but making a function `async` changes its return type to $\texttt{Promise}\langle \texttt{T} \rangle$, where T is the function's original return type. To preserve existing application behavior, all call sites to this function will need to be `await`-ed, which itself requires more functions to be made `async` and more call sites to be `await`-ed, and so on. It is imperative that *all* call sites to newly `async` functions be `await`-ed. Failing to do so will result in unexpected program behavior. As a result, the transformations proposed are an *all or nothing proposition*, and if any call sites cannot be `await`-ed, the entire transformation needs to be abandoned, and p must be discarded from the set of lazy loading candidates.

Algorithm 3 describes the process of creating the set S_{async} of functions needing to be made `async` while validating the transformation. As inputs to the algorithm, the package p is supplied along with the call graph CG of the program. First, S_{async} is initialized as the empty set (line 1), and the list F of functions yet to be processed is initialized with all functions containing references to the package p (line 2). The main loop (lines 3-15) iterates through functions $f \in F$ that have not yet been visited. First, lines 6-8 describes a special case where a function to be made asynchronous is already in a context that handles asynchrony, in which case no further transformations are required. Then, all callers of the function f are obtained from the call graph (line 9). Lines 10-12 *validates* the transformation by identifying situations that cannot support asynchrony. First, constructors cannot be `async`. Second, if f is called at the top level of the application, there is no sense in lazily loading p as the dynamic import would be executed on application startup anyway. (Also, top-level `await` expressions are only supported as of ECMAScript 2022, which may not be supported by older browsers.) Third, if f already returns a promise, the programmer is

Algorithm 3 : Validating p and building S_{async}

```
   Data : p: a package being imported dynamically
   Data : CG: the call graph of the program
1  let S_async := {};
2  let F := [functions referencing p];
3  while F not empty do
4      let f := select and remove a function from F;
5      if f not visited then
6          if f is a reaction or f is argument to promise constructor or f registered as event
            handler then
7              S_async := S_async ∪ {f};
8              continue;
9          let C_f := callers of f in CG;
10         if f is constructor or c ∈ C_f is top level or f returns promise then
11             S_async := {};
12             break;
13         S_async := S_async ∪ {f};
14         F := F ∪ C_f;
15         mark f as visited;
16 return S_async;
```

likely using it accordingly and may not want calls to it to be `await`-ed, and so it should not be transformed. In such cases, the transformation is rejected and p is not loaded lazily. If f passes this check, then f is added to S_{async}, all of f's callers are added to the list F of functions left to process, and f is marked as visited; analysis continues until F is exhausted.

4.3.3 *Code Transformations*

The application can be refactored to lazily load package p once the set S_{async} of functions that need to be made `async` is known. Several transformations are required to handle the transition to asynchronous imports, specified as declarative rewrite rules in Figure 13. The figure depicts simplified, idealized JavaScript to illustrate the salient details of the transformation. We will describe them one by one next.

ASYNC-FUNCTION: This transformation is simple: if a function `f` is in the set S_{async} of functions that need to be made `async`, the function definition gains the `async` keyword.

ASYNC-CALL: All potential calls to a function `f` $\in S_{async}$ need to have `await` expressions inserted before the call.

FOREACH-FOROF: The expression `arr.forEach(f)` calls the callback `f` on each element of `arr`, and importantly *returns nothing*, i.e., `forEach` is type `void`. If `f` were made asynchronous, the call to `forEach` would not wait for all of the asynchronous calls to resolve, and execution would simply continue past the call. In the event that `f` contains no `return` statements, the body `B` of `f` is made into the body of a `for ... of` loop that iterates over the elements of the array (the loop iterator `a` is chosen to match the argument name of `f`).

FOREACH-MAP: In the event that `f` *does* contain a `return` statement, conversion to a `for ... of` loop is not possible. Instead, the `forEach` is transformed into a `map`, and the call to `map` is surrounded in an `await`-ed `Promise.all` to ensure that all of the asynchronous callbacks fully execute before continuing.

AWAIT-MAP: Similar to the previous rule, if a callback passed to `map` is to be made asynchronous, the `map` is surrounded in an `await`-ed `Promise.all`.

INSERT-DYNAMIC-IMPORT: If a function `f` contains references $(v_0, ..., v_n)$ to a package p that is to be made dynamic ($p \in P_D$), a dynamic import to the package p is created (`const` p_{name} = `await import(p)`), where p_{name} will serve as a reference to the package in this scope. Then, declarations are created for each $v_k \in v_0, ..., v_n$ extracting the relevant component v_k^{name} from the import p_{name}. The dynamic import and associated declarations are then inserted at the beginning of the function body.

$$\frac{\texttt{f} \in S_{async} \qquad \textit{where } \texttt{f} \textit{ is not } \texttt{async}}{\texttt{fun f(A) \{B\}} \longrightarrow \texttt{async fun f(A) \{B\}}} \quad \text{(Async-Function)}$$

$$\frac{\begin{array}{c}\texttt{f} \in S_{async} \qquad \texttt{g} \text{ can resolve to } \texttt{f} \\ \textit{where } \texttt{g} \textit{ is not } \texttt{awaited}\end{array}}{\texttt{g(args)} \longrightarrow \texttt{await g(args)}} \quad \text{(Async-Call)}$$

$$\frac{\begin{array}{c}\texttt{f} \in S_{async} \qquad B \text{ body of } \texttt{f} \\ \text{no returns in } B \qquad a = \text{the single argument of } \texttt{f}\end{array}}{\texttt{arr.forEach(f)} \longrightarrow \texttt{for([i, a] of arr.entries()) \{B\}}} \quad \text{(ForEach-ForOf)}$$

$$\frac{\begin{array}{c}\texttt{f} \in S_{async} \qquad B \text{ body of } \texttt{f} \\ \text{returns in } B\end{array}}{\texttt{arr.forEach(f)} \longrightarrow \texttt{await Promise.all(arr.map(f))}} \quad \text{(ForEach-Map)}$$

$$\frac{\begin{array}{c}\texttt{f} \in S_{async} \\ \textit{where } \texttt{map} \textit{ is not in } \texttt{Promise.all}\end{array}}{\texttt{arr.map(f)} \longrightarrow \texttt{await Promise.all(arr.map(f))}} \quad \text{(Await-Map)}$$

$$\frac{\begin{array}{c}p \in P_D \qquad v_0, ..., v_n \textbf{ ref } p \in \texttt{B} \\ \texttt{dynImp} := \texttt{const } p_{name} = \texttt{await import(p)} \\ \texttt{decl}_k := \texttt{const } v_k = p.v_k^{name} \quad \forall k \in 0, ..., n\end{array}}{\texttt{fun f(A) \{B\}} \longrightarrow \texttt{fun f(A) \{dynImp; decl}_0\texttt{; ... decl}_n\texttt{; B\}}} \quad \text{(Insert-Dynamic-Import)}$$

$$\frac{\begin{array}{c}\texttt{x} \in S_{async} \qquad \texttt{f}_\texttt{B} := \texttt{async } () \Rightarrow \texttt{\{B\}} \\ \textit{where } \texttt{x} \textit{ does not return } \texttt{async iife}\end{array}}{\texttt{get x() \{B\}} \longrightarrow \texttt{get x() \{return f}_\texttt{B}\texttt{();\}}} \quad \text{(Getter)}$$

Figure 13: Transformation rules for introducing lazy loading and necessary code changes to support newly introduced asynchrony.

GETTER: Getters present a special case as they cannot be made asynchronous. A new asynchronous function `f_B` is created with the body `B` of the getter `x`. The body of `x` is then replaced with a return to the call to `f_B`—callers of `x` will `await` calls to it, and so the promise returned by `f_B` can be `await`-ed then.

The code transformation in the motivating example was determined automatically using this approach, and involved applications of rules ASYNC-FUNCTION, ASYNC-CALL, and INSERT-DYNAMIC-IMPORT. Fig. 14 shows small code examples depicting the transformations associated with the other rules: Fig. 14**(a)** and **(b)** shows rule FOREACH-FOROF, Fig. 14**(c)** and **(d)** shows rule FOREACH-MAP, Fig. 14**(e)** and **(f)** shows rule AWAIT-MAP, and finally Fig. 14**(g)** and **(h)** shows rule GETTER.

4.4 IMPLEMENTATION

This approach is realized in a tool named *Lazifier*. All static analyses, including the data flow analyses used to detect imported package usage and construct call graphs, are implemented using CodeQL [111]. The call graphs are generated with CodeQL's built-in static call graph construction algorithm for JavaScript [110], though it is known to be unsound. Code transformations are implemented in JavaScript, utilizing Babel [19] for parsing source code, manipulating ASTs, and generating the transformed output. Our code and experiments are available as an artifact [2].

4.5 EVALUATION

We pose the following research questions in order to evaluate the approach proposed in this chapter:

RQ1. How does lazy loading affect the size and initial load time of applications?

RQ2. How often does the transformation introduce unwanted behavioral changes?

RQ3. How much code is loaded lazily, and how quickly is it loaded?

RQ4. How many code changes are required to support lazy loading?

RQ5. What is the running time of *Lazifier*?

2 Link to artifact: https://doi.org/10.5281/zenodo.8264271

```
arr.forEach((e) => {
  if (e)
    foo();
  else
    bar();
});
```

(a)

```
for([i, e] of arr.entries()) {
  if (e)
    await foo();
  else
    await bar();
}
```

(b)

```
arr.forEach((e) => {
  if (e)
    return foo();
  else
    return bar();
});
```

(c)

```
await Promise.all(arr.map(async (e) => {
  if (e)
    return await foo();
  else
    return await bar();
}));
```

(d)

```
arr.map((e) => {
  if (e)
    foo();
  else
    bar();
});
```

(e)

```
await Promise.all(arr.map(async (e) => {
  if (e)
    await foo();
  else
    await bar();
}));
```

(f)

```
const o = {
  x : 1,
  get y() {
    return foo(x);
  }
}

o.y;
```

(g)

```
const o = {
  x : 1,
  get y() {
    return (async () => {
      return await foo(x);
    })();
  }
}

await o.y;
```

(h)

Figure 14: Code showing the before and after of applying select rewrite rules: **(a)-(b)** shows ForEach-ForOf, **(c)-(d)** shows ForEach-Map, **(e)-(f)** shows Await-Map, and **(g)-(h)** shows Getter.

Table 4: Information about subject applications. The first row reads: *the first application is called* ***upoint-query-builder*** *from* ***Harinathlee****, and commit hash* `f9aa0f1` *was used for the evaluation;* ***upoint-query-builder*** *has 10,341 lines of code. The initial size of the application is 0.84mb, reduced to 0.61mb after loading modules lazily, corresponding to a 27.4% size reduction. The size of the application once modules are loaded dynamically is 0.84mb. It took 201s to run* Lazifier *on this project, which required an additional 28s to build the CodeQL database.*

Project Name	Commit Hash	LOC	Initial Size (mb) Before	Initial Size (mb) After	Size Reduction	Expanded Size (mb)	Tool Run Time (s)	QLDB Time (s)
Harinathlee/upoint-query-builder [74]	`f9aa0f1`	10,341	0.84	0.61	27.4%	0.84	201	28
sadupawan1990/excelreader [197]	`4a5f9cb`	9,733	4.8	3.4	29.2%	4.8	187	44
fahimahammed/task [186]	`b641bc0`	9,747	0.94	0.48	48.9%	0.94	180	36
hongtaodai/react-excel [192]	`2d59e85`	9,685	1.9	1.5	21.1%	1.9	178	33
Abhishek312s/Movies-web-ui [7]	`58904a3`	9,789	1.4	0.96	31.4%	1.4	180	35
vishumane/ExcelSheet_Validation_Reactjs [200]	`f38cb9e`	9,942	0.90	0.40	55.6%	0.90	181	35
thewca/scrambles-matcher [198]	`1de93f7`	11,304	1.1	0.83	24.5%	1.1	188	37
hoverGecko/timetable [194]	`0fa8527`	9,932	0.60	0.38	36.7%	0.60	314	80
Akalay27/workday-schedule-exporter [10]	`97ca596`	9,718	0.90	0.44	51.1%	0.90	186	35
ultimateakash/react-excel-csv [199]	`18c6d97`	9,779	0.85	0.62	27.1%	0.85	206	34
			Average Size Reduction:		36.2%	**Average Run Time:**		240

Experimental Methodology

To address these research questions, we began by compiling a list of 10,000 open-source client-side JavaScript applications. This was done by scraping GitHub for repositories that listed JavaScript UI frameworks as dependencies. We then used the `npm-filter` [17] tool to identify projects where *Lazifier* flagged at least one package as a candidate for lazy loading, resulting in a subset of 998 projects. From this list, we manually examined projects until we identified 10 that could be successfully installed, launched, and interacted with. Most JavaScript projects on GitHub suffer from issues such as installation failures (e.g., outdated or broken dependencies), build errors (e.g., platform-specific build configurations), or runtime environment problems (e.g., reliance on unavailable external services). To ensure a thorough understanding of our subject applications, we invested significant effort in selecting projects free from these issues.

To answer **RQ1**, we begin by measuring the original application's initial size using the "bytes transferred" metric from the "Network" tab in Chrome DevTools' [70], recorded during a hard refresh of the application page. We then apply our transformation and measure the initial size of the modified application using the same method. To evaluate load time, we again use Chrome DevTools' "Network" tab, recording the "Load" time after performing a hard refresh. This measurement is taken both before and after the transformation, with 10 load times collected and averaged for each version.

To answer **RQ2**, we manually interacted with each application to trigger the execution of the code targeted for transformation, capturing screenshots after performing the relevant interactions. We then applied the transformations and repeated the same interactions, visually comparing the application's appearance and behavior before and after the changes. The application's behavior prior to transformation served as the baseline for comparison.

To answer **RQ3**, we first determine how to trigger each of the dynamic imports using the same interaction method as in **RQ2**. We then use Chrome DevTools' "Network" tab to record the size of the code chunk transferred, referencing the "bytes transferred" metric. Additionally, we note the time taken to load each chunk by consulting the "Load" time field.

To answer **RQ4**, we configured *Lazifier* to display the packages identified for lazy loading, show the dynamic import statements it inserted into the code, and log the code transformations it performed.

Finally, to answer **RQ5**, we used the Unix `time` utility to measure how long *Lazifier* took to run on each application. Since *Lazifier* 's analyses require a CodeQL database for each project, we also used the `time` utility to record the time needed to build the CodeQL database for each application.

All measurements were taken on a 2016 MacBook Pro running Catalina 10.15.7, with a 2.6GHz Quad-Code Intel Code i7 processor and 16GB RAM. We used Google Chrome version 112.0.5615.137 (Official Build) (x86_64) in incognito mode. Next, we respond to each of the **RQ**s in turn.

RQ1: *How does lazy loading affect the size and initial load time of applications?*

Lazifier's transformation takes advantage of ECMAScript 2020's support for on-demand package loading. By replacing all static imports of a package with dynamic imports, the JavaScript runtime defers fetching the package until the dynamic import is executed, meaning that the package is excluded from the initial application bundle. The initial sizes of the applications before and after refactoring are shown in the **Initial Size (mb) Before** and **After** columns in Table 4. Across all applications, we observed a significant reduction in size (an average of 36.2%), with reductions reaching up to 51.6%.

While reducing application size is beneficial on its own, we also examine how this reduction impacts the initial load time of the refactored applications. Fig. 15 presents the average load times (based on 10 trials) for each application, with three columns per subject. The first two columns are relevant here: the first shows the load time before refactoring, and the second shows the load time after refactoring. In all cases, we observe

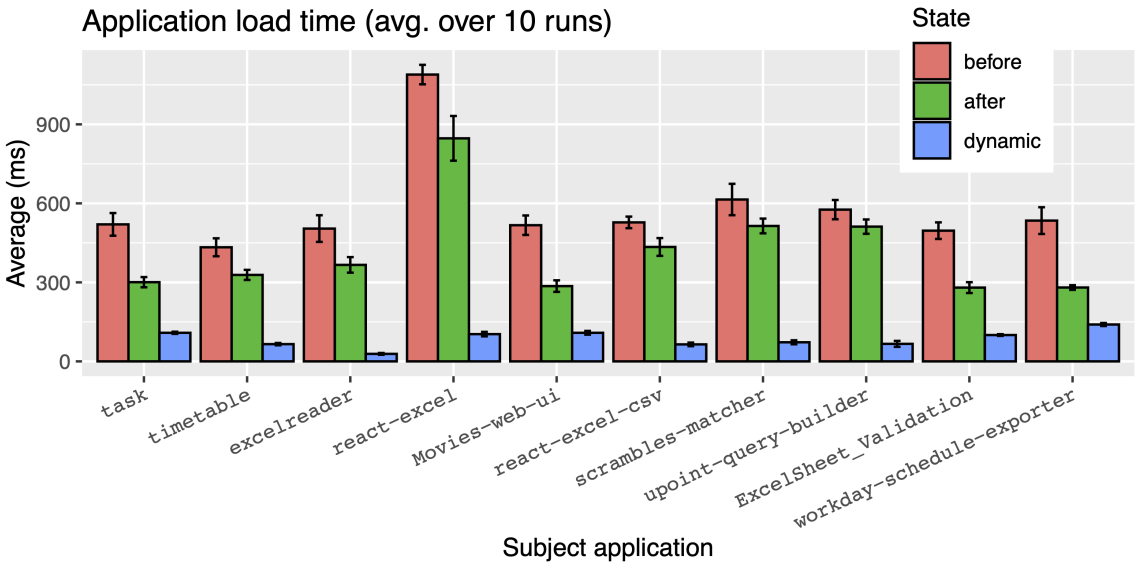


Figure 15: Load times for each subject application are depicted in this plot, with a set of three columns for each application. In each set, three times are presented: first, the time taken pre-refactoring (before), then after refactoring (after), and finally the time taken to dynamically load all packages (dynamic). These are averages over 10 runs, and error bars indicate +/- one standard deviation.

statistically significant reductions in initial load time (T-test, two-tailed, 95% confidence), with an average improvement of 29.7% and a maximum speedup of 47.5%.

Answer to RQ1: The size of refactored applications is smaller in all cases, which translates to a statistically significant reduction in application start times.

RQ2: *How often does the transformation introduce unwanted behavioral changes?*

Since the approach described in this chapter is based on unsound static analysis, *Lazifier*'s suggested transformations do not come with guarantees that application behavior will be preserved. In our evaluated applications, *Lazifier* refactored 15 packages for lazy loading, resulting in 21 dynamic import statements and 47 additional code transformations (i.e., applications of rewrite rules). We manually interacted with each application to ensure that all transformed code paths were executed and observed no differences in behavior after the transformations were applied.

Answer to RQ2: For the 10 subject applications under consideration in this evaluation, there was no evidence of behavioral differences due to unsoundness in the static analysis.

RQ3: *How much code is loaded lazily, and how quickly is it loaded?*

When a package is loaded dynamically, the application fetches and executes the package code asynchronously, making it available at runtime. This dynamic loading can lead to a larger overall application size because dynamic imports load the entire package, preventing tree-shaking optimizations that are possible with static imports. The total expanded size of each application after dynamic loading is shown in the **Expanded Size (mb)** column in Table 4. Interestingly, the total size after dynamic loading consistently matches the initial size before refactoring, indicating that tree-shaking has limited effectiveness in reducing the size of imported packages.

We also recorded the time required to complete this transfer, shown in the third column ("dynamic") of each set in Fig. 15. In all cases, the transfer size was small compared to the initial load times, averaging 85.8 milliseconds. However, it is important to note that this test did not simulate network latency, which would inevitably add overhead when transferring data over an actual network.

Answer to RQ3: The total size of the code loaded by the refactored applications (including lazily loaded packages) is comparable to the total size of the original applications, and dynamically loading packages is generally not noticeable.

RQ4: *How many code changes are required to support lazy loading?*

Since *Lazifier* proposes code changes that developers need to carefully review, it is important that the scope of these transformations remains small and manageable. Table 5 summarizes the code modifications suggested by *Lazifier* for each application, including the number of packages eligible for lazy loading (column **# Imps. Removed**), the number of dynamic import statements added (column **# Dyn. Imps.**), and the number of additional rewrite rule applications needed to support lazy loading (column **# Trans. Changes**). In all cases, the required code changes were limited, up to 15 in the largest case **upoint-query-builder**, with a median of just 6 changes per application, making the review process feasible for developers. Despite these relatively small modifications, substantial reductions in initial application size were achieved, which is promising since modest changes yielded significant improvements in load time.

Answer to RQ4: The number of code changes suggested by *Lazifier* is small, so the effort needed by programmers to review these changes is manageable.

RQ5: *What is the running time of* Lazifier*?*

The time required to run *Lazifier* is shown in the **Tool Run Time (s)** column of Table 4. This duration includes both the static analyses and the application transformation, although the transformation step itself is very fast. The time needed to build the CodeQL database is listed in the **QLDB Time (s)** column of Table 4 and represents a one-time, fixed cost per project that can be reused for other CodeQL queries.

Answer to RQ5: The run time of *Lazifier* is 240s on average, demonstrating its suitability for practical use.

4.6 THREATS TO VALIDITY

The technique presented in this chapter draws inspiration from the work presented in Chapter 3 and shares similar threats to validity. Specifically, the code transformations suggested by our approach are unsound and do not guarantee preservation of program

Table 5: Information about code transformations. The first row reads: *in* **upoint-query-builder**, *2 packages were loaded dynamically instead of statically; 3 dynamic import statements were added, and 12 applications of other rewrite rules were required to support the transition.*

Project Name	# Imps. Removed	# Dyn. Imps.	# Trans. Changes
upoint-query-builder	2	3	12
excelreader	1	1	2
task	1	1	2
react-excel	1	1	2
Movies-web-ui	2	2	5
ExcelSheet_Validation_Reactjs	2	3	7
scrambles-matcher	1	2	4
timetable	1	2	4
workday-schedule-exporter	3	4	6
react-excel-csv	1	2	3
In total:	15	21	47

behavior. This loss of soundness can stem from various factors, such as the unsound static analyses used to construct call graphs and the introduction of asynchrony, which may lead to data races. Given JavaScript's highly dynamic nature, achieving sound static analysis is inherently challenging. However, despite these limitations, our evaluation found that *Lazifier* did not introduce any behavior-altering transformations.

Additionally, our selection of subject applications may not be fully representative. To address this, we chose our applications from a nearly random sample of client-side JavaScript projects on GitHub. While we did filter this list to include only those applications that could be successfully built and run for evaluation purposes, we believe that starting with a random sample helps reduce the risk of bias.

4.7 CONCLUSION

Client-side developers aim to minimize the time users wait for a web application to load and become responsive. While existing tools like bundlers, minifiers, and tree-shakers help reduce code size by removing unused functionality, they do not address cases where certain features are potentially needed but not immediately at startup. In these situations, responsiveness can be improved by loading such functionality lazily. We have presented

an approach to identify when an entire library can be loaded on demand. This method uses static analysis to detect packages used exclusively within event-handling code and determines the necessary code changes to support lazy loading. The required transformations are specified through a set of declarative rewrite rules that guide the application's refactoring.

This approach was implemented in a tool named *Lazifier* and tested on 10 open-source client-side JavaScript applications. In every case, *Lazifier* successfully refactored the applications, achieving an average reduction of 36.2% in the initial application size and improving the start-up speed by an average of 29.7%.

4.8 PUBLICATION STATUS

This work was published in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)* [154]. It was presented at ASE 2023. The artifact for this work is available at https://doi.org/10.5281/zenodo.8264271.

5

PREVENTING SUPERFLUOUS RE-RENDERING IN REACT APPLICATIONS

5.1 INTRODUCTION

Modern-day web applications are complex pieces of software that aim to provide a rich user experience. One of the most common forms of web applications is a Single Page Application (SPA), where the content of the current page is updated dynamically instead of navigating to a new page. This approach enhances efficiency by eliminating the need to reload the foundational assets of the application—such as HTML, CSS, and JavaScript files—during navigation, thereby streamlining user interactions. This helps applications be *responsive*, one of the most desirable properties for user experience.

Building dynamic SPAs is nontrivial, and many UI libraries exist to help users develop them [1, 55, 71, 108, 131]. Typically, these libraries drastically change how developers use the language, e.g., through intricate APIs and syntactic extensions of the language. In this chapter, we focus on React [108], which is the most widely used framework for creating SPAs in JavaScript. React is currently the most popular front-end UI framework for JavaScript applications and is used by popular websites such as Instagram, Netflix, and GitHub, among others. `w3techs.com` reports that, as of January 2025, 4.7% of all websites use React [162], including 22 *million* GitHub repositories [66]. It has also garnered the attention of the research community [14, 45, 58, 103, 141, 144]; e.g., Madsen et al. specified the formal semantics of React [103], and Anastasia and Stamatia study how vulnerabilities in third-party dependencies affect clients of applications [14].

In React applications, developers define their UI in a *declarative* style. This enables the React engine to compute how much of the UI needs to be updated, or *re-rendered*, in response to user interaction. Unfortunately, the React engine is limited in its ability to detect situations where re-rendering can be avoided, and, if developers are not careful, their application may suffer from superfluous re-rendering, which may reduce responsiveness [141].

Our research aims to reduce the problem of unnecessary re-rendering in React applications. To this end, we carefully studied rendering behavior in a corpus of React applications and identified 5 anti-patterns that often give rise to needless re-rendering. We designed code transformations for each anti-pattern to reduce the number of rendering operations. We then developed a static analysis to detect instances of these anti-patterns and rewriting rules to automate code transformations necessary to eliminate them. Given

the extreme dynamism of JavaScript, the static analysis is unsound and may potentially suggest transformations that change program behavior. Therefore, similar to recent work on refactoring for JavaScript [69, 152, 154], the code transformations should be viewed as *suggestions* that should be reviewed by a developer. The technique was implemented in a tool named *Reactor* that we plan to make available as open-source software in the near future.

