Improving Code-Injection Vulnerability Detection and Confirmation in JS Programs

Nuno Sabino

CMU-CS-25-XXX

November 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

-

Thesis Committee:
Cristian-Alexandru Staicu
José Fragoso
Limin Jia, Chair
Lujo Bauer
Pedro Adão
Ruben Martins
Rui Abreu

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science.

This research was sponsored by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program (UIDB/50008/2020, Instituto de Telecomunicações, and PhD grant SFRH/BD/150692/2020).

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Code Injection Vulnerabilities, Exploit Synthesis, Dynamic Taint Analysis, Fuzzing, Symbolic Execution

Abstract

JavaScript applications face serious security risks, including client-side DOM-based Cross-Site Scripting (DOM-XSS) and server-side arbitrary command injection (ACI) and arbitrary code execution (ACE). Exploiting these vulnerabilities can lead to severe consequences, including unauthorized access to sensitive data and even full server compromise.

Dynamic taint analysis (DTA) tools have been used to identify how attacker-controlled input, such as a URL, may reach sensitive functions that lead to arbitrary code execution. Such propagations of attacker information, termed potential flows, can be good indicators of vulnerabilities. However, existing approaches struggle to (1) generate concrete inputs that exercise these flows due to limited path exploration, and (2) automatically confirm vulnerabilities, because inputs must satisfy program constraints while also triggering the intended side effects. This thesis leverages program analysis techniques to address these challenges, with tailored approaches for the distinct requirements of server and client code.

Client-side analysis is complicated by program behaviors dependent on user interactions and URL GET parameters. To overcome this, we developed a fuzzer to interact with the target web page and we employ dynamic symbolic execution (DSE) to synthesize GET parameters satisfying program constraints. Relative to our replication of prior work DOMsday, the fuzzer alone identifies 15% more vulnerabilities in a dataset of 44,480 popular pages, and the combination of fuzzing and DSE identifies 43% more vulnerabilities than DOMsday.

On the server-side, DTA-based tools miss ACI and ACE that require inputs with complex structure. We develop a novel type- and structure-aware fuzzing technique to explore Node.js packages, and an enumerator to synthesize syntactically valid payloads for ACE vulnerabilities. Extending NodeMedic with these components led to finding 1.7x more vulnerabilities.

Finally, we find that non-exploitable potential flows can still indicate real vulnerabilities, but exploitation may imply extra steps, such as bypassing sanitization or extending attacker capabilities. We introduce an exploitability metric based on a set of features of flows, designed to indicate proximity to an exploitable path, and use them to guide fuzzing and confirmation towards paths that are more likely to be automatically exploitable. Integrating this in NodeMedic-FINE results in 1% more confirmed flows, while saving 28% of the baseline confirmation time.

Acknowledgments

Many people have shaped the way I think and guided me to the conclusion of this chapter of my life. This work would not have been possible without my advisor Pedro's guidance, dedication and interest in my future. His mentoring strengthened my perseverance, determination, and methodical thinking. Still on the Portuguese side of my advising team, I also thank Rui Maranhão for his encouragment, valuable feedback, and steady guidance throughout my research. I am deeply grateful to my CMU advisor, Limin Jia, who helped me get better at identifying what truly matters, and to define clear goals in the research process. This high-level perspective has influenced me in more ways than I can express here. Finally, a special thanks to Lujo Bauer. Although not an official advisor, his objectivity, transparency, and scientific rigor have inspired me to become a better researcher.

I also thank the STT cybersecurity team and its members. The practical knowledge I gained there was critical both to the development of this thesis and to nurturing my interest for cybersecurity, a passion shared by many on the team. Thank you, Filipe, for letting me write your exploits until I was ready to capture some flags on my own.

I am grateful to my family for their patience, encouragement, and for keeping me grounded. I also feel fortunate to have lived in Pittsburgh with a group of energetic and optimistic friends who valued both intense work and intense play. Our philosophical discussions and shared experiences profoundly changed how I think and how I face life.

Finally, to my partner, Filipa: anything I could write here would be a vast understatement of how deeply you influenced me during this time. So I will just say I am glad I share my life with you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A recent survey by Stack Overflow found that JavaScript is the most commonly used programming language [99]. While originally developed as a language for client-side web scripting, JavaScript became popular as a server-side language thanks to the Node.js platform, being used by companies like Microsoft [3], Cloudflare [2] and Netflix [107]. JavaScript dominates both client- and server-side ecosystems: it is used by the majority of the world's web pages [122], estimated to number more than 900 billion [55], while on the server side the npm registry hosts over three million packages [6], making it one of the largest public package registries across all programming languages.

The JavaScript ecosystem is also very interconnected, meaning that code reuse and dependency chains link many projects together. Web pages often import vulnerable dependencies [69], e.g., outdated versions of jQuery. Similarly, packages in the npm ecosystem struggle with vulnerabilities, to the point where a single vulnerable npm package can affect a large number of dependents [138]. Each npm package typically has a set of public APIs, or entry points, specifying functions that can be called from other packages. Developers can build applications that depend on multiple npm packages and use their public APIs. In 2019, the average dependency tree size of a npm package was 86.55 [111], meaning that the average npm package depended on about 87 other packages. As developers compose applications from multiple JavaScript libraries or packages, they rely on these interconnections that amplify the frequency and impact of security vulnerabilities.

Web applications written in JavaScript are vulnerable to code injection attacks, where an attacker can illegitimately execute arbitrary code either in the browser of a user of the web page or the server hosting the web application. Approximately 5.5% of pages in 2020 were reported to be vulnerable to DOM-XSS [12], a vulnerability in client-side code where user inputs like the URL influence what code gets executed. On the server side, NodeMedic [24] found that at least 1.55% of the analyzed packages are affected by code injection vulnerabilities, while further studies corroborate the prevalence of such issues across the ecosystem [38, 58, 61, 66, 74, 75, 77, 80, 114, 115, 132]. These vulnerabilities include ACI [25] and ACE [26], which allow an attacker to execute arbitrary code or commands on the system that runs the application.

DOM-XSS vulnerabilities in the client, and ACE and ACI on the server side, are not only pervasive but dangerous when exploited. The complexity of these JavaScript ecosystems makes

it hard for humans to go through every piece of code and find all vulnerabilities, highlighting the need for automatic measures to detect code injection vulnerabilities. One way to detect these vulnerabilities automatically is by finding evidence of information propagations from certain insecure inputs to dangerous functions in the program, known as *taint flows*. An example of such a dangerous JavaScript function is `eval`, which takes a string as an argument and dynamically executes JavaScript code written on that string. Taint flows can be detected at runtime by tracking how attacker-controlled information (i.e., represented as "taint") influences program variables as operations are performed until the data reaches a sensitive function. Prior work has used these dynamic taint tracking approaches for finding flows [17, 24, 38, 61, 71, 88] but these tools often fail to thoroughly explore the application. Exploring a program thoroughly is challenging because it often requires triggering code paths that depend on highly specific inputs or interactions with the application. This general limitation manifests differently depending on whether we are looking at server or client code.

User interaction simulation and URL component synthesis. On the client side, prior work that found DOM-XSS vulnerabilities at scale in the wild, such as DOMsday [88] and others[17, 71], used a modified Chromium to find flows as code executes. However, those approaches were limited to passively navigating to each page and therefore miss vulnerabilities that require user interactions to be exploited. Furthermore, the relevance of including in the target URL keys and values of GET parameters to trigger vulnerabilities was not explored. These parameters may influence the behavior of the page, and their inclusion is key to exploring the client code more thoroughly and trigger more vulnerabilities. In this thesis, we introduce SWIPE, a tool capable of simulating user interactions, which improved DOM-XSS vulnerabilities found by 15% compared with our replication of prior work. SWIPE is also capable of synthesizing URL components like GET parameters, allowing it to find previously undiscovered DOM-XSS vulnerabilities.

Finding Node.js package inputs with complex structure. Prior work that detects ACI and ACE fails to find vulnerabilities that require input with complex structure. For example, packages often expect inputs to be objects with specific attributes. To address this limitation, we develop a fuzzer capable of reconstructing input types and object structure. Our fuzzer increases the ACI and ACE potential flows by by 70% compared to prior work. Furthermore, current approaches have difficulty exploiting ACE vulnerabilities as they require the final payload to be syntactically valid JavaScript. To address this gap, we introduce an enumerator component designed to aid synthesis of ACE payloads that are syntactically valid. This component enabled confirmation of 21% more ACE vulnerabilities. This thesis further describes our NODEMEDIC-FINE tool that integrates both the fuzzer and the enumerator components to improve code injection detection on the server side.

Finding exploitable potential flows. A significant fraction of the reported potential flows cannot be automatically exploited by the tools that find them [17, 24, 88]. In many cases, the vulnerability is not automatically exploited because of the high complexity of synthesizing a candidate exploit that works in practice. In other cases, the vulnerable program path that was

found is impossible to exploit, for example because the attacker has very limited control over the code that is executed. In this thesis, we design and implement an exploitability metric to estimate the automatic exploitability of a program path in npm packages, i.e., how easy it is to confirm flows following that program path, using NODEMEDIC-FINE's syntesis engine. We find that integrating this exploitability metric in the NODEMEDIC-FINE's fuzzer only slightly improves the number of confirmed flows over the baseline of not using it (+1%), though it significantly decreases confirmation time by a reduction of up to 28%. Our analysis shows that while the exploitability metric allows NODEMEDIC-FINE to discover flows in deeper parts of the application, it can also overprioritize analysis of unexploitable entry points that prematurely show promise of exploitability. Thus, we discuss future measures to address this limitation.

Studying dependents of packages with security warnings in their documentation. Instead of providing built-in input sanitization, we find 301 packages prefer to include a security warning in the documentation of their packages, delegating the responsibility of input sanitization to their dependents. But some of these packages have thousands of dependents that need to respect those warnings or risk being vulnerable. In this thesis, we study whether package developers heed warnings in the documentation of their dependencies, propagate those warnings, or simply pass unsanitized inputs to the vulnerable dependency entry points. This resulted in the discovery of 23 previously unknown vulnerabilities, most of which were in dependents of two particularly popular packages with warnings.

1.1 Roadmap

In Chapter 3, we present our work on automatically simulating user interactions on web pages and finding GET parameters through symbolic execution which helps uncover more DOM-XSS vulnerabilities. This work was accepted in NDSS 2026 [10].

In Chapter 4, we present our approach to overcoming the challenge of generating inputs with complex types and structure for Node.js packages. In that section, we also describe a component that automatically synthesizes syntactically valid JavaScript payloads for ACE vulnerabilities. These contributions were introduced in our paper NODEMEDIC-FINE [23], published at NDSS 2025 [9].

In Chapter 5, we describe our approach to guide fuzzing towards exploitable program paths.

Finally, in Chapter 6, we study whether package developers respect security warnings contained in the documentation of their dependencies.

1.2 Thesis Statement

In this thesis, we study how to effectively detect and confirm flows for code injection vulnerabilities in JavaScript applications. We propose methods leveraging program analysis techniques to explore applications more thoroughly, e.g., by generating Node.js package inputs of complex structure or by simulating user interactions on a web page. Finally, we leverage a set of features

of program paths to indicate their exploitability, and study whether fuzzing can be efficiently directed towards exploitable paths by prioritizing inputs that maximize those features.

Chapter 2

Attacker Model, Background and Related Work

This chapter provides an overview of the techniques, concepts, and prior work this thesis builds on. We begin by defining our attacker model in Section 2.1. Next, Section 2.2 reviews the vulnerabilities of interest and illustrates them with real-world cases. Section 2.3 surveys established approaches for detecting code injection flaws, while Section 2.4 introduces program analysis techniques for exploring multiple execution paths that help uncover vulnerabilities hidden deep within the code. We then turn to related work on exploit synthesis in Section 2.5. Finally, Section 2.6 examines vulnerability mitigation strategies.

2.1 Attacker Model

This section specifies the assumptions about what an attacker can control on the client and server sides. These assumptions define the scope of the threat model and establish the basis for how we evaluate vulnerabilities in later chapters.

In this thesis, we often refer to situations where an attacker-controlled input can influence sensitive parts of the application. We will often refer to this input as a *payload*. An *attacker* is defined as some entity that is attempting to illegitimately execute arbitrary code either on the server running a Node.js application or in the browser of a user of a web application. Depending on whether we refer to the client or the server side, there are different requirements for the payload to successfully exploit a code injection vulnerability.

2.1.1 Attacker Model on the Client Side

On the client side, we follow prior work in assuming that an attacker can influence all parts of the URL. This assumption originates in the relative ease for an attacker to send arbitrary links to victims, e.g., via email. This scheme is widely used in phishing attacks [49].

This work focuses on vulnerabilities found in a target web page that is not owned by the attacker. Thus, it is important to note, for accuracy, that there are parts of a URL that are difficult

to control for an attacker. Examples of parts of the URL that are challenging to control are the protocol [88] (usually http or https), and the host, since they must point to the vulnerable web page, and have therefore a very strict set of possible values that the attacker can set. However, GET parameters and the hash part of the URL are more flexible sources of attacker input and are the most frequent vectors of attack.

To leverage a vulnerability in a website and attack a user, the attacker typically constructs a link containing malicious data in the GET parameters or hash value. The goal of the attacker is for that malicious data to be interpreted as code by the vulnerable website code when it is loaded by the browser of the victim. The attacker sends that link to a victim and once the victim clicks on the link, their browser opens the vulnerable web page, which causes execution of attacker-controlled JavaScript contained in the link. Other sources like the cookies, referrer, local storage and the `window.name` variable were also considered by previous work on detecting code injection attacks as potential sources of attacker-controlled input, although when it comes to DOM-XSS, current approaches only automatically synthesize DOM-XSS exploits for URL-based sources [17, 71, 88].

2.1.2 Attacker Model on the Server Side

On the server side, similarly to existing work [24, 58, 80], we assume that the attacker can control the inputs to the APIs of the JavaScript packages. Such control by the attacker is reasonable because most packages (*dependencies*) can be used by any other package (*dependent*), which can import them and invoke their public APIs with arbitrary inputs. Consequently, even if no such *dependent* package exists today, at some point in the future another package could conceivably import the vulnerable package and use its insecure API with unsafe inputs.

Note that some packages may come with documentation stating that some of the package entry points are vulnerable to code injection unless input is sanitized *prior* to calling it. Typically, when a code injection detection tool flags such packages as vulnerable, they are manually classified as false positives after review of the documentation. More information regarding this type of false positives can be found in Section 4.4.7; additionally, we study in Section 6.3.1 whether developers heed such warnings in dependency documentations.

2.2 Review of Code Injection Vulnerabilities

In this section, we illustrate the vulnerabilities under study, namely DOM-XSS, arbitrary command injection, and arbitrary code execution, by providing instances of real-world code that we identified to be vulnerable to each vulnerability.

2.2.1 DOM-XSS

The Document Object Model (DOM) is a representation of the structure of a web document and its content. DOM-XSS is a cross-site scripting variant that occurs entirely within the DOM of a web page. Unlike traditional XSS vulnerabilities, where malicious scripts are injected into the HTML response generated by the server, DOM-XSS vulnerabilities are present on the client

```
1  var mapping = { // Allowed GET parameters
2    'custom1': 'key1',
3    ...
4    'custom10': 'key2'
5  };
6
7  function getJsonFromUrl() {
8    var result = {};
9    location.search.substr(1).split("&").forEach(function (part) {
10       var item = part.split("=");
11       result[item[0]] = decodeURIComponent(item[1]);
12   });
13   return result;
14 }
15
16 var urlParams = getJsonFromUrl();
17
18 function buildUrl(baseUrl) {
19   for (var key in mapping) {
20       if (!mapping.hasOwnProperty(key)) continue;
21       if (urlParams[key] != undefined) {
22           ...
23           baseUrl += encodeURI(mapping[key]) + "=" + encodeURI(urlParams[key]);
24       }
25   }
26   return baseUrl;
27 }
28
29 var basePostback = "https://alpha1trk2.com/impression/" + urlParams.custom1 + "?";
30 var postback = buildUrl(basePostback);
31 ...
32
33 document.write('<img src="' + postback + '" style="width: 0...;" />');
```

Figure 2.1: Simplified code with DOM-XSS vulnerability found in the wild.

side, typically after the web page has loaded. This also means that the server hosting the website may not even detect such attacks, especially if the malicious payload is inserted on the hash part of the URL, which is not even directly transmitted to the server and thus can be used to bypass Web Application Firewall (WAF)s [88].

DOM-XSS vulnerabilities arise when a web application manipulates the DOM in an insecure manner, allowing data from attacker-controlled sources to be inserted directly into a sensitive function within the DOM, also called a *sink*. Common examples of sensitive sinks include functions like innerHTML, eval, document.write, and setTimeout, which can dynamically execute JavaScript or result in dynamic rendering of HTML code.

In Figure 2.1 we show an example of real-world code[1] that is vulnerable to DOM-XSS. Overall, the dangerous `document.write` sink is called on line 33 and the argument depends on values that originate from the URL and therefore can be influenced by an attacker. Still, it is worth looking at the code in Figure 2.1 in depth, as it illustrates multiple challenges that need to be overcome before the vulnerability can be successfully detected and confirmed automatically. Lines 7–16 extract GET parameters from the URL and decode them, making it possible for an attacker to pass special characters that would otherwise be encoded naturally by modern

[1]For each real-world example we show, we anonymize it by renaming variables and simplifying the code, so as to not reveal information about unpatched vulnerabilities.

```
1   module.exports = {
2     sync: function(params, callback) {
3       var exec = require('child_process').exec;
4       var cmd = 'rsync';
5       if(params.flags !== undefined) {
6         cmd += ' -' + params.flags;
7       }
8       ...
9       exec(cmd, function(error, stdout, stderr) {
10        ... // call callback
11      });
12    }
13  }
```

Figure 2.2: An example ACI vulnerability

```
1   var package = require('vulnerable_package_name');
2   package.sync({"flags": "$(touch success)"});
```

Figure 2.3: An example exploit for ACI

browsers. Lines 18–30 construct a link based on the GET parameters provided by the user but all parameters are encoded again (line 23), except the one with key **custom1**, which is injected into the link without any sanitization on line 29. Finally, on line 33, that link is inserted into the DOM as the `src` attribute of an `img` tag in HTML. Therefore, an attacker can craft a malicious value for the **custom1** GET parameter that escapes the `src` attribute context and eventually execute arbitrary JavaScript. An example of a final link that an attacker may craft to execute arbitrary JavaScript in this case is: `https://vulnerable_page.com/?custom1=invalid" onerror="alert(1);`. The first part of the value of `custom1` makes the `src` attribute point to an invalid location, followed by double quotes, which escape the `src` context and allows the attacker to insert more attributes in the `img` tag. The rest of the value defines an `onerror` attribute in the `img` tag, which will be executed like normal JavaScript when the browser fails to load the image, in this case the attacker chooses to pop up an alert dialog.

2.2.2 ACI

In the context of Node.js applications, ACI is a serious security vulnerability that allows an attacker to execute arbitrary commands on a server hosting the package. ACI vulnerabilities typically arise when attacker-controlled input (i.e., entry point arguments) are used as part of the arguments of functions like `child_process.exec` or `child_process.spawn`, which are designed to execute arbitrary commands given as arguments. Exploitation of an ACI vulnerability allows an attacker to execute arbitrary commands on the server under the same privileges as the Node.js application, potentially enabling privilege escalation leading to full system compromise. An example of an ACI vulnerability is given in Figure 2.2. That figure shows an entry point called `sync`, which takes two arguments, assumed by our attacker model to be attacker-controlled. If an attacker calls the `sync` API with a carefully constructed first argument `params`, then they are able to execute arbitrary commands when line 9 is reached. This is because line 6 extends the command `cmd` with arbitrary data coming from the attacker. Note that for line 6 to execute, the attacker has to pass an object with the key `flags` as the `params` argument.

```
1  class Handler {
2    constructor(handler, router) {
3        this.handler = handler
4    }
5    ...
6    when(cond) {
7        if (typeof cond === 'string') {
8            this.cond = new Function('message', 'with(message) {return ' + cond + ';}')
9        }
10       ...
11       return this
12   }
13
14   test(message) {
15       return (!this.cond || this.cond(message))
16   }
17   ...
18 }
19 ...
20 module.exports = Handler
```

Figure 2.4: Simplified code with ACE vulnerability found in the wild.

A proof-of-concept exploit that triggers this vulnerability is given in Figure 2.3. It simply imports the package and calls the vulnerable entry point `sync` with an appropriate object as an argument. The `flags` attribute is injected in the final command that is executed, containing a payload that causes the creation of a file called *success* in the filesystem, using the `touch` command. Note that the attacker used the `$(subcommand)` syntax which allows to run arbitrary shell commands inside other commands.

2.2.3 ACE

Still in the context of Node.js packages, ACE allows an attacker to control what JavaScript code is executed. This vulnerability occurs when attacker-controlled input reaches the arguments of dangerous Node.js functions that are designed to execute arbitrary JavaScript code, like `eval` or `Function`. By exploiting an ACE vulnerability, an attacker gains the capability of executing arbitrary JavaScript. If sandboxing is inadequate or can be escaped, this capability allows access to program state, and, via Node.js APIs (e.g., `require('child_process')`), escalation to arbitrary *command* execution on the host.

In Figure 2.4 we show an example of a real-world npm package that we found to be vulnerable to arbitrary code execution. The package exports, on line 20, at least two methods of a class:

1. `when` (lines 6–12), which dynamically generates code based on the `cond` argument. This argument is assumed to be controlled by the attacker, based on our attacker model.
2. `test` (lines 14–16), calls the constructed function if it is defined. This function is intended to be a boolean condition.

This package is vulnerable to ACE: an attacker could call `when` and construct a function containing arbitrary code instead of a simple boolean formula as the package expects. A concrete exploit for this example is shown in Figure 2.5, in which an attacker imports the vulnerable package and calls the `when` API with a especially crafted argument. That argument contains code that the package

```
1  var package = require('vulnerable_package_name');
2  package.when("console.log('exploited')"); // Injects payload in a local variable
3  package.test("irrelevant"); // Calls payload
```

Figure 2.5: Example exploit for ACE.

appends to `return` before executing. Finally, once the `test` API is called, the following attacker-controlled JavaScript will be executed: `with (message) return console.log('exploited');` which, in this case, simply prints a message to the console.

2.3 Program Analysis Techniques for Signaling Code Injection Vulnerabilities

This section outlines program analysis techniques used to detect code injection vulnerabilities. These techniques are typically separated in two categories: static analysis tools examine code without executing it, while dynamic analysis tools require program execution. While some previous works used dynamic taint tracking techniques [38, 61, 109], others used static approaches [21, 58, 66, 74, 75, 77, 114, 115].

2.3.1 Static Analysis

To analyze a package, static approaches generally parse the code and construct a high-level representation (usually graph-based) and later query it to detect situations where information from attacker-controlled inputs might flow to dangerous APIs.

Since the release of Node.js in 2009, literature on code injection vulnerability detection has improved in accuracy and comprehensiveness. This was often accomplished by either improving the high-level representations of JavaScript applications, or improving the way those representations are used.

Madsen et al. (2015) [77] introduced event-based call graphs, a program representation that captures callback and event emission relationships in Node.js programs. These relationships helped reason about program behavior in Node.js, considering its event-driven, asynchronous architecture.

Synode [115] (2018) used an intraprocedural analysis to detect API calls that are possibly unsafe, starting from the call itself and backward propagating information about the set of string values that might be passed to that call, also known as *templates*, by following control flow edges in the inverse order.

Another challenge in analyzing Node.js is handling the heavy use of third-party libraries, whose internal data flows are not fully considered in the previously mentioned approaches. Taser [114] (2020) addresses this by automatically extracting taint specifications for JavaScript from test cases. By integrating these summaries, a static analysis tool can more accurately track information propagation across module boundaries, uncovering vulnerabilities that would be missed with naive modeling.

Researchers have also sought general static frameworks to detect a broad range of Node.js vulnerabilities in a unified way. ODGEN [75] (2022) builds an Object Dependency Graph (ODG) which represents objects as nodes and links them via program AST relations (e.g., property definitions and lookups). This is done using abstract interpretation, a technique that replicates code execution in an abstract domain. ODGEN's ODG graph is flexible enough to enable vulnerability detection (i.e., through querying the graph) for 13 known vulnerability types in Node.js, not limited to code injection vulnerabilities.

However, ODGEN possesses several scalability issues, as studied by a later work FAST [58]. FAST improves ODGEN's analysis by performing a bottom-up analysis that resolves dynamic call edges more coarsely by analyzing functions in isolation and then a top-down phase selectively follows promising control-flow paths from sources to sinks, pruning away irrelevant branches of the program. FAST's bottom-up analysis is responsible for constructing a control-flow graph and discovering paths from entry points to sinks. FAST's top-down phase takes each of those paths and scalably constructs a data-flow graph by focusing on the set of statements with control and data dependencies with the sink.

Recently, Graph.js [34] (2024) merged the abstract syntax tree, the control-flow graph and the data dependency graph into a new structure called Multiversion Dependency Graph (MDG). Importantly, that work shows that MDGs can be constructed in a way that soundly overapproximates the concrete execution traces. Explode.js [80] (2025) used this MDG not only to determine whether each sink call in the target package is reachable from an entry point, but also to check the existence of a data flow connecting the arguments of entry points to the arguments of a relevant sink.

2.3.2 Dynamic Analysis

We call an approach *dynamic* if it executes the code and observes its behavior. Many of the previously mentioned static tools have some kind of dynamic component. For example, Synode constructs a grammar from collected templates on packages that it cannot statically deem safe, and uses that grammar in a dynamic enforcement mechanism to block malicious input being sent to sensitive sinks during execution. Taser instead generates summaries of how data propagates through library functions, including callbacks and class methods using dynamic analysis of test cases.

Dynamic taint analysis

Dynamic Taint Analysis (DTA) is a powerful technique used to identify security vulnerabilities [38, 61, 109] by tracking the propagation of potentially attacker-controlled data through a program execution, until it reaches a sink. The core idea behind DTA is to mark (or "taint") data that originates from untrusted sources, such as user inputs, and then monitor how this tainted data propagates through the program execution. By examining how tainted data reaches sensitive operations or sinks, such as dynamic code generation functions, DTA can reveal potential vulnerabilities that could be exploitable by attackers.

DTA operates by instrumenting the program to propagate taint labels alongside the data as it moves through different functions and operations. Once the instrumented program calls a

dangerous function like `eval`, it can then check whether the arguments of that function are labeled as tainted, which signifies that they may depend on initially tainted variables. In such a scenario, the tool would report what is called a *flow*. Since initially tainted variables are the ones that we consider controllable by an attacker, a flow represents a propagation of information from attacker-controlled variables to arguments of dangerous functions that we do not want attackers to control.

Detecting code injection vulnerabilities using dynamic taint analysis

To analyze the client code, several prior works on DOM-XSS detection used DTA to detect and confirm code injection vulnerabilities. We abbreviate the project names of the three most relevant DOM-XSS detection works as follows: 25mFlows [71], DOMsday [88] and TalkGen [17]. All three implemented DTA similarly in the respective JavaScript engines, but differed on the breadth and depth of their analysis and their confirmation methodologies. 25mFlows applied byte-level taint tracking to JavaScript code and demonstrated the prevalence of DOM-XSS vulnerabilities in the wild. While 25mFlows attempted to crawl all subpages of the Alexa Top 5000 domains in 2013, DOMsday targeted a maximum of 5 subpages for each of the Alexa Top 10,000 websites in 2018. More recently in 2020, TalkGen instead used the top 100,000 domains in Tranco, targeting a maximum of 10 subpages each. We will discuss methodological differences between these works in Section 2.5.4.

Instrumenting JavaScript code vs. JavaScript interpreter

An alternative way to implement DTA is by instrumenting the JavaScript code itself. But instrumenting the interpreter (e.g., the browsers' V8 engine) has lower time and memory overhead and avoids the need to bypass page integrity checks [61, 125]. However, Chromium updates may break the taint analysis engine [60], which makes it challenging for academic researchers to keep their tools usable by others. DOMsday's original taint-enabled browser used Chromium 54, which was released in 2016 and no longer supports the latest ECMA versions used by web pages. This makes direct comparison with DOMsday very difficult. Our tool SWIPE instead uses Chromium 126 as the base browser, which was upgraded from DOMsday's original browser with substantial engineering effort.[2]

2.4 Program Exploration Techniques

Dynamic taint analysis alone acts upon a single execution path of the program. In this section, we review several techniques that are used in combination with dynamic taint analysis to explore multiple executions, such as fuzzing [36], symbolic execution [50] and even a combination of both [101]. In short, fuzzing generates and executes diverse inputs to discover unexpected behaviors, whereas symbolic execution reasons about program paths by treating inputs as symbolic variables.

[2]Credit goes to Michael Stroucken for upgrading DOMsday's browser.

2.4.1 Fuzzing

Fuzzing is a widely-used software testing technique designed to identify bugs by randomly generating diverse inputs, feeding them to the target program and observing their effect. This technique has proven to be effective at detecting a variety of issues, including those related to memory corruption, input validation, and code execution [137]. Fuzzing works by systematically feeding malformed or random data into an application and observing how it handles these inputs. The specific method for generating inputs usually varies according to the context.

Guided fuzzing Evolutionary fuzzing is a class of fuzzing techniques that iteratively refine inputs based on their effectiveness [105]. Inputs that achieve desirable outcomes, such as triggering new program behaviors, are selected and mutated to produce new candidates. Some approaches also recombine fragments of high-performing inputs to create new test cases that may inherit beneficial traits from both ”parents” [105], a technique known as *cross-over*.

The notion of effectiveness or fitness varies across fuzzers, but code coverage (i.e., the amount of program code executed by a given input) is the most common heuristic. Coverage-guided fuzzers therefore prioritize and refine inputs that exercise previously unexplored or extensive portions of the application's codebase.

Fuzzing on the client side. There are fuzzers to discover GET parameters used by a target web page [52, 81, 127] but they usually simply iterate over fixed inputs stored in a wordlist. There is also work on generating user interactions on the web page but these tools limit themselves to filling forms [82, 121, 123] or are very restricted in the actions they produce, e.g., by only being able to produce clicks, keyboard inputs or fill forms automatically [31, 79, 89, 113, 131]. Furthermore, web scanners like CrawlJax [89] and jÄk [104] fire most of their supported event handlers programmatically via JavaScript. This might lead to false positives, in situations where there is no way for a real user to execute some of the simulated interactions (e.g., the associated DOM element might be invisible). LOAD-AND-ACT [131] was the first to perform relistic simulation of user interactions on web pages although, like prior work, it only supports generating keyboard and mouse events. In Chapter 3, we introduce a systematic approach to synthesize the GET parameters expected by each target page, along with a fuzzer that not only supports realistic simulation of a wide range of user interactions but also scaled to run against thousands of pages in the wild.

Fuzzing the server side. For the server side, fuzzing tools like AFL [135] have been adapted for Node.js fuzzing [13]. These generate mostly byte sequences or strings and lack knowledge of JavaScript's rich type system. However, we find that searching the string space only is not sufficient to uncover a significant number of vulnerabilities. Other approaches for input generation rely on tests from the target package or one of its dependents [117], though such tests do not always exist. JsFuzz [56] makes it easier to generate inputs more suitable for JavaScript environments but their approach still relies on string-based input generation and a manual creation of a fuzzing target. A fuzzing target is a program that imports the target package and the fuzzer and manages their interaction, including obtaining inputs from a fuzzer and passing them

to the package entry points. Some Node.js packages have entry points that expect other specific JavaScript types like objects, arrays and especially functions, as we will show in Chapter 4. On that same chapter, we introduce a type- and structure-aware fuzzer, capable of generating inputs of a variety of types and with complex structures, e.g., objects with specific attributes that have to be themselves objects.

2.4.2 (Dynamic) Symbolic Execution

In this section, we review another technique for exploring code called *symbolic execution*. In contrast to fuzzing, symbolic execution is effective at exploring deep program paths requiring inputs that satisfy complex constraints, though it does not typically scale as well as fuzzing to large programs [16]. Symbolic execution represents some application inputs as symbolic variables, and gathers constraints while executing the program. It is then possible to feed those constraints to theorem solvers like Z3 [29] and obtain an input that satisfies the restrictions. This technique is used by several works in the context of JavaScript [76, 80, 108, 132]. To analyze the client code, this technique may be implemented by instrumenting the program (e.g., a JavaScript resource on the web) to track symbolic variables (e.g., the URL) and the operations that are performed on them, so as to accurately build the constraints.

One common variant of symbolic execution is Dynamic Symbolic Execution (DSE), which may instrument the program as described above, and then runs it with a starting input, for example by loading a page that uses that instrumented JavaScript resource. Once the page finishes executing and in order to generate new inputs, DSE collects the final list of constraints, negates one of them and passes the resulting constraints to Z3. This way, the technique can successfully find inputs that explore a different branch of the program compared with the original input that produced the initial list of constraints.

In theory, by searching all combinations of constraints and their negated variants, symbolic execution could get full coverage in the target application. In practice, this technique suffers from scalability problems [16], not only because there might be too many different program paths to explore but also the SMT solvers may have difficulty solving certain constraints. ExpoSE [76] focuses on modelling JavaScript regular expression semantics, which takes some of the heavy lifting from the SMT solver, resulting in increased code coverage. Our tool, SWIPE (Section 3), focuses on modeling string operations that are often used in URL component parsing, a common operation in web pages. Another way to address the large space of possible program paths is to create heuristics to prioritize which program paths to explore first. For example, the buggy-path-first heuristic [120] prioritizes paths that seem to contain small but unexploitable bugs.

2.5 Exploit Synthesis

Dynamic taint analysis suffers from false positives: the presence of a flow does not guarantee the presence of a vulnerability. In this section, we review approaches to confirm the exploitability of vulnerabilities. We will start by motivating the need for exploit synthesis in Section 2.5.1. In Section 2.5.2, we study how vulnerability confirmation can be reduced to synthesizing proof-of-concept exploits that exhibit particular side effects depending on the vulnerability type. On one

hand, Section 2.5.3 surveys approaches to exploit synthesis that rely on collecting and solving SMT constraints, which is particularly relevant for server-side analysis where complex back-end logic is common. On the other hand, Section 2.5.4 covers approaches for exploit synthesis focused on client-side vulnerabilities. Finally, we discuss limitations of these approaches in Section 2.5.5.

2.5.1 Overview

False positives can originate when the propagation of taint is too relaxed, also known as overtainting [57]. Thus, when DTA discovers a flow from an attacker-controllable source to a sensitive sink, we call it a *potential flow*, since it is unknown if the flow can be exploited. This highlights the importance of confirming whether the program path represented by the flow is actually vulnerable. Once we determine that a flow is exploitable, we call it a *confirmed flow*.

2.5.2 Observing Expected Side Effects to Confirm Vulnerabilities

A potential flow can be confirmed by proving that an attacker can indeed execute arbitrary code. This is often done by producing a concrete proof-of-concept exploit, which goal slightly differs depending on the vulnerability type. Arbitrary command injection vulnerabilities can be demonstrated by making the application create a specific file on the file system using the *touch* linux command [19, 24, 80]. With respect to arbitrary code execution, NodeMedic [24] leverages the vulnerability to print a specific message to the console using the `console.log` function. SecBench.js [19] and Vulcan [21] are datasets of Node.js vulnerabilities, providing proof-of-concept exploits accompanying each vulnerability. Unlike NodeMedic, SecBench.js proves arbitrary code execution vulnerabilities similarly to code injection by creating a file in the file system. This is usually done by having the exploit require the `fs` module and create a file using the `writeFileSync` function. For automatic exploit synthesis, it is simpler to synthesize an exploit that uses a `console.log` function call to prove that the attacker has arbitrary JavaScript execution capabilities, which is what our tool NODEMEDIC-FINE does. Vulcan is not as uniform in the method to confirm code injection vulnerabilities but overall, in its exploits, it attempts to show that the attacker can execute commands outside of the original functionality of the package.

2.5.3 Use of SMT Synthesis to Generate Exploits

Automatic exploit synthesis in JavaScript is an active field [17, 24, 35, 37, 71, 103]. Prior work on JavaScript exploit synthesis mostly targets cross-site scripting vulnerabilities [17, 35, 37, 72, 103] by parsing the AST of the sink call. This works well in web pages because input sources are often global (e.g., `location.search` variable, containing the GET parameters) and accessed near the sink [22]. In the case of Node.js packages, inputs are local and frequently transformed before being passed to the sink. In that situation, it is useful to produce, during dynamic taint analysis, a taint provenance graph, which has a node for each operation performed on tainted values.

Provenance graphs allow for a full reconstruction of what transformations the taint values suffered and where they came from. NodeMedic [24] used provenance graphs to build constraints

and used Z3 to produce the final exploit. FAST [58] has also used synthesis to generate proof-of-concept exploits, but, unlike NodeMedic, it is a static analysis tool and it collected control-flow constraints via abstract interpretation. FAST starts by generating an object dependence and control-flow graph through abstract interpretation and finds a path between entry points and sink functions. Then, it constructs a data flow following that path. FAST proceeds to generate an exploit by solving constraints collected from both the data flow, the control flow and the object dependence graph. Thus, FAST's constraints use only information from static analysis and therefore may miss important dynamic information. More than 90% of FAST's false negatives come from the lack of modeling of built-in functions [58], which comes free in NODEMEDIC-FINE and other dynamic analyses since they involve executing the package. Notably, by not having to install and execute a package, static approaches may be able to analyze packages that are challenging to install automatically, e.g., packages that require extra installation steps that are only described in their documentation. But that also means that not all vulnerabilities reported by FAST are necessarily exploitable, since FAST ends up not testing the exploit concretely.

PMForce [116] synthesizes ACE exploits delivered via the `postMessage` API's `event` object. PMForce gathers path constraints and uses them to fill exploit templates used for `event.data`. PMForce does not handle ACE-specific breakouts, unlike our tool NODEMEDIC-FINE which attempts to parse the constant parts of the ACE sink argument, and synthesize an attacker-controlled portion of the argument that properly completes that constraint prefix. PMForce is similar to NAVEX [15] in that they both model path constraints, although NAVEX constraints inputs to include strings from an attack dictionary, instead of using exploit templates. Unlike PMForce or NAVEX, NODEMEDIC-FINE uses provenance graphs, which encode constraints on *operations*, instead of on path constraints. This allows NODEMEDIC-FINE to solve for structured inputs with more ease.

One limitation of NODEMEDIC-FINE is that it fails to find vulnerabilities that require more than just a direct call to a package entry point. Explode.js [80] is a very recent work that leverages static analysis using Graph.js to construct drivers composed of sequences of calls. Explode.js then symbolically executes each driver, and collects the necessary constraints for attacker payload to reach the sink. Even more recently, PocGen [110] integrates Large Language Models (LLMs) in exploit synthesis pipelines. To do this, PocGen collects usage examples of the vulnerable function from the application code, and constructs prompts in natural language based on those examples, the vulnerability type, a description of the vulnerability and a high-level description of how the exploit should look like, including examples of exploits of similar vulnerabilities.

2.5.4 Existing Methodologies for DOM-XSS Vulnerability Confirmation

A flow is considered *potential* by 25mFlows, DOMsday, TalkGen and our work, if the source is URL-based, the sink is Javascript or HTML-based, and if tainted bytes reach the sink without any encoding. All these works share a generic recipe for confirming DOM-XSS potential flows: (1) Collect the URL of the resource where the potential flow was uncovered. This step is the same for each work in theory, but implemented in a different browser instrumented to use DTA. (2) Locate within that URL where exactly the attacker payload should be injected. (3) Inject the payload at the location identified in the previous step, such that when the final URL is visited, it allows to validate whether the vulnerability is real.

Tainted URL	http :// example.com/page?q=tainted&a=b
Prior work	**Example confirmation URL**
25mFlows [71]	http :// example.com/page?q=tainted&a=b#PAYLOAD
DOMsday [88]	http :// example.com/page?a=b#&q=PAYLOAD
TalkGen [17]	http :// example.com/page?q=PAYLOAD&a=b

Table 2.1: Confirmation URLs from existing methodologies.

It is on step (2) that previous works start to differ methodologically. This is evidenced in Table 2.1, which shows where each previous work injects the PAYLOAD, in a scenario where the value of the GET parameter `q` is injected on a sink argument. While DOMsday used a similar approach for DOM-XSS detection compared with 25mFlows, they improved the precision of the confirmation methodology. DOMsday uses taint provenance information to determine the exact bytes in the URL that reach the sink, maps them to a GET parameter value and injects the payload as the new value for that parameter on the final URL. While DOMsday restricted their payloads to the hash part of the URL, TalkGen improved the accuracy even further, allowing payloads on the GET parameters before the hash.

Regarding step (3), 25mFlows uses a *breakout* method where the abstract syntax tree (AST) of the final argument to the sink is parsed and analyzed to determine what characters an attacker needs to inject so as to escape the current context. To illustrate this process, suppose a web application had the following vulnerable sink call: `eval('"hey"+"'+location.search+'";')`. 25mFlows method can automatically reason that the context where the attacker payload is injected into is a JavaScript string on an `eval` call, and thus it inserts a double quote and a semi collon to enter a context where the rest of the attacker payload is executed. Such an exploit would look like this: `";report\_domxss()//`. Note that `report_domxss` is a personalized function implemented in 25mFlows's modified browser that tracks confirmed flows. TalkGen improved the analysis of the AST to support more contexts.

However, DOMsday always injects the same payload: `marker<>'"` instead of trying to execute real code to confirm vulnerabilities. This is because they considered executing actual code to be dangerous, since, unlike testing exploits in self-hosted Node.js packages, exploits on live web pages affect more than the researcher's devices. It should be noted that the special characters on that payload (i.e., the opening and closing tags, and the single and double quotes) allow to determine whether the attacker has the ability to escape the necessary context and execute JavaScript code. This is because to confirm a vulnerability, the final synthesized URL (containing the payload) is visited and then DOMsday checks whether the substring `marker<>'"` is present on the sink argument as is, without any encoding. If that happens, it is evidence that the attacker can indeed escape most contexts and insert arbitrary code to be executed, without being sanitized by application code. DOMsday authors sampled 40 cases that were flagged as vulnerabilities, and manually confirmed that they were indeed true positives, thus we use the same payload in this work.

Automatic URL-encoding and its impact on DOM-XSS exploitability

To be exploitable, the sink argument cannot be URL encoded, otherwise the attacker will not be able to inject the necessary special characters (e.g., quotes) to escape the context and execute arbitrary JavaScript code. Over the years, browsers started applying their built-in mechanisms for URL encoding on more parts of the URL.

In 2013, when 25mFlows validated exploits, they tested one third of the potential flows against Internet Explorer, purposely because that browser did not have many of these built-in URL encoding mechanisms. In 2018, DOMsday injected their payloads on the hash value, and the Chromium version used by DOMsday did not yet URL encode the hash. In 2020, TalkGen was using a modified version of Firefox for taint tracking, which purposely disabled URL encoding. TalkGen authors argued that at least one browser (Internet Explorer) still had no URL encoding mechanism built-in, and vulnerabilities that are not exploitable because of this mechanism should still be counted as true positives. This argument was reasonable in 2020, but nowadays, Internet Explorer is discontinued and the latest version of all modern browsers enforces URL encoding on both the GET parameters and the fragment value. We discuss these and other insights regarding impactful evolutions of the web for DOM-XSS detection in Chapter 3.

Detection and confirmation of other client-side JavaScript vulnerabilities

Vulnerabilities such as client-side prototype pollution [59] and client-side CSRF [64] can be detected using similar techniques as for DOM-XSS. While SWIPE's instrumented browser could be extended to include more sinks, prototype pollution requires more effort for detection and confirmation. There are detection approaches based on code-property graphs [59], but all existing work resorts to manual confirmation to the best of our knowledge.

2.5.5 Limitations of Synthesis Tools

Synthesizing exploits for arbitrary code execution is more difficult than for arbitrary command injection [24]. Exploit synthesis for ACE is harder because the final argument needs to satisfy difficult constraints, originating from the need of the final payload to be valid JavaScript code in `eval` and `Function` sinks. Our tool NODEMEDIC-FINE uses a JavaScript-syntax aware enumerator to aid in the construction of syntactically valid payloads.

Interestingly, there are several possible scenarios of program paths that are signaled as potentially vulnerable by code injection detection tools but for which no available tool can synthesize a working exploit. For example, a vulnerability may only be triggered under very specific and complex constraints that a SMT solver like Z3 may have trouble solving. Another possibility is when the final argument to an ACE sink has a fixed prefix, followed by attacker-controlled data. The combination of the need of ACE sinks to require the whole argument to be syntactically valid JavaScript, and the need of the attacker to execute arbitrary code, makes it challenging for an exploit synthesis framework to create a working proof-of-concept in those cases. In Chapter 5 we study this problem in more depth, and provide a solution that mitigates it. Instead of improving the ability of exploit synthesis frameworks to handle such cases, we instead propose

to modify detection heuristics to prioritize confirmation of potentially vulnerable program paths that appear easier to exploit.

All exploit synthesis tools still have false positives. Not every confirmed flow is a vulnerability. We define *true positive* as a confirmed flow that does not correspond to a legitimate, intended behavior of the application. There are plenty of examples of Node.js packages whose sole purpose is to execute arbitrary commands.[3] In those cases, it is possible to discover and confirm a flow while staying within the intended functionality of the package. Such cases are usually considered false positives and undeserving of CVEs. There are also packages with dangerous entry points (e.g., allowing arbitrary command execution) but at some point in their documentation maintainers state that other developers using such entry points are responsible for sanitizing their inputs. While developers may miss that warning in the documentation (Section 6.3.1), we still consider confirmed flows in such packages to be false positives. All available exploit synthesis tools are prone to finding confirmed flows that are not actually vulnerabilities, which usually leads authors to manually analyze any findings. Independently of the context (i.e., server or client side), when reporting vulnerabilities, we follow responsible disclosure guidelines to contact package or website maintainers.

2.6 Vulnerability Mitigation

In this section, we cover mitigation techniques for code injection vulnerabilities. Mitigation strategies can be categorized by the system level at which they operate:

- OS-level (Section 2.6.1).
- Engine-level (Section 2.6.2), encompassing both V8 modifications and browser security mechanisms;
- Application-level instrumentation (Section 2.6.3).
- Coding security practices (Section 2.6.4).

2.6.1 OS-Level Mitigations

Unlike browser-based JavaScript, Node.js code executes without a sandbox and has unrestricted access to the operating system. As a result, ACI and ACE vulnerabilities in Node.js can more easily escalate to full remote code execution [115]. Recent work has introduced OS-level mechanisms to protect Node.js applications against code injection attacks [11, 124].

Hodor [124] provides lightweight runtime protection by leveraging Linux kernel security mechanisms, specifically `seccomp` [112] system call filtering. It combines static and dynamic analyzes of both JavaScript (application) and C/C++ (Node.js framework) code to identify the minimal set of system calls required for benign execution. At runtime, Hodor enforces this set using `seccomp` filters, effectively whitelisting allowed system calls. This design yields negligible runtime overhead and a low false-positive rate, with most false positives stemming

[3] For example, @travist/async-shell or node-async-exec

from legitimate dynamic code generation. However, Hodor cannot fully eliminate the attack surface because certain system calls are used by the Node.js engine itself.

NatiSand [11] offers finer-grained control by additionally using newer Linux security primitives such as Landlock [28] and eBPF [20]. While these features allow the specification of more precise security policies, NatiSand requires the developers to provide a policy file specifying what resources the application is allowed to access. Furthermore, the requirements for an operating system supporting these kernel capabilities are another friction to the adoption of these systems.

2.6.2 JavaScript Engine-Level Mitigations

In contrast to OS-level mitigations, engine-level mitigations operate within the JavaScript runtime itself. This allows for finer-grained control over the behavior of executing code without relying on operating-system support for advanced security mechanisms.

COINDEF [134] exemplifies this approach in the context of Electron [98] applications. COINDEF operates in two phases: a *learning phase* and an *enforcement phase*. During the learning phase, COINDEF statically identifies the JavaScript code that should be protected and constructs a set of baseline AST profiles representing the legitimate, expected code structure. In the enforcement phase, it dynamically intercepts code at runtime and compares its AST against these precomputed profiles. If an executing script deviates from the expected structure, COINDEF blocks its execution. To accommodate benign dynamic behavior, COINDEF offers a configurable "usability-first" mode, which permits new code execution under strict sandboxing constraints. However, adopting COINDEF requires replacing the official runtime with a modified one, which limits its practical deployment.

Several mitigation mechanisms are now integrated into mainstream JavaScript runtimes. For instance, Deno [30] introduces a built-in permission system that enables developers to explicitly declare which resources (e.g., filesystem, network, environment variables) an application can access. Similarly, recent versions of Node.js include permission flags that provide comparable functionality [119]. In the browser context, modern engines implement a range of security mechanisms to mitigate code injection attacks such as XSS [128, 129], discussed in the following paragraphs.

Content Security Policy Content Security Policy (CSP) [128] is a web security standard that enables a website to instruct the browser about which resources can be loaded and how scripts may execute. CSP directives can, for example, prohibit inline event handlers or entirely disable the use of `eval`. Proper configuration of CSP is, however, the responsibility of web developers, and misconfigurations are common in practice [130].

Trusted Types Trusted types [129] is a security feature in browsers that aims to protect clients from DOM-XSS. It was first introduced in 2018, with initial support in Chromium 83. Now, it is also supported in Edge and, as of February 2025, in Firefox 135 under the `dom.security.trusted_types.enabled` flag. Trusted types, when enabled via a CSP directive, enforce the use of a trusted type policy in the creation of trusted type objects. Then, it ensures that only objects of the *trusted* type can

be passed to dangerous sinks. Trusted types are unfortunately not a complete solution to DOM-XSS, as they require developers to create policies, and their security is only as good as the policy that was defined.

2.6.3 Application-Level Mitigations

Application-level defenses mitigate code injection vulnerabilities by instrumenting the JavaScript application itself. As previously mentioned, Synode statically constructs a grammar that defines valid input structures and, at runtime, enforces this policy by blocking inputs that deviate from the grammar before they reach sensitive sinks. XGuard [133] generates a security policy that specifies which modules may invoke which APIs and with what data. A runtime monitor then tracks data provenance to ensure that sensitive operations conform to the predefined policy. Mininode [67] reduces the attack surface by removing unused code and dependencies.

2.6.4 Coding Security Practices

Many of the code injection vulnerabilities discovered in our study can be mitigated by enforcing proper sanitization of user input. In the Node.js ecosystem, developers can reduce the risk of ACI by using safer alternatives to command execution APIs. For instance, `execFile` does not invoke a shell by default [24] and explicitly separates the executable from its (potentially attacker-controlled) arguments [96]. Regarding ACE, packages should avoid dynamic code execution functions such as `eval`. The authors of Synode reported that approximately 80% of the examined uses of `eval` and `exec` could be easily refactored to eliminate these dangerous APIs altogether [115]. While the remaining `exec` usages could rely on third-party libraries to sanitize input, there is no standard sanitization method for `eval` [115].

Chapter 3

Improving Client Code Exploration for DOM-XSS Detection

In this chapter we study methods to explore client code and improve DOM-XSS detection. We develop novel approaches for client code exploration and study their effectiveness on a large scale experiment against real-world web pages.

The research described in this chapter was published at NDSS 2026, and we have open-sourced our end-to-end DOM-XSS detection tool SWIPE [106].

3.1 Overview

Prior work has measured the prevalence of DOM-XSS in the wild by analyzing web pages with modified browsers that implement dynamic taint analysis (DTA) [17, 71, 88]. However, the effectiveness of DTA for detecting code injection vulnerabilities is limited by how extensively we trigger alternative execution paths in the web application.

Existing client-side approaches fall short in this respect, as they overlook vulnerabilities with two critical features: (i) require explicit user interaction, and (ii) depend on parameters embedded in the target URL. To address this gap, we introduce an end-to-end DOM-XSS detection tool that we call SWIPE (Simulator of Webpage Interactions and Parameter Explorer), which leverages two complementary techniques: active page navigation and URL component synthesis.

Active navigation. Prior approaches for DOM-XSS detection rely on what we term *passive navigation*: the system visits a target URL, waits for the page to finish loading, and then closes the browser to process the results. While straightforward, passive navigation fails to discover vulnerabilities that require explicit user interaction before the relevant JavaScript code executes. To address this challenge, SWIPE integrates a fuzzer that actively executes event handlers by simulating user interactions on web pages (Section 3.2.5). This fuzzer substantially improves DOM-XSS detection compared to passive navigation: it increases the number of potential flows from 2023 to 2449, and confirmed flows from 72 to 83, when analyzing 44,480 real-world pages. Furthermore, our fuzzing approach is able to discover vulnerabilities in 34 domains that are not

detected by CrawlJax [89], an off-the-shelf tool that produces mouse clicks and keyboard events. These findings highlight the importance of simulating a wide range of user interactions when detecting DOM-XSS in the wild.

URL component synthesis. We find that failing to include URL components such as GET parameters and fragments (PFs) significantly hinders DOM-XSS detection effectiveness. Indeed, these PFs are not sufficiently explored in existing works that have measured the prevalence of DOM-XSS at scale. To overcome this limitation, SWIPE leverages a dynamic symbolic execution (DSE) tool [22] (Section 3.2.6) to synthesize URL elements referenced by client-side JavaScript code. When the original URL components are stripped from each target URL, DSE successfully rediscovers 26% of confirmed flows that depend on those components. Beyond rediscovery, DSE also identifies new vulnerabilities: applied to pages without known vulnerabilities, it discovers URL parameters that trigger 10 previously unseen confirmed DOM-XSS flows. Furthermore, we find that none of the PFs relevant for these 10 new confirmed flows are discovered by Wapiti [126], a web scanner that can discover GET parameters by parsing forms or links present on a target page.

Web archiving. In addition to leveraging fuzzing to simulate user actions and dynamic symbolic execution (DSE) to synthesize PFs, SWIPE takes steps to increase reproducibility and fairness in comparisons across detection approaches. This is achieved by leveraging web archiving approaches [39] to archive the page resources and replay them during analysis. While prior work has previously studied archiving for web security measurements, we address new reproducibility challenges in DOM-XSS detection, namely how web requests are modified based on user interactions or different PFs.

Contributions. To summarize, this chapter discusses the following contributions:

1. A systematic analysis of limitations in prior DOM-XSS detection techniques, focusing on the reliance on passive navigation and insufficient exploration of URL parameters and fragments.
2. The development of a novel fuzzer that simulates diverse user behaviors, uncovering vulnerabilities that depend on the execution of event handlers.
3. The integration of DSE for automated URL parameter synthesis, enabling exploration of additional program paths and consequently the detection of previously unseen vulnerabilities.
4. A discussion of the main challenges when comparing current DOM-XSS detection approaches with prior work, with special focus on how the evolution of the web and browsers impacts DOM-XSS detection. One key finding is that improved URL encoding mechanisms substantially impact exploitability of confirmed flows.
5. The design and implementation of a web archiving approach for fair and reproducible comparison across DOM-XSS detection approaches.

6. A large-scale evaluation across 44,480 URLs, offering empirical evidence of the effectiveness of SWIPE components and insights into the prevalence and characteristics of DOM-XSS vulnerabilities in the wild.

3.2 SWIPE Architecture

This section presents the architecture of SWIPE and its main components. We begin by describing the end-to-end SWIPE execution workflow for a target URL and proceed to detail each SWIPE component, namely the modified browser implementing DTA, the fuzzing and DSE engines and the web archiving mechanism.

3.2.1 Execution Modes

SWIPE operates in one of three modes, which we refer to as analysis *conditions*:

- **Passive**: Replicates prior work. The browser navigates to the target page and remains idle until the allocated time budget expires.
- **Fuzzer**: After the page loads, SWIPE activates the fuzzing module (Section 3.2.5) to simulate user interactions with the page.
- **DSE**: Executes dynamic symbolic execution (Section 3.2.6) against the target page.

SWIPE can be launched against a list of pages, specified as a set of URLs. Each target URL is analyzed independently inside a dedicated Docker container, with one container per page and per analysis condition.

3.2.2 Workflow Overview

Figure 3.1 illustrates the end-to-end execution of SWIPE for a target URL. SWIPE's execution begins by launching an instance of our modified Chromium browser (Section 3.2.3), that navigates to the target URL (step ①). Chromium is configured to route requests through a proxy that mediates all resource fetches (e.g., JavaScript, HTML, CSS) between the browser and the live page (step ②).

The proxy serves two purposes: (i). It enables reproducible analysis by archiving resources (Section 3.2.7). (ii). It can optionally instrument resources to support DSE. If a requested resource has not yet been archived, the proxy retrieves it from the live web page (steps ③-⑤). Otherwise, the resource is loaded from the archive.

Since SWIPE's behavior depends on the analysis condition, Figure 3.1 distinguishes these differences across conditions using different letters for arrow labels:

- **Passive** (a): The proxy archives and serves unmodified resources (steps ⑤a, ⑥a), after which the browser remains idle until the time budget is exhausted (step ⑦a).
- **Fuzzer** (b): Similar to Passive, resources are archived and served unmodified (steps ⑤b, ⑥b), but then the fuzzer component is launched to simulate user actions (step ⑦b, Section 3.2.5).

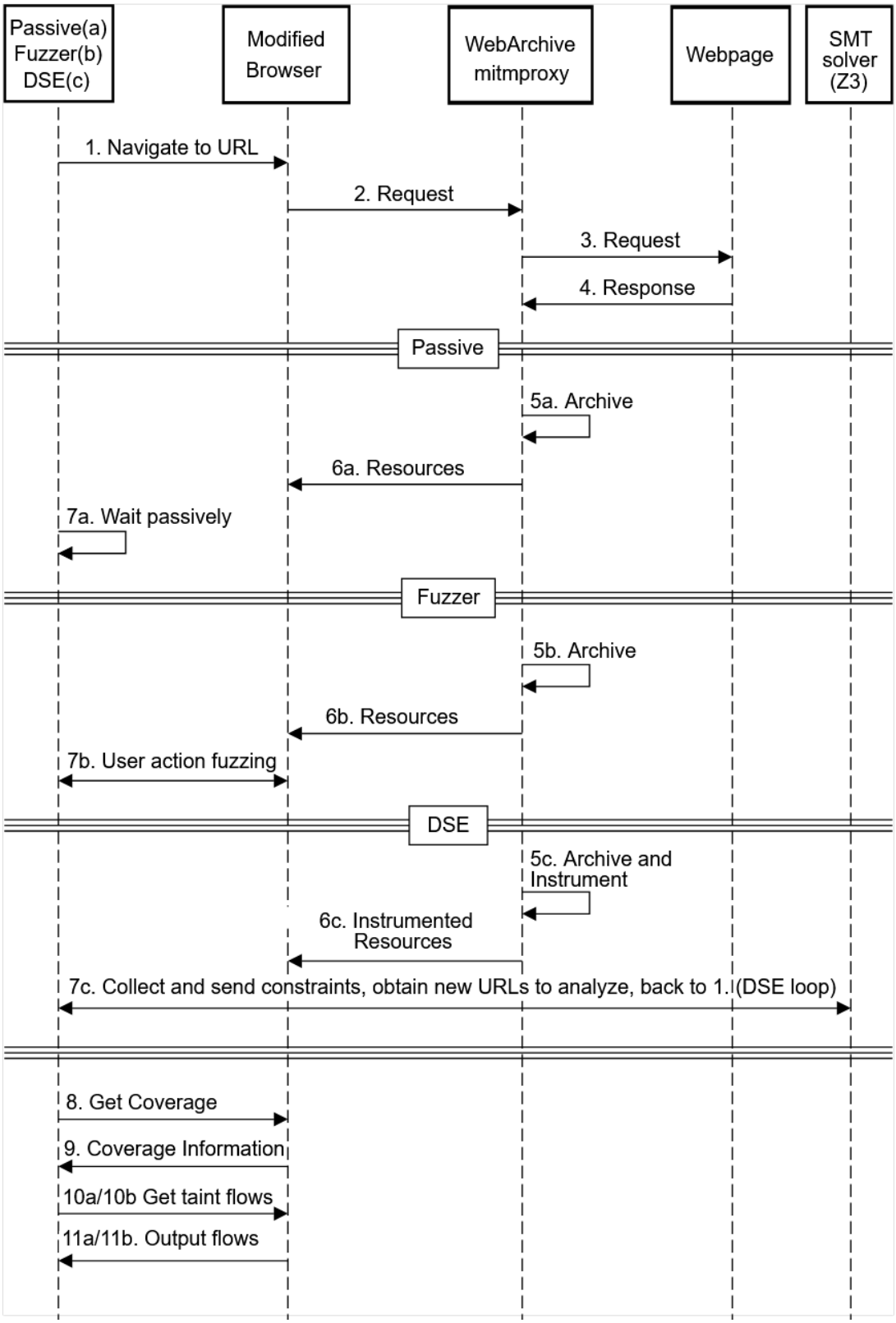

Figure 3.1: Core proxy and browser interactions in an end-to-end SWIPE workflow for a single page URL. Some workflow steps depend on the SWIPE mode that is configured. Some network requests and responses between the browser and the WebArchive are expected to occur during analysis steps (*7a*, *7b* or *7c*), even though the diagram only describes the initial requests.

- **DSE** (c): The proxy archives and *instruments* HTML and JavaScript files with logic for collecting the constraints on URL variables (step (5c)). These instrumented resources are then served to Chromium ((6c)). Once constraints are collected and solved, new variant URLs are generated, and the system returns to step (1) for analysis of those synthesized URLs. We refer to this iterative process as the *DSE loop*, described in detail in Section 3.2.6.

Execution under any condition terminates once the time budget is exhausted. At that point, final JavaScript code coverage is collected (steps (8), (9)). There are additional analysis outputs that differ across conditions: DSE returns a list of synthesized URLs, whereas Passive and Fuzzer return a list of flows observed by the modified Chromium, in steps (10a),(11a) and (10b),(11b) respectively.

3.2.3 Flow Collection

DOMsday [88] introduced a Chromium browser (based on Chromium version 54, released in 2016) with a V8 engine modified to track taint propagation across strings at runtime. SWIPE uses an updated version of DOMsday's browser, based on Chromium 126.0.6478.264, released February 7th, 2025. This instrumented browser dynamically records taint flows as JavaScript executes.

A flow can be seen as a structured object with the following fields:

- **source**: The origin of attacker-controlled input (e.g., the full URL).
- **sink**: The sensitive function or operation invoked (e.g., `eval`).
- **argument**: The final string argument passed to the sink containing at least 1 byte (character) of attacker-controlled data.
- **encoding**: Whether the source data was URL-encoded, either by Chromium's builtin URL-encoding mechanism or by the target application.
- **frame**: The URL of the frame containing the potentially vulnerable script, or containing the sink call itself in the case of an inline script.
- **script**: The URL of the script that contains the sink call. For convenience, in case the script is inline this will be the same URL as in the `frame` field.
- **location**: The position of the sink call within the script, given as a (line, column) pair. For inline scripts, this reduces to a column offset.

Whenever a tainted value reaches a sensitive sink, the modified browser logs a flow instance with the above attributes. After analyzing a target URL, all recorded flows are collected and aggregated in a database.

It should be noted that the raw count of flows does not directly correspond to the number of distinct vulnerabilities, since multiple flows may arise from repeated executions of the same vulnerable program path. To avoid overcounting, we apply deduplication following 25mFlows methodology [71]. Each flow is reduced to a *deduplication tuple*:

```
(flow.source, flow.sink, domain(flow.frame), stripPFs(flow.script), flow.location)
```

where `domain` extracts the second-level domain of the frame URL, and `stripPFs` removes the query parameters and fragment from the script URL.

Two flows are considered distinct if their tuples differ. We may also use the term *unique flows* to specify a set of flows that all have distinct deduplication tuples. We will discuss in Section 3.3.2 how this approach is imperfect and may occasionally classify flows referring to the same vulnerability as distinct, but we consider improvements on the deduplication methodology to be out of scope for this work.

Tables 3.1 and 3.2 summarize the supported attacker-controlled sources and sensitive sinks, which are actually the same as the ones supported by DOMsday [88]. Sources represent values plausibly influenced by an attacker, such as the full URL accessed via `document.location`. Sinks represent security-sensitive operations that must not be attacker-controlled. These include JavaScript primitives such as `eval` or `new Function`, HTML insertion functions such as `document.write`, and assignments to DOM properties like `innerHTML` or to event handler properties such as `onclick`.

Source	Description / JavaScript property name
Cookie	`document.cookie`
Message	`postMessage` event `data`
Message Origin	`postMessage` event `origin`
Referrer	`document.referrer`
Storage	`Storage.getItem`
URL	`location, location.href, document.URL`
	`document.documentURI, document.baseURI`
URL hash	`location.hash`
URL host	`location.host`
URL hostname	`location.hostname`
URL origin	`location.origin`
URL pathname	`location.pathname`
URL search	`location.search`
URL port	`location.port`
URL protocol	`location.protocol`
window.name	`window.name`

Table 3.1: Attacker-controlled sources supported by SWIPE.

3.2.4 Flow Confirmation

This section describes how we confirm the exploitability of potential flows. We validate each potential flow individually, following prior work methodology [17, 71, 88].

As outlined in Section 2.5.4, prior work confirms a potential flow by identifying a URL location suitable for payload injection and then inserting a payload at that location. The details of this process vary across studies, particularly in how the injection position is calculated. TalkGen [17] demonstrated that while its placement algorithm outperformed prior work, each methodology uniquely confirms flows that others miss. Thus, for each potential flow, we compute not one, but three different URLs, one for each injection position algorithm of the three relevant prior works.

Sink	Description
HTML	`document.write`, `document.writeln` function call. Assignment to `innerHTML`, `outerHTML`, `insertAdjacentHTML`.
JavaScript	`eval`, `Function` function call.
Event Handler	Assignment to any event handler attribute, e.g., `onclick`
Anchor src	Assignment to `<a>` element `href` property.
Cookie	Assignment to `document.cookie`.
Css	Assignment to `element.style`.
Embed src	Assignment to `<embed>` element `src` property.
Iframe src	Assignment to `<iframe>` element `src` property.
IMG src	Assignment to `<img>` element `src` property.
setTimeout/setInterval	`setTimeout`,`setInterval` function call.
Location	Assignment to the `location` object.
Script src	Assignment to `<script>` element `src` property.

Table 3.2: Sensitive sinks supported by SWIPE. The descriptions illustrate common cases but do not exhaustively enumerate all possible flow-generating operations.

With regards to the specific payload construction algorithm, we use DOMsday's methodology since it is open-source, which consists of simply injecting the substring `marker<>'"` in the appropriate location.

We navigate to each resulting URL containing a marker, and validate whether that marker appears unencoded on an argument to a DOM-XSS sink. This can be done by iterating over the flows found during confirmation and inspecting the `argument` and `encoding` properties of these flows. If the unencoded marker is there, we call the respective flow (found during confirmation) a *confirmed flow* and the original potential flow an *exploitable flow*.

3.2.5 Fuzzing User Interactions

Event handlers are used to define page behavior in response to user interactions. In this section, we describe our fuzzer which automatically triggers event handlers by simulating realistic user actions.

Web scanners such as CrawlJax [89] and jÄk [104] fire most of their supported event handlers programmatically via JavaScript. However, forcefully executing event handlers can lead to false positives, e.g., vulnerabilities that appear exploitable but cannot be reached by real users. This can happen for example when a triggered event handler is bound to an invisible DOM element. Our fuzzer instead focuses on discovering vulnerabilities reproducible through realistic user interaction.

Motivating example for fuzzing

```
1  function renderResult(){
2      var search = location.search;
3      query = new RegExp(/[?&]q=([^&]*)/).exec(search);
4      query = decodeURIComponent(query[1]);
5      ...
6      var b = document.createElement("a");
7      b.href = query;
8      ...
9      element.appendChild(a)
10 }
11
12 document.addEventListener("click", (function(e) {
13     var a = e.target.closest(PAGE_LINK_SELECTOR);
14     a && renderResult()
15 }));
```

Figure 3.2: Simplified vulnerable code found in the wild.

Figure 3.2 shows an anonymized code sample collected during our crawl of real websites.[1] The function `renderResult` (lines 1–10) is vulnerable to DOM-XSS. It extracts the `q` query parameter (line 3), decodes it (line 4), and assigns the resulting value to the `href` attribute of a new HTML link element (lines 6–7). The element is then appended to the DOM (line 9). Because the `q` parameter can be controlled by an attacker, the function enables injection of attacker-controlled URLs. To leverage this vulnerability and demonstrate code execution capabilities, an attacker could supply `q=javascript:alert()` in the GET parameters. Once `renderResult` is invoked and the created link clicked, the attacker's code is executed, opening an `alert` window.

Crucially, `renderResult` is never invoked during page load. It is only executed when the `onclick` event handler defined in lines 12–14 is triggered. Using passive navigation with DTA would therefore miss this vulnerability. Our fuzzer addresses this gap by detecting vulnerabilities hidden behind event-driven logic.

Collecting supported event handlers

Once a page has fully loaded, the fuzzer inspects the DOM to retrieve all registered event handlers together with the elements to which they are bound. This inspection is performed by injecting and executing a JavaScript function in every frame of the page, shown in Figure 3.3.

Lines 2–4 collect all DOM elements accessible within the frame, including both the `document` and `window` objects. The code block in lines 6–27 then iterates over these elements and extracts their associated event handlers. Specifically, for each element, line 7 invokes Chromium's built-in `getEventListeners`, which returns an object where each key corresponds to an event type (e.g., `click`), and each value is the list of functions triggered by that event. Lines 12–19 iterate over the extracted event types, obtain the exact code of each registered function, and append these to a list of handlers for the corresponding element. Finally, lines 21–24 aggregate these element-event list mappings into the overall `result` object returned by the injected function.

After this step, the fuzzer retains only those handlers it can later trigger, discarding unsupported ones. In total, the fuzzer supports 55 event handlers, which we list in Table 3.3.

[1] We anonymize all examples following the process described in Chapter 2 to avoid disclosing unpatched vulnerabilities.

```
function collect_events() {
  const all_els = Array.prototype.slice.call(document.querySelectorAll('*'));
  all_els.push(document);
  all_els.push(window);

  return all_els.reduce((result, element) => {
      const event_listeners = getEventListeners(element);
      const event_types = Object.keys(event_listeners);
      if (event_types.length !== 0) {
          events = {};

          event_types.forEach((eventType) => {
              events[eventType] = event_listeners[eventType].reduce(
                (ev, eventListener) => {
                  ev.push(eventListener.listener.toString());
                  return ev;
                }, []
              );
          });

          result.push({
              node: element,
              events: events,
          });
      }
      return result;
  }, []);
}
```

Figure 3.3: JavaScript code executed by the fuzzer after page load to collect event handlers from each frame.

Supported Events				
onclick	onmousemove	onscroll	onchange	onmouseleave
ontouchcancel	ondblclick	onmouseout	onhashchange	onfocus
oncontextmenu	onfullscreenchange	onfocusin	ontoggle	onpause
onfocusout	onsearch	onplay	onblur	onsubmit
onseeking	oncopy	onreset	onseeked	oncut
onstorage	oncanplay	onpaste	ondrag	ontimeupdate
onselect	ondragstart	onplaying	oninput	ondragend
oncanplaythrough	onkeydown	onresize	onwaiting	onkeyup
ontouchstart	onafterprint	onkeypress	ontouchend	onbeforeprint
onmousedown	onpopstate	onmessage	onmouseup	oninvalid
onfullscreenerror	onmouseenter	onmousewheel	onmouseover	onwheel

Table 3.3: Event handlers supported by the fuzzer.

Additionally, Table 3.4 describes event handlers that are not supported by our fuzzer. The reason for not supporting events falls into several categories:

1. Events that cannot be fully controlled by the user (12 total) (e.g., `onstalled` is triggered by network issues).
2. Events that would terminate the analysis because triggering them implies leaving the page (4 total).
3. Events that require a complex chain of interactions (6 total). For instance, there are 4 CSS animation events that may depend on non-trivial style transitions. It is hard to reliably determine what interactions are needed to initiate a CSS animation.
4. Drag & Drop events (4 total) blocked by a Chromium DevTools Protocol (CDP) bug.[2]
5. A touch event (`ontouchmove`) that even though it seems possible to trigger using the `touchMove` Chrome DevTools Protocol (CDP) API, it is not working properly for us in Chromium 126.
6. Two events that have no matching API in CDP that we could use.
7. One event (`onshow`) not supported by Chromium at all.

During fuzzing, unsupported event handlers can still be executed, for example, when they are programmatically invoked by the page's JavaScript code. Still, we consider an event handler unsupported if the fuzzer lacks a mapping from that handler to a sequence of high-level actions that would trigger it.

Generating high-level actions

The fuzzer operates on a set of primitive operations, which we denote as High-level actions (HLA). An example is `ClickElement`, which clicks on a specific DOM element. This HLA is implemented via CDP by obtaining the element's coordinates, moving the mouse, and pressing and releasing the left button.

[2] https://issues.chromium.org/issues/40579554

Event Name	Supported	Reason
ondurationchange	No	No user control
onprogress	No	No user control
onstalled	No	No user control
onsuspend	No	No user control
onloadstart	No	No user control
onload	No	No user control
onloadeddata	No	No user control
onloadedmetadata	No	No user control
onabort	No	No user control
onopen	No	No user control
onended	No	No user control
onerror	No	No user control
onbeforeunload	No	Terminates analysis
onunload	No	Terminates analysis
onpagehide	No	Terminates analysis
onpageshow	No	Terminates analysis
ontransitionend	No	Complex interaction
onanimationend	No	Complex interaction
onanimationiteration	No	Complex interaction
onanimationstart	No	Complex interaction
onratechange	No	Complex interaction
onvolumechange	No	Complex interaction
ondrop	No	Drag & Drop bug in CDP
ondragenter	No	Drag & Drop bug in CDP
ondragleave	No	Drag & Drop bug in CDP
ondragover	No	Drag & Drop bug in CDP
ontouchmove	No	Bug in CDP
onoffline	No	CDP insufficient
ononline	No	CDP insufficient
onshow	No	Unsupported in Chrome

Table 3.4: Event handlers not supported by the Fuzzer and for what reason

The fuzzer constructs an initial action pool by mapping supported event handlers to sets of triggering HLAs.

For instance, when encountering an `onclick` handler, it adds the composite action:

```
SequenceActions(MakeVisible(element), ClickElement(element))
```

Here, `MakeVisible` scrolls the page to ensure the element is visible. Without it, the `ClickElement` action could try to move the mouse to coordinates that are not in the visible area, deviating from what users can realistically do.

Coverage guided fuzzer

After initializing the action pool, the fuzzer enters a coverage-guided refinement loop. Each iteration executes, mutates, and prunes candidate actions based on the coverage they achieve. This process, summarized as pseudocode in Figure 3.4, consists of the following steps:

- Line 2 establishes a baseline coverage measurement.
- Lines 4–10 execute each high-level action on the current pool and compute its fitness score based on the new coverage achieved.
- Lines 12–13 retain the top `MAX_POPULATION` actions, i.e., those yielding the most coverage.
- Lines 16–27 optionally apply crossover techniques [136]. For each action, the fuzzer selects `CROSS_OVERS_PER_ELEMENT` peers and exchanges sub-actions between their composite HLA. This diversifies execution sequences and enables exploration of event handlers in alternative orders. A simplified variant of the fuzzer can be obtained by disabling this step with `config.DO_ACTION_COMBINATIONS = false`.

The refinement loop is repeated until the allotted analysis budget is consumed.

During fuzzing, we prevent the fuzzer from leaving the page by assigning a value to the `window.onbeforeunload`. This triggers a dialog whenever navigation is attempted. The fuzzer automatically closes the dialog and cancels the navigation, resuming analysis as normal.

Replaying fuzzer actions to confirm vulnerabilities

For reproducibility, each fuzzer run sets a fixed seed. This seeds the PRNG that is used by all randomized HLAs. When confirming potential vulnerabilities, the fuzzer replays the exact action sequence to reproduce the discovered execution flow.

3.2.6 Using DSE to Find PFs

A URL consists of multiple components. Some components identify the location of the server handling requests, while others encode additional data. The latter category primarily includes GET parameters and fragment values (collectively referred to as PFs). These elements can be parsed by client JavaScript code and directly influence its execution. As a result, certain execution paths may only become reachable if specific GET parameter keys and values, or fragment values, are present.

This thesis investigates whether such PFs can be automatically synthesized using a dynamic symbolic execution (DSE) tool for JavaScript, originally introduced in the work of Darion Cas-

```
function do_fuzz_step(pool){
    global.coverage = get_coverage();

    new_pool_with_fitness = [];
    for (hla in pool){
      hla.execute();
      coverage = get_coverage();
      fitness = compute_fitness_value(global.coverage, coverage);
      new_pool_with_fitness.append((fitness, hla));
    }

    sort(new_pool_with_fitness);
    new_pool_with_fitness = new_pool_with_fitness[:MAX_POPULATION];


    new_pool = [];
    for ((fitness, hla) in new_pool_with_fitness){
      if (config.DO_ACTION_COMBINATIONS){
        for (i in range(CROSS_OVERS_PER_ELEMENT)){
          hla2 = new_pool_with_fitness.get_different_elem(hla);
          new_pool.append(cross_over(hla, hla2));
        }
      }
      else{
        new_pool.append(hla);
      }
    }

    return new_pool;
}
```

Figure 3.4: Pseudocode for the fuzzing algorithm. This algorithm is repeatedly executed, mutating the pool of actions until the time budget is exhausted. The first pool is assumed to have already been constructed as previously discussed in this section.

```
1  function getJsonFromUrl() {
2    var result = {};
3    var query = location.search.substr(1);
4    query.split("&").forEach(function (part) {
5      var item = part.split("=");
6      result[item[0]] = decodeURIComponent(item[1]); });
7    return result; }
8  ...
9  var paramsMap = {'custom1':'GP1', 'custom2':'GP2', ...};
10 function buildUrl(baseUrl) {
11   var urlParams = getJsonFromUrl();
12   for (var key in paramsMap) {
13     if (urlParams[key] != undefined) {
14       baseUrl += '&';
15       baseUrl += paramsMap[key] + "=" + urlParams[key];
16     }}
17   return baseUrl;
18 ...
19 var post = buildUrl(basePost);
20 document.write('<img src="' + post + '"/>');
```

Figure 3.5: Vulnerable code requiring specific URL parameters

sel [22]. The remainder of this section motivates the use of DSE for parameter synthesis, outlines the methodology of DSE, and describes the differences between our SWIPE-DSE component and the original DSE implementation.

Motivating example for DSE

A real-world vulnerable case is shown in Figure 3.5, where the execution of a vulnerable page script depends on the presence of a specific key/value pair in its GET parameters. In this example, the script constructs a `urlParams` object (line 11) from attacker-controlled GET parameter keys and their URL-decoded values (lines 1-7). The code then selects only the GET parameter keys that are also in `paramsMap` (line 9) and assembles the selected key-value pairs into a new URL (lines 12-15). Finally, this URL — whose contents ultimately depend on attacker-controlled values in `urlParams` — is injected into the `src` attribute of a newly created `img` element (line 20). Identifying this vulnerability requires generating a URL that contains one of the permitted GET parameters in `urlParams`, such as `custom1`. Different web applications impose different constraints on PFs that must be satisfied to reach a vulnerable program path. We employ DSE [22] to systematically explore client code execution and synthesize PFs that trigger those previously unreachable code paths.

DSE loop

An overview of the DSE execution process is shown in Figure 3.1. Given a URL of the form `protocol://domain.tld/path/?search#fragment`, Chromium loads the DSE-instrumented page and treats `location.search` (query parameters) and `location.hash` (fragment) as concolic values. A concolic value represents both a symbolic and a concrete variable. During execution, DSE records constraints arising from operations that are performed on these values.

```
1  function binary(op, lhs, rhs, res) {
2    if (isConcolic(lhs) || isConcolic(rhs)) {
3      res = handle(op, lhs, rhs);
4    }
5  }
6
7  function handle(op, lhs, rhs) {
8    if (op == "+") {
9      sendConstraints("str.++", lhs.smt(), rhs.smt());
10     return lhs.concrete + rhs.concrete;
11   }
12   ...
13 }
```

Figure 3.6: DSE instrumentation for string concatenation.

The collected constraints are translated into an SMT formula and solved with Z3. If satisfiable, the solver produces concrete assignments for the symbolic components. DSE then synthesizes a new URL with updated query and fragment values and enqueues it for further exploration. If the formula is unsatisfiable, no new URL is enqueued.

This process, termed the *DSE loop*, continues until the queue is empty or the time budget for the page expires. Note that the same URL is never analyzed twice during the DSE loop.

DSE instrumentation and concrete models

Lines 15 and 20 of Figure 3.5 illustrate binary operations performed on symbolic variables. Since such operations are common for string values, we show in Figure 3.6 an example of DSE instrumentation for generating these constraints. Line 1 defines the instrumented function `binary`, which takes as input the operation (`op`), the left-hand side (`lhs`), the right-hand side (`rhs`), and a placeholder for the result (`res`). Line 2 checks whether either operand is concolic, i.e., a value derived from URL parameters or fragment values that also has an associated symbolic representation. If so, line 3 calls the `handle` function to process the symbolic operation. As partially defined in lines 7-12, `handle` generates the corresponding SMT constraints that model the operation's symbolic effect, and we illustrate the specific case of string concatenation (`op == "+"`). DSE then issues a `str.++` constraint to the solver via `sendConstraints` (line 9), using the SMT representations of the operands. The SMT representation of an operand is obtained through its `operand.smt()` method. For instance, calling `smt` on the `location.search` property yields the symbolic variable name associated with the GET parameters. Finally, line 10 evaluates the operation under a *concrete model*, which preserves the exact JavaScript semantics using the concrete components of the concolic values. This ensures that the page's JavaScript execution proceeds normally with a valid concrete value, even when dealing with concolic inputs.

Differences between SWIPE-DSE and the original DSE

SWIPE-DSE builds directly on the methodology of the original DSE [22]. Beyond bug fixes, it introduces the following extensions:

- A symbolic model for `setField` when the offset is concolic.

- The ability to turn any variable into a concolic variable.

3.2.7 Web Archiving

This section describes our web archiving component, designed to improve fairness and reproducibility.

Sources of non-determinism

A major challenge in this work is handling the inherently dynamic nature of the web, which significantly complicates the fair comparison of different DOM-XSS detection approaches. Repeated requests to the same page often yield different responses, even if with only subtle differences.

Web pages often exhibit nondeterministic behavior across multiple navigations of the same page, including the loading of different advertisements, assignment of random DOM element identifiers, and dynamic script generation based on variables such as the current timestamp. Figure 3.7 illustrates this issue with code extracted from one of the vulnerable pages that we found in our crawl. Here, an array of videos is defined (line 1); a video is then randomly selected using the `Math.random` API (line 2); and finally, the chosen video is injected into the DOM (line 3). In summary, nondeterministic factors can influence which resources are loaded and what JavaScript code is executed.

DOM-XSS-specific challenges for web archiving

Prior work has demonstrated that website archiving can enhance reproducibility in security analysis [39, 51]. However, our setting introduces challenges that standard archiving approaches do not adequately handle.

DOM-XSS vulnerabilities often involve complex client-side JavaScript execution, where resources are fetched or requests are modified dynamically based on user interactions or URL components. In our case, the fuzzing engine actively interacts with the page, triggering additional resource loads beyond those observed during a passive navigation. Additionally, the DSE engine performs repeated navigations to the same page while varying GET parameters and hash values. These URL components frequently embed timestamps or unique identifiers, which can influence what resources are requested.

To address these challenges, we integrate our dynamic analysis with a dedicated archiving and replay system. Our solution involves a novel online archive expansion mechanism that reduces missed resources and increases repeatability.

Web archiving mechanism

We now describe the design of our archiving mechanism. SWIPE components use a Chromium instance configured to route traffic through a proxy. The web archiving mechanism is implemented as an add-on for mitmproxy [27], which enables the interception of browser requests and manipulation over the responses delivered to the browser.

Archive creation. On the first visit to a page, SWIPE records all browser requests and corresponding server responses, including HTML, JavaScript, and CSS resources. These resources are stored collectively in a compressed form[3], which we call *web archive*.

Handling redirections. A browser request for a URL may result in a redirection to another URL. Without the web archive, the final URL that is served may differ across navigations. SWIPE's archiving mechanism handles redirections robustly by maintaining a mapping from requested URLs to their final redirected targets. Once a request to URL_1 is observed to redirect to URL_2, the mapping is updated accordingly.

Archive replay. Once a page has been archived, subsequent analyzes (by the same or another SWIPE component) are served from the archive rather than the live server. Upon each request, the system attempts to retrieve the corresponding stored response. This step is not always straightforward, as current requests may differ from those made during archive creation. Differences in requests may, for example, be introduced by different values of GET parameters, which can often include timestamps or unique identifiers. Following Goel et al. [39], we compute the similarity between current request URLs and stored request URLs. When a similar URL is found, the corresponding archived response is served.

Similarity metric. Given a target URL, URL_1, the web archive component selects the most similar URL, URL_2, from the set of archived URLs *archive*, such that the following 3 conditions are satisfied:

$$URL_2 \in archive, \tag{3.1}$$

$$match(URL_2, URL_1) \text{ holds}, \tag{3.2}$$

$$\forall URL_3 \in archive,\ match(URL_3, URL_1) \Rightarrow editDist(URL_3, URL_1) \geq editDist(URL_2, URL_1). \tag{3.3}$$

The predicate *match* returns `True` if the two input URLs share the same eTLD+1 (effective top-level domain plus one additional domain label). The function *editDist* corresponds to the Levenshtein distance [73] between the given string URLs.

Archive expansion during replay. If no similar request exists in the archive, we employ a novel online archive expansion phase. The request is forwarded to the live server, and the resulting response is added to the archive. This approach avoids serving incorrect responses or failing with a 404. This expansion strategy introduces a tradeoff between repeatability and realistic preservation of page behavior, and it is particularly important for our dynamic analysis, as both fuzzing and DSE can generate requests that would not occur during passive browsing.

[3] We use the WARC format [7]

Archive Exclusion During Confirmation. Prior archiving solutions cannot always perfectly reproduce live page behavior [14, 39, 51], and our mechanism does not solve that issue either. We extend on this and other limitations of our web archiving mechanism in Section 3.4.1. For vulnerability confirmation, we disable the archive entirely to ensure that detected flows represent genuine page behavior.

```
1 var vlst = ['vid_01', 'vid_02','vid_03', 'vid_04', ...];
2 var vid = vlst[Math.floor(Math.random() * vlst.length)];
3 document.write('<video poster="http://vulnerable/' + vid + '.jpg">');
```

Figure 3.7: Randomized page behavior observed in a real page from our crawl.

3.3 Evaluation

We evaluate the effectiveness of our dynamic analysis and answer the following research questions:

RQ1: Can Fuzzer generate user interactions in real-world pages that lead to the discovery of new DOM-XSS vulnerabilities (Section 3.3.2)?

RQ2: Can DSE uncover PFs in real-world pages, and how do they impact DOM-XSS detection (Section 3.3.3)?

RQ3: How does SWIPE compare to other end-to-end DOM-XSS detection tools (Section 3.3.4)?

RQ4: How do SWIPE's analysis results compare to what prior work reported, and how does the continuous evolution of the web affect the validity and consistency of such comparisons (Section 3.3.5)?

3.3.1 Experimental Setup

Setup overview

Figure 3.8 shows our experimental workflow, including our main experiments and results. In the last row, for comparison, we have included TalkGen's workflow and reported results.

Our experimental pipeline consists of four stages:

- **Dataset collection**: We collect 44,480 URLs from popular domains, in order to evaluate SWIPE and to compare with prior work.
- **Dataset augmentation**: Given a target URL, DSE synthesizes new URLs by generating additional parameters or fragments. In this stage, we apply this process to all URLs of a given input dataset.
- **Analysis**: A DOM-XSS detection tool analyzes URLs from a given dataset, producing a set of potential flows.
- **Confirmation**: We confirm exploitability of each potential flow.

The following sections describe each stage in detail.

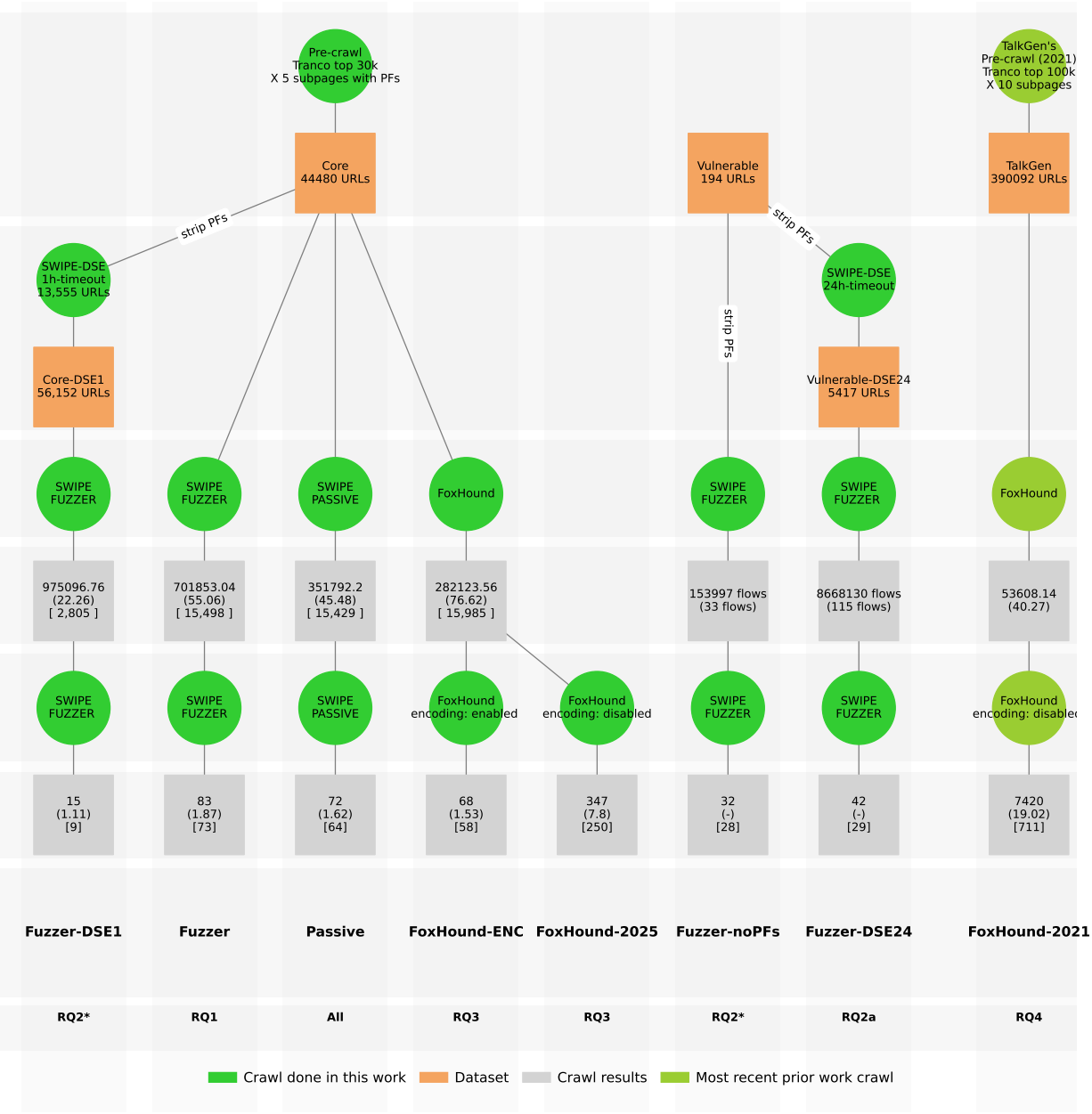

Figure 3.8: Our crawl pipeline, main results and comparison with TalkGen's crawl. Pipeline stages are marked by horizontal stripes, starting from dataset collection, DSE dataset augmentation, analysis crawls, analysis results (in terms of flows/1k URLs, unique potential flows/1k URLs in round brackets and loaded frame domains in square brackets), confirmation crawls and confirmation results. For each crawl, we give its name and which research question it helps answer.

Dataset collection stage

We first downloaded the top 30,000 domains from the Tranco list [70]. All domains were prefixed with `https://` and crawled using a script based on the crawling library Scrapy [68]. From each page, we extracted up to five links containing non-empty GET parameters or fragment values, and pointing to one of the 30,000 original domains. Pages that timed out after 3 minutes were discarded.

This produced a dataset of 44,480 URLs, including 13,396 top-level Tranco pages (i.e., reachable domains with the `https` prefix), and 31,084 subpages with PFs. We denote this dataset as the *Core dataset*.

Across all our experiments, 194 pages from the Core dataset were confirmed to contain DOM-XSS vulnerabilities. We denote this subset as the *Vulnerable dataset*.

The presence of 31,084 pages with PFs provides a baseline for DSE: by stripping known PFs from URLs and analyzing the resulting URLs with DSE, we test whether DSE can rediscover the original PFs. Removing PFs from Core and Vulnerable yields the *Core-noPFs* and *Vulnerable-noPFs* datasets respectively. Note that some of the PFs in URLs from the Core dataset are bound to only be processed server-side and therefore cannot be recovered by symbolic execution of client-side JavaScript.

Dataset augmentation stage

DSE synthesizes new URLs with additional GET parameters or fragments. By visiting these URLs, previously unobserved page behaviors may emerge, potentially exposing new vulnerabilities.

In this work, we refer to the process of synthesizing additional URLs from a given input dataset as *dataset augmentation*. Augmented datasets are analyzed with Fuzzer, and differences in vulnerability discovery between the original and the augmented datasets measure the contribution of DSE (**RQ2**).

In our experiments we have augmented two datasets:

- **Core-noPFs**: DSE was run with a 1-hour timeout per page. We allocated 100 total hours for this experiment, during which DSE analyzed 13,555 of the 44,480 URLs, producing 56,152 synthesized URLs, which we denote as the *Core-DSE1 dataset*. Analysing the results of fuzzing pages from the Core-DSE1 dataset helped us answer whether DSE can find new confirmed flows on pages in the wild.
- **Vulnerable-noPFs**: DSE was run with a 24-hour timeout per page, across all 194 pages. This produced 5,417 new URLs, forming the *Vulnerable-DSE24 dataset*. Analysing the results of fuzzing pages from the Vulnerable-DSE24 dataset helped us answer whether DSE can rediscover confirmed flows in pages that we already know are vulnerable. Since Vulnerable-noPFs is a much smaller dataset compared to Core-noPFs, we were able to allocate a much higher timeout (24X) per page for this second DSE augmentation.

Both augmentations start from datasets in which all PFs are removed: Core-noPFs and Vulnerable-noPFs. The prior stripping step is essential before augmenting Vulnerable-noPFs, as the objective of this augmentation is to validate whether DSE rediscovers known vulnerabilities. For Core-noPFs, removing PFs may reduce DSE's effectiveness, since the original Core

dataset includes valid PFs that could have been used as a starting point for DSE. We nevertheless enforce stripping for two reasons. First, it allows a consistent evaluation methodology for DSE, regardless of the dataset being augmented. Second, it generates valuable diagnostic data for future improvement: by comparing PFs in synthesized URLs against the baseline PFs from the original Core dataset, we can easily identify failure cases where DSE does not succeed (after filtering out PFs that are only processed server-side). Such data would not be so directly available if augmentation was performed directly on the unstripped Core dataset.

Analysis and confirmation stages

When we launch a DOM-XSS detection tool like SWIPE against pages of a given dataset, we aim to first obtain a set of potential flows, collected using a modified browser that implements dynamic taint analysis. All SWIPE components use the web archive component during analysis, and an archive is created the first time each page is visited with a SWIPE component. Multiple pages can be analyzed in parallel, and we prevented race conditions in the archive usage by ensuring that each page is analyzed by only one condition at a time.

Each potential flow is validated as described in Section 3.2.4, a method mostly based in DOMsday's confirmation methodology. We apply this confirmation methodology consistently to all conditions. The number of confirmed flows provides an approximation of true vulnerabilities. While DOMsday sampled 40 confirmed flows and manually validated that their confirmation methodology did not produce false positives, we also manually reviewed 10 of our confirmed flows and successfully exploited all 10 by creating a payload that spawned an alert window. Additionally, during the confirmation stage, the web archive is not used by any condition to ensure that any discovered confirmed flows are not caused by the usage of the web archive.

We reported all confirmed flows, even those that we did not attempt to manually exploit. Responsible disclosure was performed following Khodayari et al. [63] methodology. We received one reply so far, which acknowledged the vulnerability and committed to patch it.

Experiments performed

We evaluated seven main conditions:

1. **Passive**: Passive navigation, replicating DOMsday [88], on the Core dataset. Uncovered 72 confirmed flows hosted in pages from 64 distinct domains.
2. **Fuzzer**: Uses fuzzer to simulate user interactions on the Core dataset. Uncovered 83 confirmed flows in 73 domains.
3. **Fuzzer-noPFs**: Our fuzzer on the Vulnerable-noPFs dataset. Uncovered 32 confirmed flows in 28 domains. To clarify, this is the exact same fuzzer method as above, but applied on distinct datasets.
4. **Fuzzer-DSE24**: Our fuzzer on the Vulnerable-DSE24 dataset (i.e., the DSE augmented version of Vulnerable-noPFs). Uncovered 42 confirmed flows in 29 domains.
5. **Fuzzer-DSE1**: Our fuzzer on the Core-DSE1 dataset (i.e., the DSE augmented version of Core-noPFs). Uncovered 15 confirmed flows in 9 domains. Both Fuzzer-DSE1 and Fuzzer-DSE24 only apply fuzzing to URLs that are not already in the original datasets

(i.e., before augmentation). This avoids duplicate work, e.g., the Fuzzer condition already applies fuzzing to all URLs in the Core dataset. This also explains why confirmed flows resulting from those DSE crawls are smaller than 83.

6. **FoxHound-2025**: We run FoxHound [65] v.126.0, a recent version (released February 2025) of the taint-enabled browser used by TalkGen in 2021, against the pages from the Core dataset. While it reported 347 confirmed flows in 250 unique domains, we found through manual analysis that the majority of these flows are not exploitable in typical modern browsers, due to built-in URL encoding mechanisms that are disabled in FoxHound by default.
7. **FoxHound-ENC**: This is the same version of FoxHound as above except we have re-enabled URL encoding for fair comparison with our browser. This resulted in 68 confirmed flows in 58 domains.

Additionally, we aim to provide information regarding the analysis time needed to discover confirmed flows and the effectiveness of our web archive component. To this end, we performed the following experiments three months after the above crawls. We re-analyzed the Vulnerable dataset, using FoxHound-ENC, Passive-live (Passive analysis without the web archive), Fuzzer-live and also Passive and Fuzzer with the web archive enabled. We measured how long each condition took from opening the browser to rediscovering a previously found exploitable flow.

Overall, the number of detected DOM-XSS flows rises quickly at the start and plateaus a few minutes later. FoxHound-ENC is particularly faster at converging than Passive, both reaching a similar number of confirmed flows before timing out. This is expected considering the known coverage tracking overhead [48] in our Chromium, which is a necessary feature for our Fuzzer component. In this same experiment, we find that the web archive helps us rediscover exploitable flows in 95% and 93% of pages originally found vulnerable by Passive and Fuzzer respectively.

Runtime information. We have packaged each evaluation condition in a docker image. Each condition is run against a URL by launching a docker container based on the respective image. This container is restricted to using 6GB of RAM and 6 cores. With respect to timeouts, we enforce a soft analysis time budget of 3 minutes for all conditions, plus 1 minute to gracefully exit before forced termination.

3.3.2 RQ1: Importance of User Interactions

In this section we address **RQ1**: can user interactions generated by our fuzzer uncover new vulnerabilities? We refine this research question into the following sub-questions:

RQ1a: How does active navigation with simulated user interactions compare to passive navigation in terms of discovered potential and confirmed flows?

RQ1b: What is the impact of combining high-level actions in fuzzing on DOM-XSS detection effectiveness?

RQ1c: How does Fuzzer compare to existing tools for user interaction simulation?

RQ1a: Potential and confirmed flows in Passive versus Fuzzer

We first measure the effect of fuzzing through an ablation study, comparing Passive (fuzzer disabled) with Fuzzer (fuzzer enabled) on the Core dataset. We report the number of unique potential flows and confirmed flows in each condition.

Potential flows. Figure 3.9 shows the number of **unique** potential flows found: Passive discovers 2,023 (810+1213), whereas Fuzzer discovers 2,449 (1236+1213). Overall, simulating user interactions increases the number of potential flows by 21% compared to passive analysis.

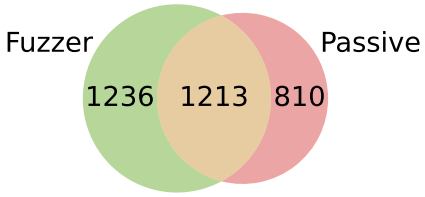

Figure 3.9: Unique potential flows found by Passive and Fuzzer.

Confirmed flows. Figure 3.10 shows the number of **unique** confirmed flows detected by our instrumented browser: Passive finds 72 (5+67), while Fuzzer finds 83 (16+67), a 15% increase.

To account for randomness in both Fuzzer and Passive analysis, we repeated each condition five times on the subset of pages containing the 21 (5+16) flows uniquely identified by one of these conditions in the first run. The goal is to test whether the detection (or lack thereof) of these flows are due to simulated interactions or non-determinism.

Out of the 16 flows initially unique to Fuzzer, 15 were never discovered by Passive in subsequent runs. Manual inspection revealed the remaining flow was actually a duplicate vulnerability: the script containing the sink call differed slightly between conditions, which caused the deduplication method to consider these two flows as different. These differences in page content can occur for example when random or time-based requests are made to the page, but our web archive component is disabled when confirming flows and thus the server can easily serve different content to the browser.

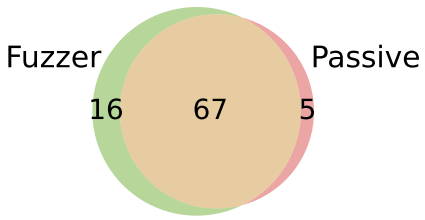

Figure 3.10: Unique confirmed flows found by Passive and Fuzzer.

For Passive, the remaining 4 of the 5 initially unique confirmed flows were eventually discovered by Fuzzer in one of the 5 runs. To summarize, when running Passive and Fuzzer 5 times,

Passive finds 72 unique confirmed flows, and Fuzzer finds a superset of those, with an additional 15 confirmed flows, a 21% increase.

Result 1a: SWIPE's fuzzer yields a 21% increase in potential flows and a 15% increase in confirmed flows over a single run compared with Passive. When accounting for non-deterministic effects over five runs, confirmed flows increase by 21% as well, compared to Passive.

RQ1b: Impact of high level action combinations

Fuzzer combines high-level actions into composite sequences and exchanges parts between these sequences (cross-over technique described in Section 3.2.5). We now evaluate the impact of action combinations on both DOM-XSS detection and JavaScript coverage.

We compare the original Fuzzer with a variant where action combinations are disabled (simpleFuzzer), using the 194 pages of the Vulnerable dataset. In simpleFuzzer, only primitive high-level actions that achieve best coverage in each round are retained. simpleFuzzer misses 7 confirmed flows detected by the original Fuzzer.

Regarding coverage, Figure 3.11 shows average executed JavaScript bytes: Passive executes 1,298,737 bytes, simpleFuzzer executes 1,364,087 bytes, and Fuzzer executes 1,468,703 bytes. This represents a 13% increase by the original Fuzzer over Passive and an 8% increase over simpleFuzzer.

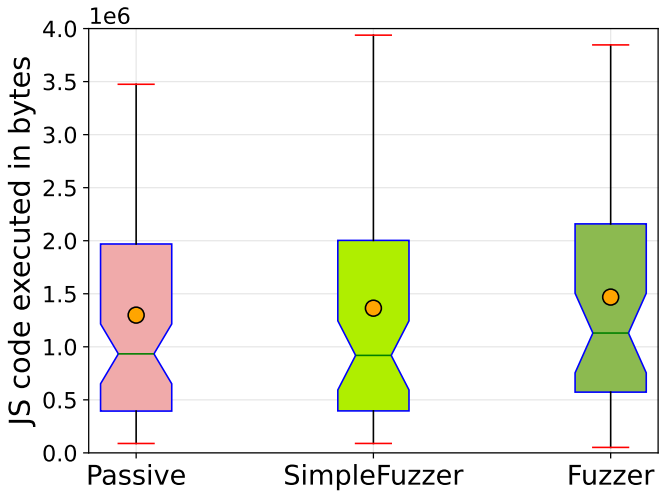

Figure 3.11: JavaScript bytes of code executed for Passive, Fuzzer and Fuzzer without action combinations in the Vulnerable dataset.

Increased code execution may originate from both direct event handler execution and indirect resource loading. For example, additional HTML and JavaScript resources may be loaded by event handler code. To isolate the impact of fuzzing on direct event handler execution, we added breakpoints in event handlers. Figure 3.12 reports counts for the top 10 supported event handlers that most frequently contained sink calls.

Passive executes a total of 291 supported event handlers, as they are triggered programmatically by the page, without user interactions. simpleFuzzer executes 1,762 handlers, a 6x increase

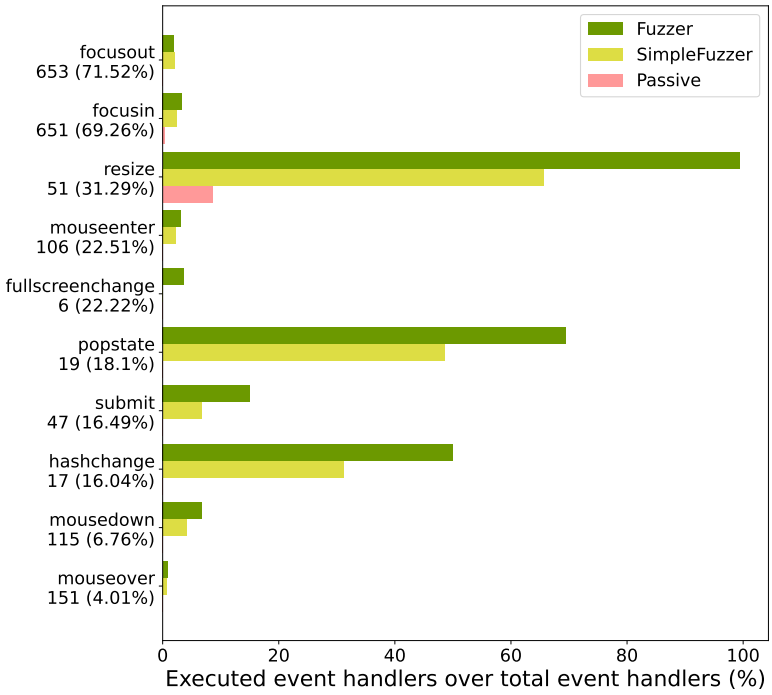

Figure 3.12: Percentage of event handlers in the Vulnerable dataset that were executed by Passive, Fuzzer and the simpleFuzzer that does not combine actions, for the 10 supported event handlers with higher sink calls frequency.

over Passive. Fuzzing while combining high level actions resulted in the execution of 2,920 handlers, a 10x increase over Passive.

We additionally report that event handlers constitute 1.56% of all JavaScript code observed in our crawl. **Supported** handlers account for 1.11% of all JavaScript. Thus, in principle, our fuzzer can simulate interactions for 71% of event handler code found in the wild. In practice, execution may fail due to hard-to-satisfy conditions, complex interaction sequences, or dynamic DOM changes (e.g., dynamic removal of elements containing handlers). Note that this is consistent with the 13% overall coverage increase: event handlers can trigger additional functions or resource loads.

Result 1b: Combinations of high-level actions increase coverage by 8% and execute 1.66x more event handlers than simple fuzzing, enabling discovery of 7 additional confirmed flows. The large fraction of unexecuted event handler code suggests further opportunities for improving DOM-XSS detection through user interaction simulation.

RQ1c: Comparison with CrawlJax

We compare the performance of Fuzzer with CrawlJax [89] with respect to DOM-XSS detection. Note that while CrawlJax systematically explores JavaScript-driven web applications through automated interaction, it lacks the ability to detect DOM-XSS vulnerabilities. To surpass that limitation and use a fair comparison, we replace the Chromium used by CrawlJax's with the

same taint-tracking Chromium used by SWIPE. We refer to this condition as *CrawlJax+Taint-tracking Chromium*. Both tools are run under the same runtime conditions against the 194 pages of the Vulnerable dataset. Results are summarized in Table 3.5: CrawlJax finds 47 confirmed flows in 40 domains, including 19 confirmed flows and 1 vulnerable domain not detected by Fuzzer. SWIPE-Fuzzer uniquely finds 55 confirmed flows and 34 vulnerable domains, which are missed by CrawlJax.

Component/Tool	**Conf Flows**	**Domains**
SWIPE-Fuzzer	83 (55)	73 (34)
CrawlJax+Taint-tracking Chromium	47 (19)	40 (1)

Table 3.5: Number of confirmed flows and vulnerable domains detected by SWIPE-Fuzzer and CrawlJax on the Vulnerable dataset. Numbers in ( ) indicate how many flows and domains are unique to each tool. CrawlJax+Taint-tracking Chromium refers to CrawlJax using our browser to detect flows.

Manual analysis shows the vulnerable domain uniquely discovered by CrawlJax was missed because SWIPE-Fuzzer resized the window to trigger an `onresize` handler, which hid a clickable DOM element required for the vulnerability. We confirmed that disabling `onresize` allowed SWIPE-Fuzzer to detect this flow.

The remaining 18 flows uniquely reported by CrawlJax were actually identified by Fuzzer but were considered distinct due to minor script variations causing different sink locations. This reflects the same deduplication issue observed in **RQ1a**, though it was a more frequent problem in this analysis. Running Fuzzer with identical browser flags and window sizes as CrawlJax recovered 5 of these 18 cases.

We suspect the slight changes in page content originate from the different way that CrawlJax interacts with the browser, compared to our SWIPE conditions. It is challenging to isolate all these differences, but we launched a version of the Fuzzer that passes identical browser flags and browser window size to CrawlJax. In this last experiment, Fuzzer found 5 of the 18 confirmed flows that used to be uniquely found by CrawlJax, as the sink location corresponded to what CrawlJax observed. When we reverted back to running Fuzzer with the original browser flags and window sizes, those 5 cases were again missed, as Fuzzer found the same sink location as in our original crawl. The number of vulnerable domains is a preferable metric for this comparison, as it is more robust against deduplication issues, ignoring exact sink locations.

Result 1c: Fuzzer discovered vulnerabilities in 18 domains not detected by CrawlJax. CrawlJax uncovered one confirmed flow missed by Fuzzer due to indirect effects of supporting the `onresize` handler.

3.3.3 RQ2: Synthesis and Impact of PFs

DSE aims to synthesize values for GET parameters and URL fragments (PFs) (Section 3.2.6). To evaluate its effectiveness, we refine RQ2 into the following sub-questions:

RQ2a: How effective is DSE at synthesizing PFs that expose DOM-XSS vulnerabilities?

RQ2b: How does DSE compare in effectiveness to off-the-shelf tools for GET parameter discovery?

We now answer each sub-question individually.

RQ2a: Effectiveness of DSE at synthesizing vulnerability-triggering PFs

The presence of specific GET parameters or fragment values can expose vulnerabilities that would otherwise remain undetected. The majority (70%) of URLs in our Core dataset already include PFs observed in the wild. One question we address is whether DSE can synthesize the required PFs when they are absent from the target URL. Answering this allows us to evaluate DSE's ability both to rediscover known vulnerabilities and to reveal new ones through parameter synthesis.

Rediscovery of known vulnerability-triggering PFs. We use the Vulnerable-noPFs dataset, where all PFs were removed from the original Vulnerable dataset, to assess whether DSE can regenerate the missing parameters needed to rediscover vulnerabilities. By comparing the results of fuzzing the Vulnerable-noPFs dataset with fuzzing the Vulnerable dataset we can obtain a set of confirmed flows triggered only when PFs are present. We then evaluate whether DSE can augment Vulnerable-noPFs into Vulnerable-DSE24 such that fuzzing the augmented dataset reproduces those flows. For this augmentation we allocated a 24-hour timeout per page.

Figure 3.13 shows the number of confirmed flows obtained with Fuzzer-noPFs on the Vulnerable-noPFs dataset, Fuzzer-DSE24 on the Vulnerable-DSE24 dataset and Fuzzer on the Vulnerable dataset (i.e., the original 194 vulnerable URLs of the Core dataset). Results show that 57 confirmed flows (42+15) appear only when PFs are included in the target URL. The remaining 26 flows (14+12) were found by Fuzzer-noPFs, showing that parameters were not needed in those cases.

When we strip parameters and apply DSE to synthesize new ones, Fuzzer-DSE24 rediscovers 15 of the 57 flows (26%). Notable, Fuzzer-DSE24 also discovers 11 unique confirmed flows not previously observed. To verify that these require synthesized parameters, we executed Fuzzer and Passive three times against the corresponding pages, and 10 out of 11 flows could never be rediscovered without DSE.

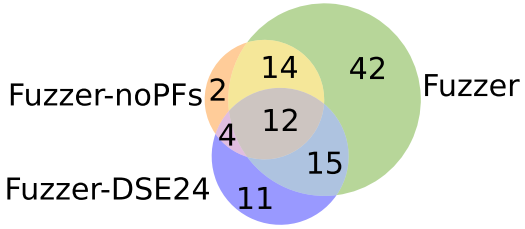

Figure 3.13: Comparison of confirmed flows found across Fuzzer, Fuzzer-noPFs and Fuzzer-DSE24 (RQ2a).

Discovery of new vulnerabilities. We next assess whether DSE can expose vulnerabilities in Core-noPFs, consisting of URLs collected from the wild with all PFs removed. For this experiment we allocated a 1-hour timeout per page, with a total budget of 100 hours for the full augmentation crawl. During that time, DSE analyzed a subset of 13,555 URLs from Core-noPFs, producing the augmented Core-DSE1 dataset with 56,152 synthesized URLs (see Section 3.3.1).

Running Fuzzer against the Core-DSE1 dataset yielded 15 confirmed flows. Manual inspection revealed that 10 correspond to previously unknown vulnerabilities across 7 pages on 5 distinct domains (4 of these new domains were not previously found to be vulnerable). The remaining 5 confirmed flows had already been discovered in the original Core dataset.

The cost of DSE synthesis per vulnerable URL ranged between 7 and 3410 seconds, with an average of 1048 seconds (17 minutes).

Result 2a: DSE is effective at synthesizing PFs that expose vulnerabilities. It successfully regenerated parameters to rediscover 26% of confirmed flows requiring specific parameters and generated new parameter combinations that revealed 10 new vulnerabilities on already vulnerable pages, as well as 10 previously undiscovered vulnerabilities in other pages of the Core dataset.

RQ2b: Comparison with off-the-shelf GET parameter discovery tools

We compare DSE with three off-the-shelf tools designed to identify GET parameters: the fuzzers ffuf [52] and wfuzz [5], and Wapiti [126], a web vulnerability scanner that includes parameter discovery functionality.

DSE vs. ffuf and wfuzz. FFuf [52] automatically discovers GET parameters through fuzzing. It constructs URL templates such as `https://.../path?{fuzz}=val` and enumerates candidate parameter names whose use yields valid responses. Similarly, Wfuzz [5] injects input into specified request fields, including GET parameters. Both tools draw their candidate values from user-supplied wordlists. We focus on the Core-DSE1 dataset, which contains 6,549 unique GET parameter keys, synthesized by DSE. We collected the default wordlists used by ffuf and wfuzz, which together contain 50,275 unique entries, covering all GET parameters these tools attempt by default.

The overlap between these wordlists and the DSE-generated parameters is limited to 287 entries. This means that 95.6% of the parameters synthesized by DSE are absent from the ffuf and wfuzz wordlists. Unlike static wordlists, each parameter combination generated by symbolic execution is expected to correspond to a distinct execution path, thereby increasing code coverage.

DSE vs. Wapiti. Wapiti discovers GET parameters mainly by statically parsing forms or links present in web pages. We evaluate DSE effectiveness by comparing vulnerabilities discovered in URLs generated by Wapiti with those in DSE-generated URLs after augmenting the Vulnerable-noPFs dataset. Both tools were given a 24-hour time budget per page. In each case, we ran Fuzzer on the generated URLs to detect vulnerabilities.

Results are summarized in Table 3.6. Wapiti discovered at least one GET parameter in 9 of the 194 pages, whereas DSE discovered at least one GET parameter in 101 of the 194 pages. Although Wapiti identified some parameters missed by DSE, none of the vulnerabilities enabled by Wapiti+SWIPE-fuzzer were new, as they had already been discovered by Fuzzer alone.

Component/Tool	Conf Flows	Domains
SWIPE-Fuzzer (baseline)	83	73
SWIPE-DSE-24 + SWIPE-Fuzzer	98 (15)	78 (5)
Wapiti + SWIPE-Fuzzer	87 (4)	73 (0)

Table 3.6: Number of confirmed flows and vulnerable domains detected by Fuzzer-DSE24 (the fuzzing by SWIPE-Fuzzer of pages augmented by SWIPE-DSE with a 24-hour timeout) and Wapiti + SWIPE-Fuzzer. Numbers in parentheses indicate the number of flows unique to each tool.

Result 2b: DSE uncovers GET parameters that off-the-shelf tools miss. ffuf and wfuzz fail to recover 95.6% of the unique parameters synthesized by DSE when augmenting 13,555 pages. Parameters generated by DSE enable more confirmed flows to be discovered than those found by Wapiti. However, because Wapiti and DSE identify different parameters, the two tools are complementary.

3.3.4 RQ3: Comparison with other DOM-XSS Detection Tools

We compare SWIPE with two other DOM-XSS detection tools: ZAP [100] and FoxHound [65]. ZAP is a web application security scanner that performs automated crawling and active scanning for vulnerabilities, including DOM-XSS. FoxHound is conceptually closer to Passive, relying on dynamic taint analysis in a modified browser and using passive navigation. Table 3.7 summarizes the results of running ZAP and FoxHound against the Vulnerable dataset and compares them with SWIPE's results.

SWIPE vs. ZAP. ZAP was executed under runtime conditions similar to Fuzzer. We configured ZAP to run the AJAX spider and DOM-XSS active scan modules. We measured the number of vulnerable pages where ZAP raised Cross-site scripting (XSS) alerts and included those results in Table 3.7. Overall, ZAP identified vulnerabilities in only 2 of the 194 vulnerable pages.

Note that ZAP reports a vulnerability only when it can successfully exploit it, i.e., when an injected payload opens an alert window. ZAP prioritizes injecting payloads in the URL fragment to bypass WAFs, but many vulnerabilities are not exploitable through this vector. Regardless, the results we have obtained are in line with what prior work DOMsday [88] reported regarding Burp [127], another web application scanner. We suspect the performance gap between ZAP and SWIPE is due to the advantages of dynamic taint analysis: Dynamic taint analysis tracks flows even when attacker input is transformed by JavaScript, and the provenance information recorded by our browser allows precise identification of where payloads must be injected.

Component/Tool	#Vuln. pages	(%)
SWIPE + FoxHound-ENC	**194**	**100%**
SWIPE-DSE + SWIPE-Fuzzer	146	75.26%
SWIPE-Passive	127	65.46%
FoxHound-ENC	120	61.86%
ZAP	2	1.03%

Table 3.7: Number of pages from the Vulnerable dataset that were deemed vulnerable by ZAP, SWIPE and FoxHound-ENC.

Passive versus FoxHound. We run FoxHound, a taint-enabled Firefox browser used by TalkGen, against the pages in the Core dataset (FoxHound-ENC). For fair comparison with Passive, we re-enabled URL encoding, a feature that was disabled in the original work TalkGen [17]. Note that Passive also has URL encoding enabled, as it is the current standard for modern browsers. Results of running Passive and FoxHound-ENC against the Core dataset are reported in the first two columns of Table 3.8.

FoxHound supports sources and sinks beyond those related to DOM-XSS, but Table 3.8 only includes results for DOM-XSS flows.

Flows. While confirmed flows aim to approximate unique vulnerabilities, the same cannot be said to normal flows (fifth row on Table 3.8) which aim to simply represent a propagation of information from an attacker-controlled source to a sensitive sink. Passive reports 15.6M flows compared to 3.8M in FoxHound-ENC. This discrepancy stems from different flow definitions: Passive emits multiple flows when all parts of the URL (i.e., protocol, host, path, query and fragment) propagate to a sink, while FoxHound aggregates them as a single flow.

Potential flows. Passive detects 2,023 potential flows, while FoxHound-ENC reports 3,408.

We manually investigated 2 potential flows that only FoxHound-ENC attempted to exploit, and found that both were false positives. FoxHound treats the entire URL as a single source, including the protocol and host sources which are not attacker-controllable. By contrast, SWIPE distinguishes URL components and discards flows with harmless sources before confirmation.

In one of the 2 manually inspected flows, a script extracted the host from `location.href` and inserted it into `innerHTML`. Since altering the host forces navigation to another site, this flow is not exploitable. In the second example, the protocol component (e.g., `https`) was inserted in a sink. Both flows were considered potential by both tools, but only FoxHound attempted exploitation.

Note that the above is not meant to indicate a limitation of FoxHound. We used the reporting mechanism for potential flows provided by FoxHound, but FoxHound additionally provides an operation tree for each flow, representing the operations that were performed on the source until it reaches the sink. Parsing the operation tree would allows us to filter unexploitable potential flows, similarly to what SWIPE does.

Metric	Passive	FoxHound ENC (2025)	FoxHound No ENC (2021) [17]	DOMsday [88]	25m Flows [71]
Date	04/2025	04/2025	09/2020	08/2017	11/2013
Domains	30,000	30,000	100,000	10,000	5,000
Sub-pages	5	5	10	5	all depth 1
Web pages	44,480	44,480	390,092	44,722	504,275
Flows	15,647,717	3,826,017	20,912,107	4,140,873	24,474,873
Flows/1k pages	351,792	86,017	53,608	92,591	48,534
Potential	2,023	3,408	15,710	5,217	?
Pot./1k pages	45.48	76.62	40.27	116.65	?
Confirmed	72	68	7,199	3,219	8,163
Conf./1k pages	1.62	1.53	18.45	71.98	16.19
Vuln. domains	64	58	711	364	480

Table 3.8: Crawling comparison between Passive and results reported by TalkGen [17] (FoxHound-2021, encoding disabled), DOMsday [88], 25mFlows [71] and FoxHound-ENC (encoding enabled), including number of flows, which include all source sink pairs considered by DOMsday, potential flows (Pot.), which only include URL sources to JavaScript or HTML sinks, and confirmed flows (Conf.).

Confirmed flows. Passive and FoxHound-ENC found 57 confirmed flows in common, but Passive uniquely found 15 confirmed flows, whereas FoxHound-ENC uniquely found 11 confirmed flows. At the domain level, Passive detects vulnerabilities in 64 domains versus 58 for FoxHound-ENC.

We sampled 5 confirmed flows that only Passive finds. Undertainting issues in FoxHound were the cause of four of these cases. The remaining case was due to overtainting: some bytes in the final sink argument were being wrongly reported by FoxHound-ENC as originating from the URL. This caused the injection algorithm to fail to locate a good position where the payload should be injected.

We also manually sampled 5 confirmed flows that only FoxHound-ENC finds. Undertainting issues in the underlying Chromium of Passive caused all 5 missing cases. These findings highlight the inherent difficulty of capturing all potential flows, given the complexity of modifying browser engines to implement taint tracking.

Result 3: Passive and FoxHound share common methodology of passively navigating to the page, although they use different modified browsers for taint-tracking, finding a similar number of confirmed flows. Out of the 194 vulnerable pages discovered by SWIPE or FoxHound, ZAP only signals 2 as vulnerable.

3.3.5 RQ4: DOM-XSS Detection Over the Years

This section compares our results with major prior studies on the prevalence of DOM-XSS in the wild. Table 3.8 aggregates results for SWIPE (2025), our replication of FoxHound (2025), and results reported by TalkGen [17] (2021), DOMsday [88] (2017), and 25mFlows [71] (2013). We highlight how the evolution of the web impacts the results and complicates direct comparisons.

SWIPE and FoxHound-ENC versus prior work. We first compared the results of SWIPE and our FoxHound replication (first two columns of Table 3.8) against the numbers reported in earlier studies (last three columns). Since the number of analyzed pages differs across studies, we normalize results by reporting confirmed flows per thousand pages (Conf./1k pages). At first glance, Table 3.8 indicates a substantial decline: Passive and FoxHound-ENC detected 1.62 and 1.53 confirmed flows per thousand pages respectively, whereas prior work reported values at least one order of magnitude higher. However, this apparent decrease is influenced by multiple factors, including dataset selection, methodological choices, and change of browser behavior over time.

Impact of URL encoding. Modern browsers encode certain URL characters before navigation (RFC 1738 [18]). We refer to this as *native URL encoding*, as browsers do it natively in contrast to application-level encoding (e.g., via the use of `encodeURIComponent` by the web application). Although originally not a security feature, native URL encoding strongly affects the exploitability of DOM-XSS vulnerabilities. To quantify its impact on DOM-XSS detection, we re-ran FoxHound's confirmation stage with URL encoding disabled, which we call *FoxHound-2025* (fifth row of Figure 3.8). Whereas FoxHound-ENC finds 68 confirmed flows, FoxHound-2025 quintuples that number to 347 (7.8 confirmed flows/1k pages). Critically, FoxHound-2025's uniquely discovered flows are not exploitable in modern browsers, as all major browsers now enforce URL encoding. This is important to discuss because while both recent results reported in Table 3.8 have URL encoding enabled, the remaining reported results from prior work disabled it one way or another:

- FoxHound-2021 completely disabled URL encoding in their browser.
- DOMsday used Chromium 54, a version that did not yet encode the hash part of the URL, and DOMSday always injected their payloads in the hash. As a result, all their payloads bypassed native URL encoding. Table 3.9 summarizes differences between Chromium 126 (used in our study) and Chromium 54 (used by DOMsday). The key change is that the fragment (i.e., the value of `window.location.hash`) is now encoded, whereas previously it was not.
- 25mflows typically used Chromium to confirm vulnerabilities that were injected via the fragment (which, again, at the time it did not use URL encoding), but to confirm flows that used other sources, 25mFlows used Internet Explorer [71], which performed no encoding anywhere at that time.

Given that URL encoding is now ubiquitous, the lower numbers in our results (Table 3.8) reflect real-world exploitability today better.

Input	Chromium 54	Chromium 126
window.location.href	Partial	Yes
window.location.hash	No	Yes
document.referrer	Partial	Yes
window.location.search	Yes	Yes

Table 3.9: URL encoding differences between our browser version (Chromium 126) and the one used by DOMsday. We found no differences between ours and the latest version.

Impact of deduplication definitions. All prior work claims to follow the deduplication method of 25mFlows, yet that method is ambiguously defined in the original paper [71], described only in a few short natural-language statements. Our experience suggests that inconsistent interpretations likely contributed to differences across studies. 25mFlows specifies that the sink location should be used for deduplication, but the meaning of *sink location*, beyond including the line and column locations of the sink, is underspecified.

One interpretation is that the full vulnerable script URL should be included as part of the sink location. This raises a key question: should GET parameters in the script URL be included? Including GET parameters may cause two flows in the same script (i.e., URLs with different GET parameters but same script content) to be counted as multiple distinct vulnerabilities. Excluding them risks underestimating diversity of flows when the server truly differentiates responses by GET parameters.

In this work, we exclude GET parameters (and fragments) from the sink location during deduplication. This decision was motivated by our DSE, which generates many new PFs. Including GET parameters when deduplicating DSE flows leads to severe inflation in confirmed flow counts. More importantly, if we include parameters in the sink location when deduplicating the confirmed flows of FoxHound-2025, the number of confirmed flows grows from 347 to 612 (13.76/1k URLs). We find that GET parameters often result in the loading of the same page, thus we argue that these 612 flows are unlikely to be truly unique. Notably, DOMsday reported that 62% of all confirmed flows were concentrated in a single domain, and the top 10 domains accounted for 84% of 3,219 confirmed flows [88]. This raises the question of whether their results would shrink under a deduplication method with parameter-agnostic sink location.

Impact of analyzing web advertisements. DOMsday reported that 82% of vulnerabilities originated in advertisement or analytics content. Since then, in-browser advertisement filtering has advanced considerably. Compared to when they run their respective crawls, a wider range of ads and trackers are now blocked by Chromium [41, 42, 44, 45, 46, 47] and Firefox [90, 91, 92, 93, 94].

DOMsday observed that 44.3% of all frames loaded were advertisements. In contrast, Passive flagged only 26.3% of loaded frames as advertisements.

Figure 3.14 categorizes vulnerable websites, frames, and scripts in our dataset, using the IAB taxonomy [54] (since Blue Coat K9, used by DOMsday, is discontinued). Our results show that advertisements are no longer the most vulnerable category of scripts. Instead, the ”Technology

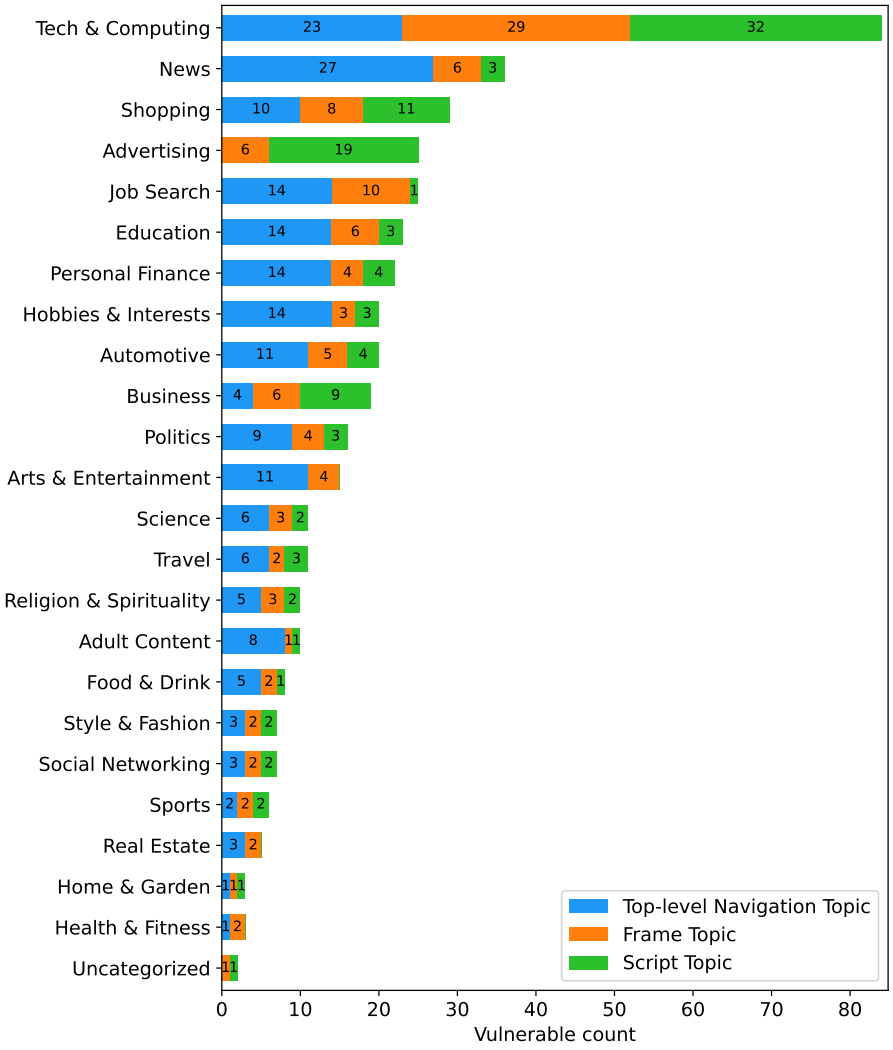

Figure 3.14: Categorization of vulnerable top-level URLs, frames and scripts.

& Computing" category ranks highest, with advertisements ranking second. We hypothesize that this difference stems from improved browser-level ad filtering. For example, DOMsday's Chromium did not filter intrusive ads, whereas current Chromium versions do [42, 44, 46]. We chose not to disable filtering, for the same reason we did not disable URL encoding: we aim for experiments to reflect modern browser behavior.

Impact of dataset differences. Dataset composition may further contribute to observed differences. We tested the Tranco top 30,000 **domains**, adding an `https://` prefix. DOMsday used HTTP instead, while TalkGen did not specify which protocol they used. This matters because since 2018 Chromium has labeled plain HTTP sites as insecure [43]. Mixed-content blocking prevents non-HTTPS advertisement frames from loading within HTTPS pages. In our Passive crawl, Chromium blocked 8,967 resources due to mixed content, affecting 931 pages (2% of the Core dataset). To better understand differences with DOMsday, we obtained from DOMsday

authors a dataset of 849 confirmed flows that DOMsdsay found in 2018. Passive confirmed only 25 of them. It failed to confirm flows in the remaining 824 cases for the following reasons:

- 104 pages are no longer reachable
- 448 pages no longer load the vulnerable script.
- Of the remaining 272, we manually examined 10:
  - In 7 cases, the vulnerable script no longer contained the sink call that it did in 2018. This was verified using the Wayback machine [4].
  - In the other 3 cases, the previously vulnerable script was considerably different from the 2018 version, and we could not manually find vulnerabilities in the current version.

Result 4: We discussed how the declining number of confirmed flows across studies reflects changes in the web ecosystem. Our evidence suggests that default browser mechanisms (e.g., URL encoding, advertisement filtering, and mixed-content blocking) prevent many vulnerabilities from being exploitable in practice. Combined with the widespread adoption of HTTPS, these factors may explain the marked decrease in DOM-XSS prevalence over the past decade. Unfortunately, it is difficult to identify with certainty all the reasons for the disparity between the number of flows across all the studies over the past decade, as datasets of confirmed flows are not usually made public for ethical reasons.

3.4 Discussion

This section discusses the limitations of our web archiving approach, the time efficiency of our dynamic analysis, and the recent DOM-XSS mitigation mechanism *Trusted Types*.

3.4.1 Limitations of the Web Archive Component

We evaluated the impact of our web archive on the reproducibility of experimental results. Specifically, we re-analyzed the Vulnerable dataset three months after the original crawl, using the same runtime conditions as before. In this experiment, we launched the following conditions:

- **Passive**: Typical SWIPE passive navigation condition using the previously created web archive.
- **Fuzzer**: Typical SWIPE fuzzing condition using the previously created web archive.
- **Passive-live**: Passive navigation without the web archive, i.e., against the live page.
- **Fuzzer-live**: Fuzzing without the web archive.
- **FoxHound-ENC**: Typical FoxHound condition without the web archive, for comparison with Passive-live in terms of flow detection efficiency.

For each condition, we measured (i) the number of previously exploitable flows rediscovered and (ii) the time elapsed between browser startup and flow detection. The results are summarized in Figure 3.15, where the Y-axis shows the cumulative number of rediscovered vulnerable pages (pages with at least one exploitable flow) and the X-axis shows detection time. Both Passive

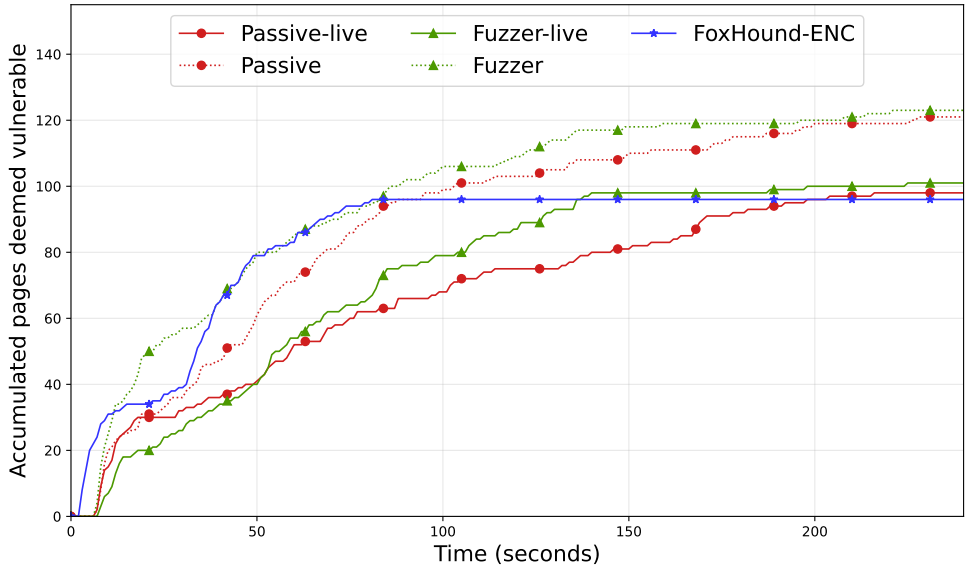

Figure 3.15: Accumulated number of pages deemed vulnerable as analysis time increases. Web archiving helps to rediscover more vulnerabilities and reproduce past results.

and Fuzzer, which rely on the archive, rediscovered substantially more vulnerabilities than their live counterparts, but not all vulnerabilities. This demonstrates that web archiving improves reproducibility of past vulnerabilities, but it is not a perfect solution.

Six pages under Passive and nine under Fuzzer were no longer found vulnerable, even when using the web archive. Some sources of non-determinism are challenging to handle and can contribute to this outcome, for example the use of the `Math.random` API, or differences in server load causing differences in page loading times.

To examine this further, we focus on the 133 pages that Fuzzer originally flagged as vulnerable. For each page, we computed the fraction of responses replayed from the archive compared to total responses during fuzzing. Figure 3.16 reports these results, with red vertical lines marking the nine pages no longer vulnerable under Fuzzer with the archive. In seven of these cases, at least one resource was served from the live server instead of the archive. These responses may have been removed (404), changed, or no longer vulnerable.

The remaining two cases highlight a different limitation: even though all requested resources were archived, vulnerabilities were not rediscovered. This suggests that reproducibility depends not only on complete archiving but also on runtime conditions and interaction patterns that our current solution does not fully capture.

3.4.2 Trusted Types

Trusted Types (TT) is a browser security mechanism designed to mitigate DOM-XSS. It prevents injection of untrusted data into vulnerable sinks by enforcing developer-defined sanitization policies. Each policy specifies transformations that render inputs safe before they reach a sink. If an input does not undergo the transformation, TT blocks its execution.

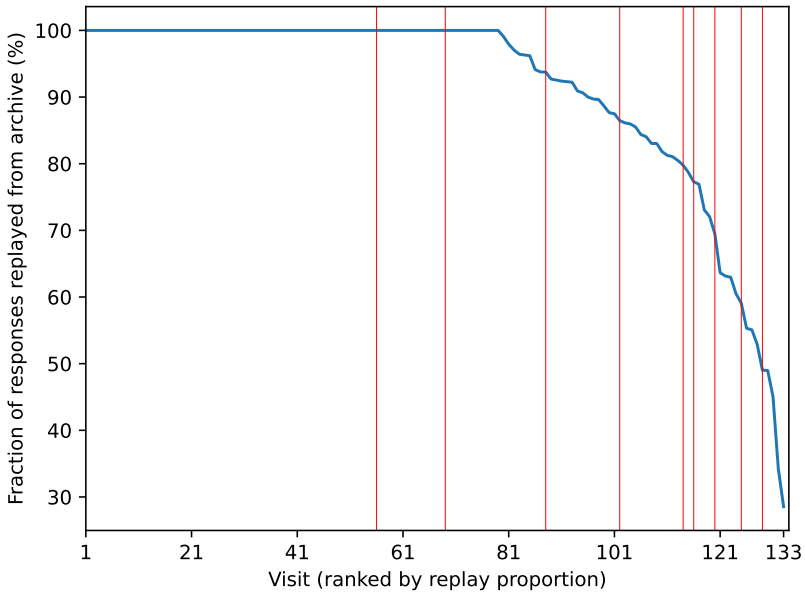

Figure 3.16: Fraction of responses replayed from the webarchive during fuzzing for each vulnerable page found by the Fuzzer. Vertical lines indicate pages where vulnerabilities were not rediscovered.

TT mitigates some DOM-XSS but is not a complete solution. Its effectiveness depends entirely on the correctness of the policies defined by developers, and adoption remains limited. We do not have precise data on how many pages in our Core dataset implemented TT, but Chrome Platform Status reports that approximately 13.3% of pages enforce TT as of 2025, up from 5.8% in 2021, when TalkGen performed their crawl [1].

3.5 Conclusions

This chapter presented SWIPE, a new infrastructure for detecting DOM-XSS, which advances prior work in three main respects. First, SWIPE incorporates a novel fuzzer that actively simulates user interactions to trigger event handlers. This fuzzing approach identified 15% more confirmed flows than passive navigation. Second, SWIPE integrates a dynamic symbolic execution tool (DSE) to systematically extract GET parameters from target pages. DSE enabled the discovery of confirmed flows that were missed by all other existing methods. Third, we analyzed how the co-evolution of web content and browsers presents significant challenges that complicate the reproducibility of prior DOM-XSS measurement studies. One example is the effect of evolving URL-encoding practices on the exploitability of DOM-XSS flows. We found that this factor is a critical variable that future studies must account for when comparing the prevalence of DOM-XSS vulnerabilities with earlier reports. SWIPE addresses several of these challenges and achieves more reliable detection results.

Chapter 4

Automatic Detection and Exploit Synthesis for Node.js Vulnerabilities

While the previous chapter emphasizes analysis on client code, this section will get us started on our goal to study how to effectively detect and confirm flows for code injection vulnerabilities in Node.js packages. The work described in this section was published at NDSS 2025 [23].

4.1 Overview

One limitation of DTA is that the detection of a flow requires execution of the application with an input that specifically triggers the vulnerable program path. Current fuzzing tools for JavaScript input generation are limited in that they either generate only string-based inputs [13, 56], or depend on the availability of test cases from the target package or its dependencies [117]. However, many packages lack comprehensive test suites [117], and some may even require inputs with complex or nested structures. To address these limitations, we extend a prior tool NODEMEDIC [24] with a type and object-structure aware fuzzer, which we describe in detail in Section 4.2. We show that our fuzzer significantly improves the number of potential Arbitrary Code Execution (ACE) and Arbitrary Command Injection (ACI) flows that are discovered (Section 4.4.3).

Prior tool NODEMEDIC additionally attempts to synthesize exploits for potential flows, so as to confirm their exploitability, but it has special difficulties when confirming ACE flows. This problem arises because exploiting ACE vulnerabilities typically involves the construction of an input that, after it is transformed by the package, results in the execution of valid JavaScript code via a ACE sink like `eval` or the `Function` constructor. To overcome this limitation, we developed a confirmation component, which we refer to as *Enumerator* (Section 4.3.2). The Enumerator takes as input a *prefix*, which is a string of code that precedes the attacker-controlled portion of the final sink argument, and completes that *prefix* with placeholders for arbitrary JavaScript statements, thereby representing the statements that an attacker could inject for code execution. Once integrated in NODEMEDIC, the Enumerator was able to complete the majority of real-world prefixes that it encountered, resulting in a 21% increase in the total number of confirmed

ACE flows.

From this point onward, we refer to the modified NODEMEDIC tool, augmented with our novel components, as NODEMEDIC-FINE.

4.2 Type and Structure Aware Fuzzer for Node.js Packages

NODEMEDIC-FINE utilizes the original NODEMEDIC instrumentation, which tracks taint propagation from package entry points arguments to dangerous sinks like `eval`. In this section we detail our coverage-guided fuzzer that is aware of input types and object structure, the first new component in NODEMEDIC-FINE. This fuzzer aims to improve vulnerability detection in Node.js packages by exploring a broader range of execution paths.

4.2.1 Motivation

Recall Figure 2.2 which illustrated an example of an ACI vulnerability. Line 5 in this figure introduces a condition on the entry point arguments `if(params.flags !== undefined)`. That line, albeit apparently simple, imposes a series of constraints on the entry point argument `params`. To satisfy that condition, `params` needs to be able to hold properties, i.e. it needs to be an object. This example highlights a situation in which the attacker must construct an object payload with a particular attribute `flags`. More generally, package entry points may require inputs of diverse types and structure. Ideally, a detection system would efficiently infer these types and structures for each argument of a target entry point; however, the absence of explicit type information in JavaScript makes it challenging to do this using static analysis. We instead design a fuzzer that concretely interacts with the target Node.js package and its entry points to dynamically discover inputs that satisfy these requirements.

Originally, NODEMEDIC instruments packages to enable taint propagation from sources to sinks, thereby implementing dynamic taint analysis. We extend this base instrumentation to additionally track which attributes are accessed on tainted variables. Doing this is useful to reconstruct what keys are expected by the package when the input must be an object.

4.2.2 Fuzzer Input Generation

Our fuzzer operates in conjunction with other components of NODEMEDIC-FINE as illustrated in Figure 4.1. Note the mention of a driver in that figure, which is an automatically generated JavaScript program that imports both the fuzzer and the target library, managing their interaction. We provide pseudocode describing the driver in Figure 4.2. Overall, for each entry point, this driver is mainly responsible for doing the following: Collecting inputs from the fuzzer (lines 5–9); Marking those inputs as tainted, by leveraging our DTA instrumentation (line 7); Invoking the entry point with the tainted inputs (line 10); Handle the situation when our taint infrastructure finds a flow (lines 11–14); Collecting coverage information to provide as feedback to the fuzzer (line 15–19).

During the input collection phase of the driver, the fuzzer is responsible for generating candidate inputs to be later supplied to the package under analysis. These inputs are generated based

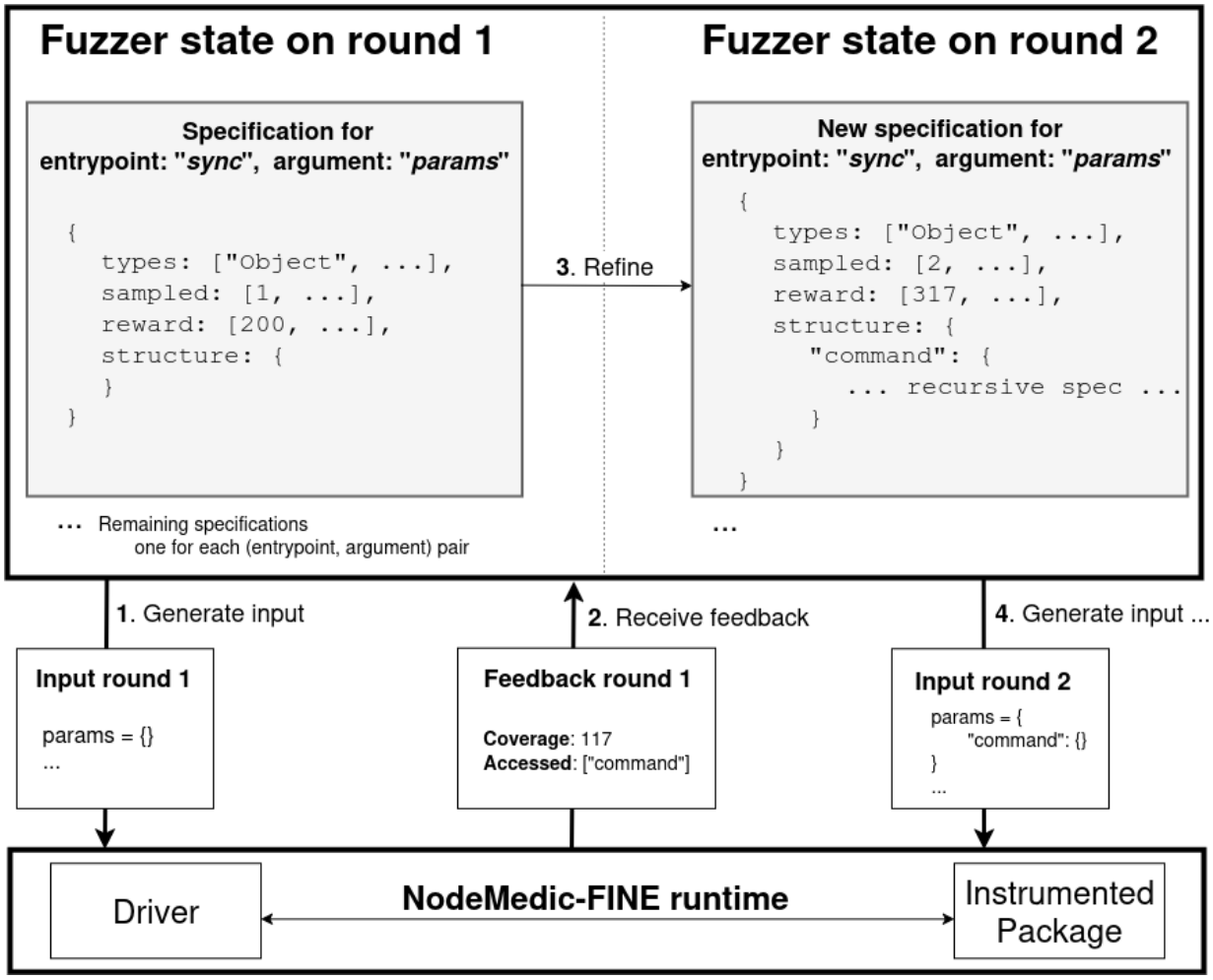

Figure 4.1: Fuzzer loop and interaction with the instrumented package, for a package with an entry point called *sync*, expecting an object argument `params` with an attribute `command`.

on a specification that the fuzzer maintains for each entry point argument. This specification was initialized in the same manner for all packages that we have analyzed in our experiments, though it can be personalized at will.

Consider the previously mentioned example package (Figure 2.2) which has an entry point *sync* expecting an object argument *params* with an attribute *flags*. In Figure 4.1 we describe an example input specification that the fuzzer creates when analyzing such a package. Input specifications generally define the set of types that should be generated for each argument, as well as the frequency with which each type should be selected. That frequency is dynamically computed using two lists:

1. **sampled**: This tracks the number of times an input of each type has been generated already. This list is initialized with all 1's to prevent a division-by-zero error during frequency calculation.
2. **reward**: This accumulates the total reward obtained from generating inputs of each type. This is initialized as described in Section 4.2.3 and is updated with the coverage that was

```
start_time = time()
while (time() - start_time < TIME_BUDGET){
    for (entrypoint in target_package.entrypoints()) {
      inputs = [];
      for (argument in entrypoint.arguments()) {
        input = fuzzer.get_input(entrypoint, argument);
        taint_infrastructure.set_taint(input);
        inputs.append(input);
      }
      target_package.call_entrypoint(*inputs)
      if (taint_infrastructure.flow_found()){
        confirm_flow();
        exit();
      }
      for (input in inputs){
        feedback = taint_infrastructure.structure_feedback(input);
        fuzzer.refine_specification(entrypoint, argument, feedback);
        taint_infrastructure.remove_taint(input);
      }
    }
}
```

Figure 4.2: Pseudocode for our driver component. Our actual driver is automatically generated specifically for the target package and its entry points, but this figure summarizes what steps the driver takes and how it interacts with the fuzzer, the target package and the taint infrastructure.

achieved when the entry point is executed with the generated input.

Each round, the fuzzer randomly generates an input based on the current specification, and refines that specification in three ways:

1. It increments the appropriate value in the **sampled** list to reflect the newly generated input of that type.
2. It adds the newly obtained coverage to the appropriate value in the **reward** list.
3. It records the attributes that were accessed by the package and adds them to the recursive specification in the **structure** component. This information is essential for accurately reconstructing expected object structure.

4.2.3 Fuzzer Weight Adjustment

Our fuzzer is coverage-guided, meaning that the coverage obtained using previous inputs influences the generation of future inputs. When deciding which new type to explore, we employ a simple yet effective method. We generate an array representing the expected coverage $\frac{reward_t}{sampled_t}$ for each type t. The elements of this array are normalized and then used as probability weights. Each fuzzing round dynamically refines these weights by updating coverage measurements. Thus,

promising inputs that achieved high coverage in the past will also have higher expected future coverage. It is important to note though that the fuzzer does not know how effective each type is at improving coverage until it generates inputs of all types. The tradeoff between continuing to generate inputs with types that seem promising versus generating inputs that were not explored much in the past is known as the exploration-exploitation dilemma [78]. Our weight adjustment method not only prioritizes input types that resulted in the execution of a significant amount of code in the past, it still makes it possible for types with little prior exploration to still be eventually chosen, even if they did not show much promise in the few times they were chosen in the past.

4.2.4 Fuzzer Weight Initialization

The initial values of the **reward** list were set based on the observation that some types are more commonly expected by Node.js package APIs with potential flows. To compute this, we performed a small scale experiment on 12k packages sampled from npm to identify the frequency of each JavaScript type that resulted in a potential flow. In this experiment, we assigned equal weight to each type and launched the fuzzer using that specification. We then measured which types were generated in inputs that caused the detection of a potential flow in any package.

Notably, just because one type is specifically prevalent in inputs that generated potential flows, it does not necessarily mean that it will also be prevalent in inputs that generate *confirmed* flows. Since actual *vulnerabilities* detected is the most important metric to measure in our context, one may wonder why we did not measure the frequency of types that were generated in inputs that caused the detection of confirmed flows, not just potential flows. The reason for that is two fold. First, potential flows may still signal real vulnerabilities, even though they could not be confirmed by NODEMEDIC or other tools. Secondly and most importantly, NODEMEDIC and other ACI and ACE vulnerability detection tools at the time when NODEMEDIC-FINE was implemented are biased to only confirm vulnerabilities with inputs that use certain types. Case in point, NODEMEDIC uses a confirmation methodology that is optimized for string solving, since that type is so commonly expected by Node.js packages. This bias is not significant during analysis, when potential flows are detected. Therefore, we aimed to produce a default specification that is not optimized for the current confirmation methodology of our tool, and instead it more generally attempts to maximize the number of potential flows.

Overall, we found that object inputs are most likely to result in potential flows, followed by strings, booleans and functions. It should also be noted that this specific initialization to fuzzer weights does not have to be followed by developers that use NODEMEDIC-FINE, as it can be personalized to individual cases. For example, one may assign higher generation probability to types that are known to be expected by a target package. In Section 4.4.3 we evaluate how supporting different input types influences the number of discovered potential flows.

4.2.5 Fuzzer Object Reconstruction

Our fuzzer aims to reconstruct the expected structure of Node.js package entry point arguments. For this, we modify NODEMEDIC's original taint instrumentation so that it also keeps track of the property names that were accessed whenever a *getField* operation is performed on a tainted

variable. This constitutes relevant feedback that we integrate in the fuzzer: At the end of each fuzzing round, the current input specification is refined so that it includes information about the attributes that were accessed in that round.

For example, in Figure 4.1, we illustrate these accessed properties that are given as feedback to the fuzzer in the `Accessed` field of the "Feedback round 1" box. In that specific example, not only 117 lines of code were executed, the `command` field was also accessed on a tainted variable. This leads to a refinement of the first specification. Thus, the next specification, which we call "New specification", now has an updated `reward` list with $200 + 117 = 317$ being assigned to the `Object` type, as well as an incremented first element of `sampled`. Importantly for this section, the `structure` field now contains a recursive specification, which is initialized in the usual way. Whenever the fuzzer generates values that have a reconstructed attribute, the feedback is used to refine the respective recursive specification, as well as the parent specification. Therefore, even if the target application attempts to access a chain of attributes from an entry point argument `argument.attrX.attrY.attrZ` all attributes are eventually reconstructed using recursive specifications.

Using this process, the fuzzer is able to generate an input with the field `command` set to a random type/value. The input specification, together with the recursive specification for the `command` attribute, will be updated in future rounds, accordingly to what types are generated for the `command` attribute and depending on the overall coverage improvements.

4.2.6 Fuzzer Generated Values

Our fuzzer generates inputs over 12 built-in JavaScript types: `String`, `Number`, `Function`, `Boolean`, `Array`, `BigInt`, `Symbol`, `Null`, `Undefined`, `Date`, `RegExp`, and `Object`. We use a seeded pseudorandom number generator to ensure reproducibility of test cases. Some types have trivial values, for example the type `Undefined` only has the value "undefined", similarly to `Null`. Simple values such as numbers and booleans are drawn from standard distributions, while strings have size ranging from 0 to the maximum integer attribute that we have seen accessed by the package. Complex types such as arrays and objects are constructed recursively according to specifications, allowing nested and structured inputs. Function types are chosen from only two values, a normal and an async function with no arguments and no return value. This satisfies a very common situation that we have observed in Node.js where packages expect a callback function and simply call it with no arguments. It could be interesting future work to study what kind of arguments packages expect and how they use any possible return values, and adapt the fuzzer accordingly.

4.3 Confirming Code Injection Flows in Node.js Packages

Prior tool NODEMEDIC already has the ability to automatically synthesize proof-of-concept exploits that demonstrate the presence of vulnerabilities. Thus, once a potential flow is found, NODEMEDIC creates a JavaScript program (known as the *exploit*) that imports the package and interacts with its entry points, in a way that proves that a vulnerable entry point exists. For ACI vulnerabilities, it verifies the existence of a vulnerability by checking whether the file `/tmp/success` was created after the exploit execution. This demonstrates that the attacker can leverage the package to create files on the filesystem, even if that was not supposed to be part of the legitimate

set of functionalities. For ACE, we can verify their presence by checking whether the exploit calls the `global.CTF` function. This is a function that is globally defined by the exploit and is never supposed to be called by the package, only through illegitimate means.

Synthesizing ACE exploits is challenging, not just for NODEMEDIC but for all prior approaches in this area. In Table 4.1 we describe the potential and confirmed flows and the exploitability rate of reported ACI versus ACE flows for several code injection vulnerability detection tools for Node.js. The key takeaway from that table is that code injection detection tools for Node.js have a harder time synthesizing exploits for ACE than for ACI vulnerabilities, in all datasets that were used. This is because ACE exploits must lead to a sink call with an argument that not only it is syntactically valid JavaScript, but also executes a specific statement. In the following sections, we will describe the novel components that we integrate in NODEMEDIC's confirmation methodology. Motivated by the above discussion, our improvements aim to address the limited effectiveness of NODEMEDIC when synthesizing exploits for ACE vulnerabilities.

Tool	Dataset alias	Potential ACE	ACE Confirmed	Potential ACI	ACI Confirmed
NodeMedic	RealWorld1	22	6 (27%)	133	102 (77%)
FAST	*Vulnerable*1	42	13 (31%)	169	86 (51%)
FAST	RealWorld2	16+5	6 (29%)	56+4	35 (58%)
Explode.js	*Vulnerable*2	24	13 (54%)	112	70 (63%)
Explode.js	RealWorld3	45	2 (4%)	151	51 (34%)
PoCGen	*Vulnerable*3	15+49+20	15 (18%)	99+67+14	99 (55%)
PoCGen	*Vulnerable*4	12+7+16	12 (34%)	85+6+1	85 (92%)

Table 4.1: Comparison of detected versus confirmed ACI and ACE flows across analysis tools. This table highlights the different exploitability rates for ACI versus ACE. Consider each row to be using different datasets of real-world npm packages or packages with vulnerabilities.

4.3.1 Usage of Polyglot Exploits for both ACI and ACE

We discovered that a significant number of packages have potential flows that can be exploited using simple polyglot input strings, which are constructed in such a way that they work under multiple different scenarios. We used the following polyglot for ACI vulnerabilities:

```
$(touch /tmp/success) #"  || touch /tmp/success #'  || touch /tmp/success
```

This polyglot can be separated in three parts. The first part `$(touch /tmp/success) #` accounts for situations when the package fails to sanitize certain shell metacharacters. This part of the ACI polyglot uses shell expansion metacharacters `$(command)` which can execute arbitrary commands in many situations. The second part `" || touch /tmp/success #` handles situations where the attacker injects input inside a double quoted string context. The final part `' || touch /tmp/success` handles single quoted string contexts. It is important to remember that for ACI, the final argument to the sink does not have to be completely syntactically valid in order to execute arbitrary commands.

With respect to ACE, we use the following polyglot:

```
global.CTF();//" +global.CTF();//' +global.CTF();//
```

This polyglot is constructed similarly to the ACI polyglot, for double quotes and single quotes so that the final argument is syntactically valid regardless of the string context, which is an important property for ACE vulnerabilities. There is no convenient shortcut like "shell metacharacters" for the ACE case.

4.3.2 Enumerator

In this section we describe the inner workings of our Enumerator, which can handle more complex ACE vulnerabilities for which the polyglot does not work. Once an ACE potential flow is found, NODEMEDIC-FINE has to come up with an input that proves the attacker has JavaScript execution capabilities. We call the final argument to the sink *objective payload*. In this case, a potential flow is assumed to have been found, and the objective payload is expected to be at least partially controlled by the attacker. Thus, whatever value is chosen for the attacker input, it must result in an objective payload that obeys all syntactic constraints of JavaScript and also executes the intended statement: `console.log('VULN FOUND')` to prove the existence of the vulnerability.

We find that many of the ACE potential flows that were not being confirmed by the original version of NODEMEDIC were in packages where attacker input is injected after a constant prefix. Once a potential flow is found, we can find what was the prefix by parsing the initial part of the sink argument that does not depend on tainted inputs. We provide this prefix to our Enumerator, which together with the remaining synthesis components has the responsibility of completing it with reasonable attacker input that may confirm the potential flow is exploitable. Note that we ignore anything that is placed after the attacker-controlled part because our payload will attempt to introduce a comment symbol (`//`) at the end. While effective in most situations, this may not work when the final sink argument has multiple lines. We discuss future work to improve this in Section 4.5.3.

4.3.3 Construction of an Objective Payload Obeying Syntactic Constraints

The objective payload must be valid JavaScript code, otherwise `eval` or the `new Function` constructor will fail with a SyntaxError. To this end, we manually constructed a graph, partly illustrated in Figure 4.3, encoding the syntax of common JavaScript primitives. Each node in the graph represents a JavaScript primitive, like a variable or a binary operation.

Overall, the task of prefix completion is divided in two stages. We provide pseudocode for both of these tasks in Figure 4.4.

The first stage is described in the `non_det_traversal` function in Figure 4.4. In this first stage, we non-deterministically traverse the Enumerator graph by starting on the *Root* node and iteratively moving to the appropriate adjacent node or nodes for each character c in the prefix. The edges of the Enumerator graph not only encode constraints on that current character c, but they also specify rules to handle an additional structure Γ_V which is used to hold the current token as if we were parsing the code with a lexer. For example, if our prefix is `"return "` then we would first

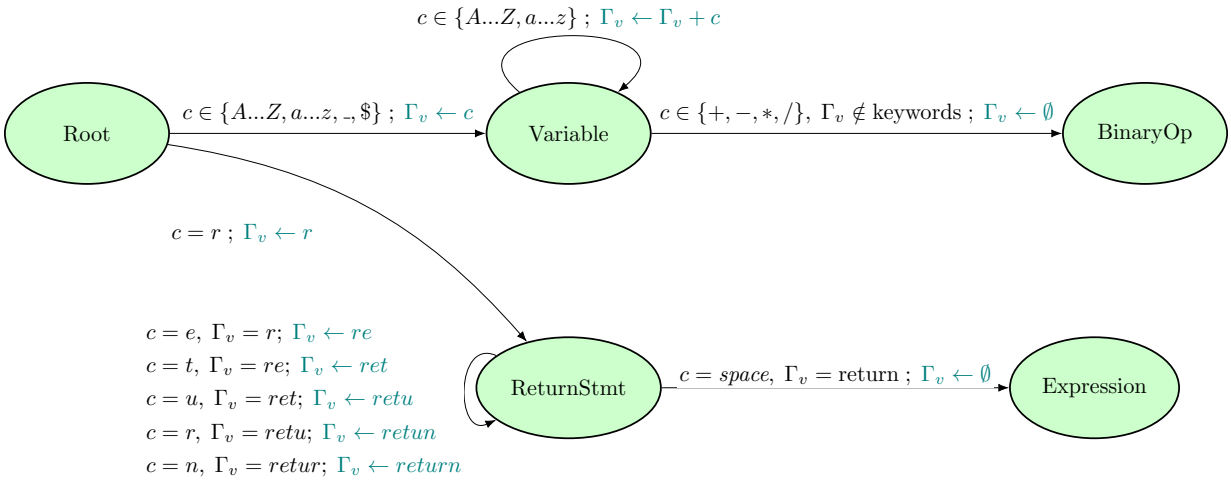

Figure 4.3: A section of the graph representation of JavaScript syntax used by the Enumerator. Edges have labels C; U where C is a condition over the current character in the prefix c and the context Γ_V. U is a context update colored in teal. Node *ReturnStmt* has 5 edges connecting it to itself, which we collapsed on a single edge with 5 labels. The term *keywords* refers to the set of reserved keywords in Node.js.

transition from *Root* to the *ReturnStmt* node, where we would stay reading characters from the prefix until the `return` token is fully seen, and we would finally end this traversal in the *Expression* node.

In another scenario, context Γ_V can, when we exit the Variable node, hold the actual variable name, which is important to encode certain constraints in JavaScript syntax. For example, Node.js forbids declaring a variable called `function`, as that is a reserved keyword. Thus, we forbid in our graph the possibility of transitioning out of the *Variable* node, if the currently parsed variable name is one among a list of keywords. This can be done using the following constraint on the current context, in edges leaving the *Variable* node: $\Gamma_V \notin$ keywords. Once all characters in the prefix are seen, our non-deterministic graph traversal will result in a list of possible *end nodes*. This list of end nodes is the output of the first stage of our prefix completion algorithm.

The second stage is responsible for synthesizing what comes after the prefix. For each end node, we find a shortest path back to *Root*, which will give us a syntactically valid completion, as long as the graph is well constructed. During that path, or at the limit, at the *Root* node in the path's end, we may run into nodes that allow execution of arbitrary statements and this is where we will try to inject the attacker payload. For example, if on our way back to the *Root* node we encounter an *Expression* node, the Enumerator can then include a placeholder for the attacker payload. This is because *Expression* nodes represent situations where any JavaScript expression would be expected, and it will be executed successfully as long as the whole code is syntactically valid.

In summary, for each node along the path, one of possibly many completions can be iteratively constructed by concatenating each node *value*, or, for special nodes, we can insert a placeholder for exploits. A node value is a value that would be allowed during non-deterministic traversal and would cause a transition to that node, from the previous node in the path. This

```
def non_det_traversal(graph, prefix):
    end_nodes = set()
    end_nodes.add((empty_context, Root(context)))

    for c in prefix:
        new_nodes = set()
        for context, node in end_nodes:
            for updated_context, next_node in graph.transitions(node, context):
                new_nodes.add((updated_context, next_node))

        end_nodes = new_nodes

    return end_nodes

def complete_prefix(graph, prefix):
    end_nodes = non_det_traversal(graph, prefix)
    for c, node in end_nodes:
        path = graph.shortest_path_to_root(node, c)
        completion = []
        for source, dest, edge, context in path:
            if can_hold_arbitrary_statement(dest):
                completion.append(EXPLOIT_PLACEHOLDER)
            else:
                completion.append(dest.get_value(source, edge, context))
        yield completion
```

Figure 4.4: Pseudocode for Enumerator's prefix completion

can be efficiently computed from a source node (i.e., previous node in the path), the destination node (i.e., current node under consideration), the context Γ_V and the edge that was followed, by ensuring that all edge context constraints are satisfied and by choosing one prefix character that satisfies prefix constraints for that edge. Thus, even one path could theoretically be used to construct multiple completions for a prefix, not just one, corresponding to all possible prefix character values that satisfy prefix constraints on the edge that was followed.

4.3.4 Integration of the Objective Payload in the Confirmation Methodology

This graph does not directly help us find payloads that bypass sanitization or other constraints on the input, but by properly designing and non-deterministically traversing it we have a way to search over multiple payloads that are at least syntactically correct. Suffice to say that when NODEMEDIC-FINE attempts to synthesize an exploit for ACE vulnerabilities, it takes the completions of the Enumerator and integrates them in a SMT statement that is passed to Z3. This SMT statement considers the placeholders for the attacker payload to be symbolic and encodes the sanitization constraints that were missing earlier while using the advantages of the Enumerator in building syntactically valid payloads.

As a minimal example of Enumerator's effect on synthesis, consider a vulnerable entry point that takes an argument `x` and executes `eval("var f = function "+ x + "(){}")`, dangerously allowing a user to choose a random name for a function. The entry point makes no effort to sanitize user input or block it from executing other code. To produce an exploit, the original NODEMEDIC attempts to synthesize a concrete `x` that embeds the payload `global.CTF();//` to call the `global.CTF` function. This fails with a SyntaxError, since `var f = function global.CTF();//(){}` is

```
1 [[<literal: '(){}+'>, <payload>]]
```

Figure 4.5: An example template produced by the Enumerator.

```
1 (declare-fun x () String)
2 (declare-fun fresh () String)
3 (define-fun payload () String "global.CTF();//")
4 (assert (= x (str.++ "(){}+" fresh)))
5 (assert (str.contains fresh payload))
6 (check-sat)
7 (get-model)
```

Figure 4.6: SMT-LIB2 encoding of a synthesis constraint using the enumerator template in Figure 4.5.

not valid JavaScript. With the Enumerator, the prefix is completed with an Enumerator template, containing a placeholder for the attacker payload. Figure 4.5 shows the template produced by the Enumerator in this case. Overall, the completion specifies that the attacker input should start with a constant literal `(){}`, followed by an arbitrary JavaScript statement. This lets the synthesis algorithm assert `x = "(){}+"+ freshvar` and inject the payload into `freshvar`. Figure 4.6 shows the synthesized SMT-LIB2 program that uses the Enumerator template. By solving it, Z3 produces a model that makes the final payload syntactically valid and run the intended statement: `var f = function (){}+global.CTF();//(){}`.

4.3.5 Addressing Efficiency Concerns

Since we are non-deterministically traversing a graph, this approach may seem like it requires intractable space usage. On the contrary, we find that the space used during traversal converges rather quickly as it iterates over each character of the prefix. Take the following prefix: `"return 1+"`, which will add 1 to whatever comes after (i.e., the attacker input). During traversal, once the `return` part of the prefix is seen, we are left with two intermediate nodes: *Variable* and *ReturnStmt*. This is because the prefix might be referring to a variable which name starts with `return`, e.g., `return_value`, or it might be the start of a return statement. The Enumerator has not seen the characters that come after `return` in the prefix, so it needs to consider all these possible situations. As soon as the space character between `return` and `1+` is seen by the Enumerator, the traversal converges on a single intermediate state: *ReturnStmt*. This is because variable names can not be reserved keywords, and so the *Variable* node can not transition to any node, including to itself. Thus, *Variable* is removed from the list of intermediate states.

All completed prefixes that were discovered in the wild were successfully completed in well under 1 second, and the space usage of the Enumerator was insignificant compared to the rest of NODEMEDIC-FINE's components. While it remains possible that the Enumerator will eventually find cases that inflate the memory usage to intractable levels, that does not seem to be the common case in npm packages.

4.4 Evaluation

In this section, we evaluate how effective NODEMEDIC-FINE is at uncovering code injection vulnerabilities in Node.js packages that require attacker-controlled inputs with complex types and structure. We then evaluate NODEMEDIC-FINE's effectiveness at confirming vulnerabilities, specifically focusing on how effectively our Enumerator aids the synthesis of working ACE exploits. Finally, we compare NODEMEDIC-FINE with prior DTA tools for Node.js and a static analysis tool with exploit generation capabilities called FAST. In summary, we aim to answer the following research questions:

RQ1: How effective is type and structure-aware fuzzing (Section 4.2) at uncovering potential ACE and ACI flows?

RQ2: How much does the usage of polyglots (Section 4.3.1) and the Enumerator (Section 4.3.2) contribute to the confirmation of ACE flows?

RQ3: How does NODEMEDIC-FINE compare to prior work that detects code injection vulnerabilities?

4.4.1 Experimental setup

Conditions. To evaluate the impact of each novel component in NODEMEDIC-FINE, we performed a series of ablation studies. We now enumerate all variations of NODEMEDIC-FINE that we have evaluated:

1. NODEMEDIC-FINE: Full version of our tool with the Fuzzer, polyglots and Enumerator components.
2. NO-POLYGLOT: Disables the polyglots.
3. NO-ENUMERATOR: Disables the Enumerator.
4. NO-OBJRECON: Disables object reconstruction capabilities in the Fuzzer.
5. NO-TYPES: Only generates string inputs during fuzzing, similarly to prior work on fuzzing JavaScript programs.
6. NODEMEDIC-MC: Baseline. Disables the Fuzzer, Enumerator and the polyglots.

NODEMEDIC-FINE does not just include the novel components that we have integrated in the analysis (the addition of the fuzzer) or the confirmation infrastructure (the polyglots and the Enumerator). It also includes engineering improvements to the original NODEMEDIC tool which significantly impacts results. This includes bug fixes, support for additional SMT models and support for implicit coercion. Our NODEMEDIC-MC condition includes those improvements and we use it as the baseline instead of the original NODEMEDIC to evaluate our novel components.

Datasets. To answer the above research questions, we use the following two datasets: (1) A dataset of real-world packages that we have collected from npm, which we call NPM-DATASET. (2) The 101 ACI and 40 ACE vulnerabilities in the SecBench.js, which is a popular dataset for server-side JavaScript vulnerabilities commonly used in prior work. We use this dataset in our comparison with FAST.

In order to obtain NPM-DATASET (1), we gathered *all* packages from npm with 1 or more weekly downloads, which amounted to 1,732,536 packages in total; Then, we kept only packages that contained calls to sinks that NODEMEDIC-FINE supports (described in Section 8.1.1). That process resulted in our NPM-DATASET dataset: a list of 33,011 packages, with size ranging from 56 bytes to 236 MB, weekly download counts from 1 to 171,158,063 and between 1 and 1366 dependencies. In Section 4.4.2 we describe the gathering process for NPM-DATASET in greater detail.

Runtime details. All experiments were performed on two Ubuntu 20.04 VMs with 12 cores each, using one Docker container per condition and per package analyzed. Packages were analyzed in parallel, each restricted to using 4GB of RAM. We analyze a package with NODEMEDIC-FINE or an ablation variation by following a sequence of steps, split by *analysis* and *confirmation* stages.

With respect to analysis, we first generate a driver (see Section 4.2.2) and then we run it. The driver will keep going until it either times out, crashes or finds a potential flow. The total timeout per package is 5 minutes. We allocate 2 minutes for fuzzing, as justified in Section 8.1.4. Note that the NODEMEDIC-MC condition does not use the fuzzer in this stage. Instead, it follows NODEMEDIC's original analysis methodology that consists of passing a single input to each of the package entry points. If a potential flow is discovered, we move on to the confirmation stage.

To confirm a potential flow, we first test the polyglots (Section 4.3.1) depending on the variant. If the polyglot is unsuccessful or if we are running the NO-POLYGLOT condition, we proceed to the next step, which is to run our synthesis algorithm. The synthesis algorithm includes the Enumerator depending on the condition and will regardless result on a proof-of-concept exploit. We run this exploit and, in order to confirm the potential flow, we validate whether the necessary side effect is observed (Section 4.3).

4.4.2 Gathering of the Evaluation Dataset

Gathering consisted of collecting a list of packages and saving each and their dependencies locally using Verdaccio. This is done to save up bandwidth, as inevitably some packages will share dependencies. From the ($> 2M$) packages in npm at the time, we gathered those that have at least 1 weekly download (1,732,536 packages).

In Table 4.2 we show the number of packages that get filtered out at each stage of the gathering pipeline, until we are left with 33,011 usable packages, our finished dataset. We show all steps of our pipeline in the same order as they run. A package stops at the **setupPackage** if it can not be downloaded; The **filterByMain** stage filters out packages that can not be imported because they do not define a main file; A package stops in the **filterBrowserAPIs** stage when it is not intended for client-side usage as it is the case for the ones that require a browser; The **filterSinks** stage discards packages that do not contain calls to ACE or ACI sinks visible to static analysis (i.e., a grep usage). Note that we also check if any of the dependencies have calls to sinks. We proceed to install the dependencies in the **setupDependencies** stage, which may error if we fail to download or install one of the dependencies; In the **getEntryPoints** stage we discard packages that do not have any public entry points defined; Finally, we gather useful metrics for

Stage	Initial	Discarded	Remaining
setupPackage	1732115	7630	1724485
filterByMain	1724485	411115	1313370
filterBrowserAPIs	1313370	459865	853505
filterSinks	853505	776288	77217
setupDependencies	77217	17173	60044
getEntryPoints	60044	27010	33034
annotateNoInstrument	33034	23	33011
runJalangiBabel	33011	0	33011

Table 4.2: Number of packages discarded at each stage, with initial and remaining counts.

Condition	Extra			Missing			All Potential flows		
	ACI	ACE	Total	ACI	ACE	Total	ACI	ACE	Total
NODEMEDIC-FINE	-	-	-	-	-	-	1788	469	2257
NO-OBJRECON	15	19	34	181	47	228	1622	441	2063
NO-TYPES	12	23	35	306	85	391	1494	407	1901
NODEMEDIC-MC	0	0	0	625	294	919	1163	175	1338

Table 4.3: Potential flows found by the fuzzer with varied configurations. Extra and missing flows are relative to the ones found by NODEMEDIC-FINE.

characterizing the dataset in the **annotateNoInstrument** stage. The last stage is the **runJalangiBabel** where we instrument the package code using Jalangi. We analyze all packages that get through to this last stage. We store the package and dependencies, together with its instrumented counterpart for all popular packages with calls to sinks. A study of the characteristics of this dataset is given in Section 8.1.3.

4.4.3 RQ1: Effectiveness of Type-Aware Fuzzing

In this section we evaluate the fuzzer's impact on identifying potential flows. We launched NODEMEDIC-FINE, NO-OBJRECON, NO-TYPES and NODEMEDIC-MC against the NPM-DATASET dataset and show the results in Table 4.3. Overall, the full fuzzer performs much better than not using the fuzzer at all, resulting in 919 additional potential flows. These are flows that benefited from exploring more execution paths with the fuzzer.

Type-awareness in the fuzzer is responsible for finding 391 extra potential flows compared to a version of our fuzzer that only generates strings. Anecdotally, we found that disabling generation of inputs of all types except strings makes the fuzzer faster at finding flows that only require strings, which explains the 35 extra flows. We validated that most of those flows would have been found by the normal fuzzer given a sufficiently longer timeout, except for 5 that crash due to out of memory.

However, object reconstruction helped find 228 extra potential flows. These were cases where

Type disabled	# Potential flows missed
Strings	609
Objects	243
Arrays	62
Functions	34
Numbers	25
Regexes	24
Booleans	20
BigInts	19
Nulls	17
Undefined	16
Symbols	16
Total	1085

Table 4.4: Potential flows missed by the fuzzer when we prevent it from generating inputs of a given type.

packages required inputs to have a certain structure. Once again, disabling this capability makes the fuzzer faster at finding flows in packages that do not require complex objects, explaining the 34 extra flows. We validated that 26 out of these 34 extra flows are found by the full fuzzer as long as the timeout is sufficiently increased, but the remaining 8 of the 34 flows crash due to out of memory. During further analysis of these 8 cases, we have also observed that in at least 2 of them the object reconstruction capability was leading the fuzzer away from inputs that would trigger the potential flow. In one such package, having a specific attribute in one of the entry point's arguments resulted in an error instead of a sink call. During that execution, our modified instrumentation sees that a `GetField` operation is performed on a tainted input and the object reconstruction feature of the fuzzer will take that into consideration and start generating inputs that are likely (though not guaranteed) to have that attribute. Thus, even though we believe that given sufficiently long timeout and sufficient memory all 34 flows could be discovered using the full fuzzer, there are situations where the object reconstruction ability actually hinders flow detection and it could be worth it to run a version of the fuzzer that disables object reconstruction.

Result 1a: Type- and object-structure aware fuzzing uncovers 2257 potential flows; 1.7x the flows of prior tool NODEMEDIC. Object reconstruction is necessary to find 228 flows. Generating diverse types yields 391 more flows compared to only generating strings.

In the following experiment, we investigate how impactful the generation of each input type is during fuzzing. For that, we launch variations of NODEMEDIC-FINE's fuzzer disabling its ability to generate inputs of a given type. We tested it against the set of packages where we previously discovered a potential flow and summarize the results in Table 4.4. Rows in the table indicate input types that the fuzzer is no longer able to generate and the respective number of potential flows that were no longer found.

Overall, the majority of flows can be triggered by more than a single input type, which is why the total goes up only to 1085, and not 2257 (i.e., the total potential flows that we have previously discovered). This highlights how loosely typed JavaScript is and how that impacts

fuzzing to detect ACI and ACE flows. Furthermore, Strings and Objects clearly seem to be the most important types that our fuzzer supports, or else we would have not discovered 609 and 243 flows respectively.

The contribution of the support for each of the other types ends up amounting to a significant number of flows discovered. For example, Node.js package entry points often receive a callback function as an argument, which can be called when an error happens during execution of that function, among other purposes. A subset of those packages really check whether the above argument is of type `function`, and a sink is called only after that check is successfull. Thus, potential flows can be missed on those packages if the fuzzer is unable to generate function types.

Result 1b: The support for a variety of types is important in the fuzzer for ACI and ACE flow detection, especially for `Strings`, `Objects` and `Arrays` types.

4.4.4 RQ2: Effectiveness of Polyglots and Enumerator for ACE Confirmation

In this section, we evaluate the impact of using polyglots and the inclusion of the Enumerator in the exploit synthesis pipeline of ACE vulnerabilities. We summarize in Table 4.5 the results of running an ablation study on NODEMEDIC-FINE's ACE confirmation components: NODEMEDIC-FINE, NO-ENUMERATOR and NO-POLYGLOT. For each condition, we report the number of extra, missing and total number of ACE potential flows that were confirmed using that condition. We now separately discuss the impact of the polyglots and the Enumerator components.

ACE polyglot. When confirming ACE potential flows, our NO-POLYGLOT condition reverts back to using the same pre-synthesis algorithm of the original NODEMEDIC: it first attempts a simple exploit `global.CTF();//` to avoid spending time performing synthesis on cases that can be exploited trivially. We improved that using our ACE polyglot (as described in Section 4.3.1) and evaluate its impact by comparing NO-POLYGLOT with NODEMEDIC-FINE which uses the new polyglot. The only difference between NO-POLYGLOT and NODEMEDIC-FINE is that NO-POLYGLOT disables the polyglot usage. Overall, not using the polyglot decreases the number of confirmed flows from 154 to 128. These improvements happen mostly in packages where the payload is inside a single or double quoted string context and at the same time the attacker input is not sufficiently sanitized by the package.

Result 2a: Our ACE polyglot was necessary to confirm 26 potential flows, increasing ACE confirmation by 20%.

Enumerator effectiveness in completing prefixes and confirming ACE flows. We include a breakdown of Enumerator results in Figure 4.7. Overall, the Enumerator was called by NODEMEDIC-FINE to complete 328 prefixes (205 unique) and came up with a valid prefix completion for 191 (58%) of those cases. The Enumerator was used to successfully confirm 38 ACE flows in NODEMEDIC-FINE, although it should be noted that the Enumerator is a completely necessary component for 27 cases, according to Table 4.5. We verified that if the Enumerator is disabled,

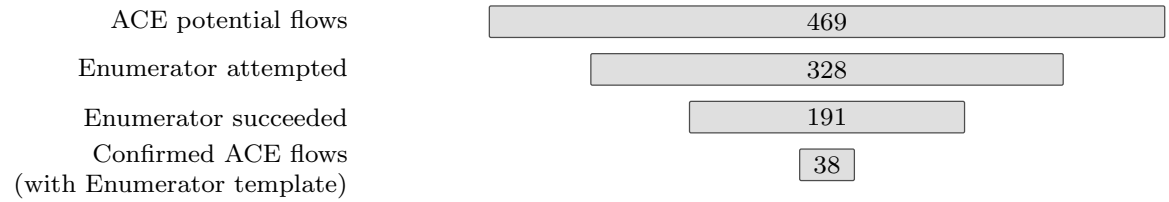

Figure 4.7: How many flows were processed by the Enumerator and how many were successfully exploited. We consider that Enumerator is successfull when it provides a correct completion to the given prefix.

Condition	Extra	Missing	Total confirmed ACE
NODEMEDIC-FINE	-	-	154
NO-ENUMERATOR	0	27	127
NO-POLYGLOT	0	26	128

Table 4.5: ACE confirmed flows found by the fuzzer with and without the Enumerator. These conditions only differ in the confirmation methodology; All conditions are attempting to confirm the same 469 potential ACE flows discovered by NODEMEDIC-FINE's analysis.

NODEMEDIC-FINE can not automatically exploit those 27 ACE flows. Note that these are necessarily cases where the ACE polyglot is not sufficient, since synthesis is only performed when polyglots fail. Instead, all 27 flows required the construction of a complex payload, which had to be injected in the right place and escape the necessary contexts appropriately. An example of one of these flows is given in Section 8.1.2.

Result 2b: The Enumerator helped NODEMEDIC-FINE complete the majority of real-world prefixes that we found in our experiments, increasing the number of total confirmed ACE flows by 21%

Manual analysis of flows where Enumerator was insufficient. There were 153 (191-38) packages that our Enumerator managed to complete the prefix, but NODEMEDIC-FINE still could not automatically exploit. We now manually investigate a random sample of 8 of those 153 cases. Four of these cases were not exploitable at all, representing false positives of the potential flow detection engine. Of the remaining four cases, two had very complex constraints which caused Z3 to timeout; 1 case used the `slice` operation with a symbolic value for the length, for which we did not implement a model; the last case required an extra step to get code execution. In that last case, the entry point returned a function that had to be called with an object argument in order for the attacker payload to be triggered. Regardless, in all these cases the Enumerator synthesized a valid completion for the given prefix, but NODEMEDIC-FINE needed to overcome additional challenges to successfully synthesize a working exploit.

Limitations of the Enumerator. There are also flows where our Enumerator did not help at all. In 137 packages with ACE potential flows, our Enumerator was unable to complete the prefix. This can happen mostly when the prefix contains JavaScript primitives that are not supported by our Enumerator. For example, 63 of these 137 failures were due to the lack of support for loops, nested objects, boolean expressions or the `+=` operator in the Enumerator's graph. In another 62 cases the Enumerator failed to support a diverse set of other JavaScript primitives, including but not limited to class definitions, try/catch statements and generator functions. In the remaining 12 cases, the prefix that was given to the Enumerator was impossible to complete. These last cases originated in a limitation of NODEMEDIC and NODEMEDIC-FINE of handling flows requiring manipulation of multiple attacker-controlled arguments to trigger the vulnerability [23].

4.4.5 RQ3: Comparison with prior work

In Table 4.6 we show a comparison between NODEMEDIC-FINE[1] and prior tools for code injection vulnerability detection NODEMEDIC [24], Ichnaea [61] and AFFOGATO [38]. The table also indicates the results for NODEMEDIC-MC, which includes small improvements on NODEMEDIC infrastructure that significantly impacted results but not our main novel components. As metrics, we use the potential flows discovered by each tool — separated by vulnerability type ACE versus ACI — and the confirmed flows that were automatically synthesized by each tool. Ichnaea and AFFOGATO numbers come from the respective reported results, which have limited scale mainly because both tools require the manual creation of a driver to analyze packages and do not support automatic confirmation of flows. Still, to the best of our knowledge, the evaluation performed for NODEMEDIC-FINE is the largest scale experiment in the Node.js ecosystem measuring ACI and ACE flows. NODEMEDIC-FINE discovered 1788 ACI and 469 ACE potential flows, for a total of 2257 flows. With respect to confirmation, NODEMEDIC-FINE successfully created proof-of-concept exploits that demonstrated the exploitability of 612 ACI flows and 154 ACE flows.

Result 3a: NODEMEDIC-FINE discovered 2257 potential flows and automatically confirmed the exploitability of 766 of those flows, in a dataset with 33,011 Node.js packages; This amounts to 1.7x the potential and 1.6x the automatically confirmed flows by NODEMEDIC-MC.

Comparison with FAST. We now compare NODEMEDIC-FINE with FAST [58] and report in Table 4.7 the flows that were discovered by running each tool on the SecBench.js dataset. From the 101 ACI and 40 ACE vulnerabilities in the SecBench.js, we consider a vulnerable package to be *valid* if all the following conditions are satisfied: it fits our attacker model; it is downloadable from a public repository like npm or GitHub; and it has a main executable file defined in the package.json file. Examples of invalid packages include one case that required command-line arguments to be attacker-controlled and, more commonly, cases that are not exploitable from the main package entry points. While there are 91 ACI and 34 ACE packages that we consider valid for this comparison, only 87 ACI and 25 ACE could be installed and run. Regardless, being

[1] Our focus in this thesis is improving ACE detection and confirmation, and ACI detection. Improvements on ACI confirmation were done as part of another thesis [22]

Tool	Year	Packages	Potential			Auto-conf.		
			ACI	ACE	Total	ACI	ACE	Total
NODEMEDIC-FINE	2025	33011	1788	469	**2257**	612	154	**766**
NODEMEDIC-MC	2025	33011	1163	175	1338	396	67	463
NODEMEDIC [24]	2023	10000	133	22	155	102	6	108
Ichnaea [61]	2018	22	9	6	15	-	-	-
AFFOGATO [38]	2018	21	-	-	17	-	-	-
Explode.js [80]	2025	32137*	151	45	196	51	2	53

Table 4.6: Overall evaluation results and comparison to other Node.js dynamic taint analysis tools.
* Packages in the dataset did not necessarily have sink calls.

executable is not a prerequisite for FAST to find potential flows, so we report all results of FAST on valid packages, not just executable packages. However, NODEMEDIC-FINE is a dynamic analysis tool and therefore needs to execute a package to analyze it.

As can be seen in Table 4.7, FAST finds a superior number of potential vulnerabilities, which is expected considering FAST is a static analysis tool and does not have to come up with an input that follows the potentially vulnerable path to signal the vulnerability. There were 3 cases where FAST found a potential flow but NODEMEDIC-FINE did not due to an undertainting issue. Still, NODEMEDIC-FINE can use dynamic information to synthesize exploits, which allowed it to perform well with respect to confirmed flows. When we focus on ACE potential flows, NODEMEDIC-FINE performs especially well. FAST generated candidate exploits for 4 ACE flows that did not work because the final argument to the sink ended up not being valid JavaScript. In those 4 cases, the Enumerator was critical to handle that challenging constraint and came up with a working completion for all respective prefixes. Another example was the package `macaddress@0.2.8`, for which the vulnerable entry point required one of the arguments to be a function. FAST failed to synthesize the right type for that argument, while NODEMEDIC-FINE's fuzzer eventually generated a function input for that argument and was able to create a working proof-of-concept exploit.

Result 3b: NODEMEDIC-FINE exclusively synthesized 5 working proof-of-concept exploits for ACE vulnerabilities with the help of Enumerator.

4.4.6 Responsible disclosure

Within the 766 confirmed flows found by NODEMEDIC-FINE, 1 ACE and 25 ACI were already known vulnerabilities. We have responsibly disclosed all vulnerabilities in stages. We have first triaged 567 ACI and 55 ACE confirmed flows, for a total of 622 confirmed flows. Within those, we have manually validated that 270 ACI and 19 ACE were true positives. Besides exploitability, a true positive is an actual vulnerability. This excludes cases of packages that simply have wrappers around `exec` for example, as executing arbitrary commands is part of the legitimate functionality of the package. In this triage batch, we received responses for 50 ACI and 6 ACE

Type	Valid	Executable	Potential		Confirmed	
			NODEMEDIC-FINE	FAST	NODEMEDIC-FINE	FAST
ACI	91	87	51	65	44	41
ACE	34	25	17	10	5	0
Total	125	112	68	75	49	41

Table 4.7: SecBench.js eval results comparing NODEMEDIC-FINE with FAST in terms of potential and confirmed flows. Valid packages are downloadable, have a main executable file defined, and the vulnerability fits in the attacker model that we share with FAST. Executable are packages that are valid and can be installed and run.

vulnerabilities. Two developers disagreed that the respective reported ACI vulnerability was real because the attacker model did not apply to them; they considered that an attacker could not possibly control the entry point's arguments. The remaining 54 developers agreed that the vulnerabilities were real. So far, 35 of these cases were patched and a new version of the package has since been published.

4.4.7 Exploring Precision in NODEMEDIC-FINE

One might expect that a system with automatic confirmation of vulnerabilities has no false positives. While that certainly is not true for FAST, since FAST exploits are not tested and can therefore not function correctly, it is not true for NODEMEDIC-FINE either. In reality, true positives correspond to flows that not only are exploitable but also must represent illegitimate behavior according to the package functionality. Identifying which potential flows are true positives provides an indication of the precision of our analysis infrastructure, including the Fuzzer and the taint instrumentation. Using the metric of weekly downloads, we sorted the list of packages by popularity and manually validated whether the 29 unconfirmed potential flows and 113 confirmed flows in the most popular packages were truly vulnerable. We summarize those results in Table 4.8.

False positives in confirmed flows. Overall, 70 out of the 113 flows are truly vulnerable (62%). In two cases with false positives, packages had a warning in the documentation pointing out that users should not pass unsanitized inputs to a certain entry point. While we have considered confirmed flows in those two cases to be false positives, we study in Section 6 whether dependents of such packages heed such warnings. Two other false positives were deprecated packages. The remaining 39 false positives were packages that either exposed a sink directly as one of their entry points, or the entry point that we found to be vulnerable was legitimately intended for arbitrary command execution.

True positives in confirmed flows. With respect to true positives, most are vulnerable due to a lack of sanitization of user inputs. Out of 54 vulnerabilities acknowledged by developers, 10 have between 100 and 3000 weekly downloads and 10 have more than 3000 weekly downloads.

	Confirmed			Not confirmed		
Sink Type	**TP**	**FP**	**Total analyzed**	**TP**	**FP**	**Total analyzed**
ACI	64 (50)	40	104	0	14	14
ACE	6 (6)	3	9	4 (3)	11	15

Table 4.8: A true positive is a flow that is manually validated to be a vulnerability. False positives include flows (potential or confirmed), in entry points that are actually just wrappers around ACI or ACE sinks. This table reports the true and false positives for both confirmed flows and potential flows that NODEMEDIC-FINE fails to confirm on NPM-DATASET. Numbers inside parenthesis represent how many of the true positives were previously unreported vulnerabilities.

We submitted packages in this last group to Snyk and so far, we were assigned 1 high severity CVE [8].

True positives in unconfirmed flows. Among the 4 unconfirmed ACE vulnerabilities, 1 requires a more complex exploit driver with multiple interactions with the API to exploit the flow; Another has complex SMT constraints and Z3 outputs `unknown`; The 2 others needed the Enumerator to support more JavaScript syntax: class definitions and passing object arguments in function calls.

False positives in unconfirmed flows. Out of 11 ACE false positives, 1 resulted from overtainting; 5 from proper sanitization; 2 packages were deprecated; 1 package called the `new Function` sink but then it did not call the resulting function, which means that the attacker input could never be triggered; The remaining 2 cases were packages where the attacker did not have sufficient control to execute arbitrary code, as it only controlled boolean and number inputs. Similarly, there was 1 ACI false positive due to overtainting; 1 with insufficient attacker control (only an attacker-controlled integer is processed), and all other ACI false positives result from the need to pass the `shell` flag in spawn sinks to execute arbitrary commands but the attacker only has control of the command argument to `spawn`.

4.5 Limitations and Future Work

We now discuss limitations of NODEMEDIC-FINE and future work to improve its techniques.

4.5.1 More Complex Drivers

The drivers and exploits generated by NODEMEDIC-FINE are limited to producing, per entry point, a simple invocation of that entry point with tainted arguments. This can miss vulnerabilities requiring a chain of interactions between different package API calls, handlers or even external interactions like using databases, the file system or network communication. While prior work on JavaScript taint analysis has encountered similar limitations [38, 61, 102, 103], Explode.js is a recent work, published after NODEMEDIC-FINE, that addresses some of these

limitations. Explode.js computes what they call a set of *vulnerable interaction schemes* (VISes), which are chains of calls (together with the types of the respective function arguments) to entry points or other functions returned by previous calls. While Explode.js performs symbolic execution on the VISes, we see no reason why NODEMEDIC-FINE could not integrate these in the fuzzing process instead. Regardless, this limitation is still not completely addressed by Explode.js, which only considers linear VISes, i.e., those where each call in the chain is either a call to an entry point or to a function returned by the previous call in the chain. Furthermore, interactions with the network, databases or the file system in-between calls to entry points are not considered.

4.5.2 Multiple Flows in the Same Package

Once NODEMEDIC-FINE finds a first potential flow, it moves on to confirmation. If confirmation fails, NODEMEDIC-FINE just stops. One example where this misses real vulnerabilities in a package, is if the package has multiple flows but the easiest flow to find is not exploitable. This is not a fundamental limitation of NODEMEDIC-FINE, as in Chapter 5 we describe further work on NODEMEDIC-FINE that implements this functionality.

4.5.3 Enumerator: completing prefixes with multiple lines

Our Enumerator can not currently handle ACE sink calls that concatenate a multi-line string to the user input. Such sink calls might look like this: `eval("prefix "+ userinput + "\n + 1")`. The Enumerator can not simply put a comment character "//" after the payload, as it will not cause the following statement to be ignored, since it is in a newline. The attacker can not insert a multiline comment either, as even if the payload ends with "/*", there is no way to insert the "*/" terminating comment token at the end of the `eval` argument. The solution here seems to be to have the Enumerator synthesize completions that are compatible with the suffix. We think it is feasible to adapt our Enumerator approach to do this, but leave it for future work.

4.6 Conclusions

By leveraging type and object-structure information gathered at runtime, NODEMEDIC-FINE is able to explore more execution paths and consequently identify more potential flows. With respect to confirmed flows, the Enumerator helped significantly improve the number of successful exploits synthesized for ACE vulnerabilities, a particularly challenging vulnerability type to exploit due to its syntactic requirements. Still, 42% of prefixes could not be completed by our Enumerator. In Chapter 5 we study how such an exploration from the fuzzer can be optimized, not only by following hints from unexploited potential flows, but also by prioritizing paths that have sink calls with easier prefixes to complete.

Chapter 5

Confirmation-Aware Analysis

In this chapter we describe our work on guiding NODEMEDIC-FINE's fuzzing engine towards paths that are more likely for the synthesis engine to generate an exploit.

5.1 Overview

In previous chapters, we examined how to integrate dynamic taint analysis (DTA) with program exploration techniques to identify code injection flows in JavaScript applications. The fuzzing engines of both NODEMEDIC-FINE and SWIPE are coverage-guided, meaning that they aim to increase the amount of JavaScript code executed as they explore the application. Coverage has been widely used by fuzzers (e.g., AFL [135] and Honggfuzz [40]) as an important metric to measure progress during program exploration to find vulnerabilities. However, even with large timeouts, fuzzers may never achieve full coverage, especially for large and complex npm packages. Given limited analysis time, selecting which parts of an application to prioritize becomes essential.

Our main goal in this chapter is to guide fuzzing towards program paths that are more likely to be *automatically exploitable*, i.e., the discovered potential flow can be confirmed by the synthesis engine. We refer to this strategy as *confirmation-aware analysis*. This idea is related to earlier heuristics, such as Thanassis et al.'s buggy-path-first approach [120] for prioritizing symbolic execution paths. The challenge here is that it remains unclear which path features best predict automatic exploitability in npm packages.

To investigate this, we sampled 63 packages from a dataset of total 630 vulnerable packages where NODEMEDIC-FINE detected a Arbitrary Command Injection (ACI) or Arbitrary Code Execution (ACE) potential flow but failed to confirm it. We identified five features of NODEMEDIC-FINE's provenance tree that correlate with automatic exploitability (sink type, flow presence, ACE prefix length, path complexity and attacker input amount).

We leverage logistic regression models to combine these features into a single exploitability metric (EM) to guide fuzzing and evaluate whether it increases the number of packages with confirmed flows, but also importantly, whether it preserves or improves the total number of packages with detected potential flows.

To evaluate EM-guided fuzzing, we performed an ablation study against the same 33,021 packages with sinks as analyzed in NODEMEDIC-FINE's dataset, and the same 5 minute timeout per package. Our evaluation shows that using EM-guided fuzzing improves the number of confirmed flows detected by 1% (from 803 without EM to 812), while perserving the number of packages with discovered potential flows (from 2314 without EM to 2319). Furthermore, we use the exploitability metric to sort the flows by their exploitability estimate before confirmation. The easier exploitability difficulty for flows discovered during EM-guided fuzzing combined with sorting the flows by their exploitability estimate improves confirmation efficiency. Concretely, that strategy reduces the number of rounds needed to discover a first confirmed flow (Welch t=2.98, p=0.001), only requiring 89% of the rounds needed by the baseline of not using the exploitability metric with the same experimental setup as above. While the overall confirmation time savings are not significant (94% of the original time required by the baseline of not using the exploitability metric when looking at the set of packages with confirmed flows found in common, t=0.68, p=0.25) they become significant when the time budget per package is increased to 1800 seconds, with 1080 seconds allocated for fuzzing (EM used 72% of the original confirmation time used by the baseline, t=2.68, p=0.004).

Finally, we discuss limitations, with a focus on the possibility that entry point prioritization may overlook exploitable paths that the model underestimates. We describe mitigations and directions for future work in Section 5.5.

5.2 Confirmation-Aware Analysis

In this section, we detail how we guide fuzzing towards paths that seem more exploitable. First, in Section 5.2.1 we provide an overview of the changes in NODEMEDIC-FINE pipeline, compared to our original work described on Chapter 4. Then, in Section 5.2.2 we motivate the features of provenance trees that we are going to use as a signal of exploitability. Finally, in Section 5.2.3 we rigorously specify the process of assigning an exploitability value to package entry points and explain how we combine the features into a single exploitability metric that can be used to adjust the probability of selecting each package entry point for analysis in a fuzzing round, depending on their exploitability estimate.

5.2.1 Iterative NODEMEDIC-FINE Pipeline

In this section, we describe the pipeline of NodeMedic-FINE-2025, which is an updated version of NODEMEDIC-FINE. In addition to bug fixing and better handling of asynchronous entry points, the analysis and confirmation pipelines have improved compared to what was described in Chapter 4. The most meaningful change was the implementation of a circular pipeline. Previously, NODEMEDIC-FINE would stop analysis at the first discovered potential flow and then it would attempt to confirm that flow. Regardless of the confirmation being successful or not, NODEMEDIC-FINE would stop after testing the produced candidate exploit, if any. Instead, NodeMedic-FINE-2025 keeps fuzzing for a fixed amount of time to allow discovery of multiple potential flows.

It is not always the case that the first input passed to an entry point results in a potential flow. The longer the timeout, the more likely it is that the fuzzer will discover (multiple) potential flows. This is because some entry points require complex inputs to reach vulnerable code paths, and fuzzing may take time to generate such inputs.

Once the fuzzing time budget is exhausted, NodeMedic-FINE-2025 attempts to confirm the first discovered potential flow, and what happens next depends on the success of the candidate proof-of-concept exploit that was produced:

1. If NodeMedic-FINE-2025 is successful at confirming the current flow, it stops
2. If the allocated time budget is exhausted for the current package, it stops
3. Otherwise, NodeMedic-FINE-2025 fails to confirm the current flow and goes back to the start of the confirmation pipeline to test the next potential flow.

The ability to detect and attempt to confirm multiple flows is necessary for the successful integration of an exploitability metric to guide analysis and confirmation, as we aim to more quickly find an exploitable flow but the first tested candidate flow may not always work.

5.2.2 Feature Design

NODEMEDIC-FINE's operation tree, also called provenance tree, is a recursive structure specifying what operations were performed on sink arguments until they reached the sink. This section describes the features we selected to estimate the exploitability of a provenance tree.

While NodeMedic [24] previously addressed the problem of assigning exploitability values to provenance trees, their approach relies on stochastic sampling, which is prohibitively expensive to integrate into our fuzzing engine. Furthermore, their method was designed to rank flows for manual inspection by human analysts rather than for use in an automated confirmation engine. It therefore fails to account for factors such as the varying difficulty faced by the SMT solver when handling different operations.

As discussed in Chapter 4, many potential flows reported by NODEMEDIC-FINE for Node.js were not automatically confirmed (see Table 4.1). However, failure to automatically confirm a flow does not imply it is unexploitable. For the set of potential flows that NODEMEDIC-FINE did not confirm, manual verification found that 47% (521/1106) of ACI and 62% (149/239) of ACE flows were actually exploitable. Typical reasons why NODEMEDIC-FINE fails to produce a successful proof-of-concept exploit include:

1. The package is indeed not exploitable.
2. Missing support for a program operation when building the SMT formula.
3. The SMT solver (i.e., Z3) cannot solve the constraints, often due to complex operations on tainted variables, for example, regex-based primitives.
4. Insufficient attacker control at the sink call. For example, some program paths may only allow a single byte of attacker input to go into the sink. Another example is that in order for the `spawn` sink to be exploitable, it requires the `shell` argument to be set to `true`. This attribute might not be attacker-controlled in the originally discovered potential flow.
5. Failure to synthesize a syntactically valid completion for ACE prefixes (see Section 4.4.4).

Manual analysis of unconfirmed potential flows. To identify features that signal exploitability, we manually analyzed a stratified sample of 63 packages drawn from the 630 vulnerable packages where NODEMEDIC-FINE reported a potential flow but failed to confirm it. Stratification considered vulnerability type and package size. For each package, we inspected the discovered potential flow and the inputs that triggered it. We then compared the flow discovered by NODEMEDIC-FINE with the flow that was exploitable, with special focus on the case where those flows differ. A flow differs from another if their respective program traces differ, i.e., they execute different program instructions or in a different order, which may impact exploitability even if both traces start from the same entry point and reach the same sink call.

Our goal was to not only define flow properties that contribute to ease of automatic exploitability, but also ascertain whether NODEMEDIC-FINE can trigger a different, exploitable flow, by mutating the original inputs that triggered the non-exploitable flows which exhibited such properties.

1. In 17.5% (11/63) of cases, the discovered flow is not exploitable. Among these:
   (a) In 81.8% (9/11) of flows, the exploitable path is in the same entry point as the original flow, and the input that triggers it can be derived from the original input. For example, in some cases, an exploitable flow can be triggered by adding a single extra attribute to the object input that triggered the originally discovered flow. One case required calling an additional entry point after the original entry point to trigger the attacker-controlled code.
   (b) In 18.2% (2/11) of flows, the exploitable path is in a different entry point.
2. In 82.5% (52/63) of cases, the discovered flow was directly exploitable. Among these:
   (a) 38.5% (20/52) involved the `spawn` sink, which requires `shell=true` to be exploitable. While NODEMEDIC-FINE confirmation could be adapted to support the synthesis of this attribute, these flows are not under focus in this work, as they typically do not require a different exploration strategy.
   (b) In 19.2% (10/52) of potential flows, there is a different, simpler exploitable path which input can be derived from the original. When comparing to the originally discovered paths, we refer to *simpler paths* as those where either the constraints are easier for Z3 to handle or the ACE prefix, if any, is easier to complete. This includes an overlapping case with the above item, where the exploitable path already passes the parameter `shell=true` in the `spawn` call, which simplifies the final exploit.
   (c) The remaining 44.2% (23/52) of cases were not automatically confirmed due to unsupported operations (missing symbolic models) or complex constraints (for example, requiring the payload to match a real filename).

Example of an exploitable potential flow which inputs can be derived from the originally discovered flow's. Figure 5.1 shows a simplified excerpt from an exploitable entry point in the popular `ejs` template engine [32]. The originally discovered potential flow calls a sink with a value sanitized via `JSON.stringify`. A small change to the `options` argument (adding `debug`, processed in lines 3-5) causes the sink to be invoked with unsanitized input and thus becomes exploitable.

Although this flow is technically exploitable, we classify it as a false positive because the `ejs`

```
1 module.exports.compile = function compile(str, options){
2   var argument = JSON.stringify(str);
3   if (options.debug){
4     argument = str;
5   }
6   var fn = new Function(argument);
7   return fn.call(this);
8 }
```

Figure 5.1: Simplified excerpt of a vulnerable entry point (`Template.compile`) in the latest version of ejs – version 3.1.10 at the time of writing.

documentation delegates responsibility of input sanitization to its dependents. Still, we use `ejs` as an illustrative example here because it does not require anonymization and one of the manually analyzed packages reuse the vulnerable part of ejs code, while failing to provide a matching security warning in their documentation. This case also typifies a broader pattern observed across some of the 63 packages that we manually analyzed: minimal changes to flow-triggering inputs discovered by NODEMEDIC-FINE fuzzer may result in confirmed flows, even when the original flows are not directly exploitable.

High-level feature description. Given the results of the manual analysis, especially on the potential flows in groups 1.(a) and 2.(b), we designed the following features as potential signals of fuzzing progress toward automatically exploitable paths:

- $f_{sinkcall}$: Which sink, if any, is being called in the operation tree. All cases of exploitable potential flows that are related to the originally discovered potential flows use the same sink but different inputs. Importantly, some sinks are harder to exploit than others: `spawn` implies the additional constraint of setting the parameter `shell=true` in the sink call; `eval` and the `Function` constructor both require the final argument to be syntactically correct, but we also noticed that prefixes for `eval` are usually simpler than those used in `Function` sink calls.
- $f_{taintedarg}$: Whether one of the sink arguments is tainted. Even though the discovered flow may not be directly exploitable, or the tainted argument is not the necessary one for exploitation, there might be a simpler, exploitable path that is reachable via small differences in the input (e.g., an additional attribute in object arguments).
- $f_{prefixlength}$: The length of the prefix that needs to be completed in the case of ACE flows. In 3 out of 63 packages, there was another exploitable path that needed a smaller, less complex ACE prefix to be completed to synthesize the final payload, compared with the path traversed by the originally discovered flow. Again, that exploitable path could be reached by introducing small differences in the input that triggered the original potential flow.
- $f_{opcomplexity}$: The complexity of the operations performed on tainted variables until the sink is reached. The effectiveness of FAST, NODEMEDIC-FINE and Explode.js's confirmation pipelines is limited by the effectiveness of the SMT solver in use (Z3 in all three tools). By confirming flows involving simpler constraints we aim to unburden the SMT solver in use, and increase the likelihood of producing successful exploits.

```
start_time = time()
while (time() - start_time < TIME_BUDGET){
  entrypoint = fuzzer.select_entrypoint(target_package);
  inputs = [];
  for (argument in entrypoint.arguments()) {
    input = fuzzer.get_input(entrypoint, argument);
    taint_infrastructure.set_taint(input);
    inputs.append(input);
  }
  target_package.call_entrypoint(*inputs);
  if (taint_infrastructure.flow_found()){
    save_flow();
  }
  var op = taint_infrastructure.get_operation_tree();
  fuzzer.estimate_exploitability(entrypoint, op);
  for (input in inputs){
    feedback = taint_infrastructure.structure_feedback(input);
    fuzzer.refine_specification(entrypoint, argument, feedback);
    taint_infrastructure.remove_taint(input);
  }
}
```

Figure 5.2: Pseudocode for our improved driver component. It allows selection of arbitrary entry points; does not stop at the first discovered potential flow; computes an exploitability estimate per entry point based on the operation tree generated by the execution of each fuzzing input.

- $f_{atkdata}$: The amount of unsanitized attacker-controlled data in the final sink argument. In 4 out of 63 packages, there was a exploitable path (again, related to the original) that allowed more attacker-controlled input to be passed to the sink. Additionally, in 3 out of 63 packages, a simpler path existed in which the attacker input was not sanitized.

We will rigorously specify these features in the next section, and in Section 5.3 we evaluate the hypothesis that these features generalize to other packages as signals indicating exploitability of an operation tree.

Ideally, a dynamic analysis tool for ACI and ACE detection should maximize both the number of confirmed flows (automatically synthesized exploits) and the number of true positive potential flows. We show in Section 5.3 that, by following paths optimizing these features, we are not losing more potential flows that we are gaining, which is an important concern to address when optimizing analysis to discover confirmed flows.

5.2.3 Assigning Weights to Exploitability Metric Features

This section describes how we assign feature values to operation trees and how we combine those values into a single exploitability metric used to steer fuzzing toward more easily exploitable paths.

As noted in Section 5.2.1, we modified NODEMEDIC-FINE's fuzzer so analysis does not stop at the first discovered potential flow. Pseudocode describing the newly generated fuzzing driver is shown in Figure 5.2 (updated from Figure 4.2).

As before, the fuzzing driver receives as input the target package and its entrypoints, and outputs a set of flows to confirm. We now summarize the driver changes and then rigorously define the exploitability metric.

Entry point selection policy. Line 3 of the generated driver permits selection of any exported entry point of the target package on each fuzzing round. When exploitability-guided sampling is disabled, selection reduces to round-robin over exported entry points, similarly to the original NODEMEDIC-FINE. When enabled, each entry point is sampled with a probability weight that depends on the entry point's estimated exploitability. Because no prior information exists at start-up, `select_entrypoint` chooses each entry point once in round-robin fashion to collect initial exploitability observations.

Estimating an entry point's exploitability. Let $\mathscr{T}$ be the set of all possible operation trees. We define an exploitability metric:

$$EM : \mathscr{T} \to \mathbb{R}_{\geq 0},$$

so that larger values indicate higher estimated exploitability.

For a fixed entry point e, let $\mathscr{T}_e^r$ denote the multiset of provenance trees produced by all calls to e observed since analysis began until round r.

$$\widehat{EM}^r(e) = \frac{1}{|\mathscr{T}_e^r|} \sum_{t \in \mathscr{T}_e^r} EM(t).$$

Our entry point-level exploitability estimate is the empirical mean of EM over $\mathscr{T}_e^r$: The higher the value of the exploitability metric for a provenance tree, the more exploitable we consider that tree to be. Our estimate of the exploitability of an entry point is based on the average exploitability for the provenance trees produced by calling that entry point, since analysis started. When fuzzing begins round $r+1$ (i.e., the $r+1$th iteration of the loop in line 2 of the driver), `select_entrypoint` uses the exploitability estimate computed from previous rounds $\widehat{EM}^r(e)$ to decide what entry point is chosen.

Structure of the exploitability metric Each feature f_i is a nonnegative real-valued function on provenance trees:

$$f_i : \mathscr{T} \to \mathbb{R}_{\geq 0}.$$

Let $\mathscr{F} = \{f_{\text{sinkcall}}, f_{\text{taintedarg}}, f_{\text{prefixlength}}, f_{\text{opcomplexity}}, f_{\text{atkdata}}\}$ denote the feature set defined in the previous section. Empirically, a linear combination of these features produced a useful signal. We found that multiplying the flow indicator by the prefix-length feature increases correlation with exploitability on our design set. Concretely, for a provenance tree $t \in \mathscr{T}$ we define

$$\begin{aligned} EM(t) = \; & w_{\text{sinkcall}} f_{\text{sinkcall}}(t) \\ & + w_{\text{flowstrength}} f_{\text{taintedarg}}(t) f_{\text{prefixlength}}(t) \\ & + w_{\text{opcomplexity}} f_{\text{opcomplexity}}(t) \\ & + w_{\text{atkdata}} f_{\text{atkdata}}(t), \end{aligned}$$

where $W = (w_{\text{sinkcall}}, w_{\text{flowstrength}}, w_{\text{opcomplexity}}, w_{\text{atkdata}})$ are nonnegative scalar weights. We now define each feature precisely and describe how the weights were obtained.

Sink	value of $f_{sinkcall}$
`eval`	0.88
`Function`	0.62
`exec`	1
`spawn`	0.1
none (default value)	0

Table 5.1: $f_{sinkcall}$ feature value, assigned to a provenance tree, depending on which sink is called.

Training data used to inform feature specifications and train the weights W. To assemble training data, we allowed NodeMedic-FINE-2025 to attempt confirmation of multiple flows per package and ran an experiment with a large timeout (1000 s total time budget, including 240 s fuzzing) against 3475 packages where a potential flow had previously been discovered with NODEMEDIC-FINE. From a subset of 281 packages that attempted to confirm several flows we extracted up to 5 exploitable and 5 non-exploitable flows per package. We paired exploitable and non-exploitable flows within each package to obtain 5483 labeled (exploitable, non_exploitable) pairs. We partitioned these into a training set of 2881 pairs from 141 packages and an evaluation set of 2602 pairs from the remaining 140 packages.

Feature f_{sinkcall}. This feature encodes which sink, if any, is invoked in the operation tree. Table 5.1 lists the assigned values. If an input results in the invocation of multiple sinks, we use the first reached sink for simplicity. The mapping in Table 5.1 was derived from the observed exploitability rates on the 63-package design set and the 141-package training set.

Feature $f_{\text{taintedarg}}$. We define $f_{\text{taintedarg}}(t) = 1$ if the operation tree shows a sink being called with the necessary argument tainted (e.g., in order for a flow using the `Function` constructor sink to be exploitable, it is required that the last argument is controlled by the attacker, not the first arguments). Otherwise, if any other sink argument is tainted, we still assign $f_{\text{taintedarg}}(t) = 0.3$. Otherwise $f_{\text{taintedarg}}(t) = 0$. Note that feature values are not independent: $f_{\text{taintedarg}}(t) = 1$ implies a sink was called; hence $f_{\text{sinkcall}}(t) > 0$ in that case. It may also seem obvious that rewarding flow-triggering inputs is a good idea, but even inputs that trigger no flows may lead to exploitable paths by introducing small changes to the input, so it is important to not over-prioritize paths that exhibit high values for these features.

Feature $f_{\text{prefixlength}}$. Let $\text{prefix}(t)$ denote the length (in bytes) of the ACE prefix that needs to be completed in operation tree t. If no sink is called or $\text{prefix}(t) = 0$ (meaning all sink-argument bytes are tainted) we set $f_{\text{prefixlength}}(t) = 1$. Otherwise we use a reciprocal decay:

$$f_{\text{prefixlength}}(t) = \frac{1}{\text{length}(\text{prefix}(t))}.$$

Weight name	Weight	Cohen-d value
$w_{sinkcall}$	3.03	0.42
$w_{flowstrength}$	1.34	0.28
$w_{opcomplexity}$	8.24	0.22
$w_{atkdata}$	0.05	0.21

Table 5.2: Exploitability metric weights computed via logistic regression on a set of 141 packages. We also show Cohen's d value for each feature in the training dataset, showing a small correlation between each feature and exploitability.

Feature $f_{\text{opcomplexity}}$. If no sink is called then $f_{\text{opcomplexity}}(t) = 0$. Otherwise we accumulate a complexity score over the operations applied to tainted values up to the sink, and again use a reciprocal decay:

$$f_{\text{opcomplexity}}(t) = \frac{1}{\text{complexity}}.$$

Counting provenance-tree nodes was insufficient to measure complexity because different operations vary greatly in SMT difficulty. Instead we assign each operation a complexity score based on observed solver difficulty and sum these scores along the tainted-provenance path to the sink. The scores are empirical and derived from the 63-package design set and the 141-package training set.

Feature f_{atkdata}. This feature measures the amount of unsanitized attacker-controlled data reaching the sink argument. If no attacker-controlled bytes reach the sink then $f_{\text{atkdata}}(t) = 0$. Otherwise we compute

$$f_{\text{atkdata}}(t) = 1 - \frac{0.90}{\ell_{\text{atk}}(t)},$$

where $\ell_{\text{atk}}(t)$ denotes the length (in bytes) of unsanitized attacker-controlled data reaching the sink. Overall, this formula reasonably rewards the presence of attacker-controlled data, with diminishing rewards as attacker data increases.

Computing the weights W. We fit W using binary logistic regression on labeled pairs of flows. The 2881 pairs of flows in the 141 packages in the training set produced the weights shown in Table 5.2.

We evaluated the resulting exploitability metric's ability to distinguish exploitable flows from non-exploitable flows on the training and evaluation partitions. Table 5.3 reports pairwise accuracy.

The metric generalizes to the held-out evaluation set. The evaluation accuracy of 63.84% corresponds to a 13.84 percentage-point distance from a 50% baseline in correctly preferring exploitable flows over non-exploitable ones.

Dataset	Accuracy (%)
Training	65.78
Eval	63.84

Table 5.3: Exploitability metric accuracy in pair-wise distinguishing exploitable versus non-exploitable flows in two datasets.

```
1 module.exports.spawncat = function spawncat(arg, options){
2   spawn("cat", [arg], options);
3 }
4 module.exports.execcat = function execcat(arg){
5   exec("cat " + arg[0]);
6 }
```

Figure 5.3: Example package with two exported entry points. `spawncat` uses `spawn`; `execcat` uses `exec`.

Prioritization of flows after fuzzing. Once fuzzing ends, NodeMedic-FINE-2025 attempts to confirm all discovered potential flows. Instead of confirming them in the order they were discovered, we sort them by their exploitability estimate (computed via EM) in descending order. This prioritization allows us to focus on the most promising flows first, increasing the chances of successful exploitation.

5.2.4 Example Analysis Run Using the Exploitability Metric.

This section shows how NodeMedic-FINE-2025 uses the exploitability metric to prioritize entry points during fuzzing. We use a hypothetical package with two exported entry points, `spawncat` and `execcat`, shown in Figure 5.3.

If the exploitability metric is disabled, the fuzzer alternates between the two entry points each round. When the metric is enabled the fuzzer collects initial observations and then estimates exploitability to guide selection.

In rounds 1 and 2 each entry point is executed once:

- Round 1: `spawncat` called with the string `"random"` produces operation tree t_1.
- Round 2: `execcat` called with the string `"anotherrandomstring"` produces operation tree t_2.

In round 3 the fuzzer computes exploitability estimates

$$\widehat{EM}^2(\text{spawncat}) = EM(t_1), \qquad \widehat{EM}^2(\text{execcat}) = EM(t_2),$$

Regarding t_1:

- The root node of t_1 will show that the sink called is `spawn`, so $f_{\text{sinkcall}}(t_1) = 0.1$.
- One of the arguments of the root note are tainted (i.e., a flow is discovered), so $f_{\text{taintedarg}}(t_1) = 1$.
- No ACE prefix needs to be completed, so $f_{\text{prefixlength}}(t_1) = 1$.

- The operations on tainted data are simple. The complexity calculation in reality is performed by summing weighted nodes of the tree depending on its operations. For simplicity, let us assume complexity $= 2$, thus $f_{\text{opcomplexity}}(t_1) = 0.5$.
- Six unsanitized attacker-controlled bytes reach the sink (i.e., the characters from the string `"random"`), so $f_{\text{atkdata}}(t_1) = 1 - \frac{0.90}{5} = 0.82$.

Hence

$$EM(t_1) = 3.03 \cdot 0.1 + 1.34 \cdot 1 \cdot 1 + 8.24 \cdot 0.5 + 0.05 \cdot 0.82 = 5.803.$$

Regarding t_2:

- The root node of t_2 will show that the sink called is `exec`, so $f_{\text{sinkcall}}(t_2) = 1$.
- A flow is discovered, so $f_{\text{taintedarg}}(t_2) = 1$.
- No ACE prefix needs to be completed , so $f_{\text{prefixlength}}(t_2) = 1$.
- The operations on tainted data are slightly more complex, with a string concatenation and an indexation. Again, for simplicity let complexity $= 2.5$, thus $f_{\text{opcomplexity}}(t_2) = 0.4$.
- One unsanitized attacker-controlled byte reaches the sink (i.e., the first character from the string `"anotherrandomstring"`), so $f_{\text{atkdata}}(t_2) = 1 - \frac{0.90}{1} = 0.10$.

Hence

$$EM(t_2) = 3.03 \cdot 1 + 1.34 \cdot 1 \cdot 1 + 8.24 \cdot 0.4 + 0.05 \cdot 0.10 = 7.671.$$

Because $\widehat{EM}^2(\text{execcat}) > \widehat{EM}^2(\text{spawncat})$, the fuzzer is more likely to select `execcat` in round 3. NodeMedic-FINE-2025's confirmation engine cannot synthesize exploits for flows through `spawn` when an `options` object is passed unless the sink call hardcodes `shell=true`. This makes focusing on `execcat` more likely to produce confirmed flows.

In later rounds the fuzzer may discover that providing a list rather than a string to `execcat` yields an operation tree with a larger attacker-controlled payload reaching the sink. That raises the exploitability estimate further and shifts more fuzzing effort to `execcat`. That input produces an exploitable flow, as a sufficient number of unsanitized attacker-controlled input reaches the sink, and the usual payloads can be injected. In this way, the exploitability metric steers fuzzing toward paths that are easier to confirm as exploitable.

5.3 Evaluation

In this section, we evaluate the effectiveness of our exploitability metric in guiding fuzzing towards more easily exploitable paths and in prioritizing exploitable flows for confirmation. We answer the following research questions:

RQ1: How does the exploitability metric impact NodeMedic-FINE-2025's ability to find confirmed flows?

RQ2: How does augmented NodeMedic-FINE-2025 compare to other tools for ACI and ACE detection and confirmation?

Condition name	EM usage	Time budget (s)		RQs involved
		Total	Fuzzing	
NMFINE-NoEM	None	300	180	RQ1,RQ2,RQ3
NMFINE-EM	Default EM	300	180	RQ1,RQ2
NMFINE-EM-Nofsinkcall	F_{sinkcall} disabled	300	180	RQ1
NMFINE-EM-Noftaintedarg	$F_{\text{taintedarg}}$ disabled	300	180	RQ1
NMFINE-EM-Nofprefixlength	$F_{\text{prefixlength}}$	300	180	RQ1
NMFINE-EM-Nofopcomplexity	$F_{\text{opcomplexity}}$	300	180	RQ1
NMFINE-EM-Nofatkdata	F_{atkdata}	300	180	RQ1
NMFINE-long-NoEM	None	1800	1080	RQ1
NMFINE-long-EM	Default EM	1800	1080	RQ1
NMFINE-long-EM-Nofsinkcall	F_{sinkcall} disabled	1800	1080	RQ1
NMFINE-long-EM-Noftaintedarg	$F_{\text{taintedarg}}$ disabled	1800	1080	RQ1
NMFINE-long-EM-Nofprefixlength	$F_{\text{prefixlength}}$ disabled	1800	1080	RQ1
NMFINE-long-EM-Nofopcomplexity	$F_{\text{opcomplexity}}$ disabled	1800	1080	RQ1
NMFINE-long-EM-Nofatkdata	F_{atkdata} disabled	1800	1080	RQ1

Table 5.4: Experimental conditions for EM ablation and timeout studies.

5.3.1 Experimental Setup

Conditions. Table 5.4 lists the NodeMedic-FINE-2025 configurations used and the research questions they address. The baseline, NMFINE-NoEM, has the exploitability metric disabled. In that baseline, entry points are selected in round-robin fashion, and flows are confirmed in discovery order, though still clustered by entry point. The baseline uses a 300s per-package timeout, following NODEMEDIC-FINE and Explode.js [80] evaluation setup. We allocated 60% of the time budget (180s) to fuzzing, as that proportion worked well in small scale experiments.

We evaluate six variants of NodeMedic-FINE-2025 with the exploitability metric enabled, varying the features used. Disabling a feature f_x is implemented by assigning the constant value $f_x(t) = 1$ for all provenance trees t. . Configurations that disable features are named with a suffix indicating which feature was disabled.

For each condition we also run a long-time alternative with a 1800s timeout, of which 1080s (60%) are allocated to fuzzing. Long-timeout conditions include the substring "-long-" in their names.

We also evaluate FAST [58] and Explode.js [80] using their latest public releases and default settings.

Datasets. We use three datasets:

- **WithSinks**: 33,021 packages containing sink calls as collected in the NODEMEDIC-FINE dataset (see Chapter 4 and Section 4.4.2). Used for EM evaluation and feature ablation (RQ1).
- **Potentials**: A collection of all 3,938 packages where some variation of NODEMEDIC-

FINE found a potential flow, either in the past or in this work. Similarly to the first dataset, we use this to evaluate the exploitability metric and its features (RQ1), but with a higher timeout for both analysis and confirmation.

- **Random120k**: The top 100,000 npm packages by weekly downloads (crawled the week of October 20, 2025) plus 20,000 randomly sampled packages. Used to compare NMFINE-EM with FAST and Explode.js.

Runtime details. Each NodeMedic-FINE-2025 condition halts after the first successful exploit. Experiments ran one Docker container per package and per condition. Packages were analyzed in parallel, each container restricted to using 4GB of RAM. Analysis and confirmation followed the pipelines described in Section 4.4.1 with the modifications in Section 5.2.1.

When comparing NodeMedic-FINE-2025 to FAST and Explode.js, we used default settings for those tools. All tools received a total timeout of 5 minutes, including 3 minutes of fuzzing for NODEMEDIC-FINE. For the Potentials dataset we used a 30-minute total timeout and, once again, allocated 60% of the budget to fuzzing (18 minutes in this case).

5.3.2 RQ1: Effectiveness of the Exploitability Metric

This section evaluates the effect of the exploitability metric (EM) on (i) the number of potential and confirmed flows discovered, and (ii) the average time to confirm flows. The evaluation proceeds in two parts: first, a short-timeout setting using the original NODEMEDIC-FINE dataset; second, a long-timeout setting applied to all packages with previously discovered potential flows.

Overall, longer timeouts amplify the benefits of EM-guided fuzzing. EM-guided fuzzing with a long timeout recovers shallow flows that may be missed with a short timeout when it overprioritizes non-exploitable entry points, while still discovering deep flows overlooked by unguided fuzzing. Additionally, because analysis of each package stops upon the first confirmed flow, EM-guided fuzzing tends to produce shorter runs that yield slightly more confirmed flows under equal time budgets, with larger gains under longer budgets.

RQ1a: Short timeout evaluation on WithSinks dataset

We executed the following configurations on the WithSinks dataset: NMFINE-EM, NMFINE-NoEM, NMFINE-EM-Nofsinkcall, NMFINE-EM-Noftaintedarg, NMFINE-EM-Nofprefixlength, NMFINE-EM-Nofopcomplexity, and NMFINE-EM-Nofatkdata. Each configuration used a total timeout of 300 seconds, with 180 seconds allocated to fuzzing. Results are summarized in Table 5.5. Overall, we find that all features contribute positively to the number of packages with discovered potential flows, though their effect on other metrics was more nuanced. Each metric is analyzed below.

Packages with observed potential flows. Since EM prioritizes entry points based on their estimated exploitability, it is essential to confirm that it does not suppress discovery of potential flows. As shown in Table 5.5, all EM-enabled configurations discovered more potential flows

NMFINE Condition	#Packages with flows			Avg. rounds to confirm (common confs.)	Avg. time to confirm (s) (common confs.)
	Pot.	Conf.	Conf. (common pots.)		
NoEM	2314	803	800	1.18	6.88
EM	2319	812	807	1.05	6.50
EM-Nofsinkcall	2310	813	808	1.05	6.62
EM-Noftaintedarg	2310	812	808	1.08	7.10
EM-Nofprefixlength	2317	813	809	1.06	6.63
EM-Nofopcomplexity	2317	810	804	1.05	6.90
EM-Nofatkdata	2318	809	806	1.07	6.33

Table 5.5: Summary of results for short-timeout evaluation on the 33,021 packages of the With-Sinks dataset. For each NodeMedic-FINE-2025 configuration, we show the number of packages with discovered potential flows, the number of packages with confirmed flows, the number of packages with confirmed flows among the 2279 packages with potential flows found in common by all conditions, the average number of rounds to confirm a flow, and the average time to confirm a flow within the 784 packages where a confirmed flow was found by all conditions.

than the baseline NMFINE-NoEM. However, each feature contributes differently, and some potential flows were missed.

Figure 5.4 shows the number of additional and missed potential flows per configuration relative to the baseline. We manually inspected a sample of missed and additional flows to determine whether EM correctly prioritized exploitable entry points.

The full EM discovered 22 additional potential flows absent in the baseline. Manual inspection of five cases revealed: one false positive (over-tainting), one case where EM avoided (by a matter of luck in input generation) an early hang in analysis suffered by the baseline, and three genuine improvements where EM prioritized entry points that contained deep flows. In these last three cases, fuzzing needs to analyze the exploitable entry point for some time until it figures out the right input structure, so prioritizing that entry point is paramount. In one such case, the $f_{complexity}$ feature was critical: The package contained 8 entry points containing sink calls. All EM-enabled conditions except NMFINE-EM-Nofopcomplexity prioritized an entry point with a simpler provenance tree, leading to successful confirmation, whereas disabling $f_{complexity}$ (EM-Nofopcomplexity) missed the potential flow completely.

Conversely, we manually sampled 5 of the 17 potential flows missed by EM but discovered by the baseline. One case involved multiple entry points with identical exploitability estimates; EM then behaved like the baseline. The baseline only discovered the flow late during fuzzing, and the condition using the exploitability estimate was not lucky enough to find the flow-triggering input in time. This potential flow was discovered by NMFINE-EM when a longer fuzzing timeout was provided, and it was also discovered when we simply changed the PRNG seed.

Another case involved two sink-calling entry points: one with a shallow potential flow and yet unexploitable; another with a harder-to-find flow (requiring input of complex structure) but exploitable. EM correctly prioritized the latter, missing the shallow potential flow seen by NMFINE-NoEM; Interestingly, using longer timeouts in this case not only results in EM find-

ing the unexploitable potential flow, but it also successfully finds and confirms the other, harder flow. Conversely, the baseline using the longer timeout still only finds the shallow unexploitable potential flow. Two other potential flows were missed due to EM partially prioritizing memory-intensive entry points that ended up crashing the process. The final case finally reflected true misprioritization: EM overemphasized an entry point that appeared exploitable early but could not actually propagate attacker input to the sink. The vulnerability was in another entry point that did not show early promise of exploitability, but it is shallow enough that, with a longer timeout, NMFINE-EM finds it even though it does not analyze it nearly as much as the other unexploitable entry point.

In Section 5.4 we include a discussion on the underlying limitation of the exploitability metric, that may result in overprioritization of entry points that exhibit early signals of exploitability. However, this manual analysis reveals empirical evidence that EM often prioritizes the exploitable entry point, and we will see later that using a long timeout for fuzzing benefits it, as many of the extra potential flows discovered by EM-guided fuzzing are deep enough that running the baseline with a larger timeout still often misses them.

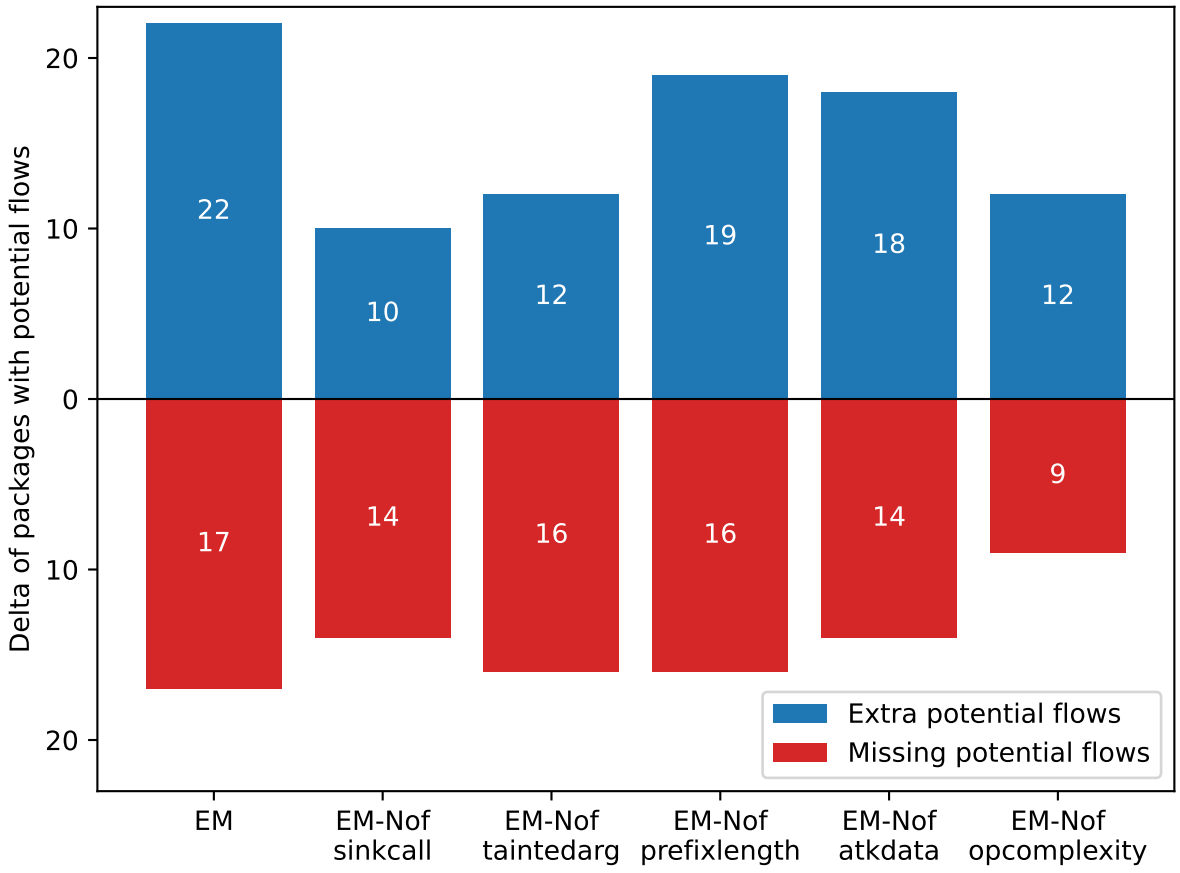

Figure 5.4: Missing and extra potential flows, compared with the baseline NMFINE-NoEM.

Finally, some flows critically depended on individual features. For example, $f_{hasflow}$ was essential in one case involving refinement of inputs that progressively increased taint propagation

to the `Function` constructor sink. The first input discovered by NMFINE-EM (and also discovered by the baseline) called that sink, but only the first argument was tainted, which is not sufficient for exploitability. Still, the EM-guided fuzzer (particularly, the $f_{hasflow}$ feature) took that as evidence that the entry point is promising to analyze, and indeed a refinement of that input led to finding a flow with the last argument tainted (i.e., the argument specifying the code of the dynamically constructed function).

Confirmed flows. Using the full EM increased confirmed flows by 1% (from 803 to 812). Although some features appear to have a positive impact (e.g., disabling the $f_{sinkcall}$ feature in NMFINE-EM-Nofsinkcall decreases confirmed flows to 809), other features seem to have a negative impact (e.g., disabling the $f_{prefixlength}$ feature actually allowed to find 813 confirmed flows, more than using the full exploitability metric). Figure 5.5 shows missing and additional confirmed flows relative to the baseline. To move from the impact of the EM on the potential flows, we now focus on the common set of packages where all conditions found a potential flow, illustrated in Figure 5.6.

Among packages where all configurations discovered potential flows, NMFINE-EM confirmed 12 additional flows relative to the baseline. Manual inspection of five cases showed that, beyond finding additional potential flows, EM gains confirmed flows in two ways: First, it refines inputs in exploitable entry points in such a way that, even though it approximately finds the same flow as the baseline (same source and sink pair but different intermediate instructions), it is able to find inputs that generate provenance trees that are simpler to exploit. This occured in two of five cases. Secondly, even when the exact same flow is discovered by both NMFINE-EM and NMFINE-NoEM, NMFINE-EM is able to prioritize flows by their exploitability estimate, which led it to successfully confirm flows (within the time budget) on the remaining three of five cases.

Furthermore, we manually analyzed all five missing confirmed flows when using the exploitability metric when looking at the common set of packages with potential flows. The first case had several exploitable entry points, and NMFINE-EM prioritized one exploitable entry point the most, but successful exploitation required the synthesis of a valid filename (i.e., the file had to exist in the filesystem). In the second and third cases, asynchronous behavior introduced non-determinism, even though both conditions found the same exploitable flow. We launched NMFINE-EM once again against the same package and it was able to confirm the same flow as NMFINE-NoEM, even though we did not even change the PRNG seed used during fuzzing. In the fourth case, both conditions attempted an exploitable flow on the first confirmation round, but the input used by NMFINE-EM froze the npm process that validated the exploit. We assign a 60 second timeout for the process of validating a single exploit, as sometimes it takes some time for the target entry point to process the input. If we give an additional 20 seconds of time budget for NMFINE-EM, it is able to validate that its first attempt is actually successful. In the fifth and final case, the exploitability metric overprioritized an entry point that showed early signs of exploitability but was not exploitable: the vulnerability was in another entry point, but the sink call was hard to reach.

Rounds to confirm and time to first confirmed flow. Full EM required significantly fewer confirmation rounds (89% of the baseline on average, Welch t-test: $t = 2.98$, $p = 0.001$), while

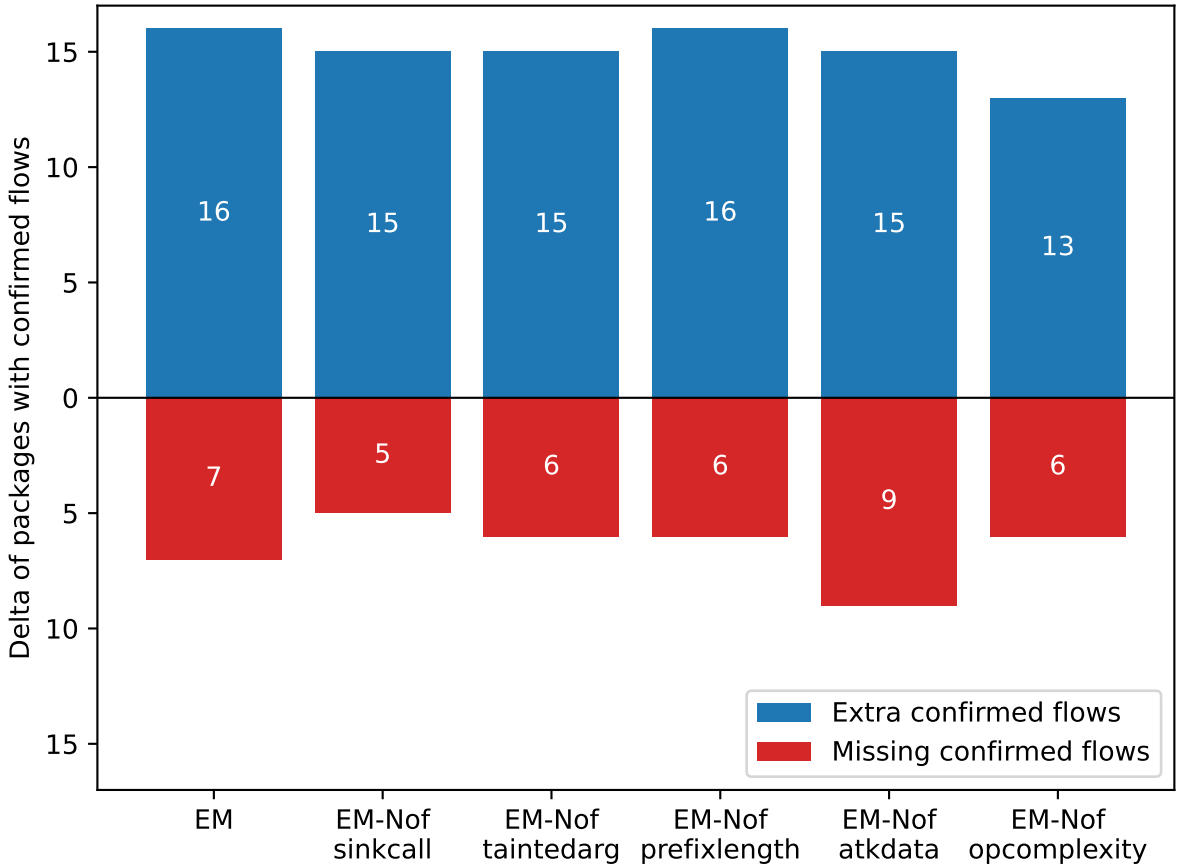

Figure 5.5: Missing and extra confirmed flows, compared with the baseline NMFINE-NoEM.

confirming slightly faster (94% of baseline time, not statistically significant - $t = 0.68$, $p = 0.25$). Recall that we stop at the first confirmed flow, and the time savings gained during confirmation when using EM-enabled conditions are not used to analyze more packages: the baseline experiment just takes longer to analyze the same set of packages.

Result 1a: EM-guided fuzzing confirmed flows in 1% more packages and required significantly fewer confirmation rounds than the baseline. Improvements stem from better entry point prioritization and consequently more refined input generation, but also from sorting the flows to confirm by their exploitability estimate.

RQ1b: Long timeout evaluation on potentials Dataset

To evaluate EM under longer budgets, we ran NMFINE-long-EM, NMFINE-long-NoEM, NMFINE-long-EM-Nofsinkcall, NMFINE-long-EM-Noftaintedarg, NMFINE-long-EM-Nofprefixlength, NMFINE-long-EM-Nofopcomplexity, and NMFINE-long-EM-Nofatkdata on the Potentials dataset. Each configuration used a total timeout of 1800 seconds (30 minutes), with 1080 seconds (60% of time budget, 18 minutes) for fuzzing. Results are summarized in Table 5.6.

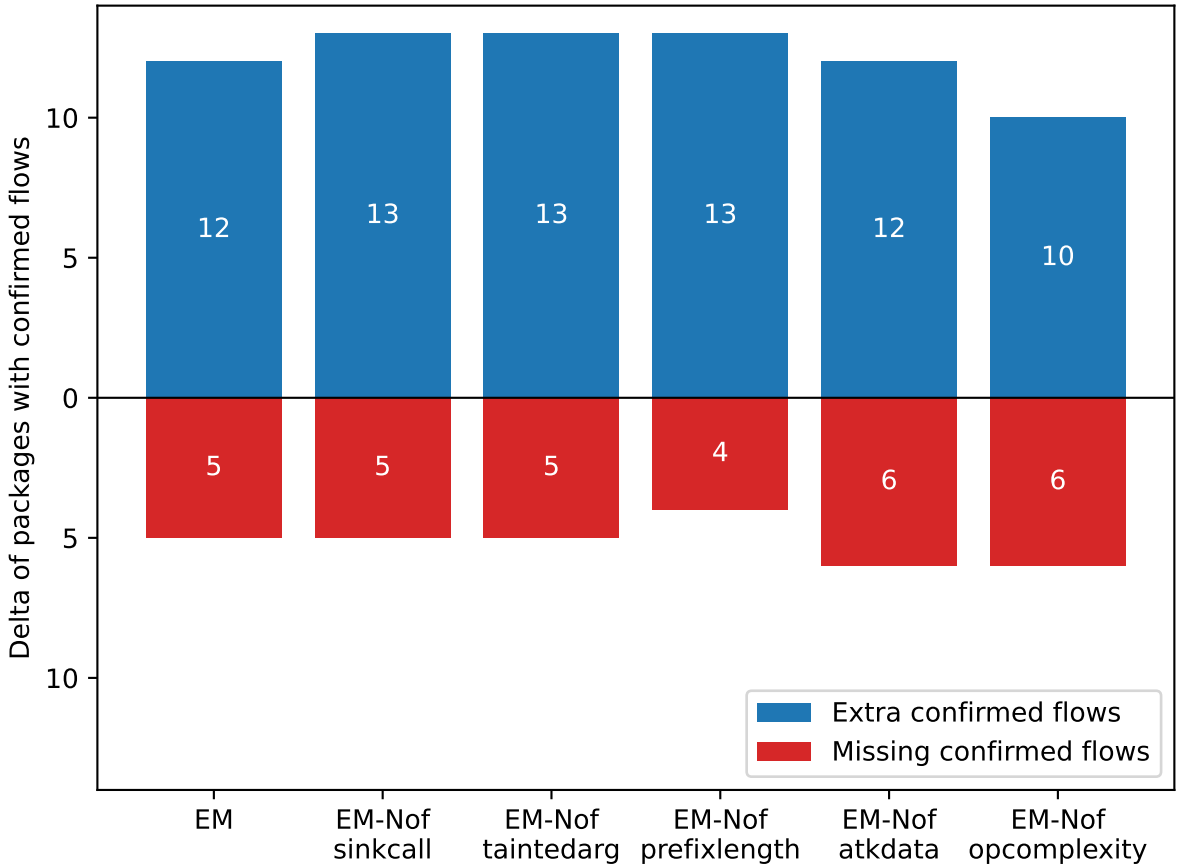

Figure 5.6: Missing and extra confirmed flows, compared with the baseline NMFINE-NoEM, but only within the set of packages where all conditions discovered a potential flow.

Potential flows. Figure 5.7 compares each EM-enabled configuration against the baseline, reporting additional and extra potential flows. The longer timeout mitigated previous misses and enabled EM to recover several flows not found under shorter runs. Even with extended time, the baseline NMFINE-long-NoEM misses 45 potential flows, which we find to be usually deep in entry points that were correctly prioritized by the exploitability metric.

Confirmed flows. Figure 5.8 shows the number of missed and extra confirmed flows discovered by each condition relatively to the baseline NMFINE-long-NoEM. If we compare those results to Figure 5.9, which shows the same data but only within the set of packages where a potential flow was discovered by all conditions, we see that a significant portion of the gained confirmed flows when using the exploitability metric come from finding a potential flow on the first place. The remaining gains in confirmed flows, once again, seem to come both from finding easier to exploit flows during fuzzing, and also from the prioritization of flows for confirmation based on their exploitability estimate.

NMFINE Condition	#Packages with flows			Avg. rounds to confirm (common confs.)	Avg. time to confirm (s) (common confs.)
	Pot.	Conf.	Conf. (common pots.)		
NoEM	3285	1638	1633	1.44	8.65
EM	3316	1656	1640	1.15	6.20
EM-Nofsinkcall	3308	1648	1634	1.23	7.43
EM-Noftaintedarg	3306	1650	1634	1.10	6.24
EM-Nofprefixlength	3321	1657	1638	1.15	6.49
EM-Nofopcomplexity	3304	1652	1637	1.18	7.58
EM-Nofatkdata	3306	1654	1641	1.17	6.56

Table 5.6: Summary of results for long-timeout evaluation on the 3,938 packages of the Potentials dataset. For each NodeMedic-FINE-2025 configuration, we show the number of packages with discovered potential flows, the number of packages with confirmed flows, the number of packages with confirmed flows among the 3248 packages with potential flows found in common by all conditions, the average number of rounds to confirm a flow, and the average time to confirm a flow within the 1,620 packages where a confirmed flow was found by all conditions.

Rounds to confirm and time to first confirmed flow. Using the exploitability metric reduced the average rounds to confirm from 1.44 to 1.23 ($t = 2.48$, $p = 0.007$) and the average confirmation time from 8.65s to 6.20s ($t = 2.68$, $p = 0.004$). No individual feature seems to be responsible for most of the decrease in the average rounds to confirm, but $f_{sinkcall}$ and $f_{complexity}$ seem to have a large impact on the average time to confirm, as disabling them results in an increase from 6.20 (full exploitability metric) to 7.43s and 7.58s respectively. With respect to $f_{complexity}$ contributions, it directly prioritizes flows that use simpler operations, and it mainly affects the SMT confirmation stage where Z3 is called. The impact of $f_{sinkcall}$ comes from the prioritization of flows with sinks that are easier to exploit, which saves time attempting confirmation on entry points using `spawn` when other flows exist.

Result 1b: With a long timeout, EM-guided fuzzing discovered nearly all potential flows found by the baseline (missing only 14), plus 45 additional ones. It confirmed 25 more flows, requiring 20% fewer confirmation rounds and 28% less confirmation time than the baseline.

5.3.3 RQ2: Comparison with Prior Work.

In this section, we compare NMFINE-EM against state-of-the-art tools for ACI and ACE detection: FAST [58] and Explode.js [80]. This evaluation uses the Random120k dataset, which includes the 100,000 most popular npm packages (by weekly downloads) and an additional 20,000 randomly sampled packages. Each tool was given a 5-minute timeout per package. For NMFINE-EM, 60% of this budget (3 minutes) was allocated to fuzzing.

Table 5.7 summarizes the results in terms of potential and confirmed flows. NMFINE-EM reports more confirmed ACE flows than all other tools. It is also worth noting that NodeMedic-FINE-2025 detected fewer flows in Random120k compared to the WithSinks dataset, as Ran-

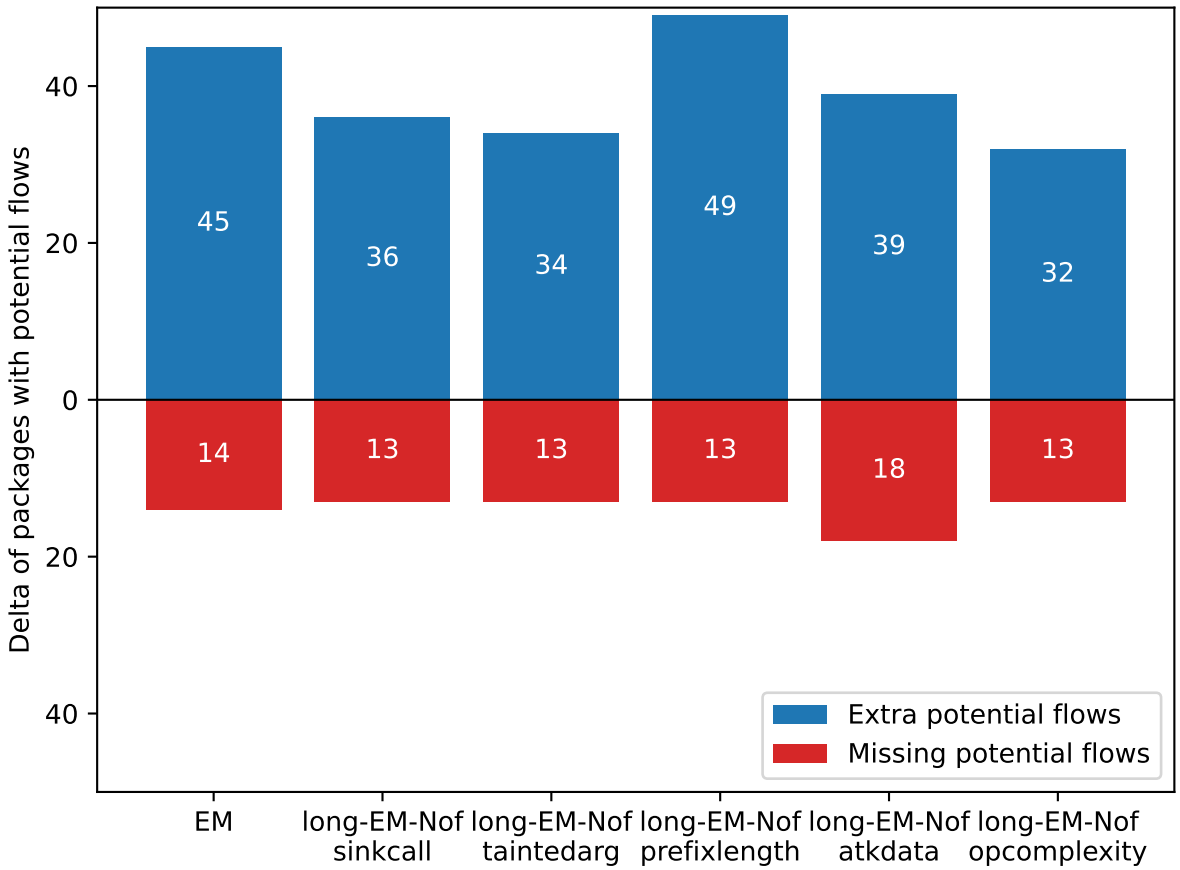

Figure 5.7: Missing and extra potential flows for conditions using the exploitability metric, compared with the baseline NMFINE-long-NoEM. This was using a large time budget of 1800 seconds per package (including 1080 seconds allocated to fuzzing).

dom120k includes many packages without sink calls, as well as packages that are not easy to install or run automatically.

FAST seeminlgy detects the largest number of potential flows and ACI confirmed flows overall. However, FAST does not execute its synthesized exploits to validate exploitability; its notion of a ”confirmed flow” is therefore more permissive.

Additionally, unlike NodeMedic-FINE-2025 and Explode.js, FAST is a purely static analysis tool that does not require package installation or execution. Finally, NodeMedic-FINE-2025 and Explode.js impose additional preconditions for successful analysis, such as requiring a valid `package.json` with a defined `main` field.

Valid packages. For the remainder of this section, we focus on packages for which detection started successfully in all tools. We define a valid package as one meeting the following tool-specific criteria:

- **Explode.js** successfully builds the graph-based representation using Graph.js, indicated by

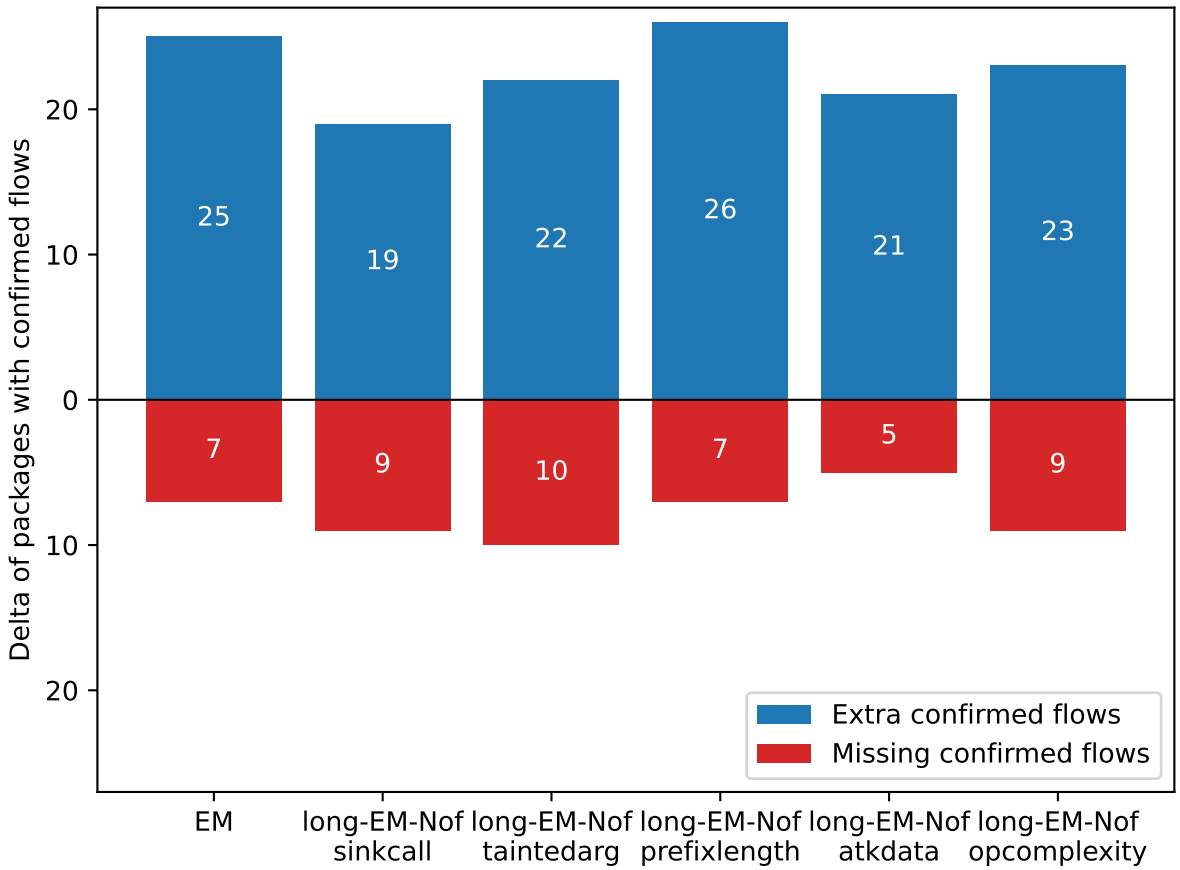

Figure 5.8: Missing and extra confirmed flows, compared with the baseline NMFINE-long-NoEM.

the presence of ”PHASE 0: VULNERABILITY DETECTION” and absence of ”Graphjs exited with non-zero code” in Explode.js output.

- **FAST** successfully generates a CFG and begins its top-down abstract interpretation phase, inferred from the presence of CFG statistics in FAST output.
- **NMFINE-EM** successfully starts fuzzing.

We call packages obeying all three criteria *valid packages*.

Figure 5.10 shows the overlap of valid packages. FAST and Explode.js rely primarily on static analysis for potential flow detection, which allows them to handle more packages. Explode.js requires code execution only for confirmation, while NodeMedic-FINE-2025 must install dependencies and execute code from the start. This explains the smaller valid set for NodeMedic-FINE-2025.

Both FAST and Explode.js support additional vulnerability types (e.g., path traversal, prototype pollution), but we restrict analysis to ACI and ACE for comparability with NodeMedic-FINE-2025.

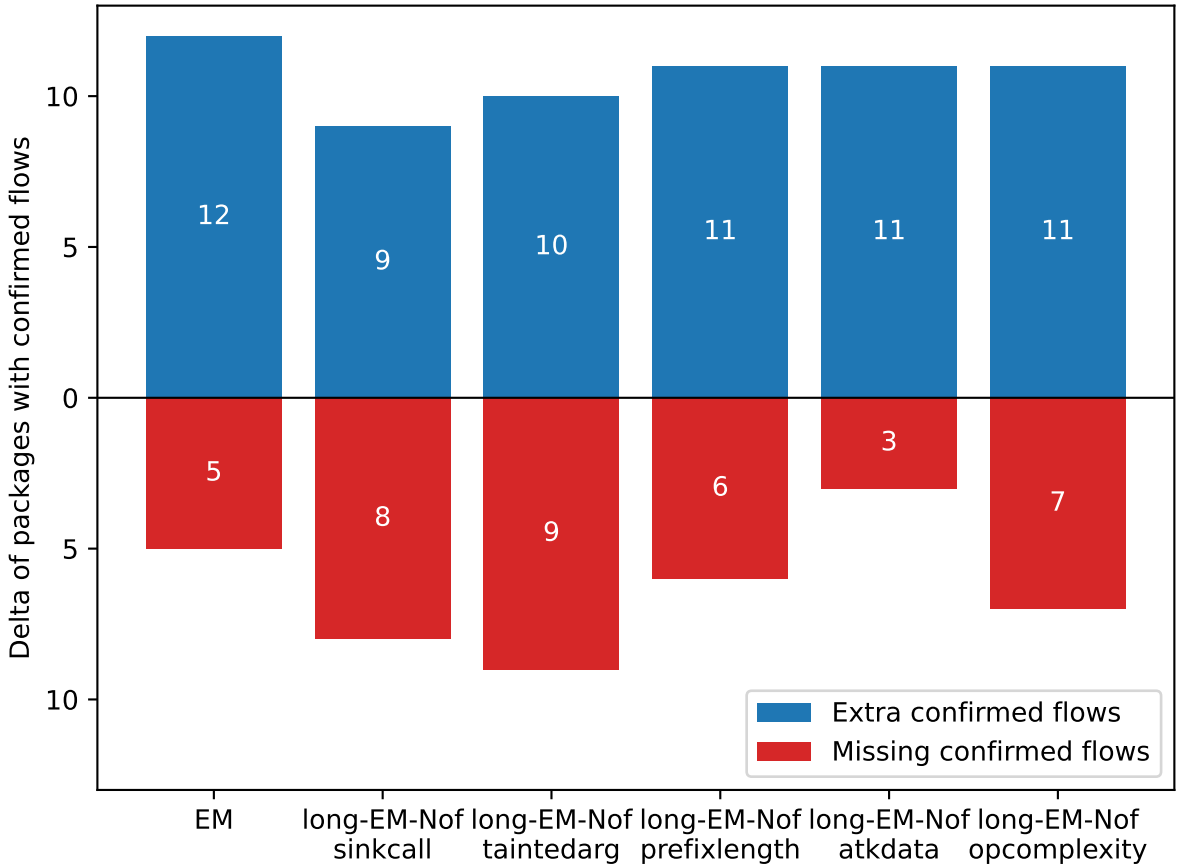

Figure 5.9: Missing and extra confirmed flows, compared with the baseline NMFINE-long-NoEM, but only within the set of packages where all conditions discovered a potential flow.

Potential flows. Figure 5.11 and Figure 5.12 show Venn diagrams of potential flows detected by each tool in the valid set, respectively for ACI and ACE flows. Each tool identifies a substantial number of unique potential flows, highlighting their complementary coverage. Explode.js detects the largest number of unique ACE potential flows (101 cases). FAST, as a static tool, also report a larger number of potential flows than NODEMEDIC-FINE.

Overall confirmed flows. We now analyze confirmed flows, including a manual inspection of flows uniquely discovered by one of the tools. Figure 5.13 and Figure 5.14 show Venn diagrams of confirmed flows for ACI and ACE, respectively.

We manually sampled up to five unique ACI and five ACE confirmed flows per tool to investigate why the others missed them.

NMFINE-EM unique confirmed flows. As shown in Figure 5.13, NMFINE-EM uniquely discovered 11 confirmed ACI flows. Manual inspection of five revealed that all involved JavaScript constructs difficult to handle statically, such as higher-order functions and asynchronous call-

Condition	ACI		ACE	
	Potential flows	**Confirmed flows**	**Potential flows**	**Confirmed flows**
NMFINE-EM	115	47	61	34
NMFINE-NoEM	116	44	62	33
FAST	179	109	112	28
Explode.js	72	4	116	2

Table 5.7: Overall results of evaluating NMFINE-EM, Explode.js and FAST against the Random120k dataset, in terms of potential and confirmed flows separated by vulnerability type.

backs. A similar pattern was observed for five of the 15 unique ACE confirmed flows (Figure 5.14). Neither Explode.js nor FAST discovered potential flows in any of the manually investigated packages with confirmed flows by NMFINE-EM. While 9 cases were true positives, `p4-oo@2.0.3` package was a false positive, since its documentation (including code documentation) already warns that inputs are not sanitized. Note that because this package has no dependents, it is not part of the DepWarnings dataset. That package exported its entry points like so: `module.exports = require('./lib/p4')(exec,path)`, where `exec` is the `child-process` sink to execute arbitrary commands. FAST does not detect this case because `lib/p4` exports entry points that call the argument passed in the constructor, and the argument is itself a function, making the `lib/p4` constructor a higher-order function. Explode.js queries to Graph.js returned no results in this case, preventing Explode.js from linking exported entry points to sinks. This is because Explode.js's underlying detection engine (based on Graph.js queries) does not handle packages that build their `module.exports` object based on the exported entry point of some internal or external library. In contrast, NodeMedic-FINE-2025 dynamically identifies entry points to analyze by simply inspecting `module.exports` at runtime.

We further analyzed the only two cases that NMFINE-EM confirmed but FAST and Explode.js only detected as potential flows. In one ACI case, Explode.js halted symbolic execution at an unsupported `await` statement, while FAST's translation to Z3 produced unsatisfiable constraints due to imprecise data-flow modeling. : The package combines a call to `require('path').join(argument)`. The other case was a package with an ACE flow. Explode.js generates two symbolic drivers. One driver assumes that the sink argument can be an array, which is incorrect because this application always calls the `search` method on the entry point argument. That method does not exist for arrays, and symbolic execution errored with TypeError. NodeMedic-FINE-2025 inference correctly infers that the correct type is string. Explode.js second driver timeouts, even after we give it an additional 20 minutes timeout. The culprit appears to be an iteration over regex search results, a difficult constraint to handle during symbolic execution. FAST failed during constraint solving. Regardeless, FAST constraints were missing syntactic validity constraints for the payload, producing an incomplete but satisfiable model.

Explode.js unique confirmed flows. Explode.js uniquely confirmed one ACE flow that neither FAST nor NodeMedic-FINE-2025 confirmed. The case involved the `vm.runInContext` ACE sink, which is similar to the `eval` sink (although it receives a context with global variables as parameter) and only Explode.js supports it for confirmation. NodeMedic-FINE-2025 detected the

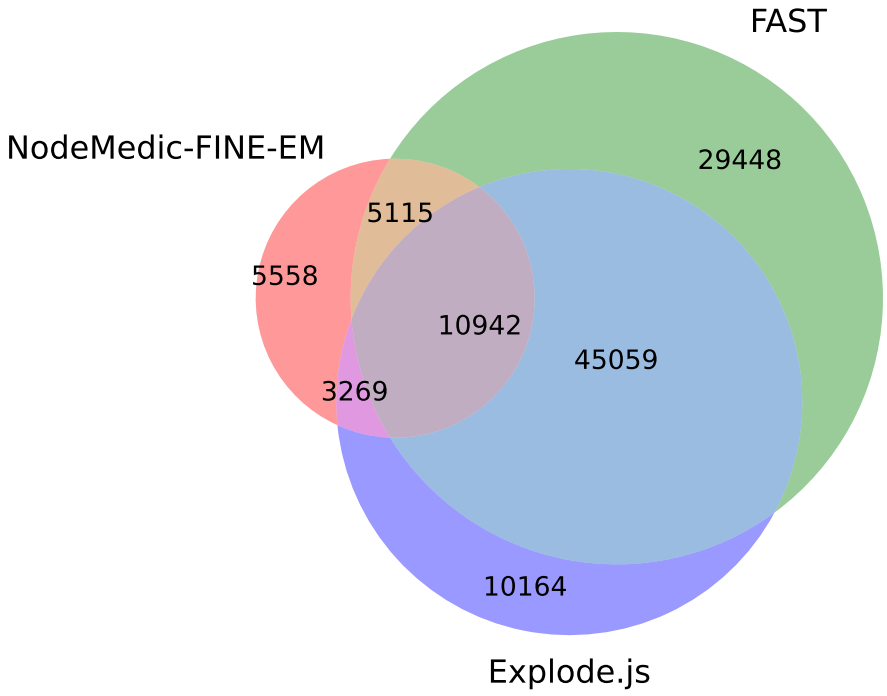

Figure 5.10: Set of valid packages for each tool. We focus our analysis on the intersection of 10,942 packages that are valid for all tools.

potential flow but lacked confirmation support for this sink. Extending NodeMedic-FINE-2025's confirmation engine with this sink (a one-line change in two files) allowed it to reproduce the same confirmed result.

FAST unique confirmed flows. We manually inspected five of the 36 ACI flows and five of the 11 ACE flows uniquely confirmed by FAST. Of these, six were false positives: although FAST reported confirmed flows, the packages were not actually exploitable (e.g., one package invoked `spawn` but never passed `shell=true`). Three were true positives where FAST correctly identified the vulnerability, but the synthesized exploit failed to execute as intended (e.g., one vulnerable entry point required escaping a single-quote context; FAST generated `"x00''; touch exploited #"`, which reentered a single-quote context immediately after escaping). Finally, one ACI flow was a true positive for which FAST's exploit succeeded. That package used `async` primitives, which prevented Explode.js from synthesizing an exploit. NodeMedic-FINE-2025 failed to detect this flow because the vulnerable function was returned by an exported entry point rather than being directly exported, a case NODEMEDIC-FINE's does not currently support, so it misses the

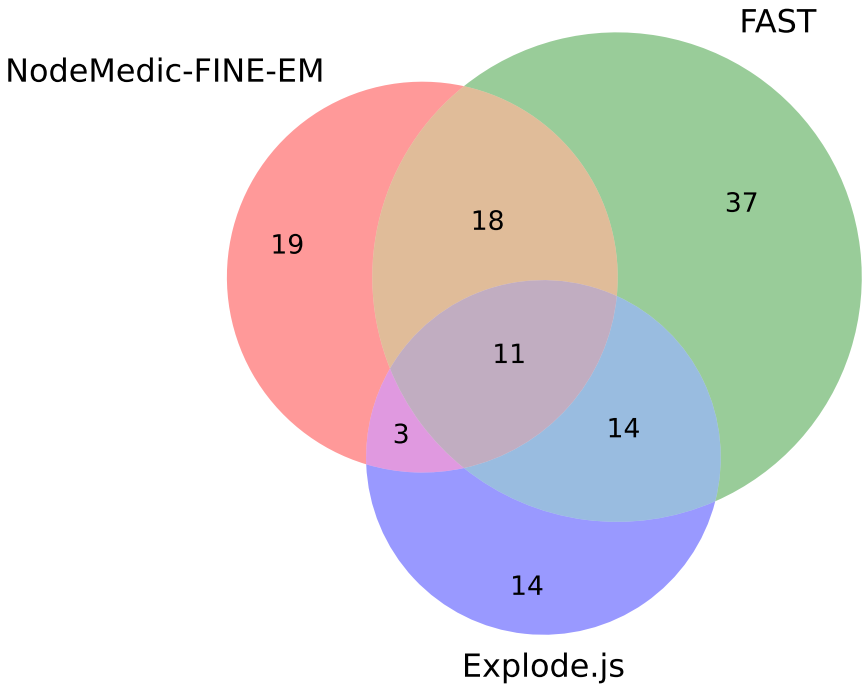

Figure 5.11: ACI potential flows discovered by NMFINE-EM, Explode.js and FAST, within the set of valid packages.

potential flow.

Result 2: The majority of confirmed flows uniquely identified by NMFINE-EM arise from its dynamic analysis, which enables it to detect vulnerabilities involving complex JavaScript primitives that static tools such as FAST and Explode.js fail to handle.

5.4 Limitations and Threats to Validity

The effectiveness of the exploitability metric depends on how accurately its features discriminate exploitable from non-exploitable paths. During manual analysis, we identified two packages in which the metric caused the fuzzer to overprioritize an entry point that initially appeared promising but was ultimately not the correct one to analyze. This behavior reflects the classical exploration-exploitation dilemma in fuzzing [78], where one must balance investing further in inputs that appear promising against exploring unexplored alternatives. To mitigate this issue, we enforce a maximum relative likelihood ratio of selecting any pair of entry points, using a fixed threshold: no entry point can become more than ten times more likely to be explored than any

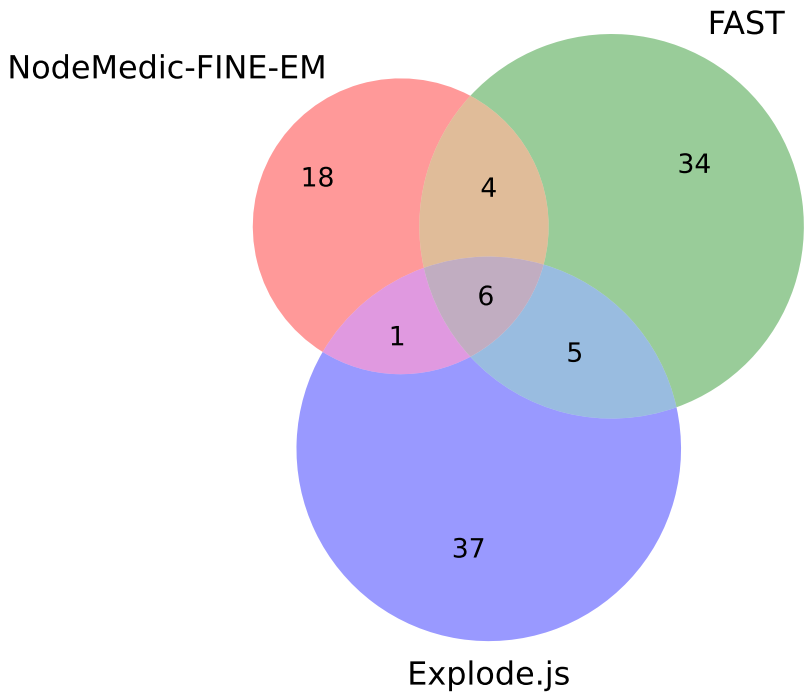

Figure 5.12: ACE potential flows discovered by NMFINE-EM, Explode.js and FAST, within the set of valid packages.

other.

Furthermore, we used the exploitability metric solely to select which entry point to analyze at each iteration. If a package contains one vulnerable entry point among N total entry points with similar runtimes, then, even when assuming optimal prioritization, the time to find the flow will be theoretically lower bounded by at least $\frac{1}{N}$ of the baseline time. . Although packages in Random120k export an average of 40.88 entry points, only about half of the packages in that dataset export more than one entry point. This limits the potential benefit of our integration strategy, which influences performance only when multiple entry points are available.

5.5 Future Work

For the threshold limiting the maximum relative likelihood between entry points, our manual investigation suggests that the optimal value to use during fuzzing likely depends on the remaining time budget for that package. As a future direction, a lower threshold could be applied early in fuzzing to promote exploration across entry points, followed by a gradual increase as time

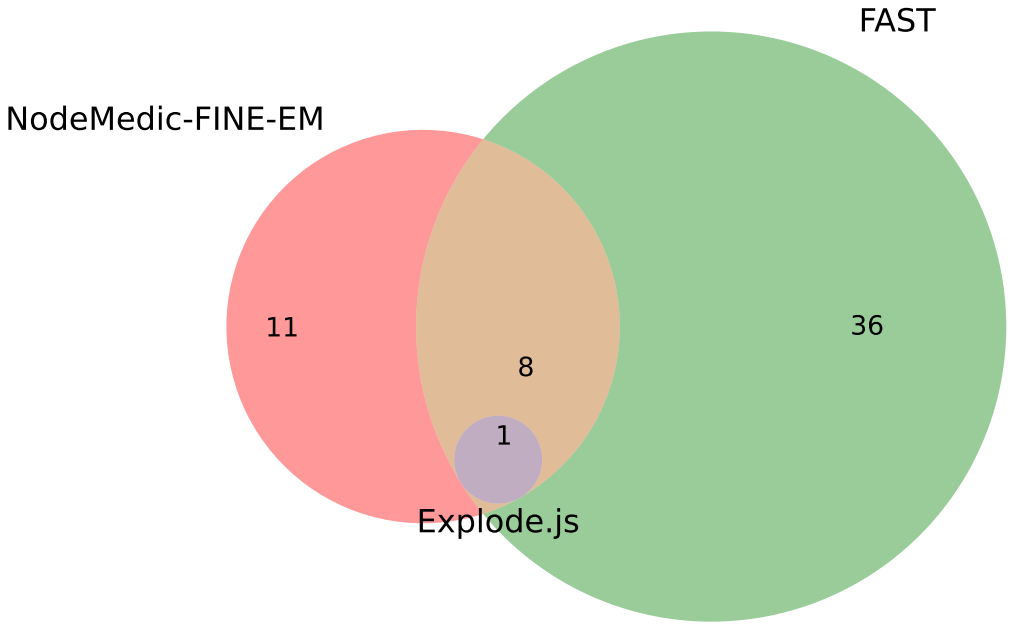

Figure 5.13: ACI confirmed flows discovered by NMFINE-EM, Explode.js and FAST, within the set of valid packages.

diminishes, allowing more intensive focus on entry points with high exploitability estimates.

A more substantial improvement may arise from giving the fuzzer more granular units of selection. Rather than selecting entry points, the metric could instead prioritize *specifications*, with each entry point associated with multiple specifications targeting diverse input populations that suit different structural characteristics of the entry point.

Further gains may also come from different or nonlinear feature combinations. Manual analysis indicates a recurring trade-off between selecting provenance trees with low complexity and those with greater amounts of attacker-controlled input passed to the sink. In practice, we find that proving that the attacker can inject a *sufficient* amount of data is often enough to ensure that a payload of interest fits. This suggests that attacker-controlled input should receive sharply increasing weight until a threshold is reached, after which additional injected input should have diminishing or no influence on the exploitability estimate. Furthermore, we currently use the same exploitability metric during fuzzing (for entry point prioritization) and after fuzzing (for flow prioritization), even though these stages have distinct purposes. Designing separate metrics for these stages may yield better performance, even if both use the same underlying features.

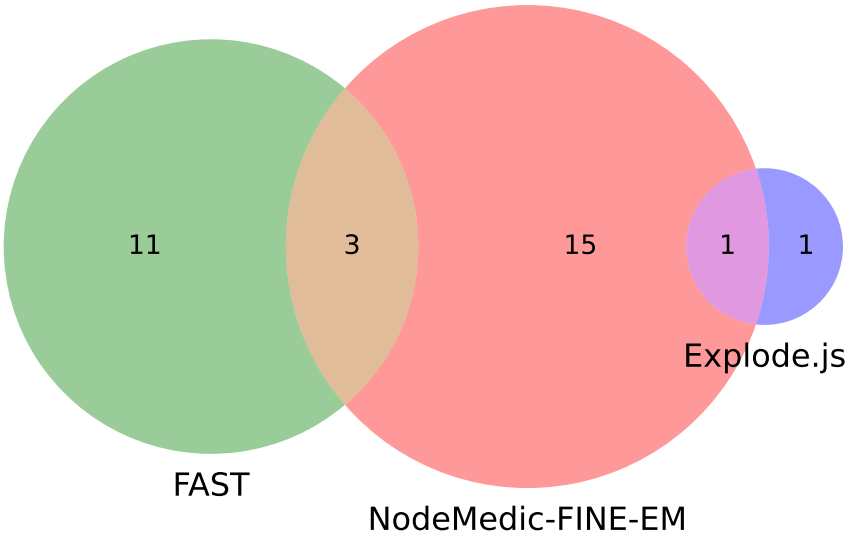

Figure 5.14: ACE potential flows discovered by NMFINE-EM, Explode.js and FAST, within the set of valid packages.

5.6 Conclusion

This work investigated a set of features that can be composed into an exploitability metric for estimating the exploitability of program paths. We integrated this metric into NODEMEDIC-FINE's fuzzing and confirmation pipeline to prioritize flows with higher exploitability estimates. A large-scale evaluation shows that incorporating this metric increases the number of confirmed flows by 1% without reducing the number of potential flows. Furthermore, the same confirmed flows can be identified more quickly, reducing confirmation time by up to 28% relative to the baseline without the metric.

Chapter 6

Responsibility of Input Sanitization

In this chapter, we study whether npm package developers respect security warnings present in the documentation of their dependencies, referring to unsafe ACE and ACI sink usages.

6.1 Overview

During reporting of NODEMEDIC-FINE vulnerabilities, one source of false positives were packages that explicitly acknowledged, in their documentation, that a security concern exists on the relevant entry points, and delegated the responsibility of sanitizing the respective inputs to the caller. Additionally, for a subset of the true positives, developers responded to our reports by adding such disclaimers in their documentation rather than fixing the underlying vulnerability.

In this work, we assess whether package developers respect security warnings in their dependencies documentations. With the help of large-language models (LLMs), we triaged the documentation of all packages currently available in the npm repository[1]. A set of 4,547 packages were signaled by GEMMA3:12B [118] and GPT-OSS:20B [97] as potentially having warnings (full gathering methodology and used prompts are described in Section 6.2). After manual validation, we obtained a set of 301 packages whose documentation explicitly warns of potential ACE or ACI issues at their entry points.

In Section 6.3, we analyzed all 31,117 dependents of those 301 packages with FAST [58], Explode.js [80] and NODEMEDIC-FINE, to determine whether these warnings are respected and how warning characteristics impact exploitability in dependent code.

Overall, we discovered 23 previously unknown vulnerabilities. These packages are not sanitizing the inputs before passing them to vulnerable dependency entry points, despite the warning. The security implications are serious: at least two packages with thousands of dependents are not sanizing their inputs internally, which facilitates the presence of security vulnerabilities.

Finally, we discuss limitations and threats to validity in Section 6.4: the accuracy of LLM triage and potential labeling bias in manual confirmation.

[1] We used the *all-the-packages* package (https://www.npmjs.com/package/all-the-packages) to extract all npm package names available in the repository.

6.2 LLM-Assisted Triage of Package Documentations

In this section, we describe the process that we have used to collect a dataset of packages with security warnings in their documentations.

Relevant warnings. Some packages documentations contain security-related warnings, regarding the unsafe usage of ACI or ACE sinks by at least one of their entry points. These warnings can be seen as delegating the input sanitization responsibility to dependent packages. We call these *relevant warnings*. We first triaged the npm repository using Large Language Models (LLMs) to identify a pool of packages that are flagged to potentially contain such warnings in their documentation, and then we manually verified all flagged packages.

First pre-filtering stage with a small model run. The LLM-assisted triage is not meant to perfectly identify all packages with warnings, but rather to build a large pool of packages that are more likely to have warnings, though small enough that we can feasibly validate manually.

We started from the complete npm repository, which had 3,540,963 packages at the time of gathering[2]. From those, we kept only the 1,670,928 packages that have at least 1 dependent package. Out of 1,670,928 packages, we successfully extracted non-empty documentations for 1,166,078 of those packages. We first ran a small, non-reasoning model (**gemma3:12b**)[3]) with a conservative prompt to reduce false negatives. Figure 6.1 describes the prompt used for this small model run.

That run flagged 51,713 documentations as potentially containing warnings.

Second pre-filtering with a larger model run. We used a larger reasoning model (**gpt-oss:20b**) to process the 51,713 documents that were signaled in the previous run. This larger model flagged 4,547 documents for manual review. The prompt used for the larger model is the same as that for the smaller model, except it does not include the instruction *"Do not summarize the provided text or provide any breakdowns of the text."*. This is because, unlike the smaller model, the larger model did not tend to include a summary of the documentation in its output, and it might also be useful for it to summarize the package documentation in its reasoning chain before producing an answer. Furthermore, gpt-oss:20b has three levels of reasoning (**Low**, **Medium**, and **High**): we used the **Medium** reasoning level. This larger model run signaled 4,547 out of the 51,713 documentations as potentially containing security-related warnings.

Manual confirmation of the presence of security warnings. We manually inspected those 4,547 packages: we filtered out cases where the signaled sentence only contained red-herrings or referred to other vulnerabilities besides ACI and ACE, and manually read the documentations of the remaining packages. We manually confirmed that 301 of those 4,547 packages indeed have relevant security warnings in their documentations. In some cases, we had to verify (by looking

[2]Gathering occured on August 17th, 2025.

[3]We also tried deepseek 1.5b and deepseek 8b, but gemma3:12b was the only tested package that correctly signaled 4 packages that we were already aware that contained warnings in their documentation, while not signaling a matching number of synthesized examples that did not contain warnings.

```
[{"role": "system", "content": """
Do not summarize the provided text or provide any breakdowns of the text. You will receive a \npm package documentation delimited by triple backticks. You MUST reply with EXACTLY:
-"yes" (lowercase, no punctuation) if the documentation contains a warning about passing unsafe, unsanitized inputs to some entry point, or warning developers to be careful in the usage of some of the entry points, for example if a function like eval, function constructor, exec or spawn is used carelessly. In this case, after 'yes', type a space and then the exact warning sentence.
-"no" (lowercase, no punctuation) if there is no such warning.
You must not add anything else.},
{"role": "user", "content": f"""Here are two examples:
DOC:
```
<positive example of a documentation with a warning>
```
Answer: """},
{"role": "assistant", "content":
"yes, 'Just be careful not to pass any unsanitized input to the \"exec_command\" entrypoint!'"},
{"role": "user", "content": """
DOC:
```
<negative example of a documentation with a warning>
```
Answer: """},
{"role":"assistant", "content": "no"},
{"role":"user", "content":f"""
DOC:
```
{doc}
```
Answer: """}]
```

Figure 6.1: Prompt for gemma3:12b model described in the chat ML language. We had to specifically request the model not to provide a summarization of the documentation: without that instruction, it tended to summarize the documentation instead of actually answering yes or not. Two examples are passed, one positive (described in Figure 6.2) and one negative (Figure 6.3). We filtered out packages for which the model output was EXACTLY "no", and perserved all other outputs for further analysis with a larger model, even those that did not start with "yes".

```
Here is the documentation for my fantastic package!

In order to use this package, users must be aware that string
   arguments are expected.
Just be careful not to pass any unsanitized input to the "
   exec_command" entry point!
```

Figure 6.2: Positive example of a warning: Responsability of sanitizing the input for some of the entry points is delegated to the dependent packages.

```
Welcome!

Sanitization is important, it prevents illness quite
   effectively.

We've got some great entry points here, for example "
   exec_command", that's a beauty. It will execute any
   commands you give it, it's perfect!
```

Figure 6.3: Negative example of a warning. It uses red-herring words like "sanitization" and "exec_command" but it is actually not warning users to sanitize the inputs; executing arbitrary commands is legitimate functionality.

at the actual code of a package) that the mentioned entry points code really used an ACE or ACI unsafely, since the documentation was sometimes ambiguous and only referred to potential security problems in a general sense. We provide a full list of the 301 packages, together with the sentence signaled by gpt-oss:20b in Table 8.1. We should note that the sentences signaled by the model were not always the exact warning sentences, or they were missing relevant context in other content of the documentation. For example, for `@iac-factory/tty-testing@0.1.9`, the signaled sentence was *"CLI utilities can be incredibly dangerous."* but the full documentation contained more information about what the danger was (use of a ACI sink).

6.3 Evaluation

In this section, we aim to answer the following research question:
RQ1: Do npm package developers heed security-related warnings contained in the documentation of their dependencies?

To answer this question, we followed the process described in Section 6.2 and collected a list of 301 packages with relevant security warnings their documentation. Then, we collected the full list of 31,117 NPM packages that, at the time of gathering, depended on at least one of those 301 packages. We refer to this final dataset of 31,117 packages as our **DepWarnings** dataset. Finally, we analyzed DepWarnings using current code injection detection tools, followed by

manual investigation of potential flows.

6.3.1 RQ1: Are ACI and ACE Security Warnings Respected by Dependent Packages?

We ran NMFINE-EM, NMFINE-NoEM, FAST, and Explode.js with a 5-minute time budget per package. We then manually inspected the reported flows[4] to determine whether dependent-package developers sanitized inputs passed to vulnerable dependency entry points.

Overview of results. Our manual inspection of all flows reported by the tools uncovered 23 previously unknown vulnerabilities: dependent packages that do not sanitize attacker-controlled inputs before passing them to vulnerable dependency entry points and that do not surface the corresponding security warnings in their own documentation. The majority of these vulnerabilities occur in dependents of `open` and `ejs`. Both packages instruct dependents to sanitize inputs, yet each has more than 10,000 dependents, all of which must consider the warning to avoid introducing vulnerabilities. Next, we investigate this situation in more depth.

Manual investigation methodology. For each dependent package with discovered flows, we manually assigned one of the following categories:

- **False positive, unexploitable**: The package is not exploitable. This occurs with unexploitable FAST flows or unexploitable potential flows from NodeMedic-FINE-2025 or Explode.js.
- **False positive, inherited_warning**: The package is technically exploitable but correctly propagates the dependency's warning in its own documentation or includes a newly added security warning.
- **False positive, legitimate**: The package is exploitable, but executing arbitrary commands or code is an intended feature.
- **False positive, atkmodel**: The package is exploitable but does not match our attacker model (i.e., attacker control of the entry point arguments is unreasonable).
- **Deprecated**: The package is deprecated.
- **True positive, irrelevant**: The package contains a vulnerability unrelated to the vulnerable dependency entry point.
- **True positive, relevant**: The package is vulnerable due to using a vulnerable dependency entry point with unsanitized input.

A breakdown of the categorization results is reported in Table 6.1.

Categorization results. The total column in Table 6.1 counts unique flows; a single package may contain flows reported by multiple tools. FAST is the only tool with unexploitable false positives among confirmed flows (55 out of 91, approximately 60%, which is consistent to what

[4]Credit for most of this manual analysis goes to Mindy Hsu. I only double-checked the true positive cases.

Category	FAST		Explode.js		NM-FINE		Total unique
	pot.	conf.	pot.	conf.	pot.	conf.	
False positive, unexploitable	95	55	47	0	48	0	184
False positive, inherited_warning	0	0	0	0	0	0	0
False positive, legitimate	9	2	4	0	42	5	53
False positive, atkmodel	18	13	8	1	1	1	26
Deprecated	1	1	0	0	1	0	2
True positive, irrelevant	10	7	18	0	25	19	50
True positive, relevant	14	13	2	0	8	3	23
Total	147	91	79	1	125	28	338

Table 6.1: Categorization of 338 potential (pot.) and confirmed (conf.) flows discovered by FAST, Explode.js and NODEMEDIC-FINE (NM-FINE) in the DepWarnings dataset. Confirmed flows are always a subset of potential flows.

we have discovered in prior manual analysis of FAST confirmed flows). No vulnerable package propagated its dependency's warning (**False positive, inherited_warning**: 0 cases). This demonstrates a fundamental weakness of relying on documentation-based warnings: users of dependent packages would need to read not only their direct dependencies documentation but also the documentation of deeper transitive dependencies, which is particularly infeasible in the npm ecosystem, where dependency trees are large. We are in the process of reporting all 73 confirmed vulnerabilities (50 + 23), including the 50 vulnerabilities not caused by vulnerable dependency entry points (i.e., **True positive, irrelevant** cases, mostly involving security issues in the dependent package's own code). We now focus on the 23 **True positive, relevant** cases in which the vulnerability stems directly from the dependent package's use of a vulnerable dependency entry point.

Analysis of relevant true positives. The relevant true positives stem from dependents of only four packages with warnings. Two of these dependencies have only a single dependent package each. We instead focus on the two popular dependencies with warnings:

- **open**: This package has 12,017 dependents. Some versions are vulnerable to ACI and 70% (16/23) of relevant vulnerabilities occur in dependents of `open`. Although the latest version is not vulnerable, most affected dependents use `open@0.0.5`, an older vulnerable version (without a CVE) that still receives 136,775 weekly downloads. Version 0.0.5 also includes a warning, though different from the original package.
- **ejs**: This package has 14,879 dependents. It is vulnerable to ACE and 22% (5/23) of relevant vulnerabilities occur in dependents of `ejs`. Four of these dependents rely on the latest version, which remains vulnerable.

One package contains two vulnerabilities and depends on both `open` and `ejs`. The issue here is clear: Thousands of dependents of these packages need to remember to sanitize the inputs, or they might be vulnerable.

Moreover, all four dependencies include warnings with clarity and ambiguity problems. For

example, the `open@0.0.5` documentation states: *"The same care should be taken when calling open as if you were calling child_process.exec directly. If it is an executable it will run in a new shell."*. This wording is, in our opinion, easy to misinterpret as a functional notice rather than a security warning.

In the case of `ejs`, the latest documentation states: *Security professionals, before reporting any security issues, please reference the SECURITY.md in this project, in particular, the following: "EJS is effectively a JavaScript runtime. Its entire job is to execute JavaScript. If you run the EJS render method without checking the inputs yourself, you are responsible for the results." In short, DO NOT submit 'vulnerabilities' that include this snippet of code: (...).*

A security researcher has looked at the `ejs` package already, finding that *"Although there is already a prompt now, it is more targeted at asking security researchers not to report, rather than asking developers not to use it in this way."* [53]. The package author has kept the same stance for years, repeatedly stating that input sanitization is the responsibility of dependents [83, 84, 85, 86, 87].

.

Result 1: We found 23 vulnerable packages that did not follow the security warnings documented in their dependencies. Most of these depend on either `ejs` or `open`, two widely used packages whose documentation places responsibility for input sanitization on dependents, facilitating situations under which ACI and ACE vulnerabilities can arise.

6.4 Threats to Validity

Regarding the completeness of the dataset of (301) packages with security warnings, although we did not encounter any instance of a warning present in a package documentation but missed by any of the LLMs, we expect such cases to exist. If true, our dataset of 301 packages with warnings would be incomplete. Despite recent improvements in LLM reading-comprehension capabilities [62], current models still suffer from hallucinations [33], preventing guarantees that all npm packages with relevant warnings are correctly signaled. In addition, documentations do not always clearly, unambiguously specify whether sanitization responsibilities lie with the dependent package, creating potential labeling bias in the construction of our dataset of 301 packages with warnings. Nevertheless, we manually verified the warnings in all four dependencies of the packages in which we discovered vulnerabