Dynamic Taint Analysis in JavaScript for JavaScript

Submitted in partial fulfillment of the requirements for

the degree of

Master of Science

in

Information Security

Wai Tuck Wong

B.S., Information Systems, Singapore Management University

Carnegie Mellon University
Pittsburgh, PA

May, 2020

Acknowledgements

I would like to thank Professor Limin Jia for supervising and guiding me through this field of research - I have learnt countless things and was exposed to new ways of thinking which will prove to be invaluable in the future. I am grateful for having Professor Lujo Bauer as a reader of my thesis, providing insights and pointing out a corner case in my implementation of the static analysis. This work is done together with the team led by Darion Cassel, together with Spencer Yu. I am really grateful for the all feedback from the folks in Professor Limin Jia's group at Cylab for my defense. Finally, I would like to thank everyone at the Information Networking Institute for being supportive throughout my journey at Carnegie Mellon University. It was a tremendously fruitful experience - one that I would remember in the days ahead. The work in this thesis is self-funded.

Abstract

In recent years, we have seen a rise in the number of applications built on JavaScript, bolstered by the increase in popularity of application frameworks such as `NODE.JS`. In particular, `NODE.JS` provides convenience for developers through packages which they can import functionality from. This has attracted malicious actors to turn their focus to find ways of exploiting such packages. In fact, code injection attacks are prevalent in the ecosystem - they allow for arbitrary code or commands to be run - and are the most critical vulnerabilities in the language. Unfortunately, JavaScript is a complicated language, and significant research effort has been invested in developing dynamic taint analysis tools to identify such vulnerabilities at scale. Prior work attempted to build such tooling on top of JavaScript engines, but such attempts are highly unmaintainable due to the rate of change in both the language and the engines they run on. Other platform independent approaches lack the flexibility in keeping track of taint for individual characters in strings. In this thesis, we propose a flexible framework for doing dynamic taint analysis purely in JavaScript that is capable of byte level tainting.

We present NodeTaintProxy, a dynamic taint analysis engine built via augmenting the instrumentation engine, Jalangi2. Here, we developed a novel approach in dealing with primitives by specifying the behavior for wrapping, and we show that this behavior respects the semantics outlined in ECMA-262. We also show how taint propagation in dynamic code generation can be handled through a mix of code rewriting and static analysis on the dynamically generated code. Finally, we evaluate our tool on existing vulnerabilities in `NODE.JS` packages and show that our tool is successful in finding these vulnerabilities.

Table of Contents

Acknowledgements **ii**

Abstract **iii**

List of Tables **vii**

List of Figures **viii**

1 Introduction **1**

1.1 NODE.JS and the NPM Ecosystem 2

1.1.1 Security Concerns in NPM 3

1.2 Existing Approaches to Finding Code Injection Vulnerabilities 6

1.2.1 Static Analysis 6

1.2.2 Dynamic Analysis 7

1.2.3 Dynamic Taint Analysis 7

1.2.4 Challenges for JavaScript 8

1.3 Outline 9

2 Overview and Background **11**

2.1 Overview 11

2.1.1 Primitives 13

2.1.2 Objects 14

2.1.3 Dynamic Code Generation 15

2.2 Instrumentation and Reflection 15

2.2.1 Reflection in JavaScript via Proxy 16

2.2.2 Instrumentation of JavaScript Operations 17

2.3 NODE.JS Features and Vulnerabilities 17
2.3.1 Code Injection Vulnerabilities 18
2.4 Dynamic Taint Analysis 19
2.4.1 Definitions 20
2.4.2 Taint Propagation 21
2.4.3 Explicit and Implicit Flow 23

3 Related Work 25
3.1 Dynamic Taint Analysis in JavaScript 25
3.1.1 Taint Analysis on the JavaScript Engine 26
3.1.2 Engine Independent Dynamic Taint Analysis for JavaScript . . 28
3.1.3 Handling of Dynamically Generated Code 33
3.2 Analysis of NPM Packages 35
3.2.1 Ecosystem Analysis of NPM Packages 35
3.2.2 Security Analysis of NPM Packages 36

4 NodeTaintProxy 38
4.1 Architecture 38
4.1.1 Overview 39
4.1.2 Uniquely Identifying Primitives via Wrapping 39
4.1.3 Initializing the Analysis 40
4.1.4 Operation Semantics of Our Tool 40
4.1.5 Taint Tracking and Propagation 42
4.1.6 Architecture by Layers 44
4.2 User Interface - Interaction via Ghost Functions 47
4.2.1 Setting up Sinks 48
4.2.2 Setting up Sources 48
4.2.3 Running the Analysis 48
4.3 Handling Tainted Dynamic Code Generation 50
4.3.1 Challenges 50
4.3.2 Overview of Approach 51
4.3.3 Limitations of Approach 58

4.4 Discussion 58
4.4.1 JavaScript Semantics 59
4.4.2 Native Functions 59
4.4.3 Precise String Tainting 60

5 Results and Discussion 61
5.1 Dataset 61
5.2 Results on Case Studies 62
5.3 Discussion 63
5.3.1 Successful Detection 63
5.3.2 Unsupported Operations 64
5.3.3 Overwrapping 64
5.4 Feasibility Study 65
5.4.1 Challenges for Large Scale Analysis of Packages 65
5.4.2 Methodology of Feasibility Study 67
5.4.3 Analysis of Feasibility Study Vulnerabilities 68
5.4.4 Insights from Feasibility Study 68
5.5 Threats to Validity 69

6 Limitations and Future Work 73
6.1 Overwrapping of Primitives 73
6.2 Better Language Support 74
6.3 Large Scale Analysis 76

7 Conclusion 77

Bibliography 80

Appendix A Sample Test Program 89

List of Tables

Table 1.1 ECMAScript Standards and Revision Dates 2

Table 5.1 NodeTaintProxy Evaluation Results 71

Table 5.2 Code Injection Vulnerabilities Found in Feasibility Study 72

List of Figures

Figure 2.1 Overview of Internals of NodeTaintProxy 12

Figure 4.1 Segregated Architecture of NodeTaintProxy 47

Figure 4.2 Output of Running the Analysis for the module *ps* 49

Symbols

M_W The Wrapper map that stores mappings of object references to their unique IDs

M_E The shared context map that maps variables to be tainted to their unwrapped values

M_T The Taint map that stores mappings of IDs to their taint entries

ID A unique identifier assigned to a particular object reference

Abbreviations

JS	JavaScript
ES5.1	ECMAScript 5.1
ES6	ECMAScript 2015/ECMAScript 6
API	Application Programming Interface
ACI	Arbitrary Command Injection
ACE	Arbitrary Code Execution
AST	Abstract Syntax Tree
NPM	Node Package Manager
XSS	Cross-site Scripting
LHS	Left-hand side

1

Introduction

JavaScript, for the seventh year in a row, is the most popular language - according to a survey done by StackOverflow on nearly 90,000 developers. In fact, JavaScript is the most commonly used programming language among both amateur and professional developers [21]. JavaScript thrives because of its versatility - first introduced in September 1995 [26], it provided interactivity to users on the web in an era where static web pages were previously the norm. It has since been standardized under the ECMAScript standard [43]. The language evolves quickly, with revisions every year and a living standard. Table 1 describes the list of ECMAScript standards published over the years and the years they were published [42].

To no one's surprise, JavaScript became the de facto standard for programming on the web, and with the introduction of frameworks such as `NODE.JS` and Electron, they soon became accessible to server-side developers and desktop application developers. In fact, `NODE.JS` is most commonly used framework among developers, according to the same StackOverflow survey [21]. With such prolific use, our thesis will focus on `NODE.JS` - we want to find vulnerabilities in this platform before the attackers do. To do that, we need to understand the ecosystem and see how our approach fits in

ECMAScript Version	Date Standardized
ECMAScript 1	1997
ECMAScript 2	1998
ECMAScript 3	1999
ECMAScript 5	2009
ECMAScript 5.1	2011
ECMAScript 2015	2015
ECMAScript 2016	2016
ECMAScript 2017	2017
ECMAScript 2018	2018
ECMAScript 2019	2019
ECMAScript.Next	Living Standard

Table 1.1: ECMAScript Standards and Revision Dates

this space, and we will elaborate more on this point in the section below.

1.1 NODE.JS and the NPM Ecosystem

NODE.JS was first introduced as a way of running JavaScript on the backend and it soon became popular among developers as a framework to use for developing server-side applications. With a budding community support, NPM (Node Package Manager) was introduced as a way of allowing developers to leverage on modules created by others in their own projects by pulling from a common NPM Registry. As of February 2019, it houses over 800,000 packages in the registry [83].

However, such a registry is not without issues. In a typical NODE.JS application, developers simply import third party modules from NPM, without verifying the functionality of these modules. In the developer's mind, they have a mental model of how the underlying package ought to behave. For example, they infer that the underlying package is well-implemented and tested from the large number of downloads. These assumptions are frequently untested and may be violated by the underlying package [33].

Developers who rely on packages from the registry implicitly trust the packages they use and all of the packages' dependencies. This has several issues. There is no guarantee of code quality of the underlying packages. It might even be the case that the contributor themselves cannot be trusted [83]. Cases of typosquatting occurred many times throughout the history of NPM, where misspelled package names led to malicious packages being installed on the unsuspecting developer's system [39]. While a powerful resource for developers, they may not understand the full repercussions of using a package from the repository, and may end up introducing vulnerabilities to their applications without them even realizing it.

1.1.1 Security Concerns in NPM

Here, we consider from the perspective of a developer the potential security problems that they might face, if they were to leverage on the library of existing packages in the NPM registry.

1. **Unvetted Publishing Model.** We first note that publishing to the NPM repository is completely unvetted. All it takes to publish is a simple command `npm publish` [3], and the package is available to the whole world. The publishers are responsible for the maintenance of the package. Code quality between packages in the ecosystem therefore varies widely when there's completely no review process. Even if a vulnerability was found in a package, the package remains on the NPM registry for download, and only a warning is displayed if it was downloaded. This decision was made to maximize availability, to allow packages that depend on the vulnerable package to continue to work. For example, the `node-serialize` module [9] continues to see over 1000 weekly downloads even after a critical vulnerability was discovered, and still sees use despite there being no patches available for it [13].

2. **Untrusted Authors.** Since the public NPM repository is unvetted, anyone,

including malicious actors, can simply upload a package containing a backdoor that runs malicious commands on the victim's machine. An unknowing developer may download such a package and find his application or even machine to be compromised. This is a real danger that developers worry about - in a survey done by NPM of 16000 `NODE.JS` developers, 77% of respondents were concerned about the security of open source code [28]. Their fears were not unfounded. In 2018, a popular `NODE.JS` module, Event-Stream, had their code backdoored with malicious code that stole private keys from the Bitcoin wallet CoPay [38]. It was undiscovered for over a month, affecting 3931 packages that depended on it [45]. We see that the highly interdependent nature of the NPM repository is fragile - a vulnerability or malicious code in a single package can have ripple effects and induce vulnerabilities in all packages that depend on it.

3. **Full Privilege of Package Code.** To exacerbate the problem, the packages not only have the same privileges as the user at the point of installation (through the install scripts), the packages themselves run without explicit authorization on what resources they are allowed to access [71]. For example, a package that provides a wrapper to delete files may do it via running the shell command `"rm -f <file>"`, thus leading to a command injection vulnerability through the `file` argument, and the unsuspecting developer using the package will not be aware of this underlying vulnerability unless they go in to audit the source code manually.

Without the proper procedures or tooling, vulnerabilities will creep into ecosystem and packages containing vulnerable code will be exploitable. In particular, JavaScript is prone to code injection attacks on many fronts - on the browser, it manifests as a cross-site scripting (XSS) attack, while on server-side applications, they could be arbitrary code execution or command injection attacks (or broadly speaking, code injection attacks). In fact, these attacks are so prevalent that the OWASP Foundation lists these vulnerabilities as the top 10 web application security risks [17].

Client-side vulnerabilities have been well studied [67][62][54] and will not be a focus in our evaluation. We will focus on code injection attacks for server-side applications in the rest of the thesis, in particular for code injection vulnerabilities for packages in `NODE.JS`. We will present more details on code injection vulnerabilities for `NODE.JS` packages in Chapter 2.

On `NODE.JS`, we see a ripe set of tools that an attacker can leverage to interact with the system. Below, we show how these vulnerabilities can manifest in a package used by the developer.

1. **Input provided to shell commands.** In implementing certain functionality, some package developers may choose to implement an external script or leverage on an existing system binary. In their code, they would pass the arguments directly to the executable file using unsafe APIs such as `exec` or `execSync`. This can lead to command injection attacks.

2. **Evaluation of input.** Package developers are known to use `eval` as a tool to evaluate user input in a power user fashion to get *"cleaner"* code, despite the known dangers of doing so [82]. An example of this is the `mongui` package [76], which uses `eval` so that users of the package have a *"more natural programming interface"*. Such unrestricted evaluation of user input can lead to arbitrary code execution.

In fact, for packages that exhibit the above behavior, very few, if any of them, use any form of sanitization, and they rarely document that untrusted input should not be passed into the interface of the package. Here, *sanitization* refers to removing or encoding some parts of the input such that it neutralizes malicious behavior that can result from the input [5]. In a study of injection attacks, `growl`, a highly popular package that passed input directly to `exec`, was discovered to have as many as 15 packages that depended on it that did not do any checks or sanitization [71]. The 4 packages that attempted to do sanitization were found to be bypassable. As we see here, these are real problems in the NPM ecosystem. In the next section, we detail potential ways of finding such vulnerabilities so we can solve the problem at hand.

1.2 Existing Approaches to Finding Code Injection Vulnerabilities

What we want to achieve is a way for developers to audit their dependencies for code injection vulnerabilities. More broadly, such a tool would also allow security analysts to probe at different packages and their interfaces and see if an input provided to the package interface would lead to code injection, such as calling `eval` on the input or the `exec` function of the `child_process` module on the input.

1.2.1 Static Analysis

One such way of determining whether an input would lead to undesired behavior is to statically analyze the source code of the application. For example, one can scan the underlying source code to see if there are any uses of `eval`, and then manually evaluate whether that usage is safe [71]. More advanced methods of static analysis (via static taint analysis) considers whether the user input could possibly flow to the function call. Given the source code of the application, a data flow graph is constructed, where an edge exists from one variable to another if and only if there is an assignment from the source variable to the target variable [36]. An advantage is that no concrete input needs to be provided to the package being analyzed. However, such an approach is would be highly inaccurate. Without runtime information, it would be very difficult to determine whether a branch would be taken, for example.

Static analysis requires us to reason about all execution paths, which inevitably leads to a state explosion, and potentially many false positives. Furthermore, we will not be able to determine actual runtime behavior where some parts of the code will not run, for example when an exception is thrown in the code or when dynamically generated code is supposed to run, which leads to us missing more cases than what we would like.

1.2.2 Dynamic Analysis

This naturally moves us to trying a different technique. A different approach uses dynamic analysis, where we have access to runtime information by providing a concrete input to the package's interface. We then check at the end whether some undesired function is triggered (e.g. `eval`) [44]. While this has the advantage of reducing false positives over the above approach, we cannot ascertain whether the input has any influence over the parameters provided to the undesired function. More concretely, we wish to figure out if the attacker/adversary has control over the arguments to the undesired function from the adversary's input alone. This however is not possible purely with Dynamic Analysis. In particular, just providing random inputs to the package's interface do not give us a principled way of analyzing the influence of the input on the end result [59].

1.2.3 Dynamic Taint Analysis

This moves us to the technique that we propose to use - dynamic taint analysis. In this case, we mark the input to the package as tainted, and we propagate the taint on the input to other variables if the variable was affected by the input [60]. At the end, we check to see if the undesired function was called with an argument that could potentially be controlled by the adversary (i.e. the argument was tainted). This gives us better results in terms of reducing false positives (since we know the argument is indeed attacker controlled), but in exchange we need a more complicated infrastructure to keep track of taint and propagate taint as the JavaScript code runs. We will discuss more details on the background of this approach, related work, and how we implement this approach in the later chapters.

1.2.4 Challenges for JavaScript

In building a platform agnostic dynamic taint analysis framework for JavaScript, we will face challenges from the language itself. Below, we summarize the key challenges in dealing with the language.

1. **Complexity of Semantics.** JavaScript is a complicated language which prevents introspection into its internals. For example, we won't be able to view the heap address of a JavaScript object, or view certain properties which are *internal* to the JavaScript object. Some properties are read-only and prevent modification from within JavaScript itself. Furthermore, JavaScript has rather complicated semantics, for instance the equality operator `==` is neither reflexive nor transitive [52]. In dealing with JavaScript, we must be careful not to make any assumptions which do not adhere to the underlying semantics of JavaScript, lest we end up with a buggy implementation of our tool.

2. **Native Functions.** Much like how C syscalls are a problem [68], native functions are the equivalent construction in JavaScript, where only the input and result of the computation are visible from JavaScript itself. This lack of transparency forces us to adopt workarounds to ensure our tool still works as expected, as we will see later.

3. **Dynamically Generated Code.** Finally, JavaScript allows for additional code to be generated and evaluated at runtime through the `eval` function and `Function` constructor. Usage of this feature is more common in reality than

once believed [65], so we will have to ensure that our tool works even in the case where code is dynamically generated. In cases where `eval` is not the sink, we want to make sure that taint is propagated correctly as well, even when the argument to `eval` is tainted.

We will outline the background required to tackle these problems in Chapter 2 and detail how we deal with the above problems in detail in Chapter 4.

JavaScript is a challenging language to analyze, but it has a large impact in the real world, where an entire ecosystem of packages could be analyzed with the right tooling. As of now, the developer has to implicitly trust the modules he utilizes without a way of determining whether the package does what he expects. We want to provide tooling to allow analysts to find code injection vulnerabilities faster, and even allow developers to find such vulnerabilities in their own code base.

We are motivated to create a state of the art dynamic taint analysis framework that allows us to analyze JavaScript code for code injection vulnerabilities; we make it platform agnostic by implementing it on top of existing instrumentation frameworks that employ source-to-source rewriting. Through this, we can do taint tracking and propagation, and we further augment it with the ability to propagate taint even when the argument to `eval` is tainted, using a novel way of tainting via program rewriting and static analysis. We show that our approach works on the vulnerabilities described earlier through a series of case studies that display code injection vulnerabilities. Our current implementation focuses on `NODE.JS`, but the implementation design should minimize effort required to port to a different JavaScript platform.

1.3 Outline

The thesis is broken down into the following parts. First, in Chapter 1, we outline the motivation in carrying out the development of a dynamic taint analysis tool for

JavaScript, specifically to analyze NPM packages. Next, we outline the necessary background needed to understand our approach in Chapter 2. In Chapter 3, we look at prior work that has been done in developing dynamic taint analysis for JavaScript in different areas, as well as look at other large scale studies done in the NPM ecosystem, and we compare our approach to theirs. In Chapter 4, we explain our tool, NodeTaintProxy, and we describe how we approach maintaining and propagating taint in JavaScript, while at the same time maximizing our adherence to the underlying semantics. We also show our novel method that propagates taint in dynamically generated code through program rewriting and static analysis. In Chapter 5, we show that our tool is able to detect code injection vulnerabilities that were made public recently, as well as case studies that were selected from prior work. We also show that our approach is promising in finding new vulnerabilities in a feasibility study in that chapter. We then round up our discussion with limitations and future work for the framework in Chapter 6. Finally, we conclude by summarizing our key findings in Chapter 7.

2

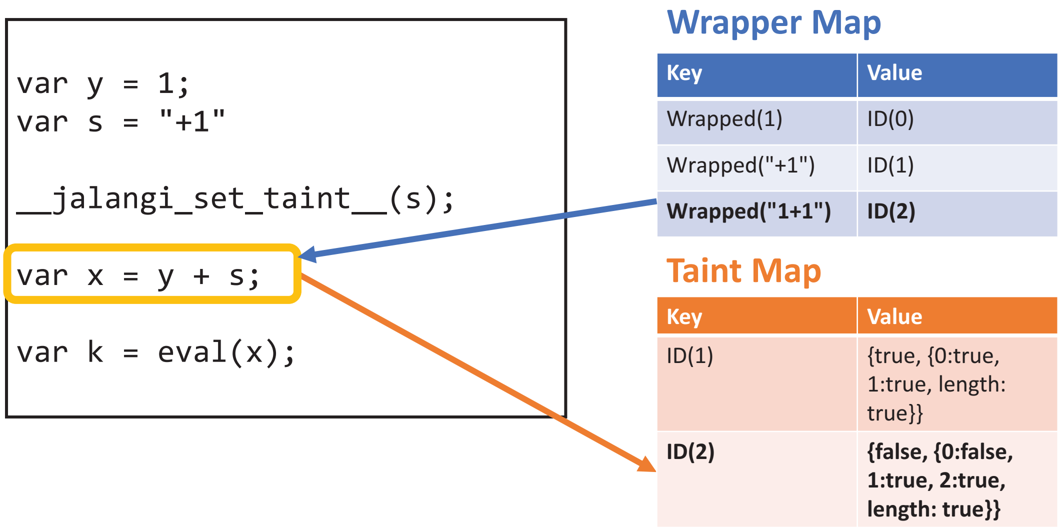
Overview and Background

In this chapter, we first briefly outline the construction that our approach will take. We will then delve deep into the background required to understand each component of our approach in the subsequent sections of this chapter.

2.1 Overview

In Figure 2.1, we provide an overview of the internals of our approach. On the left side of the diagram in Figure 2.1, we outline the source code of a simple JavaScript program running on our framework, which is a `NODE.JS` package under inspection. On the right, we show the data structures that are involved in the framework. The wrapper map keeps track of a mapping of object references to unique identities of all the values we have seen thus far, and the taint map keeps track of a mapping of the unique identities to their taint information. We will elaborate more concretely about the data structures in Chapter 4.

The code referenced simply declares two variables y and s in lines 1 and 2, taints the variable s in line 4, declares a new variable x that is set to the value `y+s` in line 6 (which is the string `‘1+1’`) evaluates the string dynamically in line 8. The data structures depicted shows the internal state of our framework after line 6 has executed. For example, x has `ID(2)` since the ID associated with the string `'1+1'` is `ID(2)` in our wrapper map. Furthermore, we note that the indices of x that are tainted are the characters `'+1'` at the end of the string. We can tell that from the taint map, which tells as that indices 1 and 2 of the string are tainted for the value associated with `ID(2)`. This makes sense, since the tainted values were derived from a tainted variable s which has the value `'+1'`. We see that s was tainted since the value of s maps to `ID(1)` in the wrapper map, and in the taint map, `ID(1)` maps to a taint entry that tells us that the whole string was tainted (as we can see from the `true` in the first entry of the tuple). We note some interesting problems that this piece of code raises in our analysis.



```
var y = 1;
var s = "+1"

__jalangi_set_taint__(s);

var x = y + s;

var k = eval(x);
```

Wrapper Map

Key	Value
Wrapped(1)	ID(0)
Wrapped("+1")	ID(1)
Wrapped("1+1")	**ID(2)**

Taint Map

Key	Value
ID(1)	{true, {0:true, 1:true, length: true}}
ID(2)	**{false, {0:false, 1:true, 2:true, length: true}}**

Figure 2.1: Overview of Internals of NodeTaintProxy

On line 1, we declared a variable y with value 1. The identity of this value 1 should differ from another variable which is also have a value of 1. We see that in this case, we have wrapped the value and mapped it to a unique ID, `ID(0)`, in our wrapper map. More generally, we need to wrap primitives which do not have references in JavaScript so we can uniquely identify them. We do so with a object class called *Proxy*. We elaborate more on what primitives are in JavaScript, and what a *Proxy* is, and how it is useful for our analysis in Sections 2.1.1 and 2.2.1 respectively.

On line 4, we tainted the variable s, and we used it to compute the value of the variable x in line 6. This means that taint needs to propagate from the variable s to the variable x, and the taint map has to be updated (as indicated by the orange arrow), so that we can keep track of taint of every value in the system. We do so by hooking each operation in JavaScript using instrumentation, which allows us to define custom behavior before and after each operation. We will explain what instrumentation is and how it is achieved in Section 2.2.2 in this chapter.

Finally, on line 8, a variable k was declared by dynamically evaluating the string `"1+1"` stored in the variable x. Since x tainted, the new variable k should also be tainted. More generally, variables introduced in the dynamically generated code should also be tainted. We look at how dynamic code generation is achieved in JavaScript and note specific details we have to take care of in our framework in Section 2.1.3.

The simple example above highlights the need to understand the underlying language and all its nuances in order to perform dynamic taint analysis correctly. Furthermore, JavaScript provides some useful classes which we will leverage on to build our framework. In this section, we will describe important aspects of JavaScript that we have to be aware of when we develop our tooling. All of the description below are based on the latest edition of the ECMAScript standard, available here [43].

2.1.1 Primitives

The simplest types in JavaScript are the built-in primitive types, which are **Undefined, Null, Boolean, Number, String** and **Symbol**. These primitive types behave as you would expect, much like normal primitives in standard imperative programming languages. They are treated as values in the context of JavaScript, which means when comparing them, we do not compare their pointers, but simply compare their values. The standard unary and bianry operators apply to JavaScript, although some of these operators are overloaded to coerce one type to another (for instance the unary operator `+` is used to coerce any type to a Number).

Primitives are not objects, and therefore do not have methods. Most operations in JavaScript are however defined on Objects, and through type coercion, primitives will be promoted to Objects before methods are called on them. For example, the primitive `1` is casted to the object `Number(1)` before calling a method like `toString` on it. Fundamentally, JavaScript is an object-based language, and we will detail what objects are in JavaScript below.

2.1.2 Objects

A JavaScript object is simply a collection of zero or more properties (key-value pairs), where the corresponding keys and values may be of any type. Each property may have attributes to control how it may be manipulated (for example, the Writable attribute for a property controls whether JavaScript can change the property).

Many built-in object types exist, some of them fundamental to the semantics of the language (such as **Object**), others serve as object representations of the underlying primitives (such as **Number**). A function is simply a special instance of a built-in object which is callable.

For many of these built-in objects, they have methods that are inherited by the child instance of the object. Such functions are often *native functions*, i.e. they are evaluated in the C++ JavaScript runtime in order to get optimal performance. For example, an operation to find the the index of an element in an array through the `indexOf` method of the Array prototype is handled in the C++ runtime (even though it can be implemented in JavaScript as well), and the result is returned as a value accessible in JavaScript. This implies that any operation done by the runtime will not be visible to any code running on JavaScript, a fact that we will have to deal with in the implementation.

Now that we have an overview of the language, we will next look at dynamic code generation, a powerful language feature of JavaScript which we will have to handle in our framework.

2.1.3 Dynamic Code Generation

In JavaScript, code can be generated and evaluated during runtime using the `eval` function and the `Function` constructor. The dynamically generated code is not isolated and executes in the current context. Hence, it is able to access and modify the current scope and global scope. This means that a variable created or modified in the dynamically generated code will be accessible to the static code executing in the same scope later. What this means for us is that taint propagation and tracking must still run even in the presence of dynamically generated code. Another feature of `eval` is that the value of the last evaluated statement becomes the return value of the `eval` function, so that must be respected as well in our framework.

With knowledge of how to handle unique cases in JavaScript, we move on to the next part of our framework - to propagate taint via reflection and instrumentation.

2.2 Instrumentation and Reflection

As mentioned in our overview, being able to intercept operations and customize behavior is key to taint tracking and taint propagation. Since the state of the system changes every time an operation occurs in JavaScript, this allows us to update our current knowledge of what variables are tainted in the system. In this section, we detail reflection done via the *Proxy* object class in JavaScript, as well as describe how instrumentation in JavaScript is done.

2.2.1 Reflection in JavaScript via Proxy

The *Proxy* class is a special kind of object class that allows us to perform reflection in the language, such as intercepting property accesses and changing the values returned of a particular object instance. By wrapping values with a Proxy, we can then define custom behavior for accesses to the properties of the value that we wrapped (the proxied value).

However, using a Proxy is also not without any downsides; in particular, we still need to ensure transparency of the proxy. When we say a Proxy is *transparent*, we mean that operations on the Proxy should be semantically equivalent to the proxied value. While this is largely true, it does not hold for certain operators. For example, the equality operator `==` is an issue in this case. The expression `1 == 1` will give unexpected results when the primitives are wrapped, and in general we want to make sure JavaScript works as expected in our framework if we wrap the primitives.

To elaborate on the point above, wrapped primitives exist as different Proxy objects which are unique JavaScript objects at runtime, so their references will not be equal [52]. To maintain semantic equivalence, we want to maintain transparency when we use proxies in our framework for interception on the underlying primitives or objects. Without modification to the semantics on the Proxy, the expression in our example will evaluate to `false`.

Proxies alone are not enough to intercept all operations in JavaScript. For instance, anything not involving objects like a binary operations (for instance, the `+` operator) on primitives will not be intercepted. To intercept those operations, we use a far more powerful framework that allows us to intercept all operations.

2.2.2 Instrumentation of JavaScript Operations

Propagating taint for each operation requires us to perform operations before and after each operation, and to do so, we need hooks into every operation in JavaScript - this is generally known as instrumentation. This can mainly be achieved in two ways. The first way looks at modifying the underlying JavaScript interpreter so that every interpreted bytecode is instrumented (i.e. there is a hook that is called before and after every instruction). This was achieved in LeakTracker's modification to Firefox's JavaScript interpreter [77]. Another way to achieve this is via program rewriting, much like in the Jalangi2 framework [69]. The framework rewrites code to install hooks for analysis, and supports JavaScript code written in ES5.1 and lower. The code to be instrumented are rewritten recursively so that both the script and its dependencies are instrumented. The rewritten program contains hooks which are triggered before and after each operation, allowing the analyst to define and customize every operation in JavaScript (for example, `invokeFunPre` is triggered before every function call, `instrumentCode` are triggered after `eval` is called). In our approach, we will be using Jalangi2. An unfortunate consequence is that Jalangi2 currently only supports up to ES5.1, so JavaScript code that uses newer features will have to be transpiled down ES5.1 to operate within our framework.

With the above background knowledge on JavaScript, we look next at the specific JavaScript context to which we wish to apply our method. We will focus our attention on code injection vulnerabilities, which are the vulnerability classes we are interested in detecting for `NODE.JS`.

2.3 NODE.JS Features and Vulnerabilities

As mentioned in Chapter 1, we will be evaluating our framework on `NODE.JS`. The APIs of the framework provide useful functions that allow a developer to interact with the underlying operating system. These APIs include functions that run shell commands, interact with the file system and send packets over the network. These functions defined in NODE.JS are leveraged by developers to create powerful applications. For example, web servers and even full-fledged command line utilities can be written with such APIs. The APIs can also be combined to create packages (such as express) that provide export functionality, which can then published on the NPM repository so that they can be used by everyone else. Unfortunately, as mentioned previously, code injection vulnerabilities are sometimes present in these packages, as we will see below.

2.3.1 Code Injection Vulnerabilities

Code injection is defined in the Common Weakness Enumeration (CWE) as a software vulnerability that allows users to provide input that modifies control flow of the underlying program [4]. This is especially relevant in NODE.JS - a large number of advisories published at vulnerability databases such as Snyk for the NODE.JS framework [70] falls under this category. Note that code injection vulnerabilities are not the only vulnerabilities that are present in this ecosystem; there are other vulnerabilities classes present in NPM packages, such as Denial of Service via regular expressions, but we will not discuss these vulnerabilities. Instead, we focus on code injection vulnerabilities, since they are the most critical and can do the most damage if exploited. We list the two relevant code injection vulnerabilities for NODE.JS below.

Arbitrary Command Injection (ACI)

When user-provided inputs are passed directly to shell commands, attackers can leverage known shell escapes and inject additional commands which will run with the same privilege as the user running the node process. For example, an attacker can insert a semicolon in his query like so: (“`rm -f file; ...`”) so that he may run additional commands on the system. This could lead to a full compromise of the system, depending on the privileges of the node process [50].

Arbitrary Code Execution (ACE)

Similarly, if user-provided inputs are directly evaluated via functions like `eval` or `new Function()`, this leads to a code injection vulnerability as well. An attacker can leverage this by injecting malicious JavaScript code that will be evaluated. For example, the code can call any function or modify any variable accessible within the scope of the call to `eval`. Furthermore, this attack even allows an attacker to run shell commands with the same privilege as the user running the `node` process, by importing the `child_process` module and running the `exec` function from the module [19].

Note that these are severe vulnerabilities that have the potential to lead to a full compromise of the targeted system, since the attacker can run any they wish. Now that we have a clearer picture of the vulnerabilities that we wish to find and their severity; we will next explain the technique used to find these vulnerabilities in our context - dynamic taint analysis.

2.4 Dynamic Taint Analysis

Dynamic taint analysis is a technique used to analyze data flow from one variable to another variable using information given or extracted at runtime [68]. We do so by keeping track of the influence of a variable in the entire system, (i.e. we can see how other values in the system are affected by the original variable). In our context, this means that we are able to keep track of how user provided input is used throughout the program and the dependencies of the program, so if a user provided input could influence the argument to sensitive functions like `eval`, we would be able to detect it and flag it as a vulnerability. Also, because it is dynamic, given the input to the program, we know that the variables that are marked as tainted at the end are indeed the variables that are likely influenced by the original input, since we know exactly which paths were taken by the program, unlike in static analysis.

While dynamic taint analysis has been used to find confidentiality leaks [40][37], here, we will mainly focus on the use of dynamic taint analysis to find vulnerabilities, like in [60]. Below, we will define some working definitions that will be used in the rest of the thesis.

2.4.1 Definitions

We first present an overview of some terminology that we will be using for the rest of the thesis. We borrow terminology used in [68] which is now standard terminology in this domain.

1. **Source.** A source refers to locations where taint will propagate from. Concretely, these are the variables that are directly controllable in some fashion by a user or attacker and are provided as input to the package's public interface. By identifying a source, we will be able to understand whether the input from the source influences the internal state of the system in a way that we deem to be sensitive or dangerous through dynamic taint analysis. An example of a source would be a string input provided by a user in an application that we wish to inspect.

2. **Sink.** Earlier, we have defined code injection vulnerabilities. These vulnerabilities are the result of passing untrusted input to vulnerable functions, like `eval`. Intuitively, when we encounter such a function and the attacker controls the argument to the function, we want flag an error to the analyst and halt execution of the program. These functions are known as *sinks* in dynamic taint

analysis. As part of our analysis, we need to identify sinks that will be used by our framework, for example, functions that are vulnerable to ACI, like the `execSync` function, which simply passes the arguments to a shell. This means that attackers can chain additional commands if they control the input to the function. When the attacker controls the argument at the source and it influences the argument at the sink, we say that the data flows from the source to sink, and we have reached the sink with a tainted input, and we flag the result to the analyst.

3. **Taint tracking.** Finally, we need a way of keeping track which variables are affected by the input, or the source. We may think of the taint as a flag that tells us whether an attacker has some influence over the variable. Obviously, the source must be tainted (it is defined to be attacker controlled!), but we must also mark values that are affected by the attacker controlled variable as tainted (for example, if they were assigned to the variable directly), and those variables that depend on tainted variables should themselves be tainted. In the next section, we will elaborate on how taint is transferred from one variable to another, in the process of taint propagation.

2.4.2 Taint Propagation

The taint flag is associated with every value *influenced* by the source, which includes the source itself. However, we have yet to define the notion of influence, and indeed this varies from implementation to implementation. Broadly speaking, we want to propagate taint from one value to the next if there is a data flow between them. For example, if a variable was assigned to a tainted value, then the variable should be tainted. When this happens, the tainted flag on the variable is set, and the taint would be propagated. However, there are many situations where the degree of influence is ambiguous (for example, in implicit flows, which we will discuss at the end of the chapter), and individual implementations have to make decisions to balance the tradeoffs of the taint analysis. These decisions are usually summarized in a *taint propagation policy* defined by the implementer [57].

Precise vs. Imprecise Tainting

Another part of tainting we have yet to discuss is the granularity of tainting. Previously, we have worked with values and variables in our definitions, but here, we will make clear what these values and variables refer to, and give two working definitions that we will use in our thesis.

1. **Precise Tainting.** We say the tainting is precise in the context of JavaScript if for a given tainted string, we know which exact bytes of the characters are tainted, and for an object, we know which properties of the object are tainted. Note that arrays are objects in JavaScript, and the array indices are properties of the array object. Intuitively, we want this notion of tainting because attacks in JavaScript tend to happen from string operations (e.g. passing a unsanitized string to a vulnerable function, as described in the above sections), so keeping track of taint on individual characters of the string allows us to capture taint at a finer grain and reduce false positives [62].

2. **Imprecise Tainting.** We say that tainting is imprecise in the context of JavaScript if we do not know which exact properties of a object or indices of string are tainted, so we assume that the whole object/string is tainted. In this case, we may have *overtainted*. In other words, some properties of the object or indices of the string may not be controlled by the source, but we have mistakenly flagged them as tainted. In this case, we might find false positives at the end of the analysis. This is however necessary if there is no way of figuring out the exact indices/properties that needs to be tainted during

taint propagation (e.g. when we have no way of observing how the input was interacted with, for example in a native function).

We want to be precise as much as we can in our analysis to reduce false positives, but at the same time we still want to propagate taint even in cases where we are unsure of the exact location of the taint. In this case, we do not want to *undertaint*, that is, we do not want to lose track of influenced variables which could eventually end up at a sink. We overcome this problem by *overtainting*, where we fall back to an imprecise taint policy to reduce false negatives.

2.4.3 Explicit and Implicit Flow

Finally, we look at cases where data flow is ambiguous, i.e. there is an influence of the variable by the source, but the influence is not a direct flow of data. If we look at the example in Listing 2.1, we see that when the variable x is $true$, then the variable y is 1. Otherwise, when the variable x is $false$, then the variable y is 0. Just by inspection, x obviously influences the variable y, so one might think our analysis should taint y if x is tainted. However, data from x never flows to y directly (e.g. via assignment). This is an *implicit flow* of data, where the tainted value is used as a control flow dependency. In general, handling taint propagation in this case is context dependent [48].

```
function implicit_flow(x) {
    var y;
    if (x) {
        y = 1;
    } else {
        y = 0;
    }
    return y
}
```

Listing 2.1: Implicit flow in JavaScript

For the rest of this thesis, we will focus on explicit flow of data. The class of vulnerabilities that we wish to study (where user provided strings are passed to vulnerable functions) are derived from explicit flow of data, where the whole string or parts of the string are passed from one function or operation to the next [71]. We argue that implicit flow of data is less of a problem in this context, and later in our evaluation in Chapter 5, we show that none of the existing case studies with exploitable code injection vulnerabilities relied on propagation of taint for implicit flows.

3

Related Work

Below, we summarize the findings and methodologies from prior work, which has significantly influenced our approach taken in our implementation. We will first discuss methods employed for doing dynamic taint analysis in JavaScript. Then, we will look at evaluation that has been done on packages in the NPM ecosystem.

3.1 Dynamic Taint Analysis in JavaScript

One of the first practical implementations of dynamic taint analysis to find vulnerabilities was introduced in 2005 in the seminal work, TaintCheck [60], to find bugs in C programs. Since then, people have expanded on the work to see if the same technique can be applied in other settings [59][35][81]. In this section, we focus on the efforts that have been made in using dynamic taint analysis in JavaScript programs. Historically, researchers have been looking at applying dynamic taint analysis for JavaScript on the client side. In recent years, with greater adoption of JavaScript on the server side, more researchers have looked into ways of analyzing JavaScript on both platforms [34], which is the direction we are aiming towards in this thesis.

In the following sections, we break down prior work on dynamic taint analysis for JavaScript into two different sections. We first look at how dynamic taint analysis have been implemented traditionally, via modifications to the JavaScript engine. Then, we look at alternative approaches that leverage on program rewriting to achieve the same effect.

3.1.1 Taint Analysis on the JavaScript Engine

Nentwich et al. was one of the earliest to implement dynamic taint analysis on JavaScript back in 2007 [58]. In their work, they modify the Firefox web browser in order to detect potential cross site scripting (XSS) confidentiality attacks. They noted that modifying the underlying JavaScript engine was *"a considerable engineering effort"*. They describe their taint propagation policies for assignment, operations on values, control structures, function calls and dynamic code generation. In particular, they taint *conservatively* - for operations that they can't be sure which variables will be tainted, they taint every operation in the program. For example, a tainted scope is created for dynamic code generated by `eval`, meaning all operations under the scope of the dynamically generated program will be tainted. A similar rule is also applied for implicit flows, and knowledge of the implicit flow via static analysis is augmented on top of their dynamic taint analysis for handling such flows. However, the authors noted the false positives that occurred due to conservative tainting, which may not be desirable in a practical implementation.

Saxena et al. presented FLAX, an alternative implementation which modifies the browser engine for Safari to detect a different class of attacks - *client-side validation (CSV)* attacks, that occurs when client-side components use untrusted data [67]. In their approach, they modified the engine to produce a trace of the program execution. The trace produced is in a custom format, JASIL, which is a simplified intermediary language which supports a subset of JavaScript that is commonly used in real world applications. This simplified language allowed them to reason about character level precise dynamic taint analysis on the program execution trace. They leverage this dynamic taint analysis to perform blackbox fuzzing over the input space to find a witness - i.e. a working exploit for the bug. In their treatment, functions that perform dynamic code generation is considered a sink, and hence no further taint propagation is needed. Their approach is largely successful, with no false positives and even discovering as many as 11 vulnerabilities in the wild, out of the 40 case studies they looked at. However, completeness is an issue in their approach, since not all JavaScript operations are supported.

Other ways were soon developed to support the full semantics of JavaScript. In particular, instrumenting the JavaScript interpreter to allow for taint tracking became a popular approach. LeakTracker is one such example; they implemented taint tracking in Firefox's SpiderMonkey engine via instrumentation of the JavaScript bytecode interpreter [77]. They note that 2400 lines of code were added to do taint propagation and taint tracking. Their approach takes into account interactions between principals via *principal-based tainting* and does not propagate taint in trusted contexts to achieve better performance. They also explicitly handle dynamic code generation by propagating tags from the principal to the generated code, before relying on the instrumented bytecode to propagate taint.

Leveraging on new developments and expanding on the success of the approach of FLAX, in 2013, Lekies et al. published a large scale study of client side vulnerabilities (in particular, *DOM-XSS*, which is cross site scripting done via injecting directly to the document object of the page) [54]. The study was driven by tooling they developed, which modifies the open-source V8 JavaScript engine for Chromium and attaches taint information to every object in the JavaScript runtime. Using this modification, they were able to achieve full coverage of JavaScript APIs and adherence to all JavaScript semantics while still allowing taint to be tracked at a character precise level. Through a validation approach similar to FLAX, they applied a context dependent exploit generation algorithm to find working exploits on their targets. Using this method on Alexa's top 5000 websites, they identified a total of 6167 vulnerabilities spread over 480 domains. However, optimizations in the JavaScript engine would have to be turned off for operations on strings as taint information would be removed otherwise.

While such implementations on JavaScript engines offer advantages (such as the ability to observe the system state completely, and significant performance benefits), the engineering effort required is significant. These engines were engineered to have performance at the forefront, so augmenting taint information to these implementations would often result in further modifications in other parts of the engine that make assumptions on the structure of the JavaScript object. With the ongoing maintenance that is required for every update to the browser and new updates to the ECMAScript specifications, such engineering efforts are often unmaintainable in the long term [51]. In the context of offline analysis where run time performance is less of an issue, alternative approaches have been proposed to allow for such analyses to be used in new contexts where JavaScript can also be applied.

3.1.2 Engine Independent Dynamic Taint Analysis for JavaScript

To solve the problem of having a engine agnostic framework, two main approaches have been explored by prior work. The first attempts to solve the gap between security needs and practical usage needs by developing a JavaScript interpreter specifically to perform taint tracking. The second approach leverages on the existing JavaScript engine to remain semantically equivalent, and instead instruments each JavaScript operation via source-to-source rewriting so taint can be tracked and propagated accordingly.

Custom Implementation of JavaScript Interpreters

One approach taken by Hedin et al. involved building a JavaScript interpreter on top of JavaScript to keep track of information flow [46]. The authors noted that building it on JavaScript enabled a general deployment for their framework. Firstly, it could run on the Firefox browser simply by installing a browser extension containing the interpreter, *Snowfox.* Secondly, it could also run on `NODE.JS` simply by installing and using a package. The interpreter covers the full ECMAScript 5 standards and passes all standard tests in the SpiderMonkey test suite, hence proving that it keeps the same semantics as the underlying JavaScript engine. They later showed that such an approach can be used to detect client side vulnerabilities on the web [47]. However, one downside is that it only supports up to ES5 in non-strict mode, meaning that newer versions of JavaScript will not be able to run on their framework, with no clear, easy way of running newer versions, since each new feature would have to be implemented in their interpreter. The authors also note the inherent tradeoffs of doing an engine independent analysis. In particular, native functions would either have to be rewritten in JavaScript (in what they call a *deep model*) or a summary must be provided on how things should be tainted (in what they call a *shallow model*).

Such an approach, while semantically equivalent, suffers from much of the same problems as modifying JavaScript engines. Researchers who wish to extend or update their framework would realize that they have to modify much of the interpreter to get the results that they want, and it soon became clear that the engineering effort required is non-trivial. Therefore, better approaches were sought in recent years, particularly through program rewriting.

Program Rewriting

Much of the engineering effort of the previous approaches is actually derived from attempting to understand and modify the interpreter without breaking functionality. For modern JavaScript engines, we have seen that this is not an easy task, and it is unclear that a custom JavaScript interpreter makes the modification easier. Program rewriting offered a way of reducing the amount of modifications needed, by deferring all JavaScript operations to the underlying JavaScript interpreter, while enabling researchers to customize the operations that happen before and after each operation is performed (so taint can be propagated, for example).

One of the first successful attempts at using program rewriting for dynamic taint analysis of JavaScript was published in 2015 by Chudnov et al. [37]. In their framework, they perform source-to-source rewriting to inline a reference monitor that enforces information flow control policies. Their framework has almost complete support of the ECMAScript 5 standard, as well as support for web APIs, which is an important part of their target domain. They introduced many ideas that we will be using in this thesis. In particular, they propose that boxing of values would be the ideal way of handling values and storing taint information. We will leverage on this idea of *boxing* values with slight modifications in our implementation. They also perform *operation emulation* to emulate parts of the complex semantics of JavaScript in their inlining code, in order to accommodate the boxed values that now exist in the environment. One key idea that we will use here is deference to the actual runtime in order to minimize implementation of the actual semantics of the operation. However, in their implementation they still ended up reimplementing parts of the ECMAScript semantics because they have to emulate some parts of the operation in their environment. Each operation is converted to a monitor operation where they specify the semantics manually. For example, addition is converted to the `opadd` monitor operation. We will improve on this in our framework and with a more general approach to handling the operations while still being semantically equivalent.

In a similar vein to the boxing ideas in [37], Kannan et al. proposes using proxies to box values and rewriting via Sweet.js macros to instrument all operations in order to simulate virtual values, which can then be used for dynamic taint tracking [49].

The boxing idea was also used in DexterJS [62], which introduces their own source-to-source rewriting to achieve character level dynamic taint analysis as well as automatic patching of DOM-XSS vulnerabilities on the client-side [61]. Their boxing approach promotes primitives to objects so that taint can be tracked on primitives. We will see a similar approach in our implementation. They also perform automatic validation of vulnerabilities much like in FLAX [67], and through this they created a robust framework that found 820 zero-day DOM-XSS vulnerabilities on Alexa's top 1000 sites. Because the rewriting was done on a proxy server before being sent back to the user-agent, the rewriting is therefore browser agnostic. We argue that there are better ways of being engine agnostic without relying on a rewriting server, such as the pure JavaScript approach taken by JSFlow [46], as we have seen earlier.

With many custom implementations of instrumentation of JavaScript, it became clear that there was a gap caused by the lack of standard tooling. This has led to the implementation and reimplementation of frameworks with the same functionality. To fill this gap, the Jalangi framework was first developed in 2013 [69] which provides a simple way for researchers to add on dynamic taint propagation and even symbolic execution through utilizing the instrumentation hooks provided by the framework, with support on both JavaScript on the server-side and the browser-side. It was further improved and the new implementation, Jalangi2 [31], which was open sourced and had full support for all JavaScript APIs up to ES5.1. It has since been used in other applications, such as finding bugs via symbolic execution in the ExpoSE framework by Loring et al. in 2017 [56].

Building on top of Jalangi2, Karim et al. proposed a platform independent way of performing dynamic taint analysis on JavaScript applications [51]. Their main motivation was to deploy the infrastructure on the Samsung Tizen smartwatch platform, at a higher performance than the work done by [37]. In order to do so, they use the hooks that are provided by the Jalangi2 framework to intercept operations and transform them into instructions on their abstract stack machine. This abstract stack machine performs the taint tracking and propagation required for the dynamic taint analysis. They applied this approach and validated vulnerabilities in 17 NPM packages. They also ensured that their approach did not flag benign packages by validating them on 5 popular safe packages. However, their approach to tainting only allowed for tainting of entire values, unlike the byte/character level tainting that was supported by previous approaches. Furthermore, their abstract machine tainting identifies variables by name and scope, which is error-prone, especially since scoping in JavaScript is complicated. We argue that such a complicated mechanism is unnecessary, and we showcase our approach which identifies values by their references, thereby simplifying taint tracking and allowing for byte level taint tracking for strings.

A dynamic taint analysis approach was later used to perform analysis on the top 1000 NPM packages [72]. In particular, Staicu et al. utilized NodeProf [73] (which uses Jalangi2 underneath the hood) to perform instrumentation and dynamic taint analysis on the test cases of the top 1000 NPM packages in order to obtain taint specifications for these libraries. This was later passed into a commercial static analysis engine to verify the vulnerabilities. However, as the focus of their paper was finding taint summaries on libraries and applying the taint summaries on a static analysis engine to find bugs, there was less of a focus on the correctness of their taint tracking and propagation. Furthermore, their approach does not support byte level tainting.

Finally, recent work in this space by Kreindl et al. looked at implementing dynamic taint analysis that worked across multiple languages (including JavaScript running `NODE.JS`) [53]. The paper proposed an implementation leveraging on the universal virtual machine, GraalVM [80], which allows for running of applications written in a wide range of languages, including JavaScript, Python and C++. They utilized the Truffle language [41] to build instrumentation required to perform the taint tracking and propagation, but such an approach is still largely untested in practice.

To summarize, we see a general trend towards platform independent approaches in dynamic taint analysis as JavaScript starts to see more use beyond the browser. JavaScript has made its way into server-side applications and more recently, in desktop applications. Therefore, we aim to build a dynamic taint analysis framework that can be deployed across different contexts with minimal modifications, yet at the same time be flexible enough to change the underlying instrumentation framework so that newer versions of JavaScript can be supported.

We also note that different approaches were applied to handle dynamic code generation. In the next section, we summarize how other work has tackled this problem in their implementation.

3.1.3 Handling of Dynamically Generated Code

Regardless of which implementation one chooses, special handling is still required for dynamic code generation. and we will have to handle it as well in our tool in this thesis. In particular, we wish to be able to propagate taint within a call to `eval`, even if the input to eval is already tainted. Below, we summarize the approaches taken by prior work.

1. **Eval as a sink.** `eval` is defined as a sink in the framework so once execution

reaches `eval` with a tainted argument, the framework raises an alert. This is the approach taken by [72], [54] and [51]. However, such an approach limits the applicability of the framework. In particular, client side scripts on the browser heavily utilize `eval` in the wild, so if the researcher was looking for a different sink in this context (for example, `document.write`), they will not be able to analyze it using such a framework.

2. **Instrumenting dynamic code.** If the framework supports it, dynamic code is instrumented and taint is propagated with some special conditions. This is the ideal case, since we can continue to analyze the code with the same granularity. LeakTracker propagates the caller tag [77] to the generated code so that the variables in the dynamically generated code can be properly tainted. FLAX also handles `eval` explicitly in their trace [67], as does DexterJS in their implementation [62].

3. **Static analysis.** One last approach combines runtime information with static analysis on the dynamically generated code. At runtime, we do know what the code generated will be, and so if we analyze the code and look at what values should be tainted, we can mark them as tainted after the call to `eval`. This is the approach taken by Chen et al. [36], and they taint all left-hand side (LHS) of assignments in the dynamically generated code. This is as good as we can do without instrumenting dynamically generated code. However, because their analysis is done in the V8 engine, we cannot apply their technique directly in a platform agnostic implementation written in JavaScript, since we don't have access to all objects or values that have been defined.

In this thesis, we will explore a new method which combines static analysis with program rewriting to achieve the same effect as the approach in [36], even without access to the underlying JavaScript engine. We do this by providing a shared context between our instrumentation framework and the instrumented code.

Having looked at the dynamic taint analysis and how they handle dynamic code, we will now look at prior work done on the context of our evaluation, the NPM ecosystem.

3.2 Analysis of NPM Packages

The NPM ecosystem is rather new, having only existed for 10 years [15]. Yet, it is one of the biggest ecosystems of software [83]. In the past 5 years, a lot of effort has been invested to better understand this ecosystem and the security of the packages in this ecosystem. We summarize the key results below, and we see how we can add value to research in this space.

3.2.1 Ecosystem Analysis of NPM Packages

Early work done in understanding the NPM ecosystem noted the high dependency between packages within the ecosystem, with 32.5% of the packages being dependent on 6 or more packages [79], and they noted an increasing trend of interdependence between packages. It was later found that many of the popular micropackages that developers depend on were mistakenly thought to be *"well implemented and tested"*, where in reality only 42.5% of these popular packages contained any form of testing in their packages [33]. Such heavy interdependence can cause ecosystem breakdowns. For example, when the `left-pad` package was removed by its author, thousands of packages that relied on it (even large, popular ones like Node and Babel) could no longer function. The disruption resulted in a total downtime of 2.5 hours, and NPM had to unpublish the package against the will of the author to fix the issue [78].

Recent analysis done by Zimmermann et al. in 2019 concluded that packages in NPM were still highly interdependent and malicious code or vulnerabilities found in a single package can have huge ripple effects in the whole ecosystem, much like before [83].

To combat this threat, researchers have been looking at different ways of weeding out vulnerabilities automatically from packages in this space. Below, we look at large scale studies of code injection attacks in NPM modules and note their findings.

3.2.2 Security Analysis of NPM Packages

The first large scale study of code injection attacks in NPM was by Staicu et al., where they performed an empirical study on 235,850 modules to see how widespread code injection vulnerabilities could be [71]. They were interested in calls to the `child_process` module and `eval` function, as these were the most prevalent code injection attack vectors. In their study, they performed a regular expression-based search that detects the usage of the above APIs, and found over 16000 modules that use `exec` or `eval` directly, and over 15% of the packages in NPM were found to be dependent of a package that calls `exec` or `eval`. However, such calls do not necessarily imply code injection vulnerabilities, since the calls may never take arguments that are user-provided, so a definitive method of verifying vulnerabilities was still required.

A larger scale analysis improves on this result, and was conducted by Gong to identify bad behavior in NPM packages [44]. In his study, he presented NodeSec, a framework that analyzes the contents of the entire package, including the install scripts of the package. The framework hooks system operations such as network activity, file system activity, and shell commands at the operating system level, and verifies that running the package and its methods do not result in any undesirable behavior. Using this technique, he analyzed over 330,000 NPM packages and found more than 300 previously unknown vulnerabilities, which were manually verified and reported to the authors of the package. However, we believe that this method can only flag potential bad behavior since all shell commands are flagged in their study and manual analysis is still required to see whether a code injection attack is possible in that context.

Finally, in recent literature, Staicu et al. presented a way of analyzing libraries at scale by extracting taint specifications via dynamic taint analysis on test cases and applying static analysis on the libraries using the extracted taint specifications [72]. Through their efforts, they analyzed the top 1000 NPM modules and their dependencies for a total of 1393 NPM modules and they created 136 new alerts which likely corresponded to actual security vulnerabilities.

All in all, we see that there is an ongoing effort in finding code injection attacks in the NPM ecosystem, though a fully principled approach has yet to be applied on a large scale for code injection vulnerabilities in this ecosystem. In the next section, we will put the pieces together and present a dynamic taint analysis framework that shows promise in discovering code injection vulnerabilities at scale.

4

NodeTaintProxy

We now present our dynamic taint analysis tool, NodeTaintProxy. We will first look at an overview of our implementation, followed by a study of the overall architecture of our implementation. We elaborate on the details of each component of our architecture and see how they fit into the whole picture in the sections below. We then look at our novel approach in performing taint analysis for dynamically generated code, particularly when the argument to `eval` or `new Function` is tainted. We finally discuss some features of our implementation, such as in our handling of native functions and byte-level tainting.

4.1 Architecture

NodeTaintProxy is a dynamic taint analysis framework written in TypeScript that uses Jalangi2 as an instrumentation framework. To start, we give an overview of the inner workings of our dynamic taint analysis framework. We define how each object/value is uniquely identified, how each unique identifier is tagged with a taint bit, and we give a description of how the analysis works.

4.1.1 Overview

At a high level, the framework first performs rewriting of the source code so that each operation is instrumented. At the beginning and end of every operation, our instrumentation updates the taint information of each value in the system via interaction with the internal data structures. This goes on until we reach a sink function like `eval`, and we determine whether a tainted argument was provided to the sink function. If it is, we alert the analyst. The specific mechanisms for each component is described below.

4.1.2 Uniquely Identifying Primitives via Wrapping

In our framework, we identify unique values by their references/pointers. A wrapper map, M_W, defines a mapping of object references to their unique IDs. Recall in Chapter 2 that primitives are not identified by references. We solve this problem by wrapping the primitive using the *Proxy* object class, and store the wrapped value in the wrapper map. The wrapper map, M_W, now defines a mapping of object references in to their unique IDs and unwrapped value. Because an instance of a Proxy is an object, each primitive in the code can be uniquely identified. This is important because primitives which have the same value should not map to the same identity. For example, the value 1 defined statically in the code should not have the same identity as the value 1 provided as a tainted input to the function.

Each time we unwrap, we push the ID for the wrapped primitive to the ID stack for that particular operation. Note that for objects, we push a dummy ID. The same goes for when we wrap - we consume an ID on the ID stack. This means that when we unwrap in one order and rewrap in the reverse order, all wrapped primitives will preserve their IDs.

The IDs are used to uniquely identify values at runtime. With this unique identity, we can use them to associate taint information with the values in run time in our framework. We will see in later sections how this identity is used for taint tracking in a later section.

With unique identities for all values in the system, we will now proceed to show how to initialize our analysis.

4.1.3 Initializing the Analysis

The analysis is composed of the package we wish to test (together with its dependencies) and an input we wish to probe the package's exported function with. The input can be thought of as a possible user-provided input that can be passed to the package if the package was deployed in production. Since this input is attacker controlled, we taint the input.

We first have to instrument the underlying package that we wish to test, together with an analysis file which contains a call to the exported function of the package that we wish to test. The analyst would taint the attacker controlled input and specify the sinks that they wish to keep track of. For our purposes, we consider calls to the `eval` function or the `exec` function of the `child_module` process as sinks, unless otherwise stated.

The instrumentation calls the hooks defined in our architecture, and allows us to transform the original JavaScript operations into our own custom defined operations. We elaborate more on how we transform each operation below.

4.1.4 Operation Semantics of Our Tool

The instrumentation intercepts all JavaScript operations to perform the transformed semantics of our tool, which modifies the original semantics to account for wrapped values in the runtime. Figure 4.1 shows the interaction between each layer for each operation that is instrumented. Referring to Figure 4.1, we perform the following steps in our transformed semantics:

1. The rewritten code **calls the relevant Jalangi2 hook** in our instrumentation layer.

2. We signal to the wrapper layer to **unwrap all arguments** involved. If the operation was a method call, we also unwrap the object that the method was called on.

3. We **perform the actual operation on the unwrapped arguments by deferring to the JavaScript run time**. We store the unwrapped result.

4. We **rewrap all values in an order that is reversed from the order in step 2**. This maintains the IDs of primitives. We also **wrap the result of the operation**.

5. We signal to the taint layer to **propagate taint**.

6. The taint layer **looks up the relevant taint semantics** that apply for the operation and taints the respective arguments and/or result, following the taint semantics.

Through this, we are able to defer all operations to the JavaScript runtime, thus reducing burden of mangling with the semantics. With this setup, we now claim the following:

Claim. *The semantics of this transformed operation is equivalent to the semantics of the original operation.*

Justification. Since the instrumentation is semantically equivalent (by [69]), we simply need to show that the result of the operation in the transformed program is the same as the original program. For any given operation in JavaScript, we only defer to the runtime once all arguments are unwrapped. Note that the unwrapped primitives have the same value as those in the original operation, and objects are not wrapped in the first place. When deferred to the runtime, the operation acts on the unwrapped arguments which are equivalent to the original arguments, hence the result follows that of the original operation, as needed. ■

The above transformed semantics allows us to achieve Proxy Transparency mentioned in Chapter 2, which we will see below.

Dealing with Proxy Opaqueness

One of the issues with Proxy as wrappers is that they are opaque - this means that comparison between two Proxy objects that wrap the same primitive will give incorrect results. This was noted as a problem in Proxy transparency in [52]. However, following the semantics of each operation above, we see that we unwrap each object before performing the original operation. Thus, the instrumented wrapped primitive comparison would behave as below:

`Proxy(1)==Proxy(1)` $\hookrightarrow^T$ `Unwrap(Proxy(1))==Unwrap(Proxy(1))` $\hookrightarrow^E$ 1==1

Above, we notate transformation of the instrumented code with $\hookrightarrow^T$ and evaluation in the JavaScript runtime as $\hookrightarrow^E$. The last statement is evaluated and we get the result `true`, as we would expect. Thus, as we perform the operation on the unwrapped primitives, this transformation gives us the correct result. Hence, we achieve transparency of the Proxy object with respect to the `==` operator.

Now that we understand how wrapping works in our infrastructure and how we maintain the semantics of the original operation, we move on to see how they are used to uniquely identify each value to their taint entries.

4.1.5 Taint Tracking and Propagation

Recall that each value is uniquely identified by an ID in our wrapper map M_W. Our taint layer defines a map M_T that stores a mapping of IDs (which are the IDs from the wrapper map M_W) to their taint entries. As an optimization, if an ID does not exist as a key in the map M_T, then the value associated with the ID is guaranteed not to be tainted.

Each taint entry is a tuple (t, M_P) where t defines a taint bit, which denotes whether the value associated with the ID is tainted. The map M_P defines a property map which maps properties to taint bits. We can think of this as an auxiliary data structure that we use to keep track of taint for characters in strings. We will use this as a mechanism to enable byte level precision for our precise string tainting.

For each operation we intercept in JavaScript, we define rules to guide us on how to taint the result of the operation. These rules (or *taint semantics*) together form a *taint propagation policy.* We describe some examples of the taint semantics we have used for the operations below.

1. **Unary operations.** If the operand is tainted, then the result is tainted.

2. **Binary operations.** If either operand of the binary operator is tainted, then the result is tainted.

3. **Field access.** If the object is tainted, or the property of the object is tainted, then the result of the field access is tainted.

As taint propagation happens after each operation is completed, taint is propagated correctly in our system with respect to direct flows. In vulnerable code, this taint will eventually propagate to a vulnerable function like `eval` or `exec`, and in the section below, we will discuss what happens in our infrastructure when this happens.

Reaching a Sink

When a tainted argument is passed into a function that allows for code injection (like `eval` or `exec`), we have reached a sink. Here, a *sink policy* defines the granularity of tainting that is required for an argument to be considered tainted when a value that can be influenced by the attacker reaches the sink. For example, we currently employ the sink policy `anyPropertiesTainted`, which states that if any property of the argument is tainted, then the argument to the vulnerable function is tainted and we have reached the sink with a tainted argument, and we flag to the analyst that the package could be vulnerable. This matches the notion of the attacker having the ability to control at least a single byte in the code injection string, which is often sufficient to do damage to the system. Stronger guarantees can also be provided, for example, the analyst can enforce that all properties of the argument must be tainted through a policy like `allPropertiesTainted` (meaning the attacker has full control of the argument), or some number of the properties must be tainted (meaning the attacker has at least partial control) before we consider the argument as tainted.

When we reach a sink with tainted arguments, the analysis terminates and an alert is flagged to the analyst. Having looked at how an analysis works, we now describe the overall architecture and understand the interaction between each of the components of our framework.

4.1.6 Architecture by Layers

Our framework consists of the following 4 layers - we designed them such that each layer serves a single purpose, and we can plug in and play experimental layers that serve the same function to see if we can achieve the different effects. Below describes the functionality of each layer:

1. **Instrumentation Layer.** We instrument operations in JavaScript so that we are able to perform our transformed semantics before and after each operation. In this layer, we simply call hooks to signal to when an operation is beginning, or completed. We currently use Jalangi2 [31] as our instrumentation layer, which rewrites the program and its dependencies so that our custom defined

hooks are called. Note that a different instrumentation framework can be used as long as the same operations are hooked before and after.

The operations we hook are as follows:

(a) **literal** - the creation of a literal value, e.g. a static string defined in the program

(b) **write** - the write to a variable or property

(c) **getField** - accessing the field/property of an object

(d) **putField** - setting the field/property of an object

(e) **invokeFun** - invocation of a function, not including `eval`/`new Function`

(f) **binary** - binary operations as defined in ECMA-262, e.g. `+`, `-`

(g) **unary** - unary operations as defined in ECMA-262, e.g. `typeof`, `!`

(h) **instrumentCode** - invocation of `eval` or `new Function`

(i) **conditional** - branching operations such as `if-then-else`, `||`

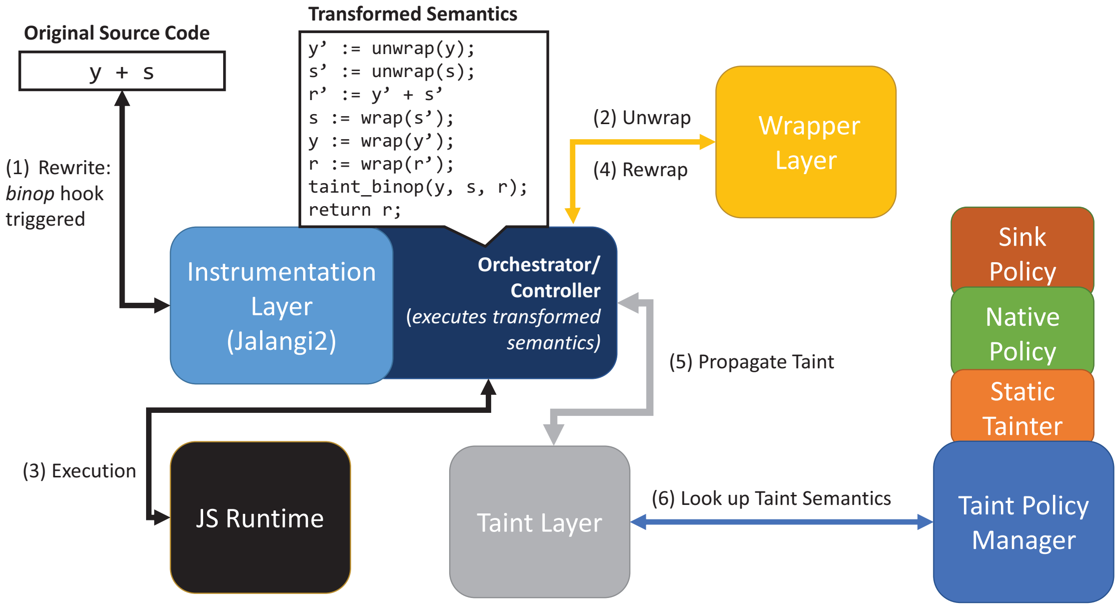
2. **Wrapper Layer.** The Wrapper layer, M_W, is defined as a mapping of object references to IDs and unwrapped values. This layer also defines the wrapping construct used, together with the wrapping and unwrapping sequences for each operation instrumented. We currently use the Proxy object class due to its in-built support for reflection, but we have also experimented with using the Object class as boxes (i.e. encapsulating a primitive in an Object like in [62]) simply by changing the wrapper construct within this layer.

3. **Taint Layer.** The taint layer is responsible for keeping track of the taint associated with each value in run time. We define it as a map M_T that stores a mapping of IDs to taint entries. Each taint entry is defined as a tuple (t, M_P) where t defines a taint bit, and M_P defines an auxiliary data structure that

maps properties to taint bits (we use this to keep track of precise string tainting). An analyst can swap out the taint layer with their own data structure used to store taint and other metadata. For example, if they want to keep track of sanitization functions used, they can incorporate that information at this layer by modifying the taint entry to capture that information, and incorporate sanitization logic into the taint semantics of the taint policy manager.

4. **Taint Policy Manager.** The taint policy manager implements the taint propagation semantics in the taint propagation policy, as well as the sink policy. An analyst that wishes to modify the taint propagation semantics for one of the operations can simply swap out the existing semantics and incorporate their own semantics at this layer. Other taint semantics can also be defined, for example for native functions or for dynamically generated code. We define a ***native policy*** module to describe semantics for native functions that we model, as well as ***static tainter*** module to taint dynamically generated code if the code generated is tainted. For sink policies, we default to using `anyPropertiesTainted` as our sink policy, but an analyst can swap this out with `allPropertiesTainted` or any other custom sink policy as they deem fit.

Finally, a controller is built on top of the instrumentation layer and presides over all interactions between the layers and the JavaScript run time. Its responsibility is to orchestrate each layer to update the data structures of each layer as needed. The architecture is summarized in Figure 4.1.



```
y' := unwrap(y);
s' := unwrap(s);
r' := y' + s'
s := wrap(s');
y := wrap(y');
r := wrap(r');
taint_binop(y, s, r);
return r;
```

Figure 4.1: Segregated Architecture of NodeTaintProxy

4.2 User Interface - Interaction via Ghost Functions

Below, we describe in steps how an analyst can write their own analysis in our framework for a package that they wish to study. We describe the usage of our framework on the NPM module `ps`, which contains a code injection vulnerability in versions $< 1.0.0$ [1].

```javascript
var cp = require('child_process');
__jalangi_set_sink__(cp.exec);
var ps = require('ps');
var evil_string = "abc"
__jalangi_set_taint__(evil_string);

ps.lookup({ pid: evil_string }, function(err, proc) {
    // this method is vulnerable to command injection
});
```

Listing 4.1: PS Module Analyzer

The analyst interacts with the framework via *ghost functions* (prefixed with `__jalangi<...>__`) that are globally defined in the framework. This allows the analyst to set up sources, sinks, and verify parts of their analysis via communication with the framework.

4.2.1 Setting up Sinks

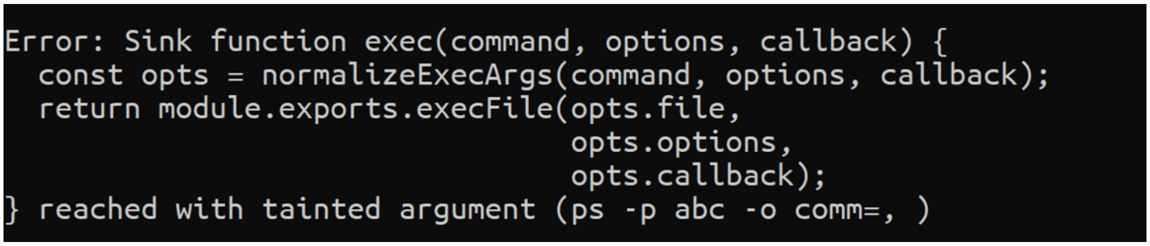
The analyst first has to define sinks that they wish to keep track of in the program. Currently, by default, `eval` is conisdered a sink since it is a code injection vector, but we show that it is trivial to add your own. Using the `__jalangi_set_sink__` function in line 2 of Listing 4.1, the analyst can specify the function they wish to keep track of as the sink by passing it as an argument to the ghost function. Because the underlying function references do not change throughout the execution of the program, we can simply compare the function reference before every function call to see if the sink has been hit with tainted arguments.

4.2.2 Setting up Sources

The analyst will then define sources, which are user controllable input to the package under investigation. In this case, the `lookup` function in the module `ps` takes in a string representing the *pid* to look up in the operation. We define a valid argument that can be passed into the library in line 4 and set the parameter to be tainted via the ghost function `__jalangi_set_taint__` as per line 5 in Listing 4.1. The source is now tainted and ready to be passed to the function under investigation.

4.2.3 Running the Analysis

In the analysis file, we import the package as required (as per line 3 in Listing 4.1) and call the function we wish to test with our tainted arguments (as per line 7 in Listing 4.1) . We provide a Makefile with the `casestudy` command that compiles our infrastructure written in TypeScript and runs the case study specified via the `FILE` parameter, and we see the results of our analysis below in Figure 4.2.



```
Error: Sink function exec(command, options, callback) {
  const opts = normalizeExecArgs(command, options, callback);
  return module.exports.execFile(opts.file,
                                 opts.options,
                                 opts.callback);
} reached with tainted argument (ps -p abc -o comm=, )
```

Figure 4.2: Output of Running the Analysis for the module *ps*

We see that the sink `exec` was reached in the program, and the tainted argument *'abc'* defined in line 4 in Listing 4.1 propagated to the sink. By our sink policy of `anyPropertiesTainted`, the argument was marked as a tainted input to our sink and an alert was flagged to the analyst. We have verified that a command injection vulnerability is present in our case study, as per the advisory in [1].We have now seen how our framework performs on a known vulnerability in practice.

However, there are special cases we wish to handle. For dynamically generated code, `eval` is typically a sink and hence if a tainted argument was provided, an alert would be raised and execution would stop. However, if the analyst does not want `eval` to be a sink, we would still have to propagate taint during the execution of the dynamically generated code. In the next section, we will a explore novel method that we implemented in our framework to propagate taint in the case where `eval` or other dynamic code generation functions are not sinks (for example, when *safe* versions of them are used [24]), and see how we handle the case when the arguments to `eval` are tainted.

4.3 Handling Tainted Dynamic Code Generation

As we have seen in Chapter 2, `eval` is a common construct in JavaScript code and sometimes use of it may be justifiable [82]. In such cases, `eval` should not be marked as a sink. To motivate our discussion, we show a simple example where a tainted argument is provided to `eval` in line 3 of Listing 4.2.

```
1 var x = "var h = 'Hello';";
2 __jalangi_set_taint__(x);
3 eval(x);
```

Listing 4.2: Original Module Code

As we can see, after the execution of the `eval` statement in line 3 of Listing 4.2, a new variable h is defined in the current scope of the program, and because the dynamic code is tainted, the variable h should be tainted. However, our taint propagation semantics currently do not describe how to taint in such an instance.

The goal is to propose a way to taint variables that were introduced or modified in the current scope by tainted code. Unfortunately, this problem turns out to be complicated as we are working through instrumentation, and we list the complications below.

4.3.1 Challenges

Ideally, in such a case, we rely on the instrumentation engine to tell us when we enter dynamically generated code, and we check to see if it is tainted. From there, we use an approach similar to the tainted scope approach in [58] to taint new variables introduced to the scope. However, prior experiments done in our framework showed that the dynamically generated code do not trigger the hooks defined in our infrastructure, so we cannot rely on the instrumentation layer for this.

We also realized that if we could get a hold of the context of the evaluation of the dynamically generated code, we could simply pass it to a `vm` construct in `NODE.JS`, which allows us to execute code in a predefined context [23]. From there, we can simply compare to see what new variables were introduced in the current scope and taint them as needed. However, the complexity of the instrumentation meant that it was difficult for us to get the scope of the currently executed operation. In particular, we are only given access to the string representing the code that will be executed in the instrumentation layer.

An alternative approach would have to be developed in order to taint the newly introduced variables and modified variables in scope. For this, we take inspiration from the static analysis approach of Mystique [36], which statically analyzes dynamically generated code and taints those variables that are introduced or written to, and we apply them in our context. However, without direct access to the JavaScript runtime internals, we implement unique workarounds via program rewriting, as we will see below.

4.3.2 Overview of Approach

To solve the issue of partial information in the instrumentation layer, we rewrite the code at different levels so that there is a known shared context between the dynamically generated code, the original program and the instrumentation engine. We leverage all the information provided to us at the instrumentation layer to achieve this effect.

Our approach takes the following steps:

1. We obtain the dynamic code generated at runtime using the hooks provided by our instrumentation layer; in particular, the `instrumentCodePre` hook gives us access to the string that will be evaluated.

2. With access to the string to be evaluated from above, we run our static analysis algorithm to form a list of variables to be tainted in the current scope and global scope. We describe the algorithm in full in the section below.

3. We rewrite the generated code so that it stores all variables that are to be tainted in a shared context map that maps the variable name to the variable value. We then execute this rewritten generated code.

4. In the original module, the code has been rewritten to obtain the variables from the shared context, so that each variable can be wrapped and tainted accordingly in our infrastructure. We then propagate the wrapped values back into the original scope of the program.

Using this approach, we ensure that the variables that exist in the runtime are wrapped and tainted as we expect them to be. The key to making this work lies in the shared context, which we elaborate on in the section below.

Shared Context

The shared context forms an important cornerstone in our approach. The shared context is defined as a map M_E that maps variable names (or more generally the stringified *left hand side*(LHS)) of assignments to their actual values.

In code, the shared context goes by the handle `__eval_taint_map__`. We see that the shared context is used in both our rewritten dynamic code in lines 4 and 6 of Listing 4.6 as well as our rewritten module code in lines 3 - 9 of Listing 4.4.

The shared context is populated by the rewritten dynamic code, and read by the rewritten module code, as we will see in the next few sections.

Module Rewriting Algorithm

In the module rewriting algorithm, we seek to read variables to be tainted from our shared context, invoke our infrastructure to taint them, and store the values back in the original context. We describe the algorithm that rewrites the modules below, in Algorithm 1.

```
Algorithm 1: Module Rewriting Algorithm
  Data: $M$: list of node.js modules under analysis,
        $tmpl$: template of transformed eval call,
        $placeholder$: placeholder value in template
  Result: $M'$: node.js modules rewritten
  initialize $M'$ to an empty list;
  foreach $m$ in $M$ do
      $r$ := read($m$);
      $ast$ := parseAST($r$);
      foreach $node$ in $ast$ do
          if $node$ is eval function call then
              $tmpl\_ast$ := parseAST($tmpl$);
              $node'$ :=$tmpl\_ast$.replace($placeholder$, $node$);
              $ast$ := replaceInAst($ast$, $node$, $node'$);
          end
      end
      $M'$.append($ast$.toJSFile());
  end
```

The template of the `eval` call passed in Algorithm 1 is as shown in Listing 4.3.

```
var __eval_taint_map__ = {};
try {
    ${insert eval node here}
} finally {
    for (var _i_ in Object.keys(__eval_taint_map__)) {
    var _k_ = Object.keys(__eval_taint_map__)[_i_];
    var __j__ = __jalangi_set_taint_eval__(
   __eval_taint_map__[_k_]);
    eval(_k_ + '=' + '__j__;');
}
```

Listing 4.3: Template for Module Rewriting

Through this algorithm, we see that the code in Listing 4.2 is rewritten to the form in Listing 4.4.

```
1  var x = "var h = 'Hello';";
2  __jalangi_set_taint__(x);
3  var __eval_taint_map__ = {};
4  try {
5      eval(x);
6  } finally {
7      for (var _i_ in Object.keys(__eval_taint_map__)) {
8      var _k_ = Object.keys(__eval_taint_map__)[_i_];
9      var __j__ = __jalangi_set_taint_eval__(
     __eval_taint_map__[_k_]);
10     eval(_k_ + '=' + '__j__;');
11 }
```

Listing 4.4: Rewritten Module Code

In Listing 4.4, we first set up the shared context in the local scope, as per line 3. We then execute the dynamically generated code in line 5. Note that the executed code will also be rewritten when we get hold of it in the instrumentation layer, and we will explain that in a later section. We wrap it in a `try-finally` block so that errors in the `eval` call will not affect our tainting, but will still be propagated to the original scope. In the `finally` block, we enumerate through each variable that was written to in line 7. We obtain the stringified LHS in line 8 in the variable `_k_`, and wrap and taint the value in our call to the ghost function `__jalangi_set_taint_eval__` in line 9 and store the result in `__j__`. In line 10, we assign the LHS to the wrapped value through an `eval` call, and we have that the original scope now has the wrapped and tainted variables that were written to or introduced.

What remains is how we pick variables to be populated in the shared context, and how the variables are populated in the shared context. In the next section, we will explain our static analysis algorithm that dictates which variables to populate in the shared context.

Static Analysis Algorithm

Recall that we are trying to taint variables that are introduced to the current scope and global scope by the dynamically generated code in this section of our analysis. Here, we will statically analyze the dynamically generated code string (provided to us by the instrumentation) to see which variables in scope are modified in the end. In Algorithm 2 defines how we do the static analysis. Note that we use `esprima` to parse the source code to an AST [6] and `escope` to verify that the variable is indeed assigned in the scope of the `eval` context [29].

Algorithm 2: Static Analysis Algorithm

```
Data: $e$: dynamically generated code string,
      $L$: currently known variables to taint
Result: $L$: list of all stringified LHS to taint
initialize $L$ to an empty list if not defined;
$ast$ := parseAST($e$);
foreach node in ast do
    if node is AssignmentExpression and in scope then
        $L$.append($node$.left.toString());
    end
    if node is CallExpression and callee is known then
        ALG($callee$.body.toString(), $L$);
    end
end
```

We note a few special cases in the below segment during our traversal of the AST:

1. **Function calls.** We treat functions as blackboxes if we do not have access to the function internals. Otherwise, we traverse the function's AST for each function call and mark variables that were written to in the global scope as variables to be tainted.

2. **Conditional branches.** In our static analysis, we do not have access to runtime information and hence are not able to determine which branch would be taken. In this case, we mark all variables that were written to in all branches

of execution.

3. **Exceptional control flow.** Much like the conditional branches, we do not make assumptions on the control flow that was taken, and we mark all variables that were written to in the `try` block, `catch` block, and the `finally` block.

At the end of the algorithm, we have a list of stringified LHS that we need to taint. With this information, we can finally move to the last part of our rewriting - the rewriting of the generated code to populate the shared context with this list of variables.

Dynamic Code Rewriting Algorithm

The generated code must populate the known variables to be tainted to our shared context so that our rewritten module code can extract it. We do this by rewriting the generated code before returning it so the rewritten code is evaluated instead. Below, we detail the algorithm for rewriting in Algorithm 3.

```
Algorithm 3: Dynamic Code Rewriting Algorithm
  Data: $e$: dynamically generated code string, $L$: list of all variables to taint
  Result: $e'$: rewritten dynamically generated code string
  $e'$ := ‘var __orig__; try {__orig__ = eval(EVAL_STMT); VARIABLES}
   catch(err){VARIABLES; throw err;} __orig__;’ ;
  $VARIABLES$ := ””;
  foreach $lhs$ in $L$ do
      $VARIABLES$ += “__eval_taint_map__[‘$lhs$’] = $lhs$;”;
  end
  $e'$.bind(‘EVAL_STMT’, $e$);
  $e'$.bind(‘VARIABLES’, $VARIABLES$);
```

Listing 4.5 shows the contents of the original dynamically generated code, which we will use as an example.

```
var h = 'Hello';
```

Listing 4.5: Original Dynamic Code

Note that the variable h is written to, and through our static analysis algorithm in Algorithm 2, we created the list containing the variable to taint (the list corresponds to `[‘h’]` in our example). The transformed code for the generated code is presented in Listing 4.6 after being rewritten by Algorithm 3.

```
1 var __orig__;
2 try{
3     __orig__ = eval('var h = "Hello";');
4     __eval_taint_map__['h'] = h;
5 } catch (err) {
6     __eval_taint_map__['h'] = h;
7     throw err;
8 }
9 __orig__;
```

Listing 4.6: Rewritten Dynamic Code

In line 1, we store the declare the return value of the `eval` statement as `__orig__`. We then attempt to evaluate the original statement and store the value in `__orig__` in line 3. We also populate the written variables in our shared context in lines 4 and 6. The statement populates the map so the rewritten module knows which variables should tainted and what their values are. The evaluation is wrapped in a `try-catch` block in case of errors in the original evaluated statement. If the evaluation of the statement throws an error, we still populate the shared context with as much information as we know in line 6, and throw back the same error in line 7. Otherwise, if there are no errors, we would still populate the shared context in line 4, and return the original evaluated value in line 9, as per what is required in the semantics of `eval`.

Thus, at the end of the instrumentation, we have that all written variables are populated in the shared context, and the rewriting of the original module ensures that the newly introduced values are wrapped and tainted as we expect.

4.3.3 Limitations of Approach

We are aware of some limitations of our approach, and we list them below:

1. **Overtainting.** Since we do not have runtime information, we cannot fully ascertain the execution path that was taken by the evaluated code. For example, in the case where there is a conditional statement like an `if-else` construct, we would have to taint both branches. The ideal case here would be to only taint the branch that was taken, but a static analysis we will not be able to easily derive this information, and hence we would have overtainted.

2. **Nested evaluation.** If an `eval` statement is nested (i.e. there is a call to `eval` in the dynamic code), since there is no instrumentation in the inner call to `eval`, we will not be able to taint variables written in that `eval`'d code. While an adversary might exploit this specific loophole to foil taint propagation, in practice, we do not see such calls being made in standard code (in particular, none of our case studies display this behavior in a normal setting). We also outline ways of mitigating this in Chapter 6.

We have tackled one of the key challenges in implementing a platform agnostic approach, which is the problem of tainted dynamically generated code. In the next section, we will look at other problems that a platform agnostic approach would have, and how we discuss how we have approached them.

4.4 Discussion

In this section, we look at other interesting design decisions we have made in our framework, particularly so in our context of being platform agnostic.

4.4.1 JavaScript Semantics

To test that we adhere to the JavaScript semantics, we studied the ECMAScript specifications [43] and implemented tests that specifically dealt with type coercion, weak typing and other special cases in the semantics flagged by the ECMAScript specifications. Our implementation passed these test cases, showing that our approach is fundamentally correct. A sample of one of the test cases for equality is provided in Appendix A.

However, without full formal semantics or rigorous testing via the Test262 test suite [32], we do not claim full semantic equivalence of our dynamic taint analysis, and there may be cases where we do not respect the semantics of JavaScript. We leave this for future work as we continue to formalize the semantics of our framework.

4.4.2 Native Functions

As we recall from Chapter 2, native functions are especially noteworthy because they are not visible to our instrumentation framework. Their occurrence in JavaScript code is frequent, and many of the functions that developers use are native. For example, the `indexOf` function in the Array prototype is a native function that finds the index of an element in the array, and we see it as a common occurrence in JavaScript code. To taint native functions precisely, we employ similar methods to that of [51] and [37], by creating models for these native functions, either by reimplementing them in JavaScript or by summarizing their taint propagation logic in our taint semantics. For native functions that we do not have models for, we opt to taint imprecisely - that is, if any of the arguments are tainted, or if a method is called on a tainted object, then the result is tainted.

4.4.3 Precise String Tainting

In addition, an improvement we implemented that was not previously available in prior platform agnostic approaches for `NODE.JS` is the ability to taint strings precisely. As mentioned earlier, in our taint entry, the property map M_P maps character indices of the string to whether they are tainted (i.e. whether they have a taint bit). Through this, we can specify which exact bytes of the string are tainted, and use that information going forward. This is however not the focus of this thesis, and the work for the taint semantics that enable this feature is still an ongoing effort.

5

Results and Discussion

Having described our tool, NodeTaintProxy, we seek to answer the following research questions:

> **RQ1:** Can NodeTaintProxy detect code injection vulnerabilities in NPM packages for language versions we support?
>
> **RQ2:** How does NodeTaintProxy compare to other tools in terms of precision of tainting?
>
> **RQ3:** Can NodeTaintProxy be applied to detect novel code injection vulnerabilities in packages in the ecosystem?

5.1 Dataset

To answer the above research questions, we evaluate our framework by determining if it is capable of detecting code injection vulnerabilities in vulnerable packages identified by prior work [51][71]. We included true negative samples as per [51] to ascertain that our results were not superfluous, i.e. we ensure that our framework compares with other frameworks and does not flag true negatives as false positives. Note that some case studies from prior work were omitted - the `printer` case study is only supported on versions of `NODE.JS` that have long been deprecated, and `modulify` is a false positive according to the author of the package [22]. We have also omitted command-line utilities from our evaluation, since they have no exported interfaces in JavaScript that our tool can leverage on to do dynamic taint analysis. We also included four new case studies not seen in prior work, showcasing recent vulnerabilities discovered in the wild. The newly introduced vulnerabilities ranges in complexity, from simple ACI in the `ps` package to complex deserialization logic involving `eval` in `node-serialize` or `js-yaml`.

5.2 Results on Case Studies

In Table 5.1, we list the results for running our tool on the dataset of case studies described above. The first column shows the name of the package under inspection, the second column shows the version of the package that was analyzed in our framework (where the vulnerability was known to exist), the third column details the origin of the case study (those linked to an advisory are known vulnerabilities in NPM that we picked as additional case studies), and the last column denotes whether the vulnerability was detected in our framework, and details the reason if it was not.

We analyzed a total of 22 packages, with 4 original case studies plus 18 case studies from previously published work. We detected code injection vulnerabilities in 11 out of the 22 packages tested. With regards to the failures in detection, 2 packages timed out because of overwrapping, and 4 failed due to a lack of support for ES6 features. In addition, 5 packages failed to run in our analysis due to known bugs in our infrastructure that can be fixed, and we believe that these bugs do not detract from the conceptual ideas of our tool. One such bug which resulted in 3 out of 5 of the failures is a bug in how we handle assignments in `growl`, where assignments to the properties of the `exports` object would be mistakenly resolved to `undefined`, hence crashing the program.

5.3 Discussion

Below, we will elaborate in detail the results of our evaluation.

5.3.1 Successful Detection

For packages written in ES5.1 and below, we were largely successful in instrumenting and finding the vulnerabilities, even if the vulnerability was complex in nature. For example, `node-serialize` suffers from a code-injection vulnerability that happens because the library attempts to call `eval` on a function that was serialized. Thus, the taint propagation would have to work through the complex deserialization logic in order to reach the `eval` sink. It is unclear from reading the source code whether any generic string would reach the sink, but our analysis clearly shows that a serialized string would always be evaluated. We also note that out of the 4 new case studies introduced, we detected code injection vulnerabilities for 3 out of 4 of them, with the last case study failing because of overwrapping.

We also support a good amount of case studies used in prior work for the language versions we support. In particular, 14 case studies out of the 18 used in prior work are packages written in ES5.1 and below, and we support all but 6 case studies from this selection. We attribute 5 of the failures to known bugs that can be fixed, while the last failure was found to have failed due to overwrapping.

RQ1: We believe NodeTaintProxy sufficiently detects code injection vulnerabilities with respect to the language versions it supports at this time.

Finally, unlike previous platform agnostic approaches which were coarse-grained [37][51], our approach supports byte level tainting. For example, in our analysis of the `ps` case study, we found that the indices tainted in the argument to the call to `exec` were precisely the indices an attacker could control, whereas previous approaches could only mark the entire argument as tainted.

RQ2: NodeTaintProxy improves upon prior work by specifying an architecture where byte level tainting can be supported in a platform agnostic approach and we show that byte level tainting has been achieved.

5.3.2 Unsupported Operations

Unfortunately, support for ES6 and above is needed to run most modern NPM packages. Jalangi2's ES6 support is lacking, and a variable defined using `let` or `const` would not be able to run in our framework and would be `undefined`, even if they were the only ES6 features that were used.

The presence of modern JavaScript features for packages in NPM is growing, and even a single line containing newer JavaScript features would prevent us from analyzing the package. An example is the `os-uptime` package, which has a single line which uses the `const` declarator and hence, Jalangi2 was not able to run the package due to a missing declaration. Other packages that use more ES6 features would also fail to run, as we would expect. We will discuss methods to resolve this in future work in Chapter 6.

5.3.3 Overwrapping

As discussed in Chapter 4, we wrap every primitive that we encounter and store their reference in the wrapper map. This however means that the overhead for each operation grows larger with the size of the package. Large packages such as `js-yaml` with about 4000 lines of code failed to load, since each operation in the package initialization is also wrapped, and the overhead of looking up unwrapped values in the wrapper map for these wrapped objects grows linearly with the number of operations performed. `mongoosify`, a small package with around 200 lines of code, depends on `lodash` which has 12000 lines of code, will also fail to load on our framework as well.

Note that we have made the above decision to wrap all primitives to ensure that our semantics for wrapping work even in large and complicated code bases. In Chapter 6, we will discuss alternative strategies to lower the overhead of wrapping.

While we recognize the limitations of our infrastructure, we see promise. In particular, we have evaluated our tool's capability in detecting code injection vulnerabilities in packages written in ES5.1. In Section 5.4, we broaden the scope and outline a feasibility study to evaluate our tool against novel code injection vulnerabilities in the wild.

5.4 Feasibility Study

Having looked at how NodeTaintProxy performs on known vulnerabilities, we want to evaluate the tool's performance on NPM packages on a large scale. However, our current implementation does not support this at this point of time. We first look at the difficulties in applying our method directly on the ecosystem of packages. We then propose a study to determine the prevalence of novel code injection vulnerabilities in the wild, and ascertain if our framework is capable of detecting the vulnerabilities. We outline how we collected the data and found the vulnerabilities, and show our results from the analysis of over 18000 packages. We then conclude that NodeTaintProxy is fundamentally able to detect novel code injection vulnerabilities, with it detecting 2 out of the 3 vulnerabilities for the JavaScript versions that it currently supports.

5.4.1 Challenges for Large Scale Analysis of Packages

There are challenges inherent in the NPM ecosystem that prevents our current implementation from being applied on packages in the wild. Below, we list two of the biggest blockers that we have encountered in our analysis.

1. **Diversity of Packages.** NPM packages come in many different flavors, not all of which are suitable for our framework. To give an example, the `watch-cli-only` package only runs from the command line. Other packages require transpilation to ES5.1 before being compatible with our framework. Most notably, packages that use newer features of JavaScript (such as `os-uptime` which is written for ES6), CoffeeScript (like `refix` [75]) or even TypeScript (like `haversine-position` [74]) would have to be compiled into a compatible version of JavaScript before our tooling would work. Another issue is that some packages run only on the browser and hence do not work in our framework.

2. **Lack of Clear Interfaces.** As noted in prior work in studying NPM packages, there are no clear interfaces on how exactly to interact with the package that we wish to target [44]. Documentation may be lacking for some packages, and only a small fraction of packages contain any kind of testing [33] which we might hope to glean potential usage patterns from.

Without a proper harness to solve the above issues, we will not be able to conduct a study at a large scale. However, before doing that, we wish to first ascertain that code injection vulnerabilities are still prevalent in this ecosystem. While previous studies noted the prolific use of APIs that cause code injection attacks [71], they did not verify that code injection attacks exist for a large portion of them. We proceeded with a similar study to see if such code injection vulnerabilities are prevalent. We verify the vulnerabilities manually and for truly vulnerable packages, and we check to see whether the vulnerability could potentially be detected by our framework.

5.4.2 Methodology of Feasibility Study

To achieve this, we employed a similar methodology to [71]. We first retrieved a dataset of over 844743 packages from an updated list of git repositories known to be associated to NPM packages [30]. From there, we pulled a subset of the dataset. In particular, we cloned the first 18031 packages, which is about 2% of the dataset. A number of packages could not be retrieved this way as their git repository link was private, down, or non-existent. We speculate that packages without a git repository tied to it would be as prone, if not more prone to such vulnerabilities.

The reason why we chose to pull repositories from git was because of the potential dangers of installing unknown NPM packages, as noted in [44]. An improvement we could have made to the analysis was to order the packages by use and pulling them starting from the packages with the highest weekly downloads.

From there, we performed a regular expression based search to find uses of the exported functions in the `child_process` module. This search provided decent fidelity for the output results, since the `child_process` module is often misused and abused, as we have seen in [71]. However, there were a large number of false positives that occurred because of the usage of the `child_process` module to run test cases, and in production this module would not be imported at runtime. We manually triaged each alert to see if a code injection vulnerability could possibly exist, and for those that we found were vulnerable, we developed proof of concepts for each of the vulnerable applications. Note that this triaging process is extremely time consuming and error-prone, and we do not believe that we have captured all code injection vulnerabilities in the dataset that we have pulled. Through the process of this triaging, we also see the potential value of being able to automatically check whether code injection exists in a package using our tool.

5.4.3 Analysis of Feasibility Study Vulnerabilities

Table 5.2 below summarizes the code injection vulnerabilities found and the details of each vulnerability. The first column shows the name of the package under inspection, the second column shows the version in which we found the vulnerability (which is the latest version at the time the thesis was written), the third column details the vulnerability, and the last column denotes whether the vulnerability was detected in our framework, and details the reason if it was not.

In total, we found 7 previously unpublished code injection vulnerabilities. For the versions of JavaScript we support, we detect 2 out of 3 of the packages containing the vulnerability. For the last package, `onion-oled-js`, an under-specified sink policy did not support the structure of calls to `apply` and `call` that was used to bootstrap promises, hence it could not detect that a *promisified* function reached the sink. We are however aware of this issue and we believe this is an isolated bug in the sink policy and does not detract from our overall approach. We believe that other vulnerabilities not detected by our infrastructure can be detected with some modifications to support other versions of JavaScript, as noted in the challenges earlier. This sends a strong signal to us to continue our work - we see that manual triaging of alerts of decent fidelity take time and the high fidelity alerts provided by our framework would cut down (if not eliminate completely) the manual labor needed to process the vulnerabilities.

RQ3: NodeTaintProxy was successful in detecting novel code injection vulnerabilities found in NPM for versions of JavaScript that we currently support, showing promise on its ability to perform in a large scale study.

5.4.4 Insights from Feasibility Study

In our analysis of the vulnerabilities, we noticed that the fixes for these packages are often simple, since the command injection in each case is quite trivial. In many cases, these applications simply run a command, passing in arguments provided to the binary they wish to run. However, because they call exec or execSync, command injection can be done by adding special characters that allow them to sequence or inject commands (like ` or ;) to the arguments passed to the function. This can easily be fixed by using a much safer function, like spawn or execFile of the child_process module, which only runs the particular binary file specified and hence are not amenable to command injection like the former vulnerable functions.

Unfortunately, despite the easy fix, unreported code injection vulnerabilities are still common in the NPM ecosystem, with potentially many more to be found. With the current state of our tool, we are able to detect vulnerabilities for packages that were written in ES5.1 and below. To that end, we believe that pursuing a large scale study on the ecosystem of NPM packages would be fruitful. We will elaborate more on the ways to conduct a large scale study in the next part of our thesis, where we discuss limitations and future work of our tool in Chapter 6.

5.5 Threats to Validity

Below, we list the threats to validity of our results.

1. **Limited real world package analysis.** While we can get results on a selected subset of real world packages as seen in prior work, this might not hold for the majority of packages in NPM. In particular, we have seen large packages in the ecosystem, as well as packages that use newer features of JavaScript that are currently not supported by our instrumentation framework. We have attempted to resolve this by introducing new case studies not looked at by prior work, plus found new vulnerabilities in our feasibility study. We showed that these new vulnerabilities can be detected in on our infrastructure. We believe that above limitations of our current approach can be circumvented, and we

outline a plan to conduct large scale studies on packages in the ecosystem - details of this can be found in Chapter 6.

2. **Overtainting.** It might be entirely possible that the positive results were caused by overtainting in the system, and hence we would always signal that the sink was reached. This would cause false positives to show up in our analysis as the framework might flag packages as vulnerable when they are not. For example, an imprecise policy might flag a value as attacker controlled when the attacker has no way in practice of controlling it. We put in place the following measures to reduce the likelihood of overtainting. First, for each case study, whenever possible, we ran the function using both tainted and untainted inputs and ensured that in the latter, we did not signal that a tainted input reached the sink. Second, if there was more than one operation required to run the exploit for a particular case study, we put in place tests to check for overtainting and ensured that only the parts we expect to be tainted are tainted after each operation in the case study. Third, we utilized benign packages such as `node-wos` as true negatives as part of our case study to ensure we did not flag every call to `exec` as malicious. Finally, we implemented and passed over 300 tests to ensure correctness of taint propagation, with each testing taint propagation for different operations in JavaScript.

Package	Version	Sink	Origin of Case Study	Detectable by NodeTaintProxy?
ps	0.0.2	`exec`	Advisory: [1]	**Yes.**
cryo	0.0.2	`eval`	Advisory: [11]	**Yes.**
js-yaml	3.13.0	`eval`	Advisory: [12]	*No.* Due to overwrapping.
node-serialize	0.0.4	`eval`	Advisory: [14]	**Yes.**
gm	1.20.0	`exec`	From [71].	*No.* Uses ES6 features.
fish	0.0.0	`exec`	From [71].	**Yes.**
git2json	0.0.1	`exec`	From [71].	**Yes.**
growl	1.9.2	`exec`	From [71].	*No.* Unsupported setting of property of export.
chook-growl-reporter	0.0.1	`exec`	From [71].	*No.* Depends on `growl`. Unsupported setting of property of export.
mqtt-growl	0.1.0	`exec`	From [71].	*No.* Depends on `growl`. Unsupported setting of property of export.
m-log	0.0.1	`eval`	From [71].	*No.* Dependency uses unsupported assignment operator.
mixin-pro	0.6.6	`eval`	From [71].	**Yes.**
mol-proto	0.0.15	`eval`	From [71].	*No.* Unsupported setting of property of export.
mongoosify	0.0.3	`eval`	From [71].	*No.* Due to overwrapping.
node-os-utils	1.0.7	`exec`	From [51].	*No.* Uses ES6 features.
node-wos	0.2.3	`execSync`	From [51]	**Yes** True negative.
osenv	0.1.5	`execSync`	From [51]	**Yes** True negative.
office-converter	1.0.2	`exec`	From [51].	**Yes.**
os-uptime	2.0.1	`exec`	From [51].	*No.* Uses ES6 features.
pidusage	1.1.4	`exec`	From [51].	**Yes.**
pomelo-monitor	0.3.7	`exec`	From [51].	**Yes.**
system-locale	0.1.0	`execFileSync`	From [51].	*No.* True negative. Uses ES6 features.

Table 5.1: NodeTaintProxy Evaluation Results

NPM Package	Version	Vulnerability Details	Detectable by NodeTaint-Proxy?
nbin [8]	0.0.4	Use of `exec` allows for command injection via `args` parameter of the exported `exec` function	**Yes**
wincred [25]	1.0.2	Use of `exec` allows for command injection via the exported `run` function	*No*, uses ES6 features.
picoTTS [18]	0.1.1	Use of `exec` allows for command injection via exported `say` function	**Yes**
remark [20]	0.1.0	Use of `exec` allows for command injection via `filepath` parameter of the exported `remove` function	*No*, uses ES6 features.
onion-oled-js [16]	0.0.2	Use of `exec` allows for command injection via `exec` in multiple exported functions	*No*, uses `apply` and `call` to implement Promises.
node-ts-ocr [10]	1.0.15	Use of `exec` allows for command injection via the `options` parameter in the `invokePdfToTiff` function	*No*, uses TypeScript.
ffmpegdotjs [7]	0.0.4	Use of `exec` allows for command injection in multiple exported functions	*No*, uses ES6 features.

Table 5.2: Code Injection Vulnerabilities Found in Feasibility Study

6

Limitations and Future Work

We break down the current limitations of our framework and see how we can address these issues in future work.

6.1 Overwrapping of Primitives

As mentioned in Chapter 5, overwrapping of primitives is an issue when running large packages. To illustrate the issue, we refer to the code snippet in Listing 6.1.

```
for (var i = 0; i < n; i++) {
    a = a + "a";
}
```

Listing 6.1: Simple For Loop For Illustration of Overwrapping

First, recall that primitives are wrapped so that they have a unique identifier in our taint map M_T. Suppose that none of the above variables in Listing 6.1 are tainted. Therefore, none of these variables have a mapping in M_T and therefore, these variables do not actually have to be wrapped! Currently, in our infrastructure, every time a literal such as ‘a’ is encountered, we wrap it. We also wrap the result of the concatenation of the variable a with the wrapped literal ‘a’. Furthermore, each time i is incremented, a new primitive is wrapped (representing the result of the increment). In the example in Listing 6.1, every iteration of the loop creates 3 wrapped primitives, for a total of $3n$ wrapped primitives at the end of the loop which are not necessary.

This incurs performance penalties for each operation, since each operation has to unwrap the wrapped object (which is a lookup on the wrapper map M_W, and with more keys in the map, lookup times would be longer). Here, we see that the number of wrapped primitives grows linear in the number of operations that the program has to execute, which would be a problem if the instrumented library is large. This implementation is unoptimized but simplifies the wrapper semantics, since we can simply treat all primitives as wrapped.

One solution to this is to optimize the wrapping semantics by wrapping on demand. Variables are wrapped if they are involved in an operation with a tainted variable, and all wrapped variables which are not tainted at the end of each operation are discarded. This would significantly reduce the overhead. We foresee that this optimization would alleviate problems in our framework caused by the large size of the underlying packages.

6.2 Better Language Support

As noted in Chapter 5, we are currently not able to run code which uses features from newer versions of JavaScript. From our feasibility study in Section 5.4, 3 out of 7 of the vulnerabilities found were in packages written in newer versions of JavaScript. If we wish to explore the space of NPM packages, better support for newer JavaScript versions must be implemented.

This can happen in multiple ways, and we elaborate more on each of the alternatives below:

1. **Transpilation.** Babel [2] is a commonly used transpiler for compiling newer versions of JavaScript to older ones. Traditionally, it was implemented to help browsers to continue to render websites even without support for the latest version of JavaScript [27]. Unfortunately, not all features can be faithfully replicated during the transpilation. For example, `let` and `const` are transpiled to `var`, which means that immutability in the `const` declaration is not guaranteed, and the original semantics may be altered [63]. If we adopt this approach, the original program might not execute correctly for some cases, which might lead to false positives or negatives, but we would largely be compatible with new code with little to no changes in our infrastructure.

2. **Modifying Jalangi2 to support newer JavaScript features.** Jalangi2 can be modified and updated to support newer JavaScript features. Some research groups are moving in this direction - for example, the ExpoSE framework uses a custom Jalangi2 framework which utilizes Babel underneath the hood to support newer versions of JavaScript [55], and a similar approach can be adopted in our infrastructure as well. However, such an update requires significant changes to the source code of the framework and might prove to be a large engineering feat.

3. **Utilizing a different instrumentation framework.** More promisingly, new virtual machine runtimes supporting `NODE.JS` that bake in instrumentation tooling are being open sourced. In particular, the GraalVM together with the Truffle language for instrumentation looks to be promising for use as the underlying instrumentation framework and runtime backend for dynamic taint analysis [53], but it is still largely untested in practice.

Currently, transpilation seems to be the most promising direction for future work, since it allows for future versions of JavaScript to be supported, while minimizing the changes made to our infrastructure. Through this, we can execute and test the majority of the NPM packages not just in the present, but also in the future, as long as we are willing to tolerate some errors caused by transpilation. In the future, alternative instrumentation frameworks (like Truffle for GraalVM) can be explored if transpilation raises too many errors.

6.3 Large Scale Analysis

In our current evaluation, we only looked at a handful of NPM packages that we know have a code injection vulnerability. We have shown that our framework succeeded in detecting the vulnerabilities, but we wish to extend this method to a larger setting to find new code injection vulnerabilities in the NPM ecosystem. However, such an analysis requires the language issue mentioned above to be solved, as well the creation of a test harness. Below, we will explore potential ideas for the test harness here.

An approach adopted by Staicu et al. used existing test cases of the package being evaluated to perform dynamic taint analysis [72], but not all packages come with test cases, and we speculate that those packages that do not have test cases are more likely to showcase such vulnerabilities. Automatic test case generation can also be explored. For example, we can try to infer the types of the arguments for an exported function using techniques from JSNice [64] and perform concolic execution to get better coverage in the testing, similar to that of [56] and [66]. This would pave the way for automatic code injection vulnerability detection that can be implemented in a repository, preventing such vulnerabilities from showing up in the ecosystem in the future.

7

Conclusion

In summary, we have shown that it is possible to build a platform agnostic framework, with a novel way of propagating taint in the face of tainted dynamic code generation. This is a particularly important contribution, especially in the backdrop of the ever increasingly popular `NODE.JS` framework. Looking at the existing NPM ecosystem, packages are often implicitly trusted by developers, even though few safeguards exists, and we see that there needs to be a way for analysts to do an offline analysis to ensure that these packages are not vulnerable to code injection attacks, and our tool, NodeTaintProxy, is there to help them achieve this goal.

Even though the ECMAScript standard is complex, inspired by [37], we developed a way of performing dynamic taint analysis that deferred execution to the JavaScript engine, thus minimizing our own need to reason about these semantics. We show our transformed semantics for each operation was equivalent to direct execution under the JavaScript engine, even in the presence of type coercion. We improved upon prior work by developing a platform agnostic, dynamic taint analysis framework that was architecturally sound. We also added support for byte-level tainting, and we do so by supporting auxiliary data structures in our taint layer. More generally, other data structures for storing taint information can also be supported, simply by swapping out the current taint layer with a different taint layer supplied by the analyst. This can be further augmented with taint propagation rules for that data structure. Our plug and play architecture for dynamic taint tracking allows for different modules to be used - new taint policies can be added seamlessly; even the underlying instrumentation framework can be changed, provided the set of interfaces that we require are correctly defined. This was something that could not be achieved in previous platform agnostic approaches in JavaScript dynamic taint analysis.

We also noted that previous platform independent approaches treated `eval` as a sink with no way of propagating taint further, but we improved upon that with a way of tainting written variables in the case where `eval` was not a sink, particularly when `eval` is called with a tainted argument. We were inspired my Mystique's approach [36] and adapted their approach to propagate taint via static analysis, and we achieve that even without access to the underlying JavaScript internals. We do this via program rewriting to make use of a shared context so that the instrumentation layer can push wrapped variables back into the local scope, and we detail the algorithms to perform this rewriting.

Our tool was successful in finding vulnerabilities for packages written in ES5.1 that were used in the evaluation of prior work, and our evaluation includes novel case studies that were added to the dataset. Furthermore, our tool does not flag benign packages that were studied in prior work, and was even able to keep track of the exact tainted indices of a given tainted string, provided that no imprecise tainting policy was applied. This is an improvement over previous coarse-grained approaches in platform agnostic dynamic taint tracking for JavaScript.

We have yet to see a large scale analysis of code injection vulnerabilities for packages in the NPM ecosystem, but our feasibility study showed that many such vulnerabilities still exist. By implementing a test harness and augmenting our tool with support for newer JavaScript features, we can run our tool on all packages in the NPM repository and find many new code injection vulnerabilities automatically. This will be the next step in our research.

To that end, we hope that the framework would eventually be the de facto standard for dynamic taint analysis in JavaScript, and that such a tool would be used by others to study and uncover new vulnerabilities in the wild, leading to a much safer JavaScript ecosystem.

Bibliography

[1] "Arbitrary Command Injection in ps | Snyk." [Online]. Available: https://snyk.io/vuln/SNYK-JS-PS-72307." [Accessed 2020-05-07].

[2] "Babel · The compiler for next generation JavaScript," library Catalog: babeljs.io. [Online]. Available: https://babeljs.io/. [Accessed 2020-05-07].

[3] "Creating and publishing scoped public packages | npm Documentation." [Online]. Available: https://docs.npmjs.com/creating-and-publishing-scoped-public-packages." [Accessed 2020-05-02].

[4] "CWE - CWE-94: Improper Control of Generation of Code ('Code Injection') (4.0)." [Online]. Available: https://cwe.mitre.org/data/definitions/94.html." [Accessed 2020-05-07].

[5] "CWE - CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection') (4.0)." [Online]. Available: https://cwe.mitre.org/data/definitions/95.html." [Accessed 2020-05-07].

[6] "Esprima." [Online]. Available: https://esprima.org/." [Accessed 2020-05-08].

[7] "ffmpegdotjs," library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/package/ffmpegdotjs. [Accessed 2020-05-07].

[8] "nbin," library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/package/nbin. [Accessed 2020-05-07].

[9] "node-serialize," library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/package/node-serialize. [Accessed 2020-05-07].

[10] "node-ts-ocr," library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/package/node-ts-ocr. [Accessed 2020-05-07].

[11] "npm advisory for cryo," library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/advisories/690. [Accessed 2020-05-07].

[12] "npm advisory for js-yaml," library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/advisories/813. [Accessed 2020-05-07].

[13] "npm advisory for node-serialize," library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/advisories/311. [Accessed 2020-05-07].

[14] "npm advsiory for node-serialize," library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/advisories/311. [Accessed 2020-05-07].

[15] "npm/npm," library Catalog: github.com. [Online]. Available: https://github.com/npm/npm. [Accessed 2020-05-06].

[16] "onion-oled-js," library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/package/onion-oled-js. [Accessed 2020-05-07].

[17] "OWASP Top Ten Web Application Security Risks | OWASP," library Catalog: owasp.org. [Online]. Available: https://owasp.org/www-project-top-ten/. [Accessed 2020-05-07].

[18] "picotts," library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/package/picotts. [Accessed 2020-05-07].

[19] "Remote Code Execution · A Roadmap for Node.js Security." [Online]. Available: https://nodesecroadmap.fyi/chapter-1/threat-RCE.html." [Accessed 2020-05-07].

[20] "@sapper-dragon/remark," library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/package/@sapper-dragon/remark. [Accessed 2020-05-07].

[21] "Stack Overflow Developer Survey 2019," library Catalog: insights.stackoverflow.com. [Online]. Available: https://insights.stackoverflow.com/survey/2019/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2019. [Accessed 2020-05-07].

[22] "Unsafe use of eval · Issue #2 · matthewkastor/modulify," library Catalog: github.com. [Online]. Available: https://github.com/matthewkastor/modulify/issues/2. [Accessed 2020-05-07].

[23] "VM (Executing JavaScript) | Node.js v14.2.0 Documentation." [Online]. Available: https://nodejs.org/api/vm.html." [Accessed 2020-05-08].

[24] “vm2,” library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/package/vm2. [Accessed 2020-05-08].

[25] “wincred,” library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/package/wincred. [Accessed 2020-05-07].

[26] “Netscape and Sun announce javascript, the open, cross-platform object scripting language for enterprise networks and the internet,” Dec. 1995. [Online]. Available: https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html. [Accessed 2020-05-02].

[27] “Transpiling ES6,” Sep. 2016, library Catalog: css-tricks.com. [Online]. Available: https://css-tricks.com/transpiling-es6/. [Accessed 2020-05-07].

[28] “Attitudes to security in the javascript community,” https://medium.com/npm-inc/security-in-the-js-community-4bac032e553b, Apr 2018.

[29] “estools/escope,” Apr. 2020, original-date: 2012-09-12T22:48:51Z. [Online]. Available: https://github.com/estools/escope. [Accessed 2020-05-08].

[30] “nice-registry/all-the-package-repos,” May 2020, original-date: 2016-12-04T21:10:04Z. [Online]. Available: https://github.com/nice-registry/all-the-package-repos. [Accessed 2020-05-07].

[31] “Samsung/jalangi2,” Apr. 2020, original-date: 2014-11-26T11:49:28Z. [Online]. Available: https://github.com/Samsung/jalangi2. [Accessed 2020-05-06].

[32] “tc39/test262,” May 2020, original-date: 2014-01-22T18:20:05Z. [Online]. Available: https://github.com/tc39/test262. [Accessed 2020-05-13].

[33] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, “Why do developers use trivial packages? an empirical case study on npm,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. Paderborn, Germany: ACM Press, 2017, pp. 385–395. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3106237.3106267. [Accessed 2020-05-06].

[34] E. Andreasen, Liang Gong, A. Møller, M. Pradel, M. Selakovic, Koushik Sen, and R.-A. Staicu, “A Survey of Dynamic Analysis and Test Generation for JavaScript,” *ACM Computing Surveys*, vol. 50, no. 5, pp. 66:1–66:36, Nov. 2017, publisher: Association for Computing Machinery. [Online]. Available: http://search.ebscohost.com/login.aspx?direct=true&db=buh&AN=125367040&site=ehost-live. [Accessed 2020-05-07].

[35] W. Chang, B. Streiff, and C. Lin, “Efficient and extensible security enforcement using dynamic data flow analysis,” in *Proceedings of the 15th ACM conference on Computer and communications security - CCS ’08*. Alexandria, Virginia, USA: ACM Press, 2008, p. 39. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1455770.1455778. [Accessed 2020-05-05].

[36] Q. Chen and A. Kapravelos, “Mystique: Uncovering Information Leakage from Browser Extensions,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto Canada: ACM, Jan. 2018, pp. 1687–1700. [Online]. Available: https://dl.acm.org/doi/10.1145/3243734.3243823. [Accessed 2020-05-02].

[37] A. Chudnov and D. A. Naumann, “Inlined Information Flow Monitoring for JavaScript,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS ’15*. Denver, Colorado, USA: ACM Press, 2015, pp. 629–643. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2810103.2813684. [Accessed 2020-03-18].

[38] C. Cimpanu, “Hacker backdoors popular javascript library to steal bitcoin funds,” https://www.zdnet.com/article/hacker-backdoors-popular-javascript-library-to-steal-bitcoin-funds/, Nov 2018.

[39] T. Claburn, “This typosquatting attack on npm went undetected for 2 weeks,” Aug 2017, library Catalog: www.theregister.co.uk. [Online]. Available: https://www.theregister.co.uk/2017/08/02/typosquatting_npm/. [Accessed 2020-05-02].

[40] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, “FlowFox: a web browser with flexible and precise information flow control,” in *Proceedings of the 2012 ACM conference on Computer and communications security - CCS ’12*. Raleigh, North Carolina, USA: ACM Press, 2012, p. 748. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2382196.2382275. [Accessed 2020-03-18].

[41] M. L. V. de Vanter, C. Seaton, M. Haupt, C. Humer, and T. Würthinger, “Fast, flexible, polyglot instrumentation support for debuggers and other tools,” *CoRR*, vol. abs/1803.10201, 2018. [Online]. Available: http://arxiv.org/abs/1803.10201.

[42] Ecma International, “Standard ECMA-262-archive.” [Online]. Available: https://www.ecma-international.org/publications/standards/Ecma-262-arch.htm.” [Accessed 2020-05-02].

[43] ——, “ECMAScript 2019 Language Specification,” Jun. 2019. [Online]. Available: https://www.ecma-international.org/ecma-262. [Accessed 2020-05-02].

[44] L. Gong, “Dynamic analysis for javascript code,” Ph.D. dissertation, University of California, Berkeley, USA, 2018. [Online]. Available: http://www.escholarship.org/uc/item/7n30n4kd.

[45] D. Grander and L. Tal, “A Post-Mortem of the Malicious event-stream backdoor | Snyk,” Dec. 2018, library Catalog: snyk.io Section: Vulnerabilities. [Online]. Available: https://snyk.io/blog/a-post-mortem-of-the-malicious-event-stream-backdoor/. [Accessed 2020-05-04].

[46] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, “JSFlow: tracking information flow in JavaScript and its APIs,” in *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC ’14*. Gyeongju, Republic of Korea: ACM Press, 2014, pp. 1663–1671. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2554850.2554909. [Accessed 2020-05-06].

[47] D. Hedin and A. Sabelfeld, “Web Application Security Using JSFlow,” in *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. Timisoara, Romania: IEEE, Sep. 2015, pp. 16–19. [Online]. Available: http://ieeexplore.ieee.org/document/7426055/. [Accessed 2020-05-06].

[48] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, “Dta++: Dynamic taint analysis with targeted control-flow propagation.” in *NDSS*. The Internet Society, 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2011.html#KangMPS11.

[49] P. Kannan, T. H. Austin, M. Stamp, T. Disney, and C. Flanagan, “Virtual values for taint and information flow analysis,” 2016.

[50] C. Karande, *Securing node applications : protecting against OWASP Top 10 risks*, first edition. ed. Sebastopol, CA: O’Reilly Media.

[51] R. Karim, F. Tip, A. Sochurkova, and K. Sen, “Platform-Independent Dynamic Taint Analysis for JavaScript,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8511058/. [Accessed 2020-05-01].

[52] M. Keil, S. Guria, A. Schlegel, M. Geffken, and P. Thiemann, "Transparent object proxies for javascript," 04 2015.

[53] J. Kreindl, D. Bonetta, and H. Mössenböck, "Towards efficient, multi-language dynamic taint analysis," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, ser. MPLR 2019. Athens, Greece: Association for Computing Machinery, Oct. 2019, pp. 85–94. [Online]. Available: https://doi.org/10.1145/3357390.3361028. [Accessed 2020-05-05].

[54] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of DOM-based XSS," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*. Berlin, Germany: ACM Press, 2013, pp. 1193–1204. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2508859.2516703. [Accessed 2020-05-05].

[55] B. Loring, "ExpoSEJS/jalangi2," Jul. 2019, original-date: 2018-05-20T11:33:05Z. [Online]. Available: https://github.com/ExpoSEJS/jalangi2. [Accessed 2020-05-08].

[56] B. Loring, D. Mitchell, and J. Kinder, "ExpoSE: practical symbolic execution of standalone JavaScript," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software - SPIN 2017*. Santa Barbara, CA, USA: ACM Press, 2017, pp. 196–199. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3092282.3092295. [Accessed 2020-05-06].

[57] J. Ming, "Pipelined symbolic taint analysis," 2016. [Online]. Available: http://search.proquest.com/docview/1847568155/.

[58] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07*, 2007.

[59] J. Newsome, S. McCamant, and D. Song, "Measuring channel capacity to distinguish undue influence," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security - PLAS '09*. Dublin, Ireland: ACM Press, 2009, p. 73. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1554339.1554349. [Accessed 2020-05-05].

[60] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity

Software," 1 2005. [Online]. Available: https://kilthub.cmu.edu/articles/Dynamic_Taint_Analysis_for_Automatic_Detection_Analysis_and_Signature_Generation_of_Exploits_on_Commodity_Software/6468716.

[61] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Auto-patching DOM-based XSS at scale," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. Bergamo, Italy: ACM Press, 2015, pp. 272–283. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2786805.2786821. [Accessed 2020-05-05].

[62] ——, "DexterJS: robust testing platform for DOM-based XSS vulnerabilities," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. Bergamo, Italy: ACM Press, 2015, pp. 946–949. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2786805.2803191. [Accessed 2020-05-06].

[63] A. Rauschmayer, "Deploying ECMAScript 6," Apr. 2015. [Online]. Available: https://2ality.com/2015/04/deploying-es6.html. [Accessed 2020-05-07].

[64] V. Raychev, M. Vechev, and A. Krause, "Predicting Program Properties from "Big Code"," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '15*. Mumbai, India: ACM Press, 2015, pp. 111–124. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2676726.2677009. [Accessed 2020-05-08].

[65] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The Eval That Men Do," in *ECOOP 2011 – Object-Oriented Programming*, M. Mezini, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6813, pp. 52–78, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-22655-7_4. [Accessed 2020-05-02].

[66] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A Symbolic Execution Framework for JavaScript," in *2010 IEEE Symposium on Security and Privacy*. Oakland, CA, USA: IEEE, 2010, pp. 513–528. [Online]. Available: http://ieeexplore.ieee.org/document/5504700/. [Accessed 2020-05-08].

[67] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications," 2010.

[68] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *2010 IEEE Symposium on Security and*

Privacy. Oakland, CA, USA: IEEE, 2010, pp. 317–331. [Online]. Available: http://ieeexplore.ieee.org/document/5504796/. [Accessed 2020-05-02].

[69] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for JavaScript," in *In ACM SIGSOFT International Symposium on the Foundations of Software Engineering (SIGSOFT FSE*, 2013.

[70] Snyk, "Vulnerability DB | Snyk." [Online]. Available: https://snyk.io/vuln?type=npm." [Accessed 2020-05-04].

[71] C.-A. Staicu, M. Pradel, and B. Livshits, "SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS," in *Proceedings 2018 Network and Distributed System Security Symposium.* San Diego, CA: Internet Society, 2018. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_07A-2_Staicu_paper.pdf. [Accessed 2020-05-02].

[72] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel, "Extracting taint specifications for JavaScript libraries," in *Proc. 42nd International Conference on Software Engineering (ICSE)*, May 2020.

[73] H. Sun, D. Bonetta, C. Humer, and W. Binder, "Efficient dynamic analysis for Node.js," in *Proceedings of the 27th International Conference on Compiler Construction - CC 2018.* Vienna, Austria: ACM Press, 2018, pp. 196–206. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3178372.3179527. [Accessed 2020-05-06].

[74] sweetim, "sweetim/haversine-position," May 2020, original-date: 2017-03-30T14:51:53Z. [Online]. Available: https://github.com/sweetim/haversine-position. [Accessed 2020-05-07].

[75] L. G. L. Thiel, "linus/refix," Aug. 2019, original-date: 2012-11-26T14:23:49Z. [Online]. Available: https://github.com/linus/refix. [Accessed 2020-05-07].

[76] J. Tortosa, "mongui," library Catalog: www.npmjs.com. [Online]. Available: https://www.npmjs.com/package/mongui. [Accessed 2020-05-02].

[77] M. Tran, X. Dong, Z. Liang, and X. Jiang, "Tracking the Trackers: Fast and Scalable Dynamic Analysis of Web Content for Privacy Violations," in *Applied Cryptography and Network Security*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos,

D. Tygar, M. Y. Vardi, G. Weikum, F. Bao, P. Samarati, and J. Zhou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7341, pp. 418–435, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-31284-7_25. [Accessed 2020-05-05].

[78] C. Williams, “How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript,” Mar. 2916, library Catalog: www.theregister.co.uk. [Online]. Available: https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/. [Accessed 2020-05-06].

[79] E. Wittern, P. Suter, and S. Rajagopalan, “A look at the dynamics of the JavaScript package ecosystem,” in *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR ’16*. Austin, Texas: ACM Press, 2016, pp. 351–361. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2901739.2901743. [Accessed 2020-05-06].

[80] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, “One VM to rule them all,” in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, ser. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, Oct. 2013, pp. 187–204. [Online]. Available: https://doi.org/10.1145/2509578.2509581. [Accessed 2020-05-05].

[81] W. Xu, S. Bhatkar, and R. Sekar, “Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks,” in *USENIX Security Symposium*, 2006.

[82] N. C. Zakas, “eval() isn’t evil, just misunderstood,” library Catalog: humanwhocodes.com. [Online]. Available: https://humanwhocodes.com/blog/2013/06/25/eval-isnt-evil-just-misunderstood/. [Accessed 2020-05-02].

[83] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, “Small World with High Risks: A Study of Security Threats in the npm Ecosystem,” 2019, pp. 995–1010. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman. [Accessed 2020-05-02].

Appendix A

Sample Test Program

The following test case is a sample test program that tests for special cases in binary operations equality, including type coercion and the lack of transitivity in the `==` operator, based on the notes in the ECMA-262 specifications [43].

```
import { __jalangi_assert_taint_true__, __jalangi_assert_taint_false__,
    __jalangi_set_taint__, __jalangi_set_prop_taint__,
   __jalangi_assert_prop_taint_false__,
    __jalangi_assert_prop_taint_true__} from "../../taint_header";
import {test_suite, test_one} from "../../test_header";
let assert = require('assert')

test_suite("- Equality Operations Correctness -", function() {
    let a = {};
    let b = {};
    test_one("NOTE 1a: coerce string comparison", function() {
        assert(""+a == ""+b)
    });

    test_one("NOTE 1b: coerce string comparison", function() {
        assert(a != b)
    });

    test_one("NOTE 1c: coerce string comparison", function() {
        assert(a !== b)
    });

    a = "1"
    b = "01"

    test_one("NOTE 1d: coerce Numeric comparison", function() {
        assert(+a == +b)
    });

    test_one("NOTE 1e: coerce Numeric comparison", function() {
        assert(a != b)
    });

    test_one("NOTE 1f: coerce Numeric comparison", function() {
        assert(a !== b)
    });

    a = "true"
    b = 1

    test_one("NOTE 1d: coerce boolean comparison", function() {
        assert (!a == !b)
    });

    test_one("NOTE 1e: coerce boolean comparison", function() {
        assert(a != b)
    });

    test_one("NOTE 1f: coerce boolean comparison", function() {
        assert(a !== b)
    });

    a = "hello";
    b = "hello";
    let c = "bye";
    let d = "bye2";

    test_one("NOTE 2a: De Morgan's and Symmetric", function() {
        assert((a != b) === !(a == b));
    });

    test_one("NOTE 2b: De Morgan's and Symmetric", function() {
        assert((c != d) === !(c == d));
    });


    test_one("NOTE 2c: De Morgan's and Symmetric", function() {
        assert((a == b) === (b == a));
    });


    test_one("NOTE 2d: De Morgan's and Symmetric", function() {
        assert((c == d) === (d == c));
    });


    test_one("NOTE 3a: Transitivity (and the lack of)", function() {
        assert(new String("a") == "a");
    });

    test_one("NOTE 3b: Transitivity (and the lack of)", function() {
        assert("a" == new String("a") );
    });

    test_one("NOTE 3c: Transitivity (and the lack of)", function() {
        assert((new String("a") == new String("a")) === false);
    });
});
```