An empirical evaluation of *Reactor* on 23 subject React applications revealed that the refactorings proposed by *Reactor* reduced the total number of rendering operations by 33.3% on average and we did not observe any behavioral differences in all but one transformed application. This reduction in renders results in 20.54% less time spent rendering on average, and an additional set of experiments on three applications with a scalable number of components shows appreciable improvements in application responsiveness as the number of components increases. The refactorings also only contribute to a modest increase in code complexity. In another experiment, the anti-pattern detection queries were run on a corpus of 7,760 React repositories. This experiment identified instances of the anti-patterns in 92.1% repositories, providing evidence that these anti-patterns are prevalent beyond the subject applications studied in the empirical evaluation.

In summary, the contributions of this work are:

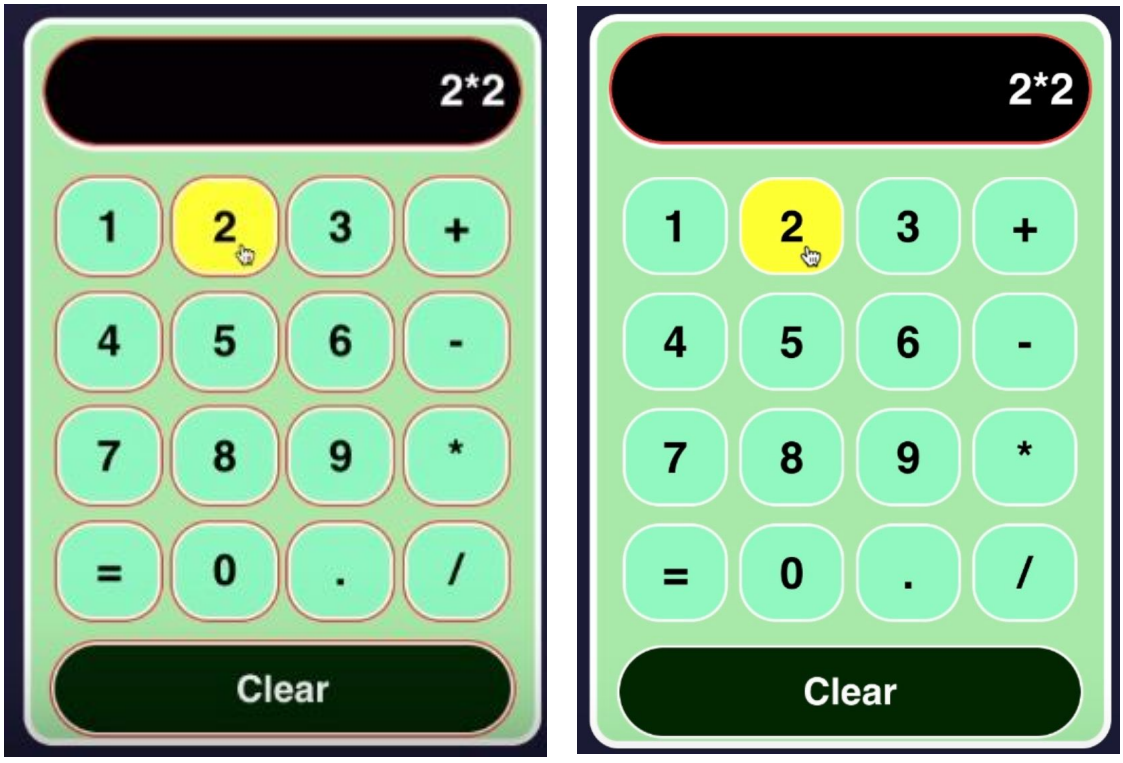
1. The identification of a set of ***anti-patterns*** that often give rise to unnecessary re-rendering in React applications,

2. A ***static analysis*** for detecting anti-pattern instances and ***rewriting rules*** describing remedial code transformations,

3. An ***implementation*** of the static analysis and rewriting rules in an automated tool called *Reactor*, which we plan to release as open-source software, and

4. An ***empirical evaluation*** of *Reactor* on 23 subject applications, demonstrating that it reduces the total number of rendering operations in most cases.

This chapter is organized as follows. Section 5.2 presents a motivating example that exhibits superfluous re-rendering and its remediation. Section 5.3 describes the identified anti-patterns. Section 5.4 describes our static analysis and code transformation techniques and Section 5.5 overviews the implementation of our tool, *Reactor*. Section 5.6 presents our evaluation. We discuss potential limitations in Section 5.7 before concluding with a summary of our findings in Section 5.9.



(a) Before transformation **(b) After transformation**

Figure 16: Re-render comparison before vs. after memoization when the "2" button is clicked with "2*" previously inputted, red indicating components that were re-rendered.

5.2 MOTIVATING EXAMPLE

To identify situations where excessive re-rendering occurs in React applications, we developed a profiling tool that highlights re-rendered components by drawing a red box around them. If a component is re-rendered after a user interaction but its visual appearance remains unchanged, it may have been re-rendered unnecessarily. Through manual inspection of a number of such situations, we identified several anti-patterns in React code that are likely to give rise to needless re-rendering, which we will present in Section 5.3.

Figure 16(a) shows a screenshot of **Calculadora** [42], a basic open-source calculator built using React in which needless re-rendering occurs. We developed a profiling tool that highlights re-rendered components by drawing a red box around them. The screenshot shows the UI while running it with our re-rendering profiler after the user has entered a formula 2*2. In Figure 16(a), each button is surrounded by a red box, indicating it was re-

```
400 import { useState } from 'react';
401 // ...
402 function App() {
403   const [input, setInput] = useState('')
        ;
404   const addInput = val => {
405     setInput(input + val);
406   };
407 // ...
408   return (
409     // ...
410     <Screen input={input}/>
411     <div className='row'>
412     <Button handleClick={addInput}>1</
        Button>
413     // ...
414   );
415 }
416 function Button(props) {
417   // ...
418 }
419
```

(a) Non-memoized

```
420 import { useState, useCallback } from '
        react';
421 // ...
422 function App() {
423   const [input, setInput] = useState('')
        ;
424   const addInput = useCallback((val) =>
        {
425     setInput((currentInput) =>
        currentInput + val);
426   }, []);
427   // ...
428   return (
429       // ...
430       <Screen input={input}/>
431       <div className='row'>
432       <Button handleClick={addInput}>1</
        Button>
433       // ...
434   );
435 }
436 const Button = React.memo(function
        Button(props) {
437   // ...
438 }
439
```

(b) Memoized

Figure 17: Example: unnecessary re-rendering in Calculadora.

rendered. Intuitively, one would expect only the display to be re-rendered when a button is pressed and the buttons' UI could remain unchanged.

To understand why the buttons re-render, we review the relevant code in Figure 17(a). Each button is represented by a `Button` component, and the calculator's display is modeled by a `Screen` component, all of which are children of a top-level `App` component. `App` maintains an `input` state, representing `Screen`'s content, also used for performing calculations. On lines 424-426, a function `addInput` is declared that concatenates a value to the `input`. This function is passed as the `handleClick` prop to each `Button` instance. Hence, clicking a button will invoke `addInput`, causing `App`'s state-setter `setInput` to be invoked. As discussed in Chapter 2, Section 2.3 any update to the `App` component's state triggers re-rendering of all its child components.

Inspecting the code reveals that `Button` does not have mutable state of its own and that the same function, `addInput`, is always passed as a property. In such cases, React's memo-

ization mechanism can be employed to prevent unnecessary re-rendering. This requires the two transformations shown in Figure 17(b):

- The `Button` component is memoized using `React.memo` (line 436). This instructs React not to re-render it when its parent `App` does, *provided that the same props are passed*.
- The `addInput` function passed as a prop from `App` to `Button` is memoized using `useCallback` (line 424). Without memoization, `App` recreates the `addInput` function on each re-render, thereby creating a new reference for the same function. Wrapping the function in `useCallback` and passing in an empty dependency list circumvents this, thus avoiding unnecessary re-renders.

We tested the optimized code and confirmed that extra re-renders were eliminated while preserving application behavior, evidenced by the absence of red borders around buttons in Figure 16(b). Eliminating these re-renders results in 31.3% less time spent on average rendering the application when a button is pressed.

5.3 ANTI-PATTERNS

The example laid out in the last section is an example of a broader *anti-pattern* that leads to re-rendering, and this section describes our methodology for identifying other such common anti-patterns that give rise to needless rerendering in React applications.

5.3.1 *Anti-pattern detection Methodology*

To identify anti-patterns, we randomly sampled 10,000 GitHub repositories that listed React as a dependency. We attempted to build the CodeQL [111] databases for these repositories, which succeeded for 7,760 repositories. Repositories for which we could not build a CodeQL database were excluded from further analysis, as a database is required to run our analysis queries. We wrote a static analysis using CodeQL to identify React components where at least one child component did not depend on all of its parent's state, representing likely scenarios with needless re-rendering. We ran this static analysis on the 7,760 repositories which identified 2,047 repositories for further analysis.

To construct a set of repositories for manual investigation, we iteratively sampled from the 2,047 candidates and applied a series of filtering criteria. A repository was retained only if it (i) was still available on GitHub, (ii) could be built and installed locally, (iii) could be executed and interacted with, (iv) exhibited visual indications of unnecessary re-rendering (as indicated by the profiler tool) despite no visible change in its appearance, and (v) did not require significant structural changes due to excessive code complexity

or deep reliance on external libraries. This process continued until we observed the same anti-patterns emerging repeatedly, at which point we considered the search saturated.

We sampled repositories uniformly at random from this corpus of 2,047 and discarded a repository if it was deleted from GitHub or if we could not install it, execute it, or interact with it. For each sampled repository, we used profiler tools to look for visual indications of needless re-rendering. We skipped repositories that showed no visual signs of needless re-rendering or had code complexity or reliance on external libraries making it impossible to eliminate re-rendering without significant structural changes. In cases where remediating re-rendering did not require such significant changes, we studied and attempted to remediate the observed re-rendering manually. We stopped investigating when the same anti-patterns began repeating themselves. This filtering process yielded 40 repositories suitable for detailed analysis, of which we found and fixed unnecessary re-rendering issues in 14.

To summarize, we began with a corpus of 10,000 GitHub repositories, out of which 7,760 repositories had CodeQL databases available. This set of repositories was further filtered down to 2,047 repositories that contained likely re-rendering scenarios. Finally, 40 repositories from this subset were randomly sampled and manually investigated, 14 of which contained re-rendering issues that could be fixed. This process resulted in the identification of 5 commonly observed anti-patterns that recurred across multiple applications. Our study of these projects, combined with our aforementioned prior knowledge and experience with React, informed the definition of anti-patterns in the subsections below.

5.3.2 *Anti-Pattern 1: Controlled Component*

Figure 3(a) shows an example of a controlled component, where a React component `Counter` declares a state variable `name` and an associated setter `setName` on line 143. In the JSX template literal returned by this component, `name` is referenced on line 153 and `setName` is invoked through `handleNameChange` referenced on line 154. React components whose state manages form data are referred to as *controlled components* [109]. In the example given here, the controlled data appears in `input` element, but controlled data may also appear in other types of form elements such as `select` and `textarea`.

A disadvantage of this approach is that each update to the input element will cause the component to re-render because one of its state-setters is called. Concretely, in Figure 3(a), each keystroke triggers a re-render, which is unnecessary.

To avoid unnecessary re-rendering in this type of scenario, one can rewrite the application so that the DOM handles the form data. This approach, commonly referred to as an *uncontrolled component*, is illustrated in Figure 3(b), and involves using React's `useRef` hook

to create a reference object with a ".current" property, allowing values to persist across re-renders without triggering new re-renders when modified. Now, changing the controlled component of Figure 3(a) into an uncontrolled component of Figure 3(b) involves the following changes: (i) importing the `useRef` hook, (ii) creating a reference `nameRef` for the `input` element (line 171), (iii) extending the form with a reference `ref` attribute bound to the reference `nameRef` (line 181), and (iv) getting/setting the value of the `input` element by accessing it via the reference object (line 185).

As discussed in Chapter 2 Section 2.3, when using a reference variable, the component will not re-render on every update to `input` and will only be re-rendered when the `CounterButton` is clicked.

5.3.3 *Anti-Pattern 2: State Variables not Affecting the UI*

```
440 import { useState, useEffect } from "
        react"
441 const CompFuncCicloVida = () => {
442   const [miVariable, setMiVariable] =
        useState<boolean|undefined>();
443   // ...
444   const updateVar = () => {
445     setMiVariable((val) => {
446       if (typeof val === "undefined")
447         return true
448       return !val
449   })}
450   return (
451     <div>
452       <button
453         onClick={updateVar}>
454         Actualiza
455       </button>
456     </div>
457   )
458 }
459
```

(a)

```
460 import {useState, useEffect, useRef}
        from "react"
461 const CompFuncCicloVida = () => {
462   const miVariableRef =
        useRef<boolean|undefined>();
463   // ...
464   const updateVar = () => {
465     miVariableRef.current =
466     typeof  miVariableRef.current ===
        "undefined"? true:
        !miVariableRef.current
467   }
468   return (
469     <div>
470       <button
471         onClick={updateVar}>
472         Actualiza
473       </button>
474     </div>
475   )
476 }
477
```

(b)

Figure 18: pattern *State Variables not Affecting the UI* example: **(a)** Uses state. **(b)** Uses reference.

Sometimes, developers use state variables for internal logic rather than updates to the UI, which may cause components to re-render unnecessarily. Figure 18(a) shows a component `CompFuncCicloVida`, taken from one of the 14 repositories under study (*16-feb-react-grupo-1*). Here, the `miVariable` state variable does not impact the UI. However, since it is

updated by a state-setter in the `updateVar` function (line 445), which is bound to the button's `onClick` handler (line 453), the `CompFuncCicloVida` component is re-rendered after each button click.

In such cases, the problem can be remediated by using React's `useRef` hook to introduce a reference object so that changes to the variable's value no longer cause re-rendering. Figure 18(b) shows how the code of Figure 18(a) can be transformed to prevent the unnecessary re-rendering by: (i) importing the `useRef` hook (line 460), (ii) replacing the declaration of the target state variable with a reference object (line 462), and (iii) replacing any access to (or modification of) that state variable with access to (or modification of) the "`.current`" property of the reference variable (lines 465, 466, 466).

5.3.4 *Anti-Pattern 3: Propless Child Component*

```
478 const Header = () => {
479   return (
480     // ...
481     <h1>My Context App</h1>
482   );
483 });
484 //...
485 export const App = () => {
486   // ...
487   return (
488     // ...
489     <Header />
490   )
491 }
492
```

(a)

```
493 const Header = React.memo(() => {
494   return (
495     // ...
496     <h1>My Context App</h1>
497   );
498 });
499 //...
500 export const App = () => {
501   // ...
502   return (
503     // ...
504     <Header />
505   )
506 }
507
```

(b)

Figure 19: pattern *Propless Child Component* example: **(a)** Non-memoized. **(b)** Memoized.

As outlined in Section 2.3, re-rendering a React component causes all its descendent components to re-render to guarantee UI consistency. In cases where components do not receive props, they inherently exhibit no change in their rendered output upon parent component re-rendering. Memoizing a prop-less component means that the shallow comparison of props trivially succeeds (given that there are none), enabling React to bypass the re-rendering process. Figure 19 shows an example of a prop-less component.

5.3.5 *Anti-Pattern 4: Child Component with Object or Array Element Prop*

```
508 function Square({handleClick, value}) {
509   // ...
510 }
511 function App() {
512   const [squares, setSquares] =
513     useState(Array(9).fill(null));
514   // ...
515   return (
516     // ...
517     <Square
518       value={squares[0]}
519       handleClick={() => handleClick(0)}
520     />
521     // ...
522   );
523 }
524
```

(a)

```
525 const Square=React.memo(({handleClick,
        value})=>{
526   // ...
527 }
528 function App() {
529   const [squares, setSquares] =
530     useState(Array(9).fill(null));
531   // ...
532   return (
533     // ...
534     <Square
535         value={squares[0]}
536         handleClick={() => handleClick(
        0)}
537     />
538     // ...
539   );
540 }
541
```

(b)

Figure 20: pattern *Child Component with Object or Array Element Prop* example: **(a)** Non-memoized. **(b)** Memoized.

If a component's props and state variables do not change, neither will its UI, so it does not require re-rendering. However, a component may still re-render if its parent re-renders, as React propagates re-rendering operations down the component tree.

Figure 20(a) shows a Tic-Tac-Toe game implementation exhibiting this anti-pattern. Here, the `App` component defines the game's grid structure, with a `Square` component for each grid square which receives a `value` prop that is either "X", "O", or empty. Each `Square` 's `value` comes from the `squares` array (a state variable in `App`) and changes only once, even though `squares` is updated with each action.

Each time a player clicks a `Square` in the game grid, the `onClick` handler, bound to the `handleClick` prop received from the `App` component, is called, which calls the `setSquares` state-setter in turn and triggers `App` to re-render. Hence, in accordance with React's strategy of re-rendering all child components when a parent component re-renders, all `Squares` re-render when their parent `App` re-renders.

Figure 20(b) shows how re-rendering can be prevented. Here, `Square` is wrapped in `React.memo` (line 525), signaling React to bypass re-rendering of `Square` components with unchanged props. Since each `Square`'s `value` prop only changes when it is first clicked, this

prevents re-rendering of the non-clicked squares as long as the value passed to `handleClick` also remains unchanged.

5.3.6 *Anti-Pattern 5: Child Component receiving Function Prop*

Figure 17 shows an example where a parent component `App` passes the `addInput` function as `handleClick` prop to its child component `Button` (line 412). When a button is clicked, the `handleClick` handler invokes `addInput`, which in turn calls `App`'s `setInput`, triggering a re-render of `App`. However, functions defined in a component are ***re-defined*** each time it is rendered. Consequently, all buttons re-render, even if `Button` is memoized, because `addInput` is recreated on each re-render of `App`, passing new references to the `Button` instances.

In such cases, wrapping a function in `useCallback` ensures that its identity remains unchanged across re-renders unless its dependencies, specified in the dependency array, change. As a result, utilizing the `useCallback` React hook can prevent these redundant re-renders by preserving the reference of the handler function across re-renders. Figure 17(b) shows how the code can be rewritten using a memoized handler(line 424) to avoid the unnecessary re-rendering. In particular, by memoizing both the `Button` component (line 436) and its `handleClick` prop ensures that only the clicked button re-renders, preventing unnecessary re-renders of the other buttons.

5.4 DETECTING AND REMEDIATING ANTI-PATTERNS

The previous section used code examples to illustrate a number of anti-patterns that give rise to needless re-rendering along with the code changes required to remediate them. This section defines the anti-patterns and changes more formally through a set of declarative rewrite rules. Our tool (*Reactor*) implements the static analyses and code transformations required to realize these rewrite rules.

5.4.1 *Controlled Component*

Rule CONTROLLED-COMPONENT is concerned with transforming controlled components into uncontrolled components and applies to a set of import statements I and a React component `F` with a body B, denoted by I ... `function F`(...) B. Reading the rule from top to bottom, the rule states that the body B of the component contains a declaration D of a state variable `v`, with associated state setter $\texttt{set}_\texttt{v}$, that is initialized to `e`. Moreover, F's return expression contains a JSX fragment R_j corresponding to either a Form or Input component, where a function `f` is passed as a change handler. Notably, `f` calls the state

$$\frac{\begin{array}{c}\text{B contains declaration } \mathrm{D} = \texttt{const}\ [\texttt{v}, \texttt{set}_\texttt{v}] = \texttt{useState}(\texttt{e}) \\ \textbf{returnExpr}(\mathrm{B}) \text{ contains } \mathrm{R_j} = <\mathrm{N}\ \ldots,\ \texttt{onChange} = \texttt{f}, \ldots> \\ \mathrm{N} \text{ is a Form or Input} \qquad \texttt{f} \text{ calls } \texttt{set}_\texttt{v} \\ \mathrm{D}' = \texttt{const}\ \texttt{v}_\texttt{ref} = \texttt{useRef}(\texttt{e}) \\ \mathrm{R_j'} = \textbf{replaceAttr}(\mathrm{R_j}, \texttt{onChange} = \texttt{f}, \texttt{ref} = \texttt{v}_\texttt{ref}) \\ \mathrm{B}' = \textbf{replaceExpr}(\mathrm{B}, \mathrm{R_j}, \mathrm{R_j'}) \qquad \mathrm{B}'' = \textbf{replaceDecl}(\mathrm{B}', \mathrm{D}, \mathrm{D}') \\ \mathrm{B}''' = \textbf{replaceVarRefs}(\mathrm{B}'', \texttt{v}, \texttt{v}_\texttt{ref}\texttt{.current.value}) \\ \mathrm{B}'''' = \textbf{replaceCalls}(\mathrm{B}''', \texttt{set}_\texttt{v}(\texttt{e}), \texttt{v}_\texttt{ref} = \texttt{e}) \\ \mathrm{I}' = \mathrm{I} \cup \{\texttt{import}\ \{\texttt{useRef}\}\ \texttt{from}\ \texttt{'react'}\}\end{array}}{\mathrm{I}\ \ldots\ \texttt{function F}\,(\ldots)\ \mathrm{B} \rightarrow \mathrm{I}'\ \ldots\ \texttt{function F}\,(\ldots)\ \mathrm{B}''''} \text{ (Controlled-Component)}$$

$$\frac{\begin{array}{c}\text{B contains declaration } \mathrm{D} = \texttt{const}\ [\texttt{v}, \texttt{set}_\texttt{v}] = \texttt{useState}(\texttt{e}) \\ \text{no data flow from v to } \textbf{returnExpr}(\mathrm{B}) \\ \mathrm{D}' = \texttt{const}\ \texttt{v}_\texttt{ref} = \texttt{useRef}(\texttt{e}) \\ \mathrm{B}' = \textbf{replaceDecl}(\mathrm{B}, \mathrm{D}, \mathrm{D}') \\ \mathrm{B}'' = \textbf{replaceVarRefs}(\mathrm{B}', \texttt{v}, \texttt{v}_\texttt{ref}\texttt{.current}) \\ \mathrm{B}''' = \textbf{replaceCalls}(\mathrm{B}'', \texttt{set}_\texttt{v}(\texttt{e}), \texttt{v}_\texttt{ref} = \texttt{e}) \\ \mathrm{I}' = \mathrm{I} \cup \{\texttt{import}\ \{\texttt{useRef}\}\ \texttt{from}\ \texttt{'react'}\}\end{array}}{\mathrm{I}\ \ldots\ \texttt{function F}\,(\ldots)\ \mathrm{B} \rightarrow \mathrm{I}'\ \ldots\ \texttt{function F}\,(\ldots)\ \mathrm{B}''''} \text{ (State-Not-Affecting-UI)}$$

$$\frac{\begin{array}{c}\texttt{F} \text{ is a React component} \\ \mathrm{I}' = \mathrm{I} \cup \{\texttt{import}\ \{\texttt{memo}\}\ \texttt{from}\ \texttt{'react'}\} \\ \mathit{where}\ \texttt{F}\ \mathrm{is\ not}\ \texttt{memoized}\end{array}}{\mathrm{I}\ \ldots\ \texttt{function F}\,()\ \mathrm{B} \rightarrow \mathrm{I}'\ \ldots\ \texttt{const F} = \texttt{memo}(\texttt{function}()\ \mathrm{B})} \text{ (Propless-Component)}$$

$$\frac{\begin{array}{c}\texttt{F} \text{ is a React component} \\ \exists\ <\mathrm{F}\ldots, \mathrm{p} = e, \ldots> \mid \texttt{e} = \texttt{v.f} \text{ or } \texttt{e} = \texttt{v[i]} \text{ for some state variable v} \\ \mathrm{I}' = \mathrm{I} \cup \{\texttt{import}\ \{\texttt{memo}\}\ \texttt{from}\ \texttt{'react'}\} \\ \mathit{where}\ \texttt{F}\ \mathrm{is\ not}\ \texttt{memoized}\end{array}}{\mathrm{I}\ \ldots\ \texttt{function F}\,(\ldots)\ \mathrm{B} \rightarrow \mathrm{I}'\ \ldots\ \texttt{const F} = \texttt{memo}(\texttt{function}(\ldots)\ \mathrm{B})} \text{ (Memoize-Component-Indexes-State)}$$

$$\frac{\begin{array}{c}\text{B contains function declaration } \mathrm{D} = \texttt{function f}\,(\ldots)\ \mathrm{B_f} \\ \text{B contains component definition } \mathrm{C} = \texttt{function N}\,(\ldots)\ \mathrm{B_N} \\ \textbf{returnExpr}(\mathrm{B}) \text{ contains } <\mathrm{N}\ \ldots,\ \texttt{attr} = \texttt{f}, \ldots> \\ \mathrm{L} = \text{all state variables referenced in } \mathrm{B_f} \\ \mathrm{D}' = \texttt{const f} = \texttt{useCallback}(\texttt{function}\,(\ldots)\ \mathrm{B_f}, \mathrm{L}) \\ \mathrm{C}' = \texttt{const N} = \texttt{memo}(\texttt{function}\,(\ldots)\ \mathrm{B_N}) \\ \mathrm{B}' = \textbf{replaceDecl}(\mathrm{B}, \mathrm{D}, \mathrm{D}') \qquad \mathrm{B}'' = \textbf{replaceDecl}(\mathrm{B}', \mathrm{C}, \mathrm{C}') \\ \mathrm{I}' = \mathrm{I} \cup \{\texttt{import}\ \{\texttt{useCallback}, \texttt{memo}\}\ \texttt{from}\ \texttt{'react'}\}\end{array}}{\mathrm{I}\ \ldots\ \texttt{function F}\,(\ldots)\ \mathrm{B} \rightarrow \mathrm{I}'\ \ldots\ \texttt{function F}\,(\ldots)\ \mathrm{B}''} \text{ (Function-Passed-As-Prop)}$$

$$\frac{\begin{array}{c}\text{B contains declaration } \mathrm{D} = \texttt{const}\ [\texttt{v}, \texttt{set}_\texttt{v}] = \texttt{useState}(\texttt{e}) \\ \text{B contains component definition } \mathrm{C} = \texttt{function N}\,(\ldots)\ \mathrm{B_N} \\ \textbf{returnExpr}(\mathrm{B}) \text{ contains } <\mathrm{N}\ \ldots,\ \texttt{attr} = \texttt{set}_\texttt{v}, \ldots> \\ \mathrm{C}' = \texttt{const N} = \texttt{memo}(\texttt{function}\,(\ldots)\ \mathrm{B_N}) \\ \mathrm{B}' = \textbf{replaceDecl}(\mathrm{B}, \mathrm{C}, \mathrm{C}') \\ \mathrm{I}' = \mathrm{I} \cup \{\texttt{import memo from}\ \texttt{'react'}\} \\ \mathit{where}\ \texttt{C}\ \mathrm{is\ not}\ \texttt{memoized}\end{array}}{\mathrm{I}\ \ldots\ \texttt{function F}\,(\ldots)\ \mathrm{B} \rightarrow \mathrm{I}'\ \ldots\ \texttt{function F}\,(\ldots)\ \mathrm{B}'} \text{ (Setter-Passed-As-Prop)}$$

Figure 21: Rewrite rules describing the remedial code transformations.

setter $\texttt{set}_\texttt{v}$. As is the case with all subsequent rules, these facts are all computed using static analysis.

Then, a new declaration $\texttt{D}'$ is defined for a reference variable $\texttt{v}_\texttt{ref}$ (also initialized to `e`); a new JSX fragment $\texttt{R}'_\texttt{j}$ is obtained from $\texttt{R}_\texttt{j}$ by replacing the `onChange` attribute with a `ref` attribute that refers to $\texttt{v}_\texttt{ref}$. The body is then modified several times, in order: $\texttt{B}'$ from `B` where the new fragment $\texttt{R}'_\texttt{j}$ replaces $\texttt{R}_\texttt{j}$, $\texttt{B}''$ from $\texttt{B}'$ where the new reference variable declaration $\texttt{D}'$ replaces the old state variable declaration `D`, $\texttt{B}'''$ from $\texttt{B}''$ where references to the state variable `v` are replaced with accesses to the "current.value" property of the new reference variable $\texttt{v}_\texttt{ref}$, and finally $\texttt{B}''''$ from $\texttt{B}'''$ where calls to the state setter $\texttt{set}_\texttt{v}$ are replaced with assignments to the new reference variable $\texttt{v}_\texttt{ref}$. The set of imports is expanded to include an import to `'useRef'`, and thus the transformed code is given by $\texttt{I}'$ ... `function F (...)` $\texttt{B}''''$.

5.4.2 *State Variables not Affecting the UI*

Many of the code transformation steps for this rule are shared with the previously discussed rewrite rule, and we will not discuss them in detail here. The primary difference lies in the second clause: the code should be transformed if there is no data flow between a state variable v and the return expression of component `F` (i.e., the state variable is not passed as a prop to any returned component).

5.4.3 *Propless Child Component*

Rule PROPLESS-COMPONENT describes the memoization of propless React components. Here, `F` is a React component, and the (function) component declaration is transformed into an assignment wrapping the function in a call to `memo` (as `memo` cannot be directly called on function declarations), and as before the imports must be extended to import `memo` from `'react'`.

5.4.4 *Child Component with Object or Array Element Prop*

Rule MEMOIZE-COMPONENT-INDEXES-STATE applies to situations where one of a component's props receives data from a state variable through property access or array access to that variable. Note that `F` is a React component. If there exists an instantiation $<\texttt{F}..., \texttt{p} = e, ...>$ of this component anywhere in the project where some prop `p` is passed either a property access `v.f` or index into an array `v[i]` of some state variable v, then the component should be memoized. As in the previous rewrite rule, the component is memoized by transforming the function declaration into an assignment of a function literal,

where the function creation is wrapped in `memo`. As before, the imports I are updated to import `memo`.

5.4.5 *Child Component receiving Function Prop*

For the last anti-pattern, we distinguish two cases: (a) a locally declared function is passed as a prop to a child component, and (b) a setter function is passed as a prop to a child component.

Rule FUNCTION-PASSED-AS-PROP applies when a function is passed as a prop to a component. Here, function body B contains both a function declaration D for a function `f` and a React component declaration C for a component `N`. The return expression of B contains a JSX expression that instantiates the component `N`, and notably, that instantiation receives `f` as a prop. To remediate this anti-pattern, first, the list L of "dependencies" of `f` is obtained, consisting of all state variables that are referenced in the body of `f`. Then, D′ is a new declaration that wraps the creation of `f` in `useCallback`, and this call to `useCallback` also takes L as an argument. Moreover, C′ is a declaration of `N` that also memoizes `N`. Two transformations are then applied: B′ is obtained from B where the function declaration D is replaced with D′, and B″ from B′ where the component declaration C is replaced with C′. The imports I are extended with imports to `useCallback` and `memo`, giving I′, and the final code is given by I′ ... `function F` (...) B″. (Note that the transformation can span multiple scopes or even files, and this simplified version is shown for illustrative purposes.)

Finally, rule SETTER-PASSED-AS-PROP applies when a function passed to a component is a state setter. The procedure to remediate this anti-pattern is similar to the previous case, except that the setter does not need to be wrapped in `useCallback`. In body B, a state variable is declared (denoted D, with state setter $\texttt{set}_\texttt{v}$) and a component `N` is declared (C). Here, the component is instantiated with $\texttt{set}_\texttt{v}$ passed as a prop (see <N ..., `attr` = $\texttt{set}_\texttt{v}$,...>). To remediate the pattern, the declaration of component `N` is wrapped in a call to `memo` (C′), and a new body B′ is obtained from the old B by replacing the declaration C with C′. I′ is obtained from I by adding an import to `memo`, and the transformed code is given by I′ ... `function F` (...) B′.

5.5 IMPLEMENTATION

These rewrite rules were implemented in a tool called *Reactor*. The implementation leverages the CodeQL [111] static analysis framework to detect the anti-patterns and to collect the data flow information required for the remedial program transformations, which are implemented using JavaScript's Babel parsing and AST infrastructure. More precisely,

CodeQL queries compute all the preconditions of a rule, and the JavaScript rewrite tool takes this information to transform the code. Our code and experiments are available as an artifact [1].

5.6 EVALUATION

This evaluation aims to answer the following research questions:

- **RQ1** How prevalent are the anti-patterns?
- **RQ2** How often do the automated transformations introduce behavioral differences?
- **RQ3** How do the transformations impact the number of re-rendering operations?
- **RQ4** What is the impact of the transformations on performance?
- **RQ5** What is the impact of the transformations on code complexity?
- **RQ6** What is the cost of the analysis?

In this Section, we start by depicting the experimental methodology, clarifying the subject selection process for our analysis in Section 5.6.1. We then attempt to answer each of the research questions with the dedicated corpus for analysis in the following subsections.

5.6.1 *Experimental Methodology*

As mentioned in Section 5.3, we initially started by aggregating a corpus of React applications by randomly sampling 10,000 GitHub repositories that listed React as a dependency, for 7,760 of which, we could successfully build a CodeQL database. CodeQL could build a database for 7,760 of these repositories, and so we ran the anti-pattern identification queries on these to answer RQ1. In Section 5.3, we focused our manual efforts on 2,047 repositories where a CodeQL query identified components where at least one child component did not depend on all of its parent's state; not wanting to restrict ourselves to this subset, we ran the anti-pattern identification queries we developed instead on all 7,760 repositories to answer RQ1.

To answer RQ1, we ran the anti-pattern identification queries we developed on all 7,760 repositories. RQs 2 and 3, however, cannot be answered "at scale" like RQ1. Although applying the analysis and transformations requires no manual effort, confirming that no behavioral differences were introduced and checking if the number of needless re-renders

1 Link to artifact: https://doi.org/10.5281/zenodo.16129074

decreased is a manual effort. Therefore, we attempted to select a subset of that corpus whose results could be representative of the overall results.

Our evaluation set began with the 14 repositories described in Section 5.3, which we had previously analyzed and manually fixed. These projects provided a basis for evaluating the effect of *Reactor* on repositories that we had already verified to be fixable manually. To assess generalizability, we then augmented this set with repositories that had not been seen during manual exploration.

We returned to the full list of 10,000 repositories and identified 75 that had published at least one release containing a downloadable asset, indicating some level of real-world usage. [2].

Reactor detects at least one anti-pattern in 57 of these projects. From these, we randomly sampled repositories until we found 9 that we could build and install successfully and set up and run locally, and did not use Class components. We relied on the Chrome Dev Tools React toolkit (the Components and Profiler tabs) and our own React Dynamic Profiler to observe and understand an application's rendering behavior.

In total, we evaluated 23 repositories: the original 14 manually analyzed projects, and an additional 9 previously unseen repositories selected through the process above.

To identify a set of subject React applications on which to evaluate the transformations suggested by *Reactor*, we started with the randomly sampled 10,000 GitHub repositories that listed React as a dependency (the same set used in Section 5.3). To find projects to test the remedial transformations, we randomly selected projects from that set that had at least one instance of an anti-pattern until we found 15 that we could install and build in order to interact with. In addition, we filtered the list to contain only projects with at least one downloaded asset (representing larger, more realistic applications); there were only 75 such projects. 57 of these projects had at least one anti-pattern detected by our tool, and from these, we randomly sampled repositories until we found 10 that installed and built and that we could set up and run so as to investigate the effect of the suggested transformations. We relied on the Chrome Dev Tools React toolkit (the Components and Profiler tabs) and our own React Dynamic Profiler to observe and understand an application's rendering behavior.

To summarize, we took the following steps to answer each of the research questions.

- For **RQ1**, we developed CodeQL [111] queries to identify each of the anti-patterns specified in Section 5.4. This large-scale analysis was done on the corpus of 7,760 repositories. (These CodeQL queries are available in the appendix.)

2 A project needs at least one release (made up of one or more assets, e.g., a zip file of source code) with at least one downloaded asset. This is quite restrictive, as it indicates the project has at least one official release that has garnered some attention; in our experience, the vast majority of React projects on GitHub do not meet this standard.

- **RQ2** and **RQ3** require non-trivial manual effort; **RQ2** requires manual inspection of projects pre- and post-refactoring. For **RQ3**, we instrumented the code to count how often a component renders before and after refactoring. Since many React projects lack tests, we created one "workflow" per studied application interacting with at least one component exhibiting an anti-pattern. We then carefully compared the application's behavior and the number of renders before and after refactoring for 23 subject repositories.
- For **RQ4**, we used the Chrome Dev Tools React Profiler tab to compute the amount of time taken to render when following the aforementioned scenarios for the 23 subject applications. In addition, we collected three additional applications with a scalable number of components to investigate the impact of re-rendering on performance as applications scale.
- For **RQ5**, we used escomplex [185] to measure different code complexity metrics such as SLOC, Cyclomatic complexity, and Halstead Complexity, both before and after transformation for the 23 subject repositories.
- Finally, for **RQ6**, we measured the time to run *Reactor* on the smaller set of 23 applications using the Unix `time` utility. We also timed the most time-consuming step of the analysis in a larger-scale study over the full set of 7,760 databases.

All the CodeQL queries were run, using NPM version v9.6.6, Node.js version v18.16.0, and CodeQL version v2.17.2.

5.6.2 *RQ1: How prevalent are the anti-patterns?*

We ran the anti-pattern detection queries on the corpus of 7,760 projects, and we found at least one instance of an anti-pattern in 92.1% of projects. The anti-patterns appeared as follows: *Controlled Component* in 30.4% of applications, *State Variables not Affecting the UI* in 42.7%, *Propless Child Component* in 91.2%, *Child Component with Object or Array Element Prop* in 6.3%, and *Child Component receiving Function Prop* in 36.7%. These results indicate that the anti-patterns occur frequently in real-world React applications.

This chapter presents five anti-patterns that were identified by manually inspecting 40 repositories. This dissertation is the first work to characterize anti-patterns leading to needless re-rendering. In the absence of ground truth, it is difficult to gauge the exhaustiveness of the anti-patterns presented in this chapter. However, we observed that these

Table 6: Overview of results. The first row of the table reads: in 16-feb-react-grupo-1, 0 instances of anti-pattern P1, 1 of P2, 11 of P3, 0 of P4, and 1 of P5 were detected. The number of component renders in the scenario we studied was reduced by 16% after refactoring. Rendering operations took on average 4.83ms before, and 4.45ms after refactoring; an improvement of 7.9%. It took 6s to build the CodeQL database, and 688s to run all queries for this application. The cyclomatic complexity changed from 325 to 338 after refactoring. Bold numbers in % Time Saved indicate a statistically significant difference between performance before and after refactoring. We omit render reductions and time spend rendering for TwitchKillMe as behavioral differences were observed in this application.

	Anti-pattern Detected					Render	Avg. Render (ms)		% Time	Cyclomatic		Analysis time (s)	
Subject Repository Name	**P1**	**P2**	**P3**	**P4**	**P5**	**Red. %**	**Before**	**After**	**Saved**	**Before**	**After**	**DB Build**	**Query**
16-feb-react-grupo-1 [2]	0	1	11	0	1	16%	4.83	4.45	7.9%	325	338	6	688
26-react-star-wars-hooks [3]	0	0	8	0	1	20%	7.72	7.43	3.8%	205	271	5	852
30days-30projects [4]	9	12	38	1	1	50%	51.61	27.87	**46%**	1107	1131	7	852
age-react [179]	0	0	3	0	2	50%	30.68	22.38	**27.1%**	164	164	5	672
AI-chat-powered-by-open-ai-api-react-node [5]	1	0	1	0	0	71%	9.06	1.54	**83%**	82	82	5	688
API-Search [6]	1	0	2	0	2	12%	13.04	13.14	-0.8%	196	207	6	852
Calculadora [43]	0	0	1	0	2	66%	20.37	13.99	**31.3%**	52	50	5	800
CashTrackr [44]	0	0	1	0	3	00%	14.15	14.94	-5.6%	153	164	5	852
CRUD-Context-React [35]	0	0	4	0	2	22%	48.67	46.78	3.9%	235	235	6	792
hackaton-euraz-2023 [188]	0	3	4	0	0	12%	83.22	38.38	**53.9%**	765	740	8	831
hacker-news-app-2023 [189]	1	1	2	0	0	15%	12.98	9.45	**27.2%**	79	79	5	852
healthCare [190]	13	14	39	0	0	23%	155.7	122.4	**21.4%**	919	964	5	672
honey-rae-repairs [191]	10	4	22	0	3	93%	24.85	5.58	**77.5%**	416	284	6	835
hooksMixed-1 [193]	0	0	2	0	0	67%	5.81	4.21	**27.5%**	78	78	5	826
TypeWriter [160]	0	1	37	0	4	22%	24.82	24.63	0.8%	616	730	13	703
farm-ng-core [187]	0	0	3	0	0	68%	95.42	97.14	-1.8%	6	28	7	604
css-fx-layout [183]	0	1	5	0	0	47%	24.65	24.93	-1.1%	50	65	6	1164
chicken-coop [181]	2	3	7	0	0	54%	6.74	5.53	**18%**	172	164	7	605
Fokasu [62]	0	0	1	0	1	57%	17.9	18.02	-0.7%	85	85	8	687
laser-shooting [195]	0	0	3	0	0	00%	207.14	219.13	-5.8%	444	444	10	1098
csi-conference-frontend [182]	0	2	13	0	0	01%	57.89	57.13	1.3%	230	351	7	623
baklava [180]	0	0	1	0	0	00%	0.52	0.46	11.5%	3	3	11	700
TwitchKillMe [158]	0	1	2	0	0	–	–	–	–	85	67	7	688
Average	**1.6**	**1.8**	**9.1**	**0.0**	**0.9**	**33.3%**	**40.25**	**34.08**	**20.54%**	**281.17**	**292.35**	**6.7**	**779.8**

five anti-patterns are extremely prevalent, as 92.1% of the 7,760 repositories we studied exhibited at least one.

Answer to RQ1: 92.1% of the 7,760 studied React applications under consideration exhibit at least one anti-pattern.

5.6.3 *RQ2: How often do the transformations cause behavioral differences?*

We manually examined each application's behavior before and after refactoring to ensure the transformations did not introduce differences. We opted for a manual approach since UI-based applications are generally not well-tested, given the complexity of simulating human interaction in tests. The goal of our manual interaction is to simulate a simple interaction to exercise code affected by our transformations and not to emulate the full use of an application. To do so, we explored every page of the application, interacting with every interactive element, exercising all transformed code. A step-by-step account of these interactions appears in the appendix.

After careful analysis of all applications, we only observed one behavioral difference in this evaluation, in the *TwitchKillMe* project. Upon deeper inspection of the code, we observed the implementation was not aligned with React's development philosophy, having a nested component definition [150] and did not use props for parent-child communication. This is not self-contained and violates the locality of behavior. It would be possible to develop a static analysis to identify instances of this anti-pattern but that is beyond the scope of this work.

Answer to RQ2: We observed only one situation where *Reactor* caused behavioral differences in our 23 carefully studied subject applications.

5.6.4 *RQ3: How do the transformations impact the number of re-rendering operations*

We automatically instrumented each React component to log the rendering operations as they were being performed, using the same scenarios that we used to answer RQ2 and noted the total number of rendering operations before and after transformation.

Table 6 summarizes the results of this experiment, in particular, the **Red. %** column indicates the % reduction in total rendering operations during our manual analysis. We see significant reductions in the number of rendering operations observed in the majority of cases. In *honey-rae-repairs*, for example, there is a large reduction as most of the application's functionality is related to forms that are transformed into uncontrolled components. *age-react* and *Fokasu* contain a timer causing some of the components and prop-less child

components to re-render every second. Memoizing these prop-less child components ensures they are rendered only once.

Notably, for *CashTrackr*, we see no change in the number of rendering operations performed. There are two major reasons for this. First, the prop-less components are only rendered once. Second, one component in the application takes functions as props (which can be wrapped in `useCallback`) but also takes an array of "expenses". Due to JavaScript's highly dynamic nature, *Reactor* only performs intra-procedural analysis and cannot fully refactor this anti-pattern.

In applications with at least one downloaded asset: For *baklava* and *laser-shooting*, we do not see any differences in the number of rendering operations, as the propless component is either a top-level component or relies on an external routing library, which prevents local optimization.

Answer to RQ3: In the 23 applications under consideration, the number of rendering operations is reduced by 33.3% on average after applying the transformations suggested by *Reactor*.

5.6.5 *RQ4: What is the impact of the transformations on performance?*

We used the Chrome Developer React Profiling Tools to collect the time spent rendering all components during each of the scenarios discussed in RQ3, reported in **Avg. Time Rendering** columns in Table 6. We also computed the % difference in time before and after refactoring, shown in column **% Time Saved** in the same table; bold entries in this column indicate statistically significant differences in render times before and after refactoring at 95% confidence using Welch's t-test (this test does not assume equal variance). Overall, we see improvements in render times in most cases, and no statistically significant negative impact to performance.

The performance impact of superfluous re-rendering becomes evident as an application scales. To illustrate, we collected an additional three applications that had a number of components that could vary dynamically: a spreadsheet, a drawing application, and a JSON editor. We configured these applications at small, medium, and large scales, and conducted experiments where we investigated the performance before and after refactoring at each scale using the Chrome Dev Tools React Profiler tab. These experiments were conducted on a 2023 MacBook Pro with an M2 Max chip and 64GB RAM. We manually confirmed that application behavior was preserved after refactoring, and these are all available in the artifact.

5.6.5.1 *Spreadsheet*

In this application [72], *every cell* of the spreadsheet was re-rendering any time a user either clicked on or entered data in *any cell*; one would expect rendering performance to become progressively worse as the sheet's size increases. *Reactor* refactors this application to avoid re-rendering unmodified cell. We prepared three configurations for this application: a small 20x20 spreadsheet, a medium 60x60 spreadsheet, and a large 100x100 spreadsheet.

In the small spreadsheet, typing one character into one cell incurred, on average, a 6.89ms re-render before refactoring, and a 2.07ms re-render after refactoring (3.3x improvement). In the medium spreadsheet, these re-renders took 43.4ms and 7.74ms on average, resp. (5.6x improvement). In the large spreadsheet, these re-renders took 125.35ms and 19.37ms on average, resp. (6.5x improvement). Before refactoring, we observed a noticeable lag in the medium scale experiment, and a significant lag in the large scale experiment. After refactoring, we noticed no lag in any case.

In terms of code complexity, the main code transformation was memoization of the component responsible for cells in the spreadsheet, and wrapping two functions passed to the cells with useCallback; together, these simple code changes result in up to a 100ms improvement in render time while typing in a cell in this spreadsheet, which we found to be noticeable.

5.6.5.2 *Drawing App*

In this application [53], users draw on a grid, and interacting with *any pixel* causes *all other pixels* to re-render; *Reactor* suggests refactoring this application to avoid re-rendering pixels that were not interacted with. We prepared three configurations for this application: a small drawing area of 32x32 pixels, a medium area of 64x64 pixels, and a large area of 128x128 pixels.

In the small configuration, drawing a single pixel caused a re-rendering operation lasting 14.51ms on average before refactoring, and 2.76ms on average after, an improvement of 5.3x. In the medium configuration, these re-renders took 51.83ms and 7.67ms on average, resp., an improvement of 6.8x. Finally, in the large configuration, these re-renders took 197.43ms and 26.49ms on average, resp., an improvement of 7.5x. We observed noticeable lag before refactoring in the large configuration, and no lag in any case after refactoring.

In terms of code complexity, the main code transformation was the memoization of the component responsible for pixels, and wrapping the function passed into the pixel components with useCallback. These are again very simple code transformations that result

in significant improvements in application performance, in particular as the drawing grid becomes larger.

5.6.5.3 *JSON Editor*

In this application [8], users load a JSON and edit in in-browser, and save it to disk. A `Form` component from the `react-bootstrap` library is created for each element in the JSON. Before refactoring, the underlying JSON object was represented with a React state variable and a new JSON object was created each time *any form* was modified, causing *all forms* to re-render. *Reactor* refactored the application to manage the JSON with a ref instead, and the only component that changes when editing a form after refactoring is the form itself. We created small, medium, and large JSONs with 100, 1000, and 10000 elements resp. for this experiment.

Application performance is greatly improved by *Reactor*. Before refactoring, writing a single character into one field incurred a 17.05ms render on average with the small JSON, a 106.74ms render on average with the medium JSON, and a 1,035.88ms render on average with the large JSON; lag was noticeable with the medium JSON, and prohibitive with the large JSON. After refactoring, editing a form is seamless in all cases as no other components on the page re-render. In fact, HTML forms (which underlie the `react-bootstrap` form component) are managed a bit differently than other components as they maintain limited internal state [3], and modifying the contents of a form does not cause a change in the VDOM unless other components depend on the contents. In a sense, these forms render their own content without going through React. After refactoring, the only component on the page that changes when a form is edited is the form itself, and the React profiling tools register no other component renders; this results in React spending oms re-rendering components after refactoring.

The main change in this application was managing the JSON object with a ref, which required *Reactor* to change how this object was referenced in 8 code locations (refactoring references to `currentJson` into `currentJson.current`). These changes were all localized to a single file, and had a dramatic effect on performance.

The impact of needless re-renders manifests itself particularly as applications scale, and small, simple code changes reduces rendering time and can make applications more responsive.

3 See `https://legacy.reactjs.org/docs/forms.html`, and `https://react-bootstrap.netlify.app/docs/forms/overview/` for more information.

5.6.6 *RQ5: What is the impact of the transformations on code complexity?*

We measured code complexity metrics such as sloc, cyclomatic complexity, and Halstead complexity. For brevity, we only present cyclomatic complexity in Table 6 but all metrics show a similar trend and are available in the Appendix. On average, we observe a slight increase in the Cyclomatic complexity of the programs from 281.17 to 292.35. Some projects such as *16-feb-react-grupo-1* and *css-fx-layout* show a small increase in cyclomatic complexity, whereas projects such as *honey-rae-repairs* and *chicken-coop* show a slight reduction in code complexity. Cyclomatic complexity can decrease, e.g., if the refactoring eliminates state setters (which eliminates function calls and callbacks being passed), or if components are made into uncontrolled components (which eliminates at least one change handler). Overall, the code transformations do not result in a significant increase in code complexity.

In the 23 applications, we observe only a slight increase in code complexity after applying the transformations suggested by *Reactor*.

5.6.7 *RQ6: What is the cost of the analysis?*

There are three aspects to the cost of *Reactor*: (i) the time to build the CodeQL database, (ii) the time taken to run all queries, and (iii) the time required to apply the transformations. Table 6 reports the first two in columns **DB Build Time** and **Total Query Time**, resp. The transformation runs in milliseconds, and the database is built in less than 15 seconds in all cases. The most expensive aspect of the analysis is the execution of CodeQL queries. On average, it takes 13 minutes to execute all queries on an application. The total time in all cases does not exceed 20 minutes.

Answer to RQ4: On average, it takes almost 13 minutes to transform one project.

5.7 THREATS TO VALIDITY

It is possible that our set of repositories do not represent all React applications, and the anti-patterns we identified may not cover all forms of re-rendering inefficiencies. To mitigate this, we randomly sampled 10,000 GitHub projects declaring React as a dependency, and selected subject applications from this pool. We further enriched our sample with repositories containing downloadable release assets to better capture real-world usage. Regarding the latter, our anti-patterns were derived from repeated manual investigations of components exhibiting superfluous re-rendering without visible UI changes. While

this process may have missed rarer or more subtle inefficiencies, the patterns we identified were prevalent across the broader 7,758-project corpus, suggesting they capture important and recurring problems. Future work could uncover additional inefficiencies; Reactor's design supports extending it with new detection and transformation rules as needed.

To assess the impact of the transformations on the number of rendering operations, we needed to observe the dynamic execution behavior of the subject applications. Testing React applications is difficult, and many React applications lack test suites, which was also the case for all of our subject applications. Therefore, we manually interacted with the applications by entering text in form fields, clicking buttons, etc. These interaction scenarios may not represent how actual users interact with the applications, which may introduce bias in our results. To mitigate this threat, we carefully documented the interactions and included them in supplemental material to enable reproducibility. Furthermore, we ensured that the interactions exercised as much of the application as possible, visiting every reachable page of each application.

The code transformation technique presented in this chapter suffers from similar threats to validity as Chapters 3 and 4. Concretely, the proposed code transformations are not guaranteed to preserve program behavior and are unsound. This unsoundness can primarily be attributed to unsoundness in the static analysis, which is inevitable due to the dynamic nature of JavaScript. Therefore, the proposed code transformations should be treated as suggestions and carefully reviewed by developers before application. In spite of this unsoundness, we found that *Reactor* proposed behavior-altering transformations in only one situation in our evaluation.

5.8 GENERALIZABILITY

While our research contributions center on formalizing and remediating re-rendering anti-patterns using CodeQL and declarative rewrite rules, we note that simpler syntactic rules could potentially be adapted into ESLint plugins. Such extensions could help developers adopt our findings directly into existing everyday workflows.

Reactor is currently tailored to React, but the problem of superfluous re-rendering is not unique to it. Other popular front-end frameworks such as Vue and Angular also follow unidirectional data flow models and exhibit similar rendering behaviors. However, there are several fundamental differences between these frameworks. For example, React and Vue both use a Virtual DOM for change detection but have different implementations of it, whereas Angular uses Zone.js [178]. Both Angular and Vue use HTML templates and prefer to extend HTML with directives and pipes, whereas React extends JavaScript with HTML instead. As a result of these differences, the solutions to the same problem

are different in each framework. As a concrete example, consider the case where a component should re-render only if some of its arguments change. To achieve this, React wraps the component in a call to `memo`, Angular uses the component configuration to set the `ChangeDetectionStrategy` to `OnPush`, and Vue would use the `v-memo` directive with the arguments as invalidation conditions. As such, it would be extremely difficult to develop a single solution that extends beyond a single framework, but the underlying ideas carry over. As future work, we plan to explore how our anti-pattern abstractions might be adapted to other popular frameworks, and whether automated transformations analogous to those in *Reactor* can be systematically derived.

5.9 CONCLUSION

We have identified 5 anti-patterns that commonly give rise to unnecessary re-rendering in React applications. For each anti-pattern, a set of code transformations was proposed to reduce how often React components are re-rendered. To detect cases where needless re-rendering may occur, we have defined static analyses for detecting instances of the anti-patterns, which are used in a set of rewrite rules that produce suggestions on how affected code fragments can be transformed. The static analyses are potentially unsound, so these suggestions must be reviewed carefully by a developer to ensure that behavior is preserved. These analysis and rewrite rules were implemented in a tool named *Reactor*.

In a large-scale evaluation of 7,760 projects, we detected at least one instance of our defined anti-pattern in 92.1% of projects, indicating their prevalence in real-world applications. In an empirical evaluation of *Reactor* on 23 subject React applications, a total of 313 instances of the anti-patterns were transformed, and the number of rendering operations was reduced by 33.3% on average. We also observed that transformations result in a small increase in cyclomatic complexity from 281.17 to 292.35 on average. This relatively small increase in complexity leads to an average decrease of 20.54% in time spent rendering, and additional case studies show that application responsiveness improves significantly as the number of components scales up. Moreover, we observed only one instance of unsoundness in our experiments. Moreover, we observed only one instance where unsoundness in analysis led to unwanted behavioral changes.

6

REWRITING WEBASSEMBLY BINARIES TO MITIGATE SECURITY VULNERABILITIES

6.1 INTRODUCTION

WebAssembly [73, 172] is a low-level bytecode format, originally designed for computationally intensive tasks in the browser, e.g., games, cryptography, and codecs. WebAssembly is typically used as a compilation target; that is, it is generally not written by hand, but compiled from different source languages like C/C++ [46, 151], and Rust [134]. Beyond browser-based applications, WebAssembly is increasingly popular on the server-side, both in Node.js [121], and in standalone runtimes like Wasmtime [168]. To facilitate practical applications, the WebAssembly System Interface (WASI) [164] allows WebAssembly programs to utilize a standardized system-call interface, including both file system and network access.

When dealing with code that runs on the web, security is of utmost importance due to considerations like parsing inputs from untrusted sources [73]. To this end, the WebAsembly designers have taken several steps to ensure some degree of safety: (*i*) each WebAssembly module is executed in a *sandboxed environment* that is separate from the host runtime, (*ii*) *memory safety* is guaranteed by providing each module with a linear memory that is disjoint from the execution stack, and (*iii*) providing *validation* of modules prior to execution to ensure type safety.

Despite these safeguards, prior work [77, 91, 92, 115] has shown that WebAssembly binaries remain vulnerable to classical memory corruption attacks. Lehmann et al. [91] identify exploitable patterns such as arbitrary memory writes and control-flow hijacking in compiled WebAssembly modules. Unlike native binaries, however, WebAssembly lacks common defenses such as stack canaries [48] or shadow memory schemes, leaving it exposed to exploit primitives that are otherwise well-defended in native settings.

Moreover, many applications that use WebAssembly modules only have access to the compiled binary, and not the source code. This makes it difficult to analyze the module for security vulnerabilities and even harder to mitigate them. In many cases, host applications only use a subset of the functionality provided by external dependencies, as discussed in Section 7.7. We posit this to be the case with WebAssembly modules as well and propose partial hardening of WebAssembly binaries to mitigate security vulnerabilities in part of the binary invoked by the host code. To this effect, we present a binary hardening approach that relies on client-provided executions or tests and aims to

mitigate memory corruption vulnerabilities in functions executed by these tests. This approach can potentially lead to unsoundness and unexpected program behavior or crashes if they test coverage is insufficient. However, a conservative approach would fail to apply any mitigation for most practical purposes due to unsoundness in analysis. Thus, we aim to thread the fine line between practical mitigation and soundness guarantees. We aim to mitigate memory corruption vulnerabilities and preserve program behavior for the execution flows present in client test cases while opening the door to potentially unsound behavior caused by unseen execution flows.

In this work, we present *Wassy*, a system for hardening WebAssembly binaries against memory corruption vulnerabilities through binary-level transformations. Our approach focuses on:

- **Memory Segmentation**: We introduce a read-only segment in linear memory this is enforced by write protection at runtime. We identify memory regions initialized at load-time and remap them to this immutable segment.
- **Per-instruction bounds enforcement**: We statically analyze memory access patterns, and insert upper-bound checks on stores within loops to prevent corruption of future addresses within the mutable segment.

We evaluate *Wassy* using a newly curated test suite of vulnerable WebAssembly programs compiled from the Juliet Test Suite [29], focusing on real-world Common Weakness Enumeration (CWE) patterns. These include memory corruption vulnerabilities such as buffer overflows, write-what-where conditions, and integer overflows. Because no publicly available dataset currently captures ground-truth WebAssembly vulnerabilities, we also release **Wasm-V**—a benchmark suite designed to support reproducible evaluation and future security research on WebAssembly binaries.

Our results show that *Wassy* is effective at preventing buffer overflows and memory corruption, while preserving benign functionality and introducing minimal overhead.

This chapter is organized as follows. Section 6.2 presents a motivating example that exhibits a buffer overflow vulnerability and its remediation. Section 6.3 describes our analysis and binary rewriting techniques and Section 6.4 overviews the implementation of our tool, *Wassy*. Section 6.5 presents our evaluation. We discuss potential limitations in Section 6.6 before concluding with a summary of our findings in Section 6.7.

6.2 MOTIVATION

The core WebAssembly specification has several features that are intended to contribute to a safe, sandboxed environment. This includes: (*i*) pointers are compiled to offsets into linear memory, so virtual addresses are hidden from applications, (*ii*) all control transfers

```
542 typedef void (*fptr)();
543
544 void print(int i); // implemented in JavaScript
545
546 void foo() { print(17); }
547 void bar() { print(18); }
548
549 char x[10] = "hello";  // can overflow into ptr below
550 fptr ptr[5] = {&foo,0,0,0,&bar};
551
552 void setX(char* val) {
553     strcpy(x, val); // vulnerable to a buffer overflow
554 }
555
556 void baz() { ptr[0](); }  // calls foo()
557
```

buffer.c

Figure 22: Illustration of a buffer overflow vulnerability in C. The `setX` function invokes the memory unsafe `strcpy` function that can overflow the character array x in Line 549. Overflowing x can overwrite the pointers on Line 550, or the instruction pointer in traditional binaries.

are first type-checked, (*iii*) there is no raw access to system calls, instead all interaction outside the VM is done through imports and exports, and (*iv*) the callstack is inaccessible, that is, return addresses from calls are not stored in memory accessible to applications. Despite these features, we have seen that WebAssembly still suffers from several types of vulnerabilities such as buffer overflows and cross-site scripting [91]. Next, we discuss buffer overflow vulnerabilities and why current WebAssembly language features are ineffective at mitigating them.

6.2.1 *Traditional Buffer Overflow Vulnerabilities*

A buffer overflow vulnerability arises when more data is written to a buffer than its allocated size. Depending on the location of the buffer overflow, it can result in significant consequences. For instance, a traditional stack-based buffer overflow can overwrite crucial data, such as return addresses of functions, stack frame pointers, local variables, function pointers, and other data on the stack. On the other hand, a traditional heap-based buffer overflow can overwrite data allocated on the heap. In some extreme cases, a buffer overflow may even redirect the execution flow of a program.

Figure 22 contains a program that is vulnerable to a stack-based buffer overflow. It defines the function `print` on Line 544 that is defined in the host environment (JavaScript). Two more functions, *foo* that prints 17 and *bar* that prints 18, are defined on Line 546 and Line 547 respectively. Line 549 creates a buffer of size 10 containing the string "hello" and Line 550 creates an array with function pointers to foo and bar. Line 556 invokes the function referenced by the first element of the pointer array, which points to `foo`. The `setX` function on Line 555 updates the buffer x using the `strcpy` function that can result in a buffer-overflow if the length of the input string is greater than the size of the buffer. In a traditional binary, this can be used to overwrite the function pointer referenced by `baz`, or the instruction pointer. However, this vulnerability is mitigated by security mechanisms such as Address Space Layout Randomization (ASLR) and stack canaries.

The buffer overflow vulnerability in Figure 22 persists when compiled to a WebAssembly binary, which does not have traditional mitigations such as ASLR and Stack Canaries. Although the stack is isolated from the program's memory in WebAssembly, an attacker can still redirect code execution by overwriting sensitive data stored on the heap, such as function table indices. The next section describes how the above vulnerability can result in control-flow hijacking in WebAssembly.

6.2.2 *Persistence of Traditional Buffer Overflows in WebAssembly*

Figure 23(a) shows the compiled WebAssembly module for the program shown in Figure 22. Only the relevant parts of the binary are shown for brevity. The function `print` is imported from JavaScript with the identifier `0`. The functions `foo`, `setX`, and `baz` are compiled to WebAssembly with the identifiers `2`, `4`, and `5` respectively. There are four important aspects of the compiled binary that contribute to the persistence of a buffer overflow vulnerability in WebAssembly:

1. The function pointers in Figure 22 are converted to function table indices. Line 576 maps indices to functions such that index 1 points to `foo` and index 2 points to `bar`.

2. In the absence of a stack, the character array `x` and function pointer array `ptr` are stored in linear memory. The string `"hello"` is stored in linear memory as an array of bytes on Line 577. Similarly, the function pointers to `foo` and `bar` are converted to function table indices and stored in linear memory on Line 578 and Line 579. Since the function table indices are of type `i32`, they occupy 4 bytes and are stored in little endian format. Figure 24(a) shows the memory layout for Figure 23(a).

3. The vulnerable `strcpy()` function is implemented as a `store8` instruction inside a loop as shown on Line 567.

```
(module
    (import "env" "print" (func (;0;) (type 1)))
    (func (;2;) (type 0) ;; foo
        i32.const 17
        call 0)
    (func (;4;) (type 1) (param i32) ;; setX
        ...
        loop  ;; label = @4
            ...
            i32.store8
            ...
    )
    ...
    (func (;5;) (type 0) ;; baz
        i32.const 1040
        i32.load
        call_indirect (type 0))
    ...
    (elem (;0;) (i32.const 1) func 2 3 1)
    (data (;0;) (i32.const 1024) "hello")
    (data (;1;) (i32.const 1040) "\01")
    (data (;2;) (i32.const 1056) "\02"))
```

(a) buffer.wasm

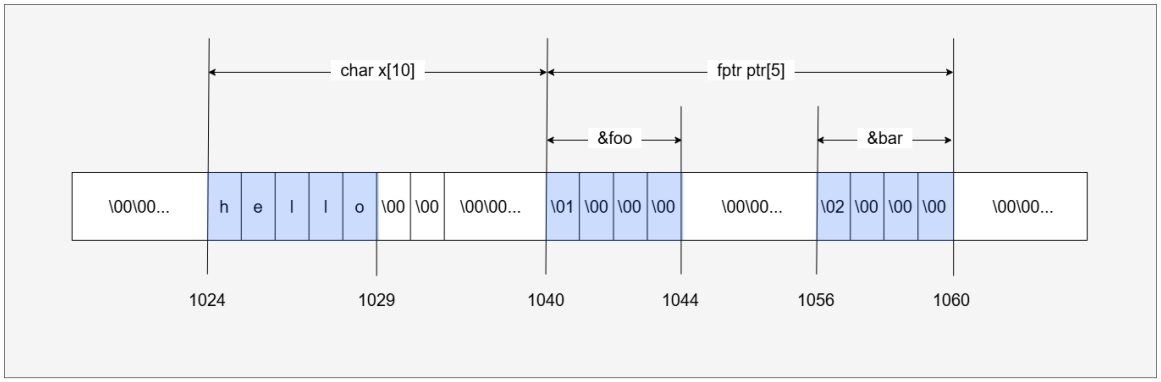
```
const MAX_SIZE = 17;

const importObj = { env : { print: console.log } };
const wasmBuffer = fs.readFileSync('buffer.wasm');
WebAssembly.instantiate(wasmBuffer, importObj).then(wasmObj => {
  const {memory, setX, baz } = wasmObj.instance.exports;

  baz(); // prints "17"

  const bufX = new Int8Array(memory.buffer, 0, MAX_SIZE);
  const encode = (str) => new TextEncoder().encode(str);
  const str = String.fromCharCode(2).repeat(MAX_SIZE);
  bufX.set(encode(str));
  setX(bufX.byteOffset); // buffer overflow, overwrites ptr[0]
  baz();  // prints "18"
});
```

(b) exploit.js

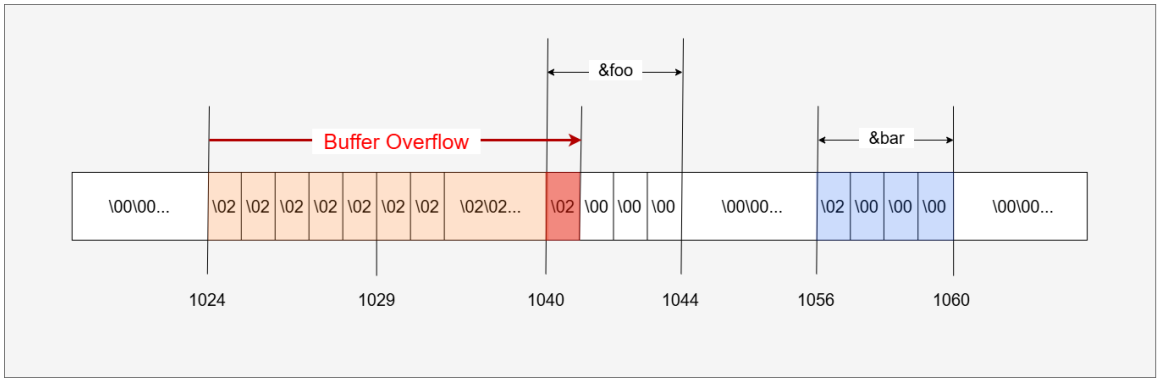
Figure 23: Illustration of a buffer overflow vulnerability in WASM where the `call_indirect` function can be redirected: (a) shows the function `baz` (func 5) using a `call_indirect` instruction to read a function table index at address 1040 in linear memory to invoke `foo()`, (b) shows the exploit code in JavaScript to overflow the buffer at address 1024 in linear memory with "
02" through `setX` (func 4), redirecting control flow from `foo()` to `bar()`.

4. Lastly, the array lookup in C is converted to a `load` instruction and the call to function pointer is implemented as a `call_indirect` instruction on Lines 573 and 574.

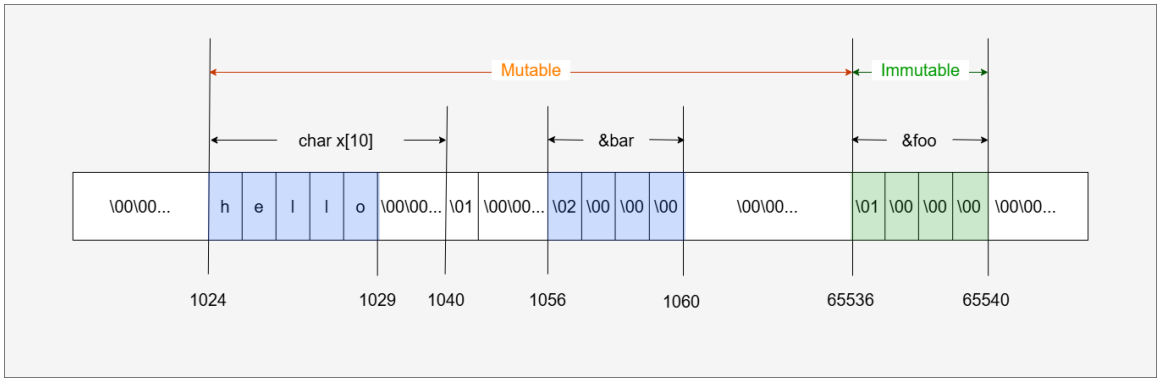
The function `setX` is vulnerable to a buffer overflow since it performs an unbounded write operation without checking the length of the input string. Since linear memory is a contiguous block of writable memory, overflowing the string `"hello"` would allow an attacker to overwrite the function table index at address 1040. Figure 23(b) exploits this vulnerability. Figure 23(b) invokes `setX` with a string containing the character \02 repeated 17 times. As a result, `setX` begins writing \02 in linear memory from address 1024, fills the buffer `x`, and overwrites the function table index at address 1040 with \02. Figure 24(b)



(a) Memory Layout of buffer.wasm



(b) Memory Layout after exploit



(c) Memory Layout after hardening

Figure 24: Memory layout of buffer.wasm:
(a) Memory Layout of buffer.wasm showing the string `"hello"` at address 1024, the function table index (01) to `foo` at address 1040, and the function table index (02) to `bar` at address 1056.
(b) Memory Layout of buffer.wasm after exploiting the vulnerability in `setX` and overflowing the string `"hello"` to overwrite the function table index `01` with `2`.
(c) Memory Layout of buffer.wasm after hardening. The read-only function table index `01` is now moved to the immutable segment at address 65536.

shows the state of linear memory after the exploit. Thus, now when the function `baz` in invoked, it retrieves the function table index 2 instead of 1, and invokes the function `bar` instead of `foo`.

The above example is analogous to a confirmed vulnerability [116], illustrating that buffer overflows do occur in practice. This instance demonstrates how traditional buffer overflow vulnerabilities can surface when the host code fails to adhere to the assumptions made by a WebAssembly library. This case shows how a buffer overflow can redirect control flow, leading to the invocation of an incorrect function. However, this is not the only issue that can arise; there could be other runtime errors, such as incorrect results due to data corruption and unexpected program termination due to a mismatch in the type of the function, as well.

6.2.3 *Preventing Buffer Overflows*

```
598 (type (;4;) (func (param i32) (param i32) (
        result i32) (result i32)))
599 ...
600 (func (;10;) (type 4) (param i32) (param
        i32) (result i32) (result i32)
601
602     block ;; label = @1
603       local.get 0     ;; get store addr
604       i32.const 65536  ;; readonly start
605       i32.lt_u        ;; addr < start
606       br_if 0 (;@1;)  ;; break if true
607       local.get 0     ;; get store addr
608       i32.const 65540  ;; readonly end
609       i32.gt_u        ;; addr > end
610       br_if 0 (;@1;)  ;; break if true
611       unreachable     ;; exit
612     end
613     local.get 0       ;; return addr
614     local.get 1)      ;; return value
615
```

(a) readonly_bound_check()

```
616 (type (;5;) (func (param i32) (param i32) (
        param i32) (result i32) (result i32)))
617 ...
618 (func (;14;) (type 5) (param i32) (param
        i32) (param i32) (result i32) (result
        i32)
619
620     block ;; label = @1
621       local.get 0      ;; get store addr
622       local.get 2      ;; get upper bound
623       i32.lt_u         ;; addr < upper
        bound
624       br_if 0 (;@1;)   ;; break if true
625       unreachable      ;; exit
626     end
627     local.get 0        ;; return addr
628     local.get 1)       ;; return value
629
```

(b) upper_bound_check()

Figure 25: Illustration of helper functions: **(a)** shows the function readonly_bound_check() that prevents writing to the read-only segment, **(b)** shows the function upper_bound_check() that prevents writing beyond a given address in linear memory. The functions are inserted in **buffer.wat** in Figure 26 on Line 650 and Line 651 respectively.

Linear memory in WASM is a contiguous block of memory to which data can be written and read. Data in linear memory is written from lower addresses to higher addresses. Although different kinds of data such as mutable values, constants, and function table indices reside in linear memory, no distinction is made between them. In the absence of an instruction pointer, control flow hijacking in WebAssembly occurs through indirect function calls that reference linear memory. To mitigate the impact of a buffer overflow, we introduce two protection mechanisms: (i) Memory Segmentation, and (ii) Per-instruction bounds enforcement.

MEMORY SEGMENTATION We introduce a read-only segment in linear memory, i.e., an immutable section that stores data that is never mutated. We enforce that the read-only segment is immutable by terminating the program if an attempt is made to write to this segment. The introduction of a read-only segment involves the following steps:

- Addition of a function (*readonly-bound-check*) to the binary that validates whether an address is within the read-only segment. If an attempt is made to write to such an address, the program is terminated by invoking `unreachable`.
- Remapping immutable values in linear memory to a new address in the read-only segment.
- Updating `load` instructions to reference the remapped address in the read-only segment.
- Invoking the (*readonly-bound-check*) function before every `store` instruction.

These changes ensure that immutable data in linear memory cannot be corrupted by a buffer overflow. In Figure 26, we move the function table index `\01` to the read-only segment, preventing control flow hijacking.

PER-INSTRUCTION BOUNDS ENFORCEMENT Although memory segmentation prevents the corruption of read-only data, buffer overflows can still occur within the mutable segment, primarily through `store` instructions in loops. `store` instructions within loops are analogous to functions such as `strcpy` in C and are most likely candidates for improperly bound writes. `store` instructions outside loops can perform directed writes, which are harder to detect and prevent. Nevertheless, all store instructions are prevented from writing to the read-only segment. To mitigate memory corruption within the mutable segment, we introduce per-instruction bound enforcement on such store instructions within loops. This involves the following steps:

- Addition of a function (*upper-bound-check*) that validates that an address is less than a specified upper bound. If an attempt is made to write beyond this bound, the program is terminated by invoking `unreachable`.

- Identifying the upper bound for `store` instructions within loops and invoking the (*upper-bound-check*) with the identified bound before the `store` instruction.

These changes would limit the impact of buffer overflows within the mutable segment of linear memory. In Figure 26, this would prevent the corruption of linear memory address `1040` and beyond.

6.2.4 *Hardening buffer.wat*

This section describes the application of previously described protection mechanisms to the vulnerable binary in Figure 23.

The approach adds two new functions to the WebAssembly binary: (*i*) a function `ReadonlyBoundCheck` to validate whether an instruction is writing to the read-only section of memory, as shown in Figure 25(a), and (*ii*) a function `UpperBoundCheck` to check if an instruction is attempting to write beyond a specified linear memory address, as shown in Figure 25(b). The start address of the read-only segment is the starting address of the next unused page in linear memory and the size of the read-only segment is just large enough to accommodate all the immutable data. In our example, the last used memory address is 1056, which is on the first page that ends at address 65535, and the only known immutable data is the function table index at address 1040, which is 4 bytes in size. As a result, Figure 25(a) defines the start of the read-only segment at address 65536 and the end at 65540. If a store instruction attempts to write within this address range, the program is terminated. This function is defined in the WebAssembly binary as function `10` on Line 650 in Figure 26. Figure 25(b) compares a linear memory address against the specified upper limit and terminates the program if an attempt is made to write to an address beyond this limit. This function is defined in the WebAssembly binary as function `14` on Line 651 in Figure 26. In addition to these functions, Figure 26 copies read-only values in linear memory to addresses within this range. The instruction on Line 657 initializes the address `65536` in the read-only section with the value `\01` from address `1040`. The new layout of linear memory after binary hardening is shown in Figure 24(c). The above changes provide the pre-requisites necessary to mitigate the buffer overflow. Next, we need to modify the existing instructions to utilize bound-checking functions and modified linear memory to mitigate the buffer overflow.

First, we need to update the `load` on Line 647 to read from the read-only segment of linear memory. This is achieved by updating the input value of the `load` instruction by adding an offset to the previous address on Line 645 and Line 646, such that the new effective address points to *65536*. Next, we need to add validations to the `store` instructions to prevent them from writing over independent allocations and the read-only segment. To prevent writing to the read-only segment, we invoke the `readonly_bound_check()` func-

```
630 (module
631   ...
632     (func (;4;) (type 1) (param i32) ;; function definition: setX()
633         ...
634         loop  ;; label = @4
635             ...
636             i32.const 1040 ;; upper bound on store
637             call 14 ;; invoke upper_bound_check()
638             call 10 ;; invoke readonly_bound_check()
639             i32.store8
640         ...
641     )
642     ...
643     (func (;5;) (type 0) ;; function definition: baz()
644         i32.const 1040
645         i32.const 64496 ;; offset to 65536
646         i32.add ;; Effective address = 65536
647         i32.load
648         call_indirect (type 0))
649     ...
650     (func (;10;) ...) ;; function definition: readonly_bound_check()
651     (func (;14;) ...) ;; function definition: upper_bound_check()
652     ...
653     (data (;0;) (i32.const 1024) "hello")
654     (data (;1;) (i32.const 1040) "\01")
655     (data (;2;) (i32.const 1056) "\02"))
656     ;; readonly segment starts at 65536
657     (data (;3;) (i32.const 65536) "\01") ;; copy \01 from addr 1040
658
```

Figure 26: Illustration of security measures implemented to mitigate the buffer overflow in **buffer.wat**: (i) The function readonly_bound_check() from Figure 25(a) is inserted in the binary on Line 650. (ii) The function `upper_bound_check()` from Figure 25(b) is inserted in the binary on Line 651. (iii) The hardened binary uses the functions readonly_bound_check() and upper_bound_check() along with read-only address remapping on Line 646 to mitigate buffer overflows and control flow hijacking.

tion before the store instruction as shown on Line 638. Since we know that the data `\01` is independent of the string `hello` in linear memory, we set the upper bound of the `store` instruction as *1040*. To prevent overwriting the data at address *1040*, we inject the constant *1040* as shown on Line 636 and invoke the `upper_bound_check()` function before the store instruction as shown on Line 637. Thus, the introduction of memory segmentation and per-instruction bounds checking together mitigate the buffer overflow as shown in Figure 26.

Concretely, mitigating the buffer overflow in Figure 23(a) involves the following changes highlighted in Figure 26:

1. Adding two utility functions to validate linear memory addresses as shown on Line 650 and Line 651.
2. Initialization of the constant "\01" in the read-only segment at the memory location *65536* as shown on Line 657.
3. Updating the linear memory reference for the `load` instruction by adding the constant *64496* to *1040* on Line 646.
4. Adding a constant upper bound with the value *1040* for the store instruction in the *SetX* function on Line 636.
5. Invoking function *14* on Line 637 to ensure that the store instruction does not violate the upper bound.
6. Invoking function *10* on Line 638 to ensure that the store instruction does not write to the read-only segment.

6.3 APPROACH

The previous section demonstrated a buffer overflow vulnerability along with the proposed changes to mitigate it. In this section, we present the following three step approach for automatically hardening a WebAssembly binary:

COLLECT EXECUTION TRACES AND STATIC METADATA The first step in our approach is to collect execution traces and memory access information based on test cases for a WebAssembly module. In addition to the dynamic analysis, we infer properties about the WebAssembly module such as the number of functions and the instructions present inside loops using static analysis.

IDENTIFY TRANSFORMATION CANDIDATES AND COMPUTE BOUNDS The information collected using dynamic analysis and static analysis is combined to produce values

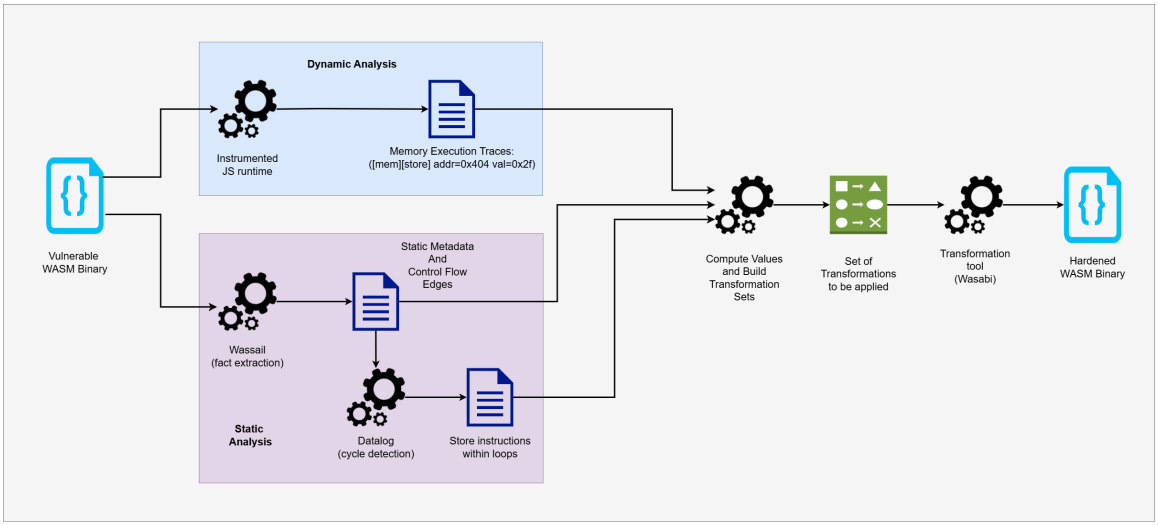


Figure 27: High level architecture of Wassy

that are necessary for the binary transformation. This includes identifying valid indices for injecting the helper functions, determining the address and size for the read-only segment, and identifying upper bounds on store instruction present in loops.

TRANSFORM THE BINARY The last step is to analyze the transformation sets and applying the transformations. This includes injecting helper functions into the binary, remapping linear memory, and injecting bound checks on store instructions.

Figure 27 shows the high-level architecture of *Wassy*.

6.3.1 *Assumptions for Behavior Preservation*

Wassy is a tool that operates on the WebAssembly binary and aims to mitigate security vulnerabilities in cases where the source code is not available. *Wassy* uses a combination of static and dynamic analysis to inject bounds checking and enforce memory segmentation. Ideally, we would want soundness guarantees about the transformations but there are several challenges which make this infeasible. Once a program is compiled to a WebAssembly module from a source language, a lot of useful information is lost along the way. For example, the source language contains information about the size and type of variables and can distinguish between values and pointers. However, this information is no longer available in WebAssembly, where there are no explicit boundaries between variables and no distinction between integers and pointers. This lack of type information,

combined with external dependencies add to unsoundness in static analysis, especially when dealing with values. If we prioritize behavior preservation, the unsoundness in static analysis will almost always mark all memory as mutable, and thus prevent any meaningful mitigation. As a result, we complement the static analysis with dynamic analysis by collecting execution traces based on client test cases. While the static analysis suffers from unsoundness, the dynamic analysis is often incomplete. If we prioritize security based on the dynamic analysis, most mitigated binaries would end up being overly restrictive, permitting only the execution flows present in the test cases, making them unusable in practice.

We aim to find a middle ground between the unsoundness of static analysis and incomplete dynamic analysis such that we can mitigate some security vulnerabilities without being overly restrictive while minimizing deviation from expected behavior. To achieve this, we harden the binary (and linear memory) for the execution traces observed during test execution. We choose to be permissive about linear memory addresses by treating then as mutable in situations where sufficient information is unavailable. This approach ensures that the transformation is behavior preserving for the execution traces observed, but makes a best effort to preserve behavior for executions not seen before. The correctness of the transformed binary is heavily reliant on the dynamic analysis being complete, i.e., we assume that the test cases have sufficient coverage all permitted use cases. Insufficient coverage of permitted use cases can lead to potentially unsound behavior and program crashes. This approach assumes the presence of adequate coverage of permitted executions for the transformations to be sound. Considering the lack of source level information and unsoundness in analysis, we believe it is a reasonable requirement to expect adequate coverage of permitted execution to preserve behavior while mitigating security vulnerabilities. This approach is similar to Control Flow Integrity (CFI) [33, 114, 119, 120, 174] enforcement approaches in the sense that control flow will always be redirected to indices in linear memory that appear to be read-only. However, this approach does not strictly enforce CFI and can result in some previously unseen executions. Future work may include enforcing integrity of control flow by replacing `call_indirect` instructions with `call` instructions where we observe the function table index to be constant during program execution.

6.3.2 *Collect Execution Traces and Static Metadata*

Wassy performs a hybrid of dynamic instrumentation and static control-flow analysis to guide and evaluate binary transformations that harden WebAssembly WASM modules.

Each WASM module is paired with a JavaScript driver that invokes the module. *Wassy* modifies this driver to insert instrumentation hooks, enabling the capture of runtime

behavior. During execution, all linear memory loads and stores are recorded along with their effective address, value, and context. These memory traces are structured as entries like `[mem][store] addr=0x404 val=0x2f`.

In addition to runtime logging, *Wassy* performs static analysis over each WASM module to extract control-flow graphs and memory access metadata. This is done by translating each module into a Datalog fact representation. Control-flow edges (`cfg_edge.facts`) are analyzed to identify cyclic paths—i.e., loops—using a graph cycle detection pass. Memory operations from `memory_access.facts` are then cross-referenced to determine which store instructions occur within loops.

All instrumentation outputs—including memory access traces, stdout, crash state, and transformation metadata—are stored in a relational database. This unified structure supports evaluation pipelines that compare baseline behavior against transformed variants for correctness and mitigation, as described in Section 6.5.

6.3.3 *Identify Transformation Candidates and Compute Bounds*

The information collected in Section 6.3.2 is used to compute the labels and constants necessary for applying the transformations.

IDENTIFIERS FOR HELPER FUNCTIONS One of the transformations involves injecting helper functions into the binary. The count of existing functions in the module obtained from static analysis is used to generate the identifiers for helper functions.

UPPER BOUNDS FOR STORE INSTRUCTIONS IN LOOPS For each store instruction located in a loop, *Wassy* locates the next memory-accessing instruction outside the loop (typically a load) and uses its lowest effective address as an inferred upper bound. The lowest effective address is the lowest address accessed by the instruction during dynamic analysis. This information is used to insert instruction-specific bounds checks before the corresponding store. These checks prevent stores within loops from writing beyond dynamic memory ranges observed to be valid.

BOUNDS FOR READ-ONLY SEGMENT Separately, memory regions that are initialized but not written to after startup are inferred as immutable. These are remapped to protected read-only segments by transforming the WASM binary and applying custom enforcement functions. This ensures that accidental or malicious writes to static data structures (e.g., strings, constants) are prevented at runtime. The dynamic analysis informs the starting address (L) of the read-only segment and is computed using the formula `math.ceil((last_offset + 1) / PAGE_SIZE) * PAGE_SIZE`. The size of the read-only segment (S) is computed based on the number of bytes in linear memory that are accessed by `load` instructions but not by `store` instructions. The upper bound (U) of the

read-only segment is computed as `L + S`. The lower and upper bound are referenced in the rewrite rule READONLY-BOUND-CHECK as L and U respectively.

Once all the necessary values are computed and transformation candidates identified, we apply the transformations.

6.3.4 *Rewriting the Binary*

In this section, we present the proposed transformations formally as a set of declarative rewrite rules that can be applied to any WebAssembly binary. The approach consists of two mechanisms for overflow mitigation (i) Memory Segmentation, and (ii) Per-instruction bounds enforcement. The rewrite rules rely on certain facts extracted or computed based on the hybrid static and dynamic analysis. Our tool (*Wassy*) implements the necessary static and dynamic analyses along with the binary transformations to realize these rules. The rewrite rules can be found in Figure 28 and Figure 29.

6.3.4.1 *Memory Segmentation*

This section describes the necessary transformation rules for introducing memory segmentation and enforcing a read-only segment in linear memory.

READONLY-BOUND-CHECK: Memory segmentation involves creating a read-only segment in linear memory. This rule introduces a set of functions that check whether a linear memory address references this immutable section of memory.

This function is a write protection mechanism and is invoked before a `store` instruction. Let the bound checking function being inserted be defined as `func(A){B}`, where A represents the parameters of the function and B represents the body of the function. The function defines two parameters in A := (`loc : i32, val : i32`), *loc* and *val*, such that the linear memory address passed to the store instruction will be bound to loc and the value bound to *val*. The body B of the function contains two constants L and U that represent the start and end of the immutable segment, respectively. The values of L and U are computed based on the dynamic analysis and described in Section 6.3.3 The body of the function compares the input address *loc* with the bounds of the read-only segment and if *loc* is within the read-only segment, an error is thrown. If the input address is outside the read-only segment, the function pushes the input linear memory address and the value back on the stack. Figure 25(a) shows the implementation of the bound checking function in WAT syntax. This function is inserted into the binary after all other functions for every WebAssembly module. Four variants of this function are injected into the binary to handle the four different types of values, namely *i32*, *i64*, *f32*, and *f64*.

$$\frac{\begin{array}{c}L := \mathit{Const}(\mathit{lower\_bound}) \qquad U := \mathit{Const}(\mathit{upper\_bound}) \\ \text{Let A be the parameters of the helper function,} \\ \mathit{such\ that}\ A := (\mathit{loc}: \mathit{i32},\ \mathit{val}: \mathit{i32}) \\ \text{Let B be the body of the helper function,} \\ \mathit{such\ that}\ B := \mathit{if}\ (L < \mathit{loc} < U)\ \mathit{then}\ (\mathit{loc},\ \mathit{val})\ \mathit{else}\ \mathit{unreachable} \\ \mathit{then}\ \texttt{func}_k := \texttt{func(A)\{B\}} \\ \forall m : \mathit{WebAssembly\ Modules} \qquad \mathit{where}\ \texttt{func}_k \notin m\end{array}}{\texttt{module}((\texttt{func}_1; ...; \texttt{func}_n;)) \longrightarrow \texttt{module}((\texttt{func}_1, ..., \texttt{func}_n; \texttt{func}_k);)}$$

(READONLY-BOUND-CHECK)

$$\frac{\begin{array}{c}S_{load} : \text{Set containing elements (i, l, v) from execution traces for load instructions} \\ S_{store} : \text{Set containing elements (i, l, v) from execution traces for store instructions} \\ \mathit{where},\ i = \text{instruction},\ l = \text{address},\ v = \text{value} \\ \mathrm{Loc} : (i, l, v) \longrightarrow l \\ S_{readonly} := \mathrm{Loc}(S_{load}) \setminus \mathrm{Loc}(S_{store}) \\ l' : \text{ Fresh address in read-only segment} \\ l \in \mathrm{Loc} \qquad \mathit{where}\ l\ \mathit{not\ previously\ remapped}\end{array}}{(\texttt{module}(...(\texttt{data l v}); ...) \longrightarrow (\texttt{module}(...(\texttt{data l v}); (\texttt{data l' v}); ...)}$$

(INITIALIZE-READONLY-SEGMENT)

$$\frac{\begin{array}{c}f \in S_{ReadonlyBoundCheck} \qquad instr \in S_{store} \\ k := \mathrm{Identifier}(f) \qquad \text{such that, } \mathrm{typeof}(instr) == \mathrm{typeof}(f) \\ \forall instr \in S_{store} \qquad \mathit{where}\ \texttt{instr}\ \mathit{not\ preceded\ by}\ (\texttt{call k})\end{array}}{\texttt{func}(...(\texttt{i32.store}); ...) \longrightarrow \texttt{func}(...(\texttt{call k}); (\texttt{i32.store}); ...)}$$

(INSERT-READONLY-SEGMENT-CHECK)

$$\frac{\begin{array}{c}S_{load\_remap} : \text{Set of remapped (i, l, l'), where} \\ \text{i = load instruction, l = old address, l' = remapped address} \\ \text{such that,} \qquad \text{data accessed by l has moved to l'} \\ \text{and offset := l' - l} \\ \forall (i, l, l') \in S_{load\_remap} \\ \mathit{where}\ \texttt{i}\ \mathit{not\ preceded\ by}\ (\texttt{const offset});\ (\texttt{i32.add});\end{array}}{\texttt{func}(...(\texttt{i32.load}); ...) \longrightarrow \texttt{func}(...(\texttt{const offset}); (\texttt{i32.add}); (\texttt{i32.load}); ...)}$$

(REMAP-READONLY-LOADS)

Figure 28: Rewrite rules describing the transformations for memory segmentation.

INITIALIZE-READONLY-SEGMENT: This rule involves copying values from a location in linear memory to a section that is defined as read-only.

Let S_{load} and S_{store} be a set of execution traces for *load* and *store* instructions respectively, and let **Loc()** be an auxiliary function that returns the linear memory address accessed by these instructions. Then, the set of read-only locations $S_{readonly}$ is a set of all linear memory addresses accessed by a *load* instruction, but not by a *store* instruction i.e., the subtraction of S_{store} from S_{load}. For every location l in $S_{readonly}$, we assign a fresh memory location l' in the read-only segment and initialize l' with the value present at location l in linear memory.

INSERT-READONLY-SEGMENT-CHECK: This rule inserts a function call to the function defined by the rule *readonly-bound-check*. Let $S_{ReadonlyBoundCheck}$ be a set of bound check functions and S_{store} be a set of store instructions in the program. Let k be the identifier for the read-only bound check function for a `store` instruction of type `i32`. Then, for every store instruction in S_{store}, we insert a call to function k before the store instruction.

REMAP-READONLY-LOADS: This rule updates `load` instructions to reference the newly created read-only segment in linear memory instead of a mutable address.

Let S_{load_remap} be a set of $(\mathtt{i}, \mathtt{l}, \mathtt{l}')$ such that `load` instruction `i` accesses the linear memory address `l` and the value at location `l` has been copied to a new linear memory address `l'` in the read-only section of linear memory. Then, for every instruction `i`, we compute the offset, such that, the addition of `l` and `offset` results in the new effective address `l'`. The transformation is applied for every `i` where a constant `offset` and `i32.add` instruction is injected before the load instruction.

6.3.4.2 *Per-instruction bounds enforcement*

This section describes the necessary transformation rules for introducing per-instruction bounds enforcement for mitigating buffer overflows in the mutable segment.

UPPER-BOUND-CHECK: Similar to the READONLY-BOUND-CHECK rule, this rule inserts a set of functions into the binary that enforce a dynamic upper bound on the linear memory addresses that may be mutated by a *store* instruction.

Let the bound checking function being inserted be defined as `func(A){B}`, where A represents the parameters of the function and B represents the body of the function. The function defines three parameters in $A := (\mathtt{loc} : \mathtt{i32}, \mathtt{val} : \mathtt{i32}, \mathtt{upper\_bound} : \mathtt{i32})$, *loc*, *val*, and *upper_bound*, such that the linear memory address passed to the store instruction will be bound to loc, the value passed to the store instruction will be bound to *val*, and the highest memory address that the store instruction is permitted to write will be bound to *upper_bound*. The *upper_bound* for the store instruction is computed as described in Section 6.3.3. The body B of the function compares the location *loc* with the upper bound

$$\frac{\begin{array}{c}\text{Let A be the parameters of the helper function,}\\ \text{such that } A := (\text{loc}: \text{ i32}, \text{ val}: \text{i32}, \text{ upper\_bound}: \text{ i32})\\ \text{Let B be the body of the helper function,}\\ \text{such that } B := \text{if } (\text{loc} < \text{upper\_bound}) \text{ then } (\text{loc}, \text{ val}) \text{ else unreachable}\\ \text{func}_k := \text{func}(A)\{B\}\\ \forall m : \text{WebAssembly Modules} \qquad \text{where } \texttt{func}_k \notin m\end{array}}{\texttt{module}((\texttt{func}_1; ...; \texttt{func}_n;)) \longrightarrow \texttt{module}((\texttt{func}_1, ..., \texttt{func}_n; \texttt{func}_k);)}$$

(UPPER-BOUND-CHECK)

$$\frac{\begin{array}{c}f \in S_{\textit{UpperBoundCheck}} \qquad \text{Identifier} : \text{instr} \rightharpoonup \text{identifier} \qquad \text{Identifier}(\text{func k}) \longrightarrow k\\ S_{\textit{mem\_access}} : \text{Set of (i, l), where } i \in \{\text{store}, \text{load}\} \text{ such that}\\ i = \text{unique instruction}, \ l = \text{lowest memory address accessed}\\ \forall (i_1, l_1) \in S_{\textit{mem\_access}}\{\exists (i_2, l_2) \in S_{\textit{mem\_access}} \mid (l_1 < l_2) \wedge (i_1 \neq i_2) \wedge (i_1 = \text{store})\} \text{ then,}\\ L: \text{ Collection of all } l_2 \text{ for every } i_1 \qquad S_{\textit{upper\_bounds}} = \{i_1, \min(L)\}\\ \text{Now, } S_{\textit{upper\_bounds}} : \text{Set of (instr, ub), where instr == store}\\ k := \text{Identifier}(f) \qquad \text{such that, typeof}(\text{instr}) == \text{typeof}(f)\\ \forall (\text{instr}, \text{ub}) \in S_{\textit{upper\_bounds}} \qquad \text{where instr is inside loop}\\ \text{where } \texttt{instr} \text{ not preceded by } (\texttt{const ub}); \ (\texttt{call k});\end{array}}{(\texttt{loop}(...(\texttt{i32.store}); ...)) \longrightarrow (\texttt{loop}(...(\texttt{const ub}); \ (\texttt{call k}); \ (\texttt{i32.store}); ...))}$$

(INSERT-UPPER-BOUND-CHECK)

Figure 29: Rewrite rules describing the transformations for per-instruction bounds enforcement.

upper_bound and throws an unreachable error if the location is not within the bound. If the input address is within the bound, the function pushes the linear memory address and the value back on the stack. Figure 25(b) shows the implementation of the bound checking function in WAT syntax. This function is inserted into the binary after all other functions for every WebAssembly module. Four variants of this function are injected into the binary to handle the four different types of values, namely *i32*, *i64*, *f32*, and *f64*.

INSERT-UPPER-BOUND-CHECK: This rule identifies the highest linear memory address that a *store* instruction can mutate and inserts a call to a function defined by the rule *upper-bound-check* to enforce it.

First, we need to find the upper bound for *store* instruction. Let S_{mem_access} be a set of pairs consisting of a unique *load* or `store` instruction `i` and the lowest address in linear memory `l` accessed by `i`. Each unique linear memory address in this set is considered the beginning of a new allocation in linear memory. Then, for every unique instruction (i_1, l_1) in S_{mem_access} that is a *store* instruction, we find another pair of (i_2, l_2), such that i_2 is a different instruction and l_2 is higher in linear memory than l_1. For every instruction i_1, we collect all locations l_2 that satisfy the previous condition in a collection L. Thus, for instruction i_1, the collection L contains the starting addresses of data allocations on the heap by other instructions that may get written over by an overflow. Consequently, we select the closest allocation $\mathtt{min}(L)$ from this set as the upper bound for i_1 and add it to the set S_{mem_access} as a pair (i_1,$\mathtt{min}(L)$). Finally, for every instruction in S_{mem_access} that is present inside a loop, we identify the appropriate function *k* from *upper-bound-check* and apply the code transformation. The code transformation adds the static upper bound as a constant and a call to function k immediately before the store instruction.

6.4 IMPLEMENTATION

This approach is implemented in a tool called *Wassy*. The dynamic analysis is performed by hooking into the JavaScript harness to extract runtime memory access. The static analysis is implemented in Wassail by extracting control-flow graphs and memory access metadata. The control-flow graphs and metadata are expressed as facts which are consumed by Datalog to perform a cycle detection pass to identify loops. The binary transformation is implemented in Rust using Wasabi to parse code, manipulate ASTs, and emit transformed code.

6.5 EVALUATION

In this section we evaluate *Wassy* by addressing the following research questions:

- **RQ1:** To what extent do runtime memory traces extracted from original WASM executions provide sufficient information to support effective security hardening?
- **RQ2:** To what extent does *Wassy* successfully mitigate memory corruption vulnerabilities in WASM binaries?
- **RQ3:** Does *Wassy* preserve benign program functionality when applied to non-vulnerable (good) binaries?
- **RQ4:** What is the runtime overhead introduced by our binary transformations and memory protections?

6.5.1 *Testset and Experimental Setup*

We evaluate *Wassy* using the Wasm-V [165] dataset, a curated suite of WebAssembly modules derived from C programs with known memory safety issues. The dataset contains the following 6 categories of memory safety issues:

- **CWE-121 (Stack-based Buffer Overflow) [36]**: A stack-based buffer overflow condition is a condition where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).
- **CWE-122 (Heap-based Buffer Overflow) [37]**: A heap overflow condition is a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory, generally meaning that the buffer was allocated using a routine such as malloc().
- **CWE-123 (Write-what-where Condition) [38]**: Any condition where the attacker has the ability to write an arbitrary value to an arbitrary location, often as the result of a buffer overflow.
- **CWE-124 (Buffer Underwrite) [39]**: The product writes to a buffer using an index or pointer that references a memory location prior to the beginning of the buffer.
- **CWE-190 (Integer Overflow or Wraparound) [40]**: The product performs a calculation that can produce an integer overflow or wraparound when the logic assumes that the resulting value will always be larger than the original value.
- **CWE-680 (Integer Overflow to Buffer Overflow) [41]**: The product performs a calculation to determine how much memory to allocate, but an integer overflow can occur that causes less memory to be allocated than expected, leading to a buffer overflow.

Each C program is compiled to a WebAssembly module along with two input variations:

- A **good variant**: that follows a safe execution path representative of benign use.
- A **bad variant**: that triggers a vulnerability during execution (e.g., overflow, underflow) indicative of malicious use.

The dataset consists of 7,519 WebAssembly modules, of which, 2,389 were discarded because: (i) they contained unsupported instructions such as `memory.copy` or sign-extension operations, or (ii) they exhibit unsafe behavior on the good variant. The evaluation is performed on the remaining set of 5,130 WebAssembly modules. Table 7 shows a breakdown of discarded modules by CWE.

The evaluation pipeline consists of:

1. **Static Analysis**: Extract control flow and memory access information from the input WASM binary. We use a Datalog-based fact generation pipeline to identify Control Flow Graph (CFG) cycles and memory-modifying instructions.
2. **Dynamic Instrumentation**: Execute each WASM module using a custom-instrumented JavaScript harness that logs runtime memory accesses.
3. **Transformation**: Harden the bad variant by:
   - Introducing Memory Segmentation.
   - Introducing Per-instruction Bounds Enforcement.
4. **Post-transformation Execution**: Execute the hardened binary and collect memory access traces.

All experiments were conducted on a 6-core 2.4GHz Intel Xeon CPU, with 32GB of RAM, running Ubuntu 22.04. Execution times are averaged over multiple runs to account for variance.

6.5.2 *RQ1: Runtime Trace Collection Effectiveness*

Our first research question investigates whether memory access traces collected from unmodified WebAssembly binaries are sufficiently informative to drive effective hardening transformations. Specifically, can *Wassy* reliably extract memory traces that provide the necessary insight for secure binary rewriting?

To answer this question, we execute each WebAssembly binary using a modified JavaScript harness that records memory access events during runtime. Instead of altering the binaries themselves, *Wassy* instruments the surrounding JavaScript environment to observe address-value pairs dynamically. We assess the traces based on two criteria:

- **Trace Completion:** Whether memory events are consistently recorded for all test cases.
- **Trace Fidelity:** The average number of memory events per binary, indicating trace richness and analytical utility.

Trace richness is one many metrics that can be used to measure the quality of trace collection. Augmenting trace richness with other coverage metrics such as the percentage of memory access instructions covered, and the percentage of linear memory covered would increase confidence in the effectiveness of our dynamic analysis. Table 7 presents trace collection statistics across CWE categories.

Table 7: Breakdown of discarded WASM Modules and Memory Trace Collection Results by CWE Category. The first row reads: *The **CWE-121** category contains **2074** WASM modules, **627** of which had to be discarded and **1447** were transformed. During dynamic analysis, **1,445,483** execution traces were captured across the 2074 modules with **696.95** linear memory accesses recorded for each module on average.*

	WASM Modules			Trace Collection	
CWE	Total	Discarded	Final	Traces Captured	Avg. Observations
CWE-121	2074	627	1447	1,445,483	696.95
CWE-122	2642	770	1872	2,327,942	881.12
CWE-123	39	39	0	22,372	573.64
CWE-124	787	201	586	671,409	853.12
CWE-190	1892	667	1225	1,453,442	768.20
CWE-680	85	85	0	73,620	866.12
Total/Average	7519	2389	5130	5,994,268	797.21

Across all evaluated categories, *Wassy* successfully collected runtime memory traces for every binary. Further, the average number of memory observations, i.e., the execution traces for load and store instructions per binary indicates that the traces are sufficiently detailed to inform transformation logic. Categories such as CWE-122 and CWE-124 yielded the richest traces, suggesting extensive memory activity during execution—critical for supporting fine-grained hardening.

Answer to RQ1: *Wassy* successfully collects memory traces for all modules and generates sufficiently rich memory logs to support downstream hardening across all evaluated WebAssembly binaries.

6.5.3 *RQ2: Hardening Effectiveness*

The second research question examines the security impact of applying *Wassy*'s hardening transformations. Specifically, can *Wassy* prevent memory corruption vulnerabilities from manifesting at runtime?

To answer this question, we consider 3 test cases for each WASM module:

- **Base Good:** A vulnerable version that is executed with safe inputs that do not trigger any vulnerability. This serves as the ground truth.
- **Base Bad:** A vulnerable version that is executed with unsafe inputs that trigger memory corruption.
- **Hardened Bad:** The transformed version of the vulnerable binary after applying *Wassy* that is executed with the same unsafe inputs as in the case of *base bad*.

Each test case is executed under our instrumentation harness to collect memory traces and standard output.

After hardening, we analyze the memory traces and classify the behavior based on the following 3 categories:

- **Ameliorated**: The hardened binary exhibits behavior (execution trace) that is identical to the base good when invoked with the unsafe inputs, indicating that the unsafe inputs did not corrupt memory.
- **Detected**: The hardened binary detects unsafe behavior and terminates the program by executing a trap.
- **Unmitigated**: The hardened binary neither executes a trap nor has a trace identical to the *base good* case. This is an over-approximation of unsafe behavior since any mitigated binary with a different execution trace would be considered unmitigated.

The vulnerability is considered mitigated if the execution is classified as either *ameliorated* of *detected*. Table 8 summarizes effectiveness by CWE category.

Wassy successfully mitigated memory corruption vulnerabilities across all major CWE classes. The combination of Memory Segmentation and per-instruction bounds enforcement proved effective in capturing critical memory violations. We observed that in 62.50% of the cases, the hardened binaries mirrored the benign behavior of the good variant—producing

Table 8: Breakdown of Hardening effectiveness and Runtime Overhead by CWE Category. The first row reads: *The **CWE-121** category contains **1447** that were transformed. After hardening, **859 (59.36%)** modules presented benign behavior under malicious inputs and a buffer overflow was detected in **10 (00.69%)** modules, both of these cases are considered mitigated. The average execution time for the WASM modules before hardening was **0.3144** seconds and after hardening was **0.3243** seconds indicating an overhead of **3.15%**.*

		Hardening Effectiveness			**Runtime Overhead**		
CWE	Modules	Ameliorated	Detected	Total Mitigated	Before (s)	After (s)	Overhead
CWE-121	1,447	859 (59.36%)	10 (00.69%)	**869** (60.06%)	0.3144	0.3243	3.15%
CWE-122	1872	1,805 (96.42%)	4 (00.21%)	**1,809** (96.63%)	0.3169	0.3250	2.55%
CWE-123	0	–	–	–	–	–	–
CWE-124	586	434 (74.06%)	6 (1.02%)	**440** (75.09%)	0.3160	0.3259	3.13%
CWE-190	1,225	109 (8.90%)	532 (43.43%)	**641** (52.33%)	0.3161	0.3253	2.93%
CWE-680	0	–	–	–	–	–	–
Total/Average	5,130	3,207 (62.50%)	552 (10.76%)	3,759 (73.29%)	**0.3159**	**0.3251**	**2.94%**

similar traces or outputs. In 10.76% of the cases, the memory corruption vulnerability was prevented by terminating execution through a trap. The vulnerabilities remain unmitigated in 26.72% of the cases and can generally be attributed to incomplete trace coverage (e.g., missed store instructions or unexercised paths), indicating future benefits from symbolic execution or fuzzing to expand dynamic visibility.

Answer to RQ2: *Wassy* prevented memory corruption in over **73%** of tested WebAssembly programs. In most cases, hardened variants preserved the expected benign behavior, demonstrating the effectiveness of our automated transformations.

6.5.4 *RQ3: Correctness and Preservation of Benign Behavior*

The third research question evaluates whether *Wassy* alters the intended behavior of benign programs. Specifically, does *Wassy* preserve functional correctness when applied to non-vulnerable WebAssembly binaries?

To answer this question, we consider the WASM binaries and their corresponding safe inputs from the Wasm-V dataset. These test cases (binary-input pairs) are expected to execute safely and produce correct output. For each test case, we:

- Execute the **original good** binary and collect its standard output and runtime memory traces.

- Apply *Wassy*'s full transformation pipeline to produce a **hardened** variant.
- Re-execute the hardened variant with the same inputs and compare behavior.

A transformation is considered correct if:

- **Output Equivalence:** The hardened binary produces the same standard output as the original.
- **Trace Compatibility:** The memory access trace is functionally equivalent or semantically aligned.

Wassy consistently preserved correct behavior across all benign programs in the test suite. No regressions, runtime errors, or unexpected outputs were introduced by the hardening process. These results highlight the precision of our binary rewriting:

- Transformations are guided by observed memory behavior and guarded by static checks.
- Inserted protections (e.g., bounds checks, read-only segments) do not interfere with normal control or data flow.
- The hardened binaries remain semantically equivalent to their original counterparts.

While these results are encouraging, future work could incorporate formal equivalence checking, alternative runtimes, or concolic testing to strengthen behavioral guarantees.

Answer to RQ3: *Wassy* maintained **100%** functional correctness across all tested non-vulnerable WebAssembly programs, confirming the safety and precision of its transformations.

6.5.5 *RQ4: Runtime Overhead*

The final research question assesses the performance cost of applying our hardening transformations. Specifically, what is the runtime overhead introduced by *Wassy*'s transformations on WebAssembly binaries?

We measure the execution time of each test case before and after applying our full hardening pipeline, including read-only memory protection and per-instruction bounds checks. To account for execution variance, we execute each test case **20 times** under identical conditions. We report the **average runtime**, discarding the minimum and maximum

values to mitigate the impact of outliers. All measurements are performed using a stable, Node.js (`v22.15.0`) runtime on a quiet system. Runtime overhead is then computed as:

$$\text{Overhead (\%)} = \frac{T_{\text{hardened}} - T_{\text{base}}}{T_{\text{base}}} \times 100$$

Table 8 presents the observed runtime overheads across all CWE categories. Runtime costs are generally low, with variation depending on memory access intensity and frequency of stores within loops. We observe that the runtime overhead is roughly 3% in all cases.

Answer to RQ4: Across all evaluated cases, the average runtime overhead of *Wassy* was under **3%**, on average, with heavier memory workloads exhibiting slightly higher overhead due to bounds checking.

6.6 DISCUSSION

While *Wassy* achieves practical hardening of WebAssembly binaries, several limitations constrain its current generality and precision.

TRACE COVERAGE Our reliance on dynamic memory traces means effectiveness depends on input quality. Unexercised code paths may lead to unprotected memory regions. Incorporating input fuzzing or concolic execution could expand trace coverage and strengthen protection guarantees.

STATIC HEURISTICS We infer upper bounds for loop-contained stores using control-flow cycle detection and subsequent memory accesses. This heuristic works well in many cases but may fail for complex or irregular control flow. More precise static dataflow analysis may improve the accuracy and applicability of this technique.

WASM FEATURE SUPPORT Our transformation tools currently do not support some WASM extensions, such as sign-extension operators or SIMD. Supporting a broader set of WASM features is necessary to ensure compatibility with a wider range of real-world modules.

DATASET SCOPE Evaluation is based on Wasm-V, a curated dataset derived from compiled Juliet test cases. While useful for controlled analysis, these do not capture the full diversity of modern WASM usage. Expanding to real-world binaries and fuzzed programs will help evaluate generality and robustness.

RESULT INTERPRETATION We determine mitigation success by comparing memory traces and stdout. While practical, this may miss subtle behavioral divergences. Future work may benefit from more semantic equivalence testing or runtime invariant checking.

Wassy provides a promising base for binary-level WASM hardening. Future improvements will focus on hybrid analysis, broader dataset evaluation, and expanded transformation support.

6.7 CONCLUSION

In this work, we presented *Wassy*, a binary-level framework for hardening WebAssembly modules against memory corruption vulnerabilities. By combining dynamic analysis with targeted rewriting, *Wassy* instruments WebAssembly binaries to enforce memory segmentation and per-instruction bounds checks that proactively guard against buffer overflows and related memory violations.

Our evaluation demonstrates that *Wassy* is capable of mitigating real vulnerabilities across a range of representative test cases, while preserving the correctness of benign binaries. The system operates directly at the WASM binary level, without requiring source code or compiler modification, and introduces minimal runtime overhead.

These results highlight the potential of post-deployment binary transformation as a practical defense mechanism for WebAssembly applications. We believe this work lays the foundation for future advances in WASM security tooling, and opens promising directions for hybrid analysis, fine-grained memory safety enforcement, and formal reasoning about binary-level protections.

7 RELATED WORK

7.1 DECLARATIVE SOFTWARE REWRITING

There are several language-specific declarative rewriting tools as well as generic software transformation toolsets and DSLs for writing transformation tools. Stratego [30] is a language and toolset for program transformation that is comprised of two parts, a DSL for expressing program transformations and a set of reusable transformation components that can be derived from declarative specifications. Spoofax [79], built on top of Stratego, is a language workbench for DSLs and integrates language processing techniques into an IDE. Rascal [85] is a DSL for source code analysis and program manipulation that aims to reduce boilerplate and abstract away from the tight integration of tools and techniques for program analysis. DMS [24] is a software engineering toolkit that can define arbitrary specifications and apply source-to-source and procedural transformations at scale. Coccinelle [90] is a program matching and transformation tool for C that allows developers to write code manipulations using a generalization of the patch syntax. Coccinelle4J [78] is a variant of Coccinelle for Java. TXL [47] is another DSL for writing tools and applications for program transformation. This DSL uses a combination of first order functional programming and term rewriting for source code manipulation. Cubix [88] is a multi-language, source-to-source program transformation system that uses a new program representation called incremental parametric syntax. Piranha [129] is an automated code refactoring tool designed to delete code that corresponds to stale feature flags. PolygotPiranha [81] proposes a new DSL to bridge the gap between language specific rewrite tools and lightweight match-replace tools by extending the functionality and increasing the expressivity of lightweight match-replace tools.

7.2 IMPERATIVE SOFTWARE REWRITING

Several software rewriting tools are based on an imperative paradigm. Balaban, Tip, and Fuhrer [22] present a method for migrating legacy classes in Java to replacements specified by the programmer. This method is implemented in the Eclipse IDE. JDeodorant [61] is another Eclipse plugin that identifies opportunities to extract cohesive classes from a large complex class and applying the refactoring chosen by the developer. LambdaFicator [63] is a tool that automatically migrates anonymous inner classes and loops over

collections to Lambda functions. Ketkar et al. [80] present an approach for automated type migration in a program by using a graph representation of type structure and using parallel MapReduce to scale the analysis. OpenRewrite [126] is a multi-language tools for software rewriting that is operates on the lossless syntax tree and performs transformations based on reusable rewrite recipes. Ossendrijver, Schroevers, and Grelck [127] present an approach that combines ErrorProne [9] and Refaster [170] for library migrations.

7.3 EXAMPLE GUIDED SOFTWARE REWRITING

Refaster [170] is a tool for rewriting Java programs based on a set of compatible before-and-after code examples. Lase [106] is a tool that creates context-aware edit scripts from two or more examples to automatically identify edit locations and transform the code. Blue-Pencil [112] is a tool that observes user behavior to identify repetitive edits and suggests similar transformations at other locations. Overwatch [177] is a similar edit tool that observes user behavior in an IDE and suggests similar transformations. Melt [130] is a tool that generates lightweight API migration rules from pull requests in popular library repositories. TC-infer [82] infers information about type migrations from open source repositories and generates rewrite rules to perform these transformations. SOAR [118] is a tool for migrating Data Science APIs that learns API representations and data mappings between APIs from documentation

7.4 INTRODUCING ASYNCHRONY IN SYNCHRONOUS CODE

To our knowledge, our work is the first technique in the latter category that is concerned with refactoring synchronous APIs into asynchronous equivalents, and that is concerned with JavaScript. Dig et al. [51] present *Relooper*, a refactoring tool for converting sequential loops into parallel loops in Java programs. Java introduced a `ParallelArray`, capable of applying a function to each element in parallel, and their tool performs a variety of program analyses to establish the safety of the transformation—both our and their tools present transformations as suggestions to mitigate potential unsoundness. When Java introduced parallel streams, Khatchadourian et al. [83] proposed a refactoring tool for migrating between sequential and parallel streams, in a similar spirit to how *Desynchronizer* proposed refactorings from synchronous to asynchronous APIs. Wloka, Sridharan, and Tip [173] is concerned with a whole-program transformation to make programs *reentrant*, which crucially allows them to be safely deployed in parallel. While this is a sync-to-async transformation tool like *Desynchronizer*, the use case is quite different, as we introduce asynchrony on a smaller scale. Dig, Marrero, and Ernst [50] introduce a refactoring tool called *Concurrencer* for introducing data structures from the `java.util.concurrent` library.

Concurrencer can parallelize arbitrary code, though users need to explicitly specify shared data for *Concurrencer* to work. In contrast, *Desynchronizer* only refactors synchronous API calls, but generates refactoring suggestions automatically.

7.5 TRANSLATION WITHIN ASYNCHRONOUS PROGRAMMING PARADIGMS

Gallaba et al. [64] explored how to refactor older, callback-based JavaScript code to use promises. Where their work is concerned with converting one style of asynchronous programming to another, ours is concerned with converting synchronous code to asynchronous code. Most previous research on refactoring of asynchronous or concurrent code is concerned with languages such as C#, Java, and Android. Okur et al. [123] present *Asyncifier*, a tool that converts callback-based code to use the .NET `async`/`await` constructs. Lin, Radoi, and Dig [97] present Asynchronizer, a tool for extracting long running computations in Android applications using the `AsyncTask` asynchronous programming construct, using a static points-to analysis to establish the safety of transformations. Lin, Okur, and Dig [96] also present a tool for correcting misuse of old asynchronous programming idioms, and modernizing the code. These approaches focus on converting older asynchronous code to newer asynchronous code, whereas *Desynchronizer* introduces asynchrony where none was present. Other work on C# includes work by Okur, Erdogan, and Dig [122] on two refactoring tools, *Taskifier* and *Simplifier*, for introducing TASK abstractions and transforming them into higher-level design patterns. This is another example of a migration from one asynchronous paradigm to another (from low level `Thread` abstractions to higher level asynchronous design patterns).

7.6 UNDERSTANDING ASYNCHRONOUS PROGRAMMING

There is wide body of work on *understanding* asynchronous applications, e.g., work by Alimadadi et al. [13] on understanding event-based asynchrony in JavaScript applications, on understanding asynchrony on the entire application stack [12], and on understanding the effects of DOM-sensitive changes [11]. This is complementary to our work, as *Desynchronizer* and *Lazifier* presents refactorings (that introduce asynchrony!) as suggestions to be vetted by programmers.

As Java's support for concurrent programming matured, tools were needed to help programmers choose the right constructs: Schäfer et al. [136] present *Relocker*, an automated tool that assists programmers with refactoring synchronized blocks into `ReentrantLocks` and `ReadWriteLocks`, to facilitate the performance tradeoffs associated with different types of locks. Other work by Schäfer et al. [135] introduces synchronization dependencies that refactoring engines must respect to preserve the correctness of com-

monly used refactorings in the presence of concurrency. More abstractly, Song et al. [142] propose a workflow refactoring to change the layout of regions of code to maximize concurrency and block-structuredness of applications. Zhang et al. [175] also explore this abstract space, detailing tooling to separate concurrency from the application design. This would allow programmers to decouple parallelism from core application functionality.

7.7 DEBLOATING AND LAZY LOADING

Software debloating is well-studied. Many applications contain far more code than is required, commonly referred to as "dead code", and the study of debloating is the study of how to determine and safely remove this dead code. Besides increasing application size, dead code is undesirable as it increases the "attack surface" of an application, i.e., more code provides more opportunities for an attacker to take advantage of a system. For example, Bhattacharya et al. [27] studies situations where functions accumulate more features than are strictly necessary, yielding poor performance when spurious functionality is not needed. Koo et al. [87] propose configuration-driven software debloating, where application configurations are linked with feature-specific libraries, and libraries are only loaded when the appropriate configuration criteria are met. This is a semi-automated process, and the code itself is not changed. Doloto [101] proposes an approach that leverages developer-supplied application traces to automatically refactor applications to load entire "routes" lazily, only when they are needed; their approach performs dynamic loading synchronously, which is disallowed in the modern web standard. Soto et al. [143] propose an approach to automatically specialize Java dependencies according to how they are used by the application's test suite, and Sharif et al. [138] propose a technique that leverages constant value configuration data to specialize applications.

Some recent work has been concerned with debloating JavaScript applications. Stubbifier [153], for example, leverages an application's test suite to determine "probably unused" code and replace this code with small *stubs* that can dynamically fetch and execute the code if it was actually needed. Stubbifier cannot debloat client-side applications (which, incidentally, rarely have test suites). Malavolta et al. [105] propose a technique to debloat client-side JavaScript applications with various levels of optimization; first, dead code is determined by consulting a call graph of the application, and one of the optimization levels proposed in the work replaces dead code with snippets to load the code lazily. Vasquez et al. [161] propose a technique that flags external library functions as being potentially dead, and removes them once a programmer confirms that they are truly unused. These pieces of work are concerned with removing *unused* functionality, and often lazily loading the dead code if they were wrong about the code being dead, whereas *Lazifier* removes *conditionally* used functionality, and none of these tools would not remove the

packages identified by our approach as they are used in the application. In a sense, these approaches are complementary.

7.8 FRAMEWORK-SPECIFIC STUDIES

Reactor being focused on React, a predominant JavaScript framework, aligns with the broader academic practice of deeply studying influential libraries and frameworks within a programming language. An example of such foundational tools is the research done on TensorFlow. Lagouvardos et al. introduced Pythia, a static analysis tool for TensorFlow to track tensor shapes and detect dynamic type errors, identifying shape-related bugs [89]. This approach closely models TensorFlow's value-flow semantics, demonstrating a concrete application of static analysis in identifying practical bugs within TensorFlow applications. Zhang et al. performed an empirical analysis on TensorFlow program bugs, offering insight into common defect categories, their symptoms, and root causes, as well as evaluating debugging strategies [176]. Filus et al., using static analysis tools, identified six common types of security threats in TensorFlow applications [59]. They provided insights into their causes and impacts and offered recommendations to improve code security and quality. Liu et al. introduced SHAPETracer, a static analysis and constraint-based tool developed to detect the top three types of TensorFlow bugs in real-world industrial settings, demonstrating rapid and effective bug identification [99].

7.9 ANALYZING REACT AND ITS SEMANTICS

React is the most popular JavaScript front-end framework and is used by over 1.3 million websites globally [45] and offers great research opportunities for optimization. Several studies have explored React's architecture and principles in the context of optimization to enhance its performance and usability and to understand its underlying behavior, highlighting the importance of this area of research. Anastasia et al. studied 5,572 vulnerability reports from React.js, focusing on how third-party dependencies impact security. This study reveals that managing updates effectively can mitigate common vulnerabilities [14]. Ferreira et al. present an empirical study of refactoring in React applications, identifying a catalog of 25 refactoring operations tailored to React alongside 17 adapted traditional refactoring operations, providing actionable insights for enhancing the design and maintainability of React-based web applications [141]. Ferreira et al. identified 12 unique code smells specific to React-based applications, demonstrating that these issues are not addressed by general-purpose catalogs [58]. In [144], the authors explored the development principles and architecture of React, highlighting its component-based structure, one-way data flow, and functional programming principles that enhance performance and devel-

oper usability. Madsen et al. presented a formal semantics for a core subset of React, capturing the essence of its component life-cycle, state changes, and reconciliation, facilitating program understanding, testing, and bug detection [103]. These studies provide a general examination of React's architecture and formal semantics, while *Reactor* specifically focuses on component re-rendering upon user interactions and UI updates.

7.10 UI OPTIMIZATION

Optimizing User Interface frameworks is critical for interactive applications to ensure responsiveness and enhance user experience, leading to extensive research in this area. Diniz-Junior et al. conducted a comparative study of three Java-Script UI frameworks: React, Vue, and Angular. They found that Vue had the fastest DOM manipulation, React had the best interaction time, and Angular had the largest bundle size [52]. Similarly, Siahaan et al. [140] conducted an analysis on the rendering performance of React, Vue, Next.js, and Nuxt.js. They found that Vue had the fastest initial rendering times, while React excelled in speed, index and user interactions. Next.js was best in server-side rendering. Additionally, Ollila et al. [124] compared rendering strategies of modern web frameworks, including Angular, React, Vue, Svelte, and Blazor, finding significant differences in performance, particularly in how they handle updates and static content rendering. They noted frameworks like Vue and Svelte, which automatically track dirty components and only process data bindings on updates, perform significantly better than those like React and Blazor, which re-render entire subtrees and do not differentiate between static and dynamic content. These studies compare rendering performance of various JavaScript UI frameworks, whereas *Reactor* is focused on improving React's responsiveness by optimizing performance and reducing unnecessary re-renders during UI updates.

7.11 JAVASCRIPT PERFORMANCE OPTIMIZATION

Addressing performance issues in JavaScript is crucial for optimizing web applications, as demonstrated by several studies focusing on inefficient API usage, asynchronous operations, and anti-pattern detection. Selakovic et al. [137] demonstrated that inefficient API usage falls among JavaScript's most important performance issues. They found that most optimizations require minimal code changes. Turcotte et al. [156] identified anti-patterns in JavaScript's promises and async/await, leading to inefficiencies and developed DrAsync, a tool to detect and visualize these issues. Arteca et al. [16] demonstrated that sub-optimal scheduling of asynchronous I/O operations in JavaScript can be improved by reordering them using static side-effect analysis, resulting in significant performance gains. Ferreira et al. [141] provided a comprehensive catalog of React-specific refactoring

by studying top GitHub projects, offering insights into best practices for maintaining and improving the quality and performance of React applications. While these studies focus on various performance issues such as API usage, asynchronous operations, and general refactoring practices, *Reactor* specifically targets enhancing React application performance by addressing re-rendering anti-patterns and optimizing UI update processes.

7.12 THE REACT COMPILER

Meta's React team recently announced the React Compiler [107], an experimental extension to React that aims to render React applications more efficiently. However, the compiler does not aim to provide perfectly optimal re-rendering with zero unnecessary computation. However, it's important to note that while our work is a research prototype exploring various optimization techniques, their tool is in the experimental stage. Their method introduces a new compilation layer, which inherently limits the range of optimizations due to the need to guarantee correctness at the compiler level, while our work suggests code refactoring that leverages preexisting and well-tested methods. Furthermore, to leverage the complete set of features, the React Compiler requires users to use React version 19. Instead, *Reactor* only requires using functional components, which have been the preferred paradigm since the introduction of life-cycle hooks in React 16.8, instead of class components.

7.13 BINARY INSTRUMENTATION

Binary instrumentation frameworks allow developers to build binary analysis tools like checkers and profilers. Well-known frameworks for x86 binaries include Valgrind [117], which is primarily used for memory debugging and leak detection in C programs, and DynamoRIO [32], which supports run-time code transformations. Lehmann et al. [92] developed Wasabi, an instrumentation framework for dynamic analysis of WebAssembly binaries. Wasabi inserts instrumentation hooks into the WebAssembly binary to allow the development of analyses such as instruction profiling and dataflow analysis. While these tools are useful for finding issues in binaries and developing analysis techniques, unlike *Wassy* they do not implement mitigations for these issues, specifically in a compiled binary.

7.14 WEBASSEMBLY SECURITY

Kolosick et al. [86] present a technique for Software-Based Fault Isolation (SFI) that relies on the identification of zero-cost conditions that, if satisfied, enable the implementation of context-switching using ordinary function calls. Stievenart et al. [149] report that compiling existing C programs to WebAssembly for cross-platform distribution may require source code adaptations to prevent security vulnerabilities. Lehmann and Pradel [93] present a learning-based approach for assisting security experts with understanding WebAssembly binaries by predicting precise high-level type information that goes beyond WebAssembly's basic low-level int and float types. Bhansali et al. [26] explore a range of obfuscation techniques for WebAssembly binaries. Romano et al. [133] present Wobfuscator, a technique for obfuscating JavaScript code in a way that evades modern malware detectors by translating (malicious) JavaScript code into WebAssembly. Swivel [115] is a compiler framework that hardens WebAssembly against Spectre-style attacks for modules running outside of the browser. Johnson et al. [77] present an approach for verifying the memory of WebAssembly binaries post-compilation along with a formal proof of the verifier soundness using the Coq theorem prover.

Fuzzm Lehmann, Torp, and Pradel [94] presents a binary instrumentation technique that injects canaries in the heap and stack to detect vulnerabilities and combines it the input generation algorithm from AFL fuzzer in a WebAssembly VM. WAFL Haßler and Maier [75] is a coverage-guided fuzzer for WebAssembly based on AFL++ that modifies the WebAssembly VM to generate coverage data. WasmFuzz Bauckholt and Holz [23] explores the use of WebAssembly as a compilation target for source code and fuzzing harnesses, treating WebAssembly as an intermediate representation.

Several widely used industry tools, WABT [163], Binaryen [28], and Twiggy [157], use static analysis for optimizing and reducing the size of WebAssembly binaries. Stievenart et al. presented a static information flow analysis for detecting security vulnerabilities [148], a static slicing technique for WebAssembly with applications in reverse engineering, code comprehension, and security [146], and a general-purpose static analysis framework for WebAssembly [147]. Andrès et al. Andrès et al. [15] present owi and Ningyu et al. present Seewasm, which are tools for symbolic execution of WebAssembly modules. Owi is build on a modular, monadic interpreter and can execute WebAssembly with concrete and symbolic values. Brito et al. Brito et al. [31] present Wasmati, a static analysis tool and security scanner based on code property graphs that can used for finding security vulnerabilities in WebAssembly modules. Baek et al. Baek et al. [21] present Wasm-R3, a technique to record and replay Wasm binaries by instrumenting the binary to collect execution traces and producing an executable standalone replay module.

8 CONCLUSION

In this thesis, we set out to show that *Declarative software rewriting can be used to automatically improve the performance, responsiveness, and security of Web applications*. We used declarative software rewriting to describe three performance optimization techniques in JavaScript applications and one security optimization application in WebAssembly.

8.1 INTRODUCING ASYNCHRONY IN JAVASCRIPT APPLICATIONS

In Chapter 3, we developed a technique to introduce asynchrony in synchronous JavaScript applications. JavaScript libraries offer both synchronous and asynchronous APIs for many common I/O operations. Although asynchronous APIs are generally preferred for their superior responsiveness and performance, developers often choose synchronous APIs due to their simplicity and ease of use. This technique uses unsound static analysis to identify synchronous JavaScript APIs and applies a set of rewrite rules to transform them to asynchronous equivalents. The technique is implemented in a tool called *Desynchronizer*.

In an empirical evaluation on 12 applications, *Desynchronizer* identified 316 synchronous API calls. 256 of these API calls were identified as transformation candidates, 244 of which were transformed successfully. Only 12 transformations resulted in behavioral changes, most of which can be attributed to unsoundness in the call graph analysis. As such, the transformations are presented as suggestions to the user and should be carefully reviewed before application.

8.2 INTRODUCING LAZY LOADING IN JAVASCRIPT APPLICATIONS

In Chapter 4, we present a technique for introducing lazy-loading in client-side JavaScript applications. Smaller bundle sizes help to minimize the time users wait for a web application to load and become responsive. Many existing tools such as bundlers and minifiers aim to reduce bundle size by eliminating dead code, but do not handle cases where the the code is required eventually, but not at application start-up. We implement this technique in a tool called *Lazifier*.

Lazifier uses unsound static analysis to identify external libraries that are used only in the context of event handlers and required conditionally. We identify the set of changes required to load such dependencies lazily and apply a set of transformations that are

specified as declarative rewrite rules. We evaluated *Lazifier* on 10 open source client-side JavaScript applications. *Lazifier* successfully transformed all the applications, reducing the bundle size by 36.2% on average, resulting in improvement in start-up time by 29.7%

8.3 PREVENTING SUPERFLUOUS RE-RENDERING IN REACT APPLICATIONS

In Chapter 5, we present a technique to identify and remediate superfluous re-rendering in React Applications. React is one of the most popular frameworks for creating single page applications in JavaScript and uses a declarative style for creating a UI. This enables react to compute what parts of the UI have changed and need to be re-rendered in response to user interactions. However, if developers are not careful, this can lead to unnecessary computation and superfluous re-rendering. We identified 5 anti-patterns that cause superfluous re-rendering in React and developed a tool to remediate them. This technique is implemented in a tool called *Reactor*.

Reactor uses unsound static analysis to identify situations where superfluous re-rendering may occur. For each anti-pattern, *Reactor* applies a set of rewrite rules that transform the code and remediate that identified anti-pattern. We evaluated *Reactor* in a targeted study on 23 open source React projects and performed a large-scale evaluation to study the prevalence of anti-patterns in 7,760 open source projects. In the targeted evaluation, the tool transformed 313 anti-pattern instances, resulting in an average reduction of 33.3% in rendering operations. We observed unsoundness in only one instance. In the large-scale study, we found at least one instance of an anti-pattern in 92.1% of the projects.

8.4 INTRODUCING MEMORY SEGMENTATION TO WEBASSEMBLY

In Chapter 6 we present a technique binary rewriting technique for hardening WebAssembly binaries. WebAssembly contains several safety features by design, but is still susceptible to buffer overflows and control flow hijacking. Moreover, the vulnerabilities can be difficult to detect and remediate if the source code is unavailable. We introduce memory segmentation and implement bound checks on store instructions to mitigate buffer overflow vulnerabilities. This technique is implemented in a tool called *Wassy*.

Wassy uses a combination of static and dynamic analysis to record execution traces and memory access information. This information is used to identify immutable data in linear memory and bounds for store instructions in loops. *Wassy* uses this information and applies a set of rewrite rules to transform the binary to mitigate buffer overflows. *Wassy* relies on client provided test cases to drive the dynamic analysis and assumes adequate coverage of permitted executions. Inadequate execution traces can lead to potential unsoundness in transformed binaries.

Wassy is evaluated on a curated dataset of 5,130 WebAssembly binaries with known vulnerabilities. *Wassy* mitigated vulnerabilities in 73% of binaries while adding a runtime overhead of only 3%. We did not observe unsoundness in the transformed binaries in our evaluation.

8.5 DISCUSSION

8.5.1 *Unsoundness in Analysis*

Chapters 3-5 present techniques for optimizing JavaScript applications using static analysis. JavaScript is a dynamically typed programming language with dynamic and highly permissive semantics. Although efforts have been made to extend JavaScript with static typing (*Flow* [60] and *TypeScript* [159]), many real-world programs either lack static types completely or often use imprecise types such as `any`. This makes a sound and precise analysis of JavaScript extremely difficult. An analysis is considered sound if and only if it does not produce any false negatives, i.e., a sound analysis is guaranteed to report every entity that satisfies the constraints of the analysis, while potentially reporting false positives. Sound analysis often leads to over-approximation to prevent false negatives, which can result in less useful information such as reporting "Top" as the type for variables or reporting all functions as targets in a call graph. Precision, on the other hand, deals with how precise the information reported by an analysis is. For example, an analysis that reports the type as "number" is considered more precise than an analysis that reports "any". In the context of a call graph, a precise call graph would report fewer targets than an imprecise analysis. Ideally, we want the analysis to be sound and precise, but this is often computationally intensive and does not scale to larger programs.

In this work, we rely on unsound static analysis to extract information about control and data flow in programs, ensuring that the analysis can scale and terminate in a reasonable amount of time. This ensures that the tools presented in this work can be used to transform real-world programs. However, the unsoundness can result in missed edges in the call graph or unsound assumptions about certain program behaviors. We acknowledge that this can lead to unsound transformations and present the changes as suggestions to be reviewed by developers. During our evaluation, we observed that incorrect transformations are not very common and conclude that some soundness in the analysis is worth sacrificing to make the tools fast and usable in practice.

An interesting direction for this research could be to use other techniques such as dynamic analysis or partial execution in conjunction with static analysis and observe the impact on reliability of transformations.

Unlike static analysis, dynamic analysis does not contain false positives. However, dynamic analysis can often be incomplete, i.e., the available set of inputs or test cases may not cover all the functionality. In Chapter 6, we use incomplete dynamic analysis of WebAssembly binaries to harden partial execution flows through binary rewriting. In practice, incomplete dynamic analysis might be sufficient to reliably transform binaries if the client application exercises all parts of the external module that it intends to use. This is similar to the guarantees associated with control flow integrity. Nevertheless, using techniques such as Concolic execution could be used to achieve higher coverage of the binary and more reliable transformations.

Thus, although the analysis used in this manuscript is unsound, we believe that it is sufficiently sound to provide useful insights to perform meaningful transformations in a reasonable amount of time.

8.5.2 *Towards a Unified Framework for Declarative Software Rewriting*

In this manuscript, we use Declarative Rewrite Rules to formally define the transformations for different techniques. This is a convenient and language-agnostic way of describing the analysis and code changes that abstracts away low-level details of the implementation. In practice, the implementation of the program analysis and associated transformations is often quite complex and does not share the elegance and simplicity of the rewrite rules. For example, code transformations may be implemented as modifications on the AST. This may be implemented imperatively as an AST traversal that modifies nodes instead of string manipulation. Such an approach ensures that the resulting transformations are well-formed and do not result in syntax errors. Many rewriting techniques choose to develop new DSLs to describe and implement the rewrite rules. This often results in users having to learn a new DSL which can, at times, be fairly complex and difficult to learn.

In this work, we focus primarily on describing the transformation techniques declaratively, but leave the implementation open ended. In our implementation, we choose to keep the analysis and transformation tool as separate entities. The analysis produces a set of source locations that are tagged with the transformation rule applicable at that location. The transformation tool consumes this set of tagged entries and performs the necessary manipulations on the code. The only constraint that we enforce in our implementation is that the analysis component and the transformation component must agree on the interface to communicate the information necessary for transformation. As a result, we are free to use static or dynamic analysis with no constraints on the framework as long as we produce an output that the transformation component can consume. Similarly, we can use any method for source manipulation as long as it can infer the necessary informa-

tion from the defined interface. Thus, if we use declarative tools for analysis and code manipulation, the implementation can also become declarative.

Concretely, we have a well-defined software analysis and transformation pipeline that we use to implement the rewrite rules from Chapters 3-5. The transformation pipeline uses CodeQL for static analysis and our own implementation of an AST based code manipulation tool. The two components use a specially formatted string for communication as shown below:

```
"/path/to/file.js:<line,col>--<line,col>~~~await_call???\{\"key\":\"value\"\}"
```

The string contains the path to file containing the source code, and identifies the node to be transformed using the line and column information for the start and end of the node. Next, the string tags the source location with the applicable rewrite rule, that is AWAIT-CALL in this example. Finally, the static analysis may provide additional optional information as a set of key-value pairs to assist the transformation. The static analysis, although often much more complicated than described in the rewrite rules, is implemented as a CodeQL query. The below code shows the query for AWAIT-CALL:

```
import javascript
import client_side_debloating_lib
import write_call_graph
/*
    ~~~ find call sites of newly async functions that now need to be awaited
*/
from
    ImportDeclarationToMakeDynamic dynImpDecl, DataFlow::InvokeNode ce
where
    // get all call sites of functions that are transformed
    ce.getACallee() = reverseCallTree(dynImpDecl.getASpecifier().getLocal().getVariable().
    getAnAccess().getEnclosingFunction())
select
    buildTransformationString(ce.asExpr()) as transformation
```

The auxiliary function `reverseCallTree` and the custom class `ImportDeclarationToMakeDynamic` abstract away much of the complexity of building the call graphs and identifying the focal nodes for our transformation. Lastly, we implemented the transformation tool in typescript using babel [20] for AST parsing, traversal, and code generation. However, our implementation abstracts away many of the low level details such as parsing the input from the analysis, file reads and writes, AST parsing and code generation, and AST traversal. The user only needs to create a file for every rewrite rule and implement the code for making the necessary changes to the AST node. The implementation for the AWAIT-CALL is shown below:

```
export default function makeAsync(td: TransformationDetails) {
```

```
    const theNode: NodePath<T.Node> = td.astNodeAfter;

    if (!theNode.isFunction()) { // node is not what we expected.
        logger.log"Expected a function, got a " + theNode.type));
        return;
    }

    // Make it async.
    theNode.node.async = true;

    // framework handles writing out the changes.

}
```

Thus, we have a flexible software rewriting pipeline that is synergistic with the declarative rewrite rules. We would like to extend this tool and add support for different languages going forward. It would also be interesting to explore techniques other than AST manipulation for manipulating the code.

8.5.3 *Scalability of Transformations*

Rewrite rules are generally very precise about the entities they transform. We can generalize parts of the rewrite rule to refer to a set of constructs that are a part of the core JavaScript specification. For example, we may use a generic term `func` in the rewrite rules to refer to both, function definitions and arrow function definitions. However, the generalizability of entities is rather limited. JavaScript is arguably the most popular programming language in the world [145]. As a result, it has a rich ecosystem with a large number of open-source contributions. The NPM contains more than 3 million packages. Rewrite rules cannot be automatically scaled up to support all or even a majority of NPM packages since this involves explicitly adding rules or conditions to the rules to handle them. Additionally, the underlying infrastructure that implements these rules must also support extension to accommodate these packages.

We find that declarative rewrite rules are an effective way to describe the changes necessary for a particular transformation. But, extending rewrite rules to support a larger set of APIs and packages is a non-trivial task and requires significant effort to update the rules, analysis, and transformations. In its current state, the Rewrite rules do not scale to support a large number of packages and are ill-equipped to handle the long tail. However, it would be interesting to explore the use of synthesis techniques for mining new rules to extend the support for transformation. Considering the declarative nature of these rules, Large Language Models (LLMs) could potentially be leveraged to extend support for different NPM packages. However, synthesizing the analysis and transformation from such

a rule is still a non-trivial task. It would be interesting to study the affinity of Declarative Software Rewriting to synthesis techniques and LLMs as tools for generating automated end-to-end transformation pipelines.

8.6 CLOSING THOUGHTS

In this manuscript, we present declarative rewrite rules as an effective method for describing source code transformations. In Chapter 3, we demonstrate the use of declarative software rewriting for improving the performance of server-side JavaScript applications by introducing asynchrony. Chapter 4 uses declarative software rewriting to transform client-side JavaScript applications to lazily load potentially unused dependencies, resulting in faster load times and improved responsiveness of web applications. In Chapter 5, declarative software rewriting is used to identify and remediate anti-patterns in React applications, thereby reducing superfluous computation, resulting in improved performance and responsiveness. Finally, in Chapter 6, we use declarative software rewriting to introduce memory segmentation and bounds enforcement in WebAssembly binaries to mitigate buffer overflows and improve security. Based on our empirical evaluation of these approaches, we conclude that *Declarative software rewriting can be used to automatically improve the performance, responsiveness, and security of Web applications*.

BIBLIOGRAPHY

[1] . *Build interactive web UIs using C#*. See https://dotnet.microsoft.c/. 2024.

[2] *16-feb-react-grupo-1*. See https://github.com/rerendering-evaluation/16-feb-react-grupo-1. 2024.

[3] *26-react-star-wars-hooks*. See https://github.com/rerendering-evaluation/26-react-star-wars-hooks. 2024.

[4] *30days-30projects*. See https://github.com/rerendering-evaluation/30days-30projects. 2024.

[5] *AI-chat-powered-by-open-ai-api-react-node*. See https://github.com/rerendering-evaluation/AI-chat-powered-by-open-ai-api-react-node. 2024.

[6] *API-Search*. See https://github.com/rerendering-evaluation/API-Search. 2024.

[7] Abhishek312s. *Movies-web-ui*. See https://github.com/Abhishek312s/Movies-web-ui/58904a3. 2023.

[8] AccessiTech. *local-json-editor*. See https://github.com/AccessiTech/local-json-editor. 2025.

[9] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. "Building Useful Program Analysis Tools Using an Extensible Java Compiler." In: *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. SCAM '12. USA: IEEE Computer Society, 2012, 14–23. ISBN: 9780769547831. DOI: 10.1109/SCAM.2012.28. URL: https://doi.org/10.1109/SCAM.2012.28.

[10] Akalay27. *workday-schedule-exporter*. See https://github.com/Akalay27/workday-schedule-exporter/97ca596. 2023.

[11] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. "Hybrid DOM-sensitive change impact analysis for JavaScript." In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

[12] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. "Understanding asynchronous interactions in full-stack JavaScript." In: *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 1169–1180.

[13] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. "Understanding JavaScript event-based interactions." In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 367–377.

[14] Terzi Anastasia and Bibi Stamatia. "Managing Security Vulnerabilities Introduced by Dependencies in React.JS JavaScript Framework." In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*. IEEE. 2024, pp. 126–133.

[15] Léo Andrès, Filipe Marques, Arthur Carcano, Pierre Chambart, José Fragoso Santos, and Jean-Christophe Filliâtre. "Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly." In: *The Art, Science, and Engineering of Programming* 9.1 (Oct. 2024). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2025/9/3. URL: http://dx.doi.org/10.22152/programming-journal.org/2025/9/3.

[16] Ellen Arteca, Frank Tip, and Max Schaefer. "Enabling Additional Parallelism in Asynchronous JavaScript Applications." In: *35th European Conference on Object-Oriented Programming (ECOOP 2021)* (2021). Ed. by Manu Sridharan. DOI: 10.4230/LIPIcs.ECOOP.2021.7. URL: https://par.nsf.gov/biblio/10285906.

[17] Ellen Arteca and Alexi Turcotte. "Npm-Filter: Automating the Mining of Dynamic Information from Npm Packages." In: *Proceedings of the 19th International Conference on Mining Software Repositories*. MSR '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, 304–308. ISBN: 9781450393034. DOI: 10.1145/3524842.3528501. URL: https://doi.org/10.1145/3524842.3528501.

[18] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. "QL: Object-oriented Queries on Relational Data." In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. 2016, 2:1–2:25. DOI: 10.4230/LIPIcs.ECOOP.2016.2.

[19] Babel. *Babel*. See https://babeljs.io/. 2023.

[20] BabelJS. *Babel Parser Documentation*. https://babeljs.io/docs/en/babel-parser. Accessed 2021-03-20. 2021.

[21] Doehyun Baek, Jakob Getz, Yusung Sim, Daniel Lehmann, Ben L. Titzer, Sukyoung Ryu, and Michael Pradel. "Wasm-R3: Record-Reduce-Replay for Realistic and Standalone WebAssembly Benchmarks." In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: 10.1145/3689787. URL: https://doi.org/10.1145/3689787.

[22] Ittai Balaban, Frank Tip, and Robert Fuhrer. "Refactoring support for class library migration." In: *SIGPLAN Not.* 40.10 (Oct. 2005), 265–279. ISSN: 0362-1340. DOI: 10.1145/1103845.1094832. URL: https://doi.org/10.1145/1103845.1094832.

[23] Florian Bauckholt and Thorsten Holz. "WebAssembly as a Fuzzing Compilation Target (Registered Report)." In: *Proceedings of the 3rd ACM International Fuzzing Workshop*. FUZZING 2024. Vienna, Austria: Association for Computing Machinery, 2024, 23–32. ISBN: 9798400711121. DOI: 10.1145/3678722.3685531. URL: https://doi.org/10.1145/3678722.3685531.

[24] Ira D. Baxter. "DMS: program transformations for practical scalable software evolution." In: *Proceedings of the International Workshop on Principles of Software Evolution*. IWPSE '02. Orlando, Florida: Association for Computing Machinery, 2002, 48–51. ISBN: 1581135459. DOI: 10.1145/512035.512047. URL: https://doi.org/10.1145/512035.512047.

[25] Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi, and Arjun Guha. "Putting in all the stops: execution control for JavaScript." In: *SIGPLAN Not.* 53.4 (June 2018), 30–45. ISSN: 0362-1340. DOI: 10.1145/3296979.3192370. URL: https://doi.org/10.1145/3296979.3192370.

[26] Shrenik Bhansali, Ahmet Aris, Abbas Acar, Harun Oz, and A. Selcuk Uluagac. "A First Look at Code Obfuscation for WebAssembly." In: *WiSec '22: 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks, San Antonio, TX, USA, May 16 - 19, 2022*. Ed. by Murtuza Jadliwala, Yongdae Kim, and Alexandra Dmitrienko. ACM, 2022, pp. 140–145. DOI: 10.1145/3507657.3528560. URL: https://doi.org/10.1145/3507657.3528560.

[27] Suparna Bhattacharya, Kanchi Gopinath, and Mangala Gowri Nanda. "Combining concern input with program analysis for bloat detection." In: *ACM SIGPLAN Notices* 48.10 (2013), pp. 745–764.

[28] *Binaryen*. https://github.com/WebAssembly/binaryen. Accessed 4/2023.

[29] Tim Boland and Paul E. Black. "Juliet 1.1 C/C++ and Java Test Suite." In: *Computer* 45.10 (2012), pp. 88–90. DOI: 10.1109/MC.2012.345.

[30] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. "Stratego/XT 0.17. A language and toolset for program transformation." In: *Sci. Comput. Program.* 72.1–2 (June 2008), 52–70. ISSN: 0167-6423. DOI: 10.1016/j.scico.2007.11.003. URL: https://doi.org/10.1016/j.scico.2007.11.003.

[31] Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. "Wasmati: An efficient static vulnerability scanner for WebAssembly." In: *Computers & Security* 118 (2022), p. 102745. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2022.102745. URL: https://www.sciencedirect.com/science/article/pii/S0167404822001407.

[32] Derek Bruening and Timothy Garnett. "Building dynamic instrumentation tools with DynamoRIO." In: *Proc. Int. Conf. IEEE/ACM Code Generation and Optimi zation (CGO), Shen Zhen, China*. 2013.

[33] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. "Control-Flow Integrity: Precision, Security, and Performance." In: *ACM Comput. Surv.* 50.1 (Apr. 2017). ISSN: 0360-0300. DOI: 10.1145/3054924. URL: https://doi.org/10.1145/3054924.

[34] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V. Madhyastha, and Vyas Sekar. "Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices." In: *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. USENIX Association, 2015, pp. 439–453. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/butkiewicz.

[35] *CRUD-Context-React*. See https://github.com/rerendering-evaluation/CRUD-Context-React. 2024.

[36] *CWE-121 (Stack-based Buffer Overflow)*. https://cwe.mitre.org/data/definitions/121.html. Accessed: 2025-07-06.

[37] *CWE-122 (Heap-based Buffer Overflow)*. https://cwe.mitre.org/data/definitions/122.html. Accessed: 2025-07-06.

[38] *CWE-123 (Write-what-where Condition)*. https://cwe.mitre.org/data/definitions/123.html]. Accessed: 2025-07-06.

[39] *CWE-124 (Buffer Underwrite)*. https://cwe.mitre.org/data/definitions/124.html]. Accessed: 2025-07-06.

[40] *CWE-190 (Integer Overflow or Wraparound)*. https://cwe.mitre.org/data/definitions/190.html]. Accessed: 2025-07-06.

[41] *CWE-680 (Integer Overflow to Buffer Overflow)*. https://cwe.mitre.org/data/definitions/680.html]. Accessed: 2025-07-06.

[42] *Calculadora*. See https://github.com/Contimp/Calculadora/blob/master/src/App.js. 2024.

[43] *Calculadora*. See https://github.com/rerendering-evaluation/Calculadora. 2024.

[44] *CashTrackr*. See https://github.com/rerendering-evaluation/CashTrackr. 2024.

[45] Citrusbug. *React Statistics – An In-Depth Look at the Trends*. See https://citrusbug.com/blog/react-statistics/#:~:text=In%202024%2C%20there%20are%20about,for%20their%20front-end%20framework.. 2024.

[46] *Clang: a C language family frontend for LLVM*. Available from https://clang.llvm.org/. 2022.

[47] James R. Cordy. "TXL - A Language for Programming Language Tools and Applications." In: *Electron. Notes Theor. Comput. Sci.* 110.C (Dec. 2005), 3–31. ISSN: 1571-0661.

[48] Crispan Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks." In: *Proceedings of the 7th USENIX Security Symposium (USENIX Security 98)*. 1998. URL: https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention.

[49] Douglas Crockford. *jsmin*. See https://www.crockford.com/jsmin.html. 2023.

[50] Danny Dig, John Marrero, and Michael D. Ernst. "Refactoring sequential Java code for concurrency via concurrent libraries." In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. 2009, pp. 397–407. DOI: 10.1109/ICSE.2009.5070539.

[51] Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph E. Johnson. "Relooper: refactoring for loop parallelism in Java." In: *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. 2009, pp. 793–794. DOI: 10.1145/1639950.1640018.

[52] Raimundo NV Diniz-Junior, Caio César L Figueiredo, Gilson De S Russo, Marcos Roberto G Bahiense-Junior, Mateus VL Arbex, Lanier M Dos Santos, Raimundo F Da Rocha, Renan R Bezerra, and Felipe T Giuntini. "Evaluating the performance of web rendering technologies based on JavaScript: Angular, React, and Vue." In: *2022 XVLIII Latin American Computer Conference (CLEI)*. IEEE. 2022, pp. 1–9.

[53] Emily Doherty. *Pixelsketch*. See https://github.com/emilydoh/Pixelsketch. 2025.

[54] *ECMAScript 2020 Language Specification*. https://www.ecma-international.org/ecma-262/11.0/. 2020.

[55] Evan You, Vue.js developers. *cybernetically enhanced web apps.* See https://vuejs.org/. 2024.

[56] *Express Web Framework for JavaScript*. http://expressjs.com/. 2021.

[57] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. "Efficient construction of approximate call graphs for JavaScript IDE services." In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 2013, pp. 752–761. DOI: 10.1109/ICSE.2013.6606621.

[58] Fabio Ferreira and Marco Tulio Valente. "Detecting code smells in React-based Web apps." In: *Information and Software Technology* 155 (2023), p. 107111.

[59] Katarzyna Filus and Joanna Domańska. "Software vulnerabilities in TensorFlow-based deep learning applications." In: *Computers & Security* 124 (2023), p. 102948.

[60] *Flow*. See https://flow.org/. 2025.

[61] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. "JDeodorant: identification and application of extract class refactorings." In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, 1037–1039. ISBN: 9781450304450. DOI: 10.1145/1985793.1985989. URL: https://doi.org/10.1145/1985793.1985989.

[62] *Fokasu*. See https://github.com/react-rerendering-benchmarks/fokasu. 2024.

[63] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. "LambdaFicator: From imperative to functional programming through automated refactoring." In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 1287–1290. DOI: 10.1109/ICSE.2013.6606699.

[64] Keheliya Gallaba, Quinn Hanam, Ali Mesbah, and Ivan Beschastnikh. "Refactoring Asynchrony in JavaScript." In: *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 2017, pp. 353–363. DOI: 10.1109/ICSME.2017.83.

[65] Dennis F. Galletta, Raymond M. Henry, Scott McCoy, and Peter Polak. "Web Site Delays: How Tolerant are Users?" In: *J. Assoc. Inf. Syst.* 5.1 (2004), p. 1. URL: http://aisel.aisnet.org/jais/vol5/iss1/1.

[66] GitHub. *Prevalence of React*. See https://github.com/facebook/react/network/dependents. Accessed Jun 07 2024. 2024.

[67] Gitnux. *Statistics About The Most Popular Front End Frameworks*. See https://gitnux.org/most-popular-front-end-frameworks/. 2024.

[68] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. *Automatic Migration from Synchronous to Asynchronous JavaScript APIs (Artifact)*. 2021. URL: https://doi.org/10.5281/zenodo.5502210.

[69] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. "Automatic migration from synchronous to asynchronous JavaScript APIs." In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–27.

[70] Google. *Chrome DevTools*. See https://developer.chrome.com/docs/devtools/. 2023.

[71] Google. See https://angular.io/. 2024.

[72] Agnieszka Goralczyk. *react_spreadsheet*. See https://github.com/ajgoralczyk/react_spreadsheet. 2025.

[73] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. "Bringing the Web up to Speed with WebAssembly." In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 185–200. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062363. URL: http://doi.acm.org/10.1145/3062341.3062363.

[74] Harinathlee. *upoint-query-builder*. See https://github.com/Harinathlee/upoint-query-builder/f9aa0f1. 2023.

[75] Keno Haßler and Dominik Maier. "WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots." In: *Reversing and Offensive-Oriented Trends Symposium*. ROOTS'21. Vienna, Austria: Association for Computing Machinery, 2022, 23–30. ISBN: 9781450396028. DOI: 10.1145/3503921.3503924. URL: https://doi.org/10.1145/3503921.3503924.

[76] *Jest: A delightful JavaScript Testing Framework*. https://jestjs.io/. 2021.

[77] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. "SFI safety for native-compiled Wasm." In: *Network and Distributed Systems Security (NDSS) Symposium*. 2021.

[78] Hong Jin Kang, Ferdian Thung, Julia Lawall, Gilles Muller, Lingxiao Jiang, and David Lo. "Semantic Patches for Java Program Transformation." In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 22:1–22:27. ISBN: 978-3-95977-111-5. DOI: 10.4230/LIPIcs.ECOOP.2019.22. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2019.22.

[79] Lennart C.L. Kats and Eelco Visser. "The spoofax language workbench: rules for declarative specification of languages and IDEs." In: *SIGPLAN Not.* 45.10 (Oct. 2010), 444–463. ISSN: 0362-1340. DOI: 10.1145/1932682.1869497. URL: https://doi.org/10.1145/1932682.1869497.

[80] Ameya Ketkar, Ali Mesbah, Davood Mazinanian, Danny Dig, and Edward Aftandilian. "Type migration in ultra-large-scale codebases." In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE '19. Montreal, Quebec,

Canada: IEEE Press, 2019, 1142–1153. DOI: 10.1109/ICSE.2019.00117. URL: https://doi.org/10.1109/ICSE.2019.00117.

[81] Ameya Ketkar, Daniel Ramos, Lazaro Clapp, Raj Barik, and Murali Krishna Ramanathan. "A Lightweight Polyglot Code Transformation Language." In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656429. URL: https://doi.org/10.1145/3656429.

[82] Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryksin. "Inferring and applying type changes." In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, 1206–1218. ISBN: 9781450392211. DOI: 10.1145/3510003.3510115. URL: https://doi.org/10.1145/3510003.3510115.

[83] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. "Safe automated refactoring for intelligent parallelization of Java 8 streams." In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. Ed. by Joanne M. Atlee, Tevfik Bultan, and Jon Whittle. IEEE / ACM, 2019, pp. 619–630. DOI: 10.1109/ICSE.2019.00072.

[84] Hee Yeon Kim, Ji Hoon Kim, Ho Kyun Oh, Beom Jin Lee, Si Woo Mun, Jeong Hoon Shin, and Kyounggon Kim. "DAPP: automatic detection and analysis of prototype pollution vulnerability in Node.js modules." In: *Int. J. Inf. Sec.* 21.1 (2022), pp. 1–23. DOI: 10.1007/s10207-020-00537-0. URL: https://doi.org/10.1007/s10207-020-00537-0.

[85] Paul Klint, Tijs van der Storm, and Jurgen Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation." In: *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. SCAM '09. USA: IEEE Computer Society, 2009, 168–177. ISBN: 9780769537931. DOI: 10.1109/SCAM.2009.28. URL: https://doi.org/10.1109/SCAM.2009.28.

[86] Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. "Isolation without taxation: near-zero-cost transitions for WebAssembly and SFI." In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–30. DOI: 10.1145/3498688. URL: https://doi.org/10.1145/3498688.

[87] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. "Configuration-driven software debloating." In: *Proceedings of the 12th European Workshop on Systems Security*. 2019, pp. 1–6.

[88] James Koppel, Varot Premtoon, and Armando Solar-Lezama. "One tool, many languages: language-parametric transformation with incremental parametric syntax." In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276492. URL: https://doi.org/10.1145/3276492.

[89] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. "Static analysis of shape in TensorFlow programs." In: *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. 2020.

[90] Julia Lawall and Gilles Muller. "Coccinelle: 10 Years of Automated Evolution in the Linux Kernel." In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 601–614. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/lawall.

[91] Daniel Lehmann, Johannes Kinder, and Michael Pradel. "Everything Old is New Again: Binary Security of WebAssembly." In: *Proceedings of the 29th USENIX Security Symposium*. USENIX Security 20. USENIX Association, Aug. 2020, pp. 217–217. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

[92] Daniel Lehmann and Michael Pradel. "Wasabi: A framework for dynamically analyzing webassembly." In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 1045–1058.

[93] Daniel Lehmann and Michael Pradel. "Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries." In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI '22. New York, NY, USA: Association for Computing Machinery, 2022. DOI: 10.1145/3519939.3523449.

[94] Daniel Lehmann, Martin Toldam Torp, and Michael Pradel. *Fuzzm: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly*. 2021. arXiv: 2110.15433 [cs.CR]. URL: https://arxiv.org/abs/2110.15433.

[95] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. "Detecting Node.js Prototype Pollution Vulnerabilities via Object Lookup Analysis." In: *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. Ed. by Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta. ACM, 2021, pp. 268–279. DOI: 10.1145/3468264.3468542. URL: https://doi.org/10.1145/3468264.3468542.

[96] Yu Lin, Semih Okur, and Danny Dig. "Study and Refactoring of Android Asynchronous Programming (T)." In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 2015, pp. 224–235. DOI: 10.1109/ASE.2015.50.

[97] Yu Lin, Cosmin Radoi, and Danny Dig. "Retrofitting concurrency for Android applications through refactoring." In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 2014, pp. 341–352. DOI: 10.1145/2635868.2635903.

[98] Gitte Lindgaard, Gary Fernandes, Cathy Dudek, and Judith M. Brown. "Attention web designers: You have 50 milliseconds to make a good first impression!" In: *Behav. Inf. Technol.* 25.2 (2006), pp. 115–126. DOI: 10.1080/01449290500330448. URL: https://doi.org/10.1080/01449290500330448.

[99] Chen Liu, Jie Lu, Guangwei Li, Ting Yuan, Lian Li, Feng Tan, Jun Yang, Liang You, and Jingling Xue. "Detecting TensorFlow program bugs in real-world industrial environment." In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 55–66.

[100] Zhicheng Liu and Jeffrey Heer. "The Effects of Interactive Latency on Exploratory Visual Analysis." In: *IEEE Trans. Vis. Comput. Graph.* 20.12 (2014), pp. 2122–2131. DOI: 10.1109/TVCG.2014.2346452. URL: https://doi.org/10.1109/TVCG.2014.2346452.

[101] Benjamin Livshits and Emre Kiciman. "Doloto: Code Splitting for Network-Bound Web 2.0 Applications." In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '08/FSE-16. Atlanta, Georgia: Association for Computing Machinery, 2008, 350–360. ISBN: 9781595939951. DOI: 10.1145/1453101.1453151. URL: https://doi.org/10.1145/1453101.1453151.

[102] Magnus Madsen, Ondrej Lhoták, and Frank Tip. "A model for reasoning about JavaScript promises." In: *PACMPL* 1.OOPSLA (2017), 86:1–86:24. DOI: 10.1145/3133910. URL: http://doi.acm.org/10.1145/3133910.

[103] Magnus Madsen, Ondrej Lhotak, and Frank Tip. "A Semantics for the Essence of React." In: *European Conference on Object-Oriented Programming*. 2020.

[104] Magnus Madsen, Frank Tip, and Ondrej Lhoták. "Static analysis of event-driven Node.js JavaScript applications." In: *Proc. of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. 2015, pp. 505–519. DOI: 10.1145/2814270.2814272. URL: http://doi.acm.org/10.1145/2814270.2814272.

[105] Ivano Malavolta, Kishan Nirghin, Gian Luca Scoccia, Simone Romano, Salvatore Lombardi, Giuseppe Scanniello, and Patricia Lago. "JavaScript Dead Code Identification, Elimination, and Empirical Assessment." In: *IEEE Transactions on Software Engineering* (2023), pp. 1–23. DOI: 10.1109/TSE.2023.3267848.

[106] Na Meng, Miryung Kim, and Kathryn S McKinley. "LASE: locating and applying systematic edits by learning from examples." In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 502–511.

[107] Meta Platforms, Inc. *React Compiler*. See https://github.com/facebook/react/tree/main/compiler. 2024.

[108] Meta Platforms, Inc. *React: The library for web and native user interfaces*. See https://react.dev/. 2024.

[109] Meta Platforms, Inc. *React: The library for web and native user interfaces*. See https://legacy.reactjs.org/docs/forms.html. 2024.

[110] Microsoft. *CodeQL JavaScript Data Flow Library*. See https://github.com/github/codeql/tree/7323d4e/javascript/ql/lib/semmle/javascript/dataflow. 2023.

[111] Microsoft. *CodeQL*. See https://codeql.github.com/. 2023.

[112] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. "On the fly synthesis of edit suggestions." In: vol. 3. OOPSLA. New York, NY, USA: Association for Computing Machinery, Oct. 2019. DOI: 10.1145/3360569. URL: https://doi.org/10.1145/3360569.

[113] *Mocha - the fun, simple, flexible JavaScript test framework*. https://mochajs.org/. 2021.

[114] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. "Opaque Control-Flow Integrity." In: *NDSS*. Vol. 26. 2015, pp. 27–30.

[115] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. "Swivel: Hardening webassembly against spectre." In: *arXiv preprint arXiv:2102.12730* (2021).

[116] *National Vulnerability Database – CVE-2018-14550 Detail*. Accessed 4/2023. 2018. (Visited on 02/14/2020).

[117] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[118] Ansong Ni, Daniel Ramos, Aidan Z.H. Yang, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. "SOAR: A Synthesis Approach for Data Science API Refactoring." In: *Proceedings of the 43rd International Conference on Software Engineering*. ICSE '21. Madrid, Spain: IEEE Press, 2021, 112–124. ISBN: 9781450390859. DOI: 10.1109/ICSE43902.2021.00023. URL: https://doi.org/10.1109/ICSE43902.2021.00023.

[119] Ben Niu and Gang Tan. "Modular control-flow integrity." In: *SIGPLAN Not.* 49.6 (June 2014), 577–587. ISSN: 0362-1340. DOI: 10.1145/2666356.2594295. URL: https://doi.org/10.1145/2666356.2594295.

[120] Ben Niu and Gang Tan. "Per-Input Control-Flow Integrity." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, 914–926. ISBN: 9781450338325. DOI: 10.1145/2810103.2813644. URL: https://doi.org/10.1145/2810103.2813644.

[121] *Node.js with WebAssembly*. Available from https://nodejs.dev/en/learn/nodejs-with-webassembly. 2022.

[122] Semih Okur, Cansu Erdogan, and Danny Dig. "Converting Parallel Code from Low-Level Abstractions to Higher-Level Abstractions." In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. 2014, pp. 515–540. DOI: 10.1007/978-3-662-44202-9_21.

[123] Semih Okur, David L. Hartveld, Danny Dig, and Arie van Deursen. "A study and toolkit for asynchronous programming in C#." In: *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 2014, pp. 1117–1127. DOI: 10.1145/2568225.2568309.

[124] Risto Ollila, Niko Mäkitalo, and Tommi Mikkonen. "Modern web frameworks: A comparison of rendering performance." In: *Journal of Web Engineering* 21.3 (2022), pp. 789–813.

[125] OpenJS Foundation. *Node.js*. https://nodejs.org/en/. 2021.

[126] *OpenRewrite by Moderne | Large Scale Automated Refactoring*. https://docs.openrewrite.org/. Accessed: 2025-07-08.

[127] Rick Ossendrijver, Stephan Schroevers, and Clemens Grelck. "Towards automated library migrations with error prone and refaster." In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. SAC '22. Virtual Event: Association for Computing Machinery, 2022, 1598–1606. ISBN: 9781450387132. DOI: 10.1145/3477314.3507153. URL: https://doi.org/10.1145/3477314.3507153.

[128] Joonyoung Park, Inho Lim, and Sukyoung Ryu. "Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild." In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. Ed. by Laura K. Dillon, Willem Visser, and Laurie A. Williams. ACM, 2016, pp. 61–70. DOI: 10.1145/2889160.2889227. URL: https://doi.org/10.1145/2889160.2889227.

[129] Murali Krishna Ramanathan, Lazaro Clapp, Rajkishore Barik, and Manu Sridharan. "Piranha: reducing feature flag debt at uber." In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '20. Seoul, South Korea: Association for Computing Machinery, 2020, 221–230. ISBN: 9781450371230. DOI: 10.1145/3377813.3381350. URL: https://doi.org/10.1145/3377813.3381350.

[130] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. "MELT: Mining Effective Lightweight Transformations from Pull Requests." In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Sept. 2023, 1516–1528. DOI: 10.1109/ase56229.2023.00117. URL: http://dx.doi.org/10.1109/ASE56229.2023.00117.

[131] Rich Harris. See https://svelte.dev/. 2024.

[132] Rollup. *Tree Shaking*. See https://rollupjs.org. also see https://rollupjs.org/faqs/#what-is-tree-shaking for tree-shaking. 2023.

[133] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. "Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly." In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 1574–1589. DOI: 10.1109/SP46214.2022.9833626. URL: https://doi.org/10.1109/SP46214.2022.9833626.

[134] *Rust: A language empowering everyone to build reliable and efficient software*. Available from https://www.rust-lang.org/what/wasm. 2022.

[135] Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. "Correct Refactoring of Concurrent Java Code." In: *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*. 2010, pp. 225–249. DOI: 10.1007/978-3-642-14107-2_11.

[136] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. "Refactoring Java programs for flexible locking." In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. Ed. by Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic. ACM, 2011, pp. 71–80. DOI: 10.1145/1985793.1985804.

[137] Marija Selakovic and Michael Pradel. "Performance issues and optimizations in JavaScript: an empirical study." In: *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 61–72.

[138] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. "TRIMMER: Application Specialization for Code Debloating." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE '18. Montpellier, France: Association for Computing Machinery, 2018, 329–339. ISBN: 9781450359375. DOI: 10.1145/3238147.3238160. URL: https://doi.org/10.1145/3238147.3238160.

[139] SheetJS. *xlsx*. See https://www.npmjs.com/package/xlsx. 2023.

[140] Mangapul Siahaan and Ryan Kenidy. "Rendering performance comparison of react, vue, next, and nuxt." In: *Jurnal Mantik* 7.3 (2023), pp. 1851–1860.

[141] Fabio da Silva Ferreira, Hudson Silva Borges, and Marco Tulio Valente. "Refactoring React-Based Web Apps." In: *Marco Tulio, Refactoring React-Based Web Apps* ().

[142] W. Song, H. Jacobsen, S. C. Cheung, H. Liu, and X. Ma. "Workflow Refactoring for Maximizing Concurrency and Block-Structuredness." In: *IEEE Transactions on Services Computing* (2018), pp. 1–1.

[143] César Soto-Valero, Deepika Tiwari, Tim Toady, and Benoit Baudry. "Automatic Specialization of Third-Party Java Dependencies." In: *arXiv preprint arXiv:2302.08370* (2023).

[144] CACM Staff. "React: Facebook's functional turn on writing Javascript." In: *Commun. ACM* 59.12 (2016), 56–62. ISSN: 0001-0782. DOI: 10.1145/2980991. URL: https://doi.org/10.1145/2980991.

[145] *Statista*. See https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/. 2025.

[146] Quentin Stiévenart, Dave Binkley, and Coen De Roover. "Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries." In: *The 44th International Conference on Software Engineering*. ICSE 2022. 2022. DOI: 10.1145/3510003.3510070.

[147] Quentin Stiévenart and Coen De Roover. "Wassail: a WebAssembly Static Analysis Library." In: ProWeb21. Fifth International Workshop on Programming Technology for the Future Web. Mar. 22, 2021. URL: https://2021.programming-conference.org/home/proweb-2021.

[148] Quentin Stiévenart and Coen De Roover. "Compositional Information Flow Analysis for WebAssembly Programs." In: *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020, pp. 13–24. DOI: 10.1109/SCAM51674.2020.00007. URL: https://doi.org/10.1109/SCAM51674.2020.00007.

[149] Quentin Stiévenart, Coen De Roover, and Mohammad Ghafari. "Security risks of porting C programs to WebAssembly." In: *SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022*. Ed. by Jiman Hong, Miroslav Bures, Juw Won Park, and Tomás Cerný. ACM, 2022, pp. 1713–1722. DOI: 10.1145/3477314.3507308. URL: https://doi.org/10.1145/3477314.3507308.

[150] React Dev Team. *Nesting and organizing components*. See https://react.dev/learn/your-first-component#nesting-and-organizing-components. Accessed Oct 15 2024. 2024.

[151] *The LLVM Compiler Infrastructure*. Available from https://llvm.org/. 2022.

[152] Alexi Turcotte, Mark W. Aldrich, and Frank Tip. "Reformulator: Automated Refactoring of the N+1 Problem in Database-Backed Applications." In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. Rochester, MI, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. DOI: 10.1145/3551349.3556911. URL: https://doi.org/10.1145/3551349.3556911.

[153] Alexi Turcotte, Ellen Arteca, Ashish Mishra, Saba Alimadadi, and Frank Tip. "Stubbifier: debloating dynamic server-side JavaScript applications." In: *Empirical Software Engineering* 27.7 (2022), p. 161.

[154] Alexi Turcotte, Satyajit Gokhale, and Frank Tip. "Increasing the Responsiveness of Web Applications by Introducing Lazy Loading." In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 459–470.

[155] Alexi Turcotte, Satyajit Gokhale, and Frank Tip. *Lazifier Artifact*. See https://zenodo.org/badge/latestdoi/680260614. 2023.

[156] Alexi Turcotte, Michael D Shah, Mark W Aldrich, and Frank Tip. "DrAsync: identifying and visualizing anti-patterns in asynchronous JavaScript." In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 774–785.

[157] *Twiggy: A code size profiler for Wasm*. https://rustwasm.github.io/twiggy/. Accessed 4/2023.

[158] *TwitchKillMe*. See https://github.com/react-rerendering-benchmarks/TwitchKillMe. 2024.

[159] *TypeScript*. See https://www.typescriptlang.org/. 2025.

[160] *TypeWriter*. See https://github.com/react-rerendering-benchmarks/TypeWriter. 2024.

[161] H.C. Vázquez, A. Bergel, S. Vidal, J.A. Díaz Pace, and C. Marcos. "Slimming javascript applications: An approach for removing unused functions from javascript libraries." In: *Information and Software Technology* 107 (2019), pp. 18–29. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.10.009. URL: https://www.sciencedirect.com/science/article/pii/S0950584918302210.

[162] W3Techs. *Comparison of the usage statistics of React vs. Vue.js vs. Angular for websites*. See https://w3techs.com/technologies/comparison/js-angularjs,js-react,js-vuejs. 2024.

[163] *WABT: The WebAssembly Binary Toolkit*. See https://github.com/WebAssembly/wabt. Accessed 4/2023.

[164] *WASI – The WebAssembly System Interface*. See https://wasi.dev/. (Visited on 04/12/2022).

[165] *Wasm-V: A Vulnerability-Labeled Benchmark Suite for Security Analysis of WebAssembly Binaries*. https://github.com/williamsryan/Wasm-V. Accessed 2025-04-22. 2025.

[166] *WasmEdge Runtime*. https://wasmedge.org/. Accessed: 2025-04-22.

[167] *Wasmer: The Universal WebAssembly Runtime*. https://wasmer.io. Accessed: 2025-04-22.

[168] *Wasmtime: A fast and secure runtime for WebAssembly*. See https://wasmtime.dev/. 2022. (Visited on 11/16/2022).

[169] *Wasmtime*. https://docs.wasmtime.dev/. 2025.

[170] Louis Wasserman. "Scalable, example-based refactorings with refaster." In: *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*. WRT '13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, 25–28. ISBN: 9781450326049. DOI: 10.1145/2541348.2541355. URL: https://doi.org/10.1145/2541348.2541355.

[171] *WebAssembly Language Support Matrix*. https://developer.fermyon.com/wasm-languages/webassembly-language-support. 2025.

[172] *WebAssembly*. 2020. URL: https://webassembly.org/ (visited on 03/31/2022).

[173] Jan Wloka, Manu Sridharan, and Frank Tip. "Refactoring for reentrancy." In: *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*. 2009, pp. 173–182. DOI: 10.1145/1595696.1595723.

[174] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. "Practical Control Flow Integrity and Randomization for Binary Executables." In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 559–573. DOI: 10.1109/SP.2013.44.

[175] Yang Zhang, Dongwen Zhang, Weixing Ji, and Yizhuo Wang. "Refactoring for Separation of Concurrent Concerns." In: *Algorithms and Architectures for Parallel Processing*. Ed. by Guojun Wang, Albert Zomaya, Gregorio Martinez, and Kenli Li. Cham: Springer International Publishing, 2015, pp. 105–118. ISBN: 978-3-319-27137-8.

[176] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. "An empirical study on tensorflow program bugs." In: *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 2018, pp. 129–140.

[177] Yuhao Zhang et al. "Overwatch: learning patterns in code edit sequences." In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022). DOI: 10.1145/3563302. URL: https://doi.org/10.1145/3563302.

[178] *Zone.js*. See https://github.com/angular/angular/tree/main/packages/zone.js. 2025.

[179] *age-react*. See https://github.com/rerendering-evaluation/age-react. 2024.

[180] *baklava*. See https://github.com/react-rerendering-benchmarks/baklava. 2024.

[181] *chicken-coop*. See https://github.com/react-rerendering-benchmarks/chicken-coop. 2024.

[182] *csi-conference-frontend*. See https://github.com/react-rerendering-benchmarks/csi-conference-frontend. 2024.

[183] *css-fx-layout*. See https://github.com/react-rerendering-benchmarks/css-fx-layout. 2024.

[184] eligrey. *file-saver*. See https://www.npmjs.com/package/file-saver. 2023.

[185] *escomplex*. See https://github.com/escomplex/escomplex. 2025.

[186] fahimahammed. *task*. See https://github.com/fahimahammed/task/b641bc0. 2023.

[187] *farm-ng-core*. See https://github.com/react-rerendering-benchmarks/farm-ng-core. 2024.

[188] *hackaton-euraz-2023*. See https://github.com/rerendering-evaluation/hackaton-euraz-2023. 2024.

[189] *hacker-news-app-2023*. See https://github.com/rerendering-evaluation/hacker-news-app-2023. 2024.

[190] *healthCare*. See https://github.com/rerendering-evaluation/healthCare. 2024.

[191] *honey-rae-repairs*. See https://github.com/rerendering-evaluation/honey-rae-repairs. 2024.

[192] hongtaodai. *react-excel*. See https://github.com/hongtaodai/react-excel/2d59e85. 2023.

[193] *hooksMixed-1*. See https://github.com/rerendering-evaluation/hooksMixed-1. 2024.

[194] hoverGecko. *timetable*. See https://github.com/hoverGecko/timetable/0fa8527. 2023.

[195] *laser-shooting*. See https://github.com/react-rerendering-benchmarks/laser-shooting. 2024.

[196] mishoo. *uglify-js*. See https://www.npmjs.com/package/uglify-js. 2023.

[197] sadupawan1990. *excelreader*. See https://github.com/sadupawan1990/excelreader/4a5f9cb. 2023.

[198] thewca. *scrambles-matcher*. See https://github.com/thewca/scrambles-matcher/1de93f7. 2023.

[199] ultimateakash. *react-excel-csv*. See https://github.com/ultimateakash/react-excel-csv/18c6d97. 2023.

[200] vishumane. *ExcelSheet_Validation_Reactjs*. See https://github.com/vishumane/ExcelSheet_Validation_Reactjs/f38cb9e. 2023.

[201] webpack. *Tree Shaking*. See https://webpack.js.org. Also, see https://webpack.js.org/guides/tree-shaking/#root for tree shaking. 2023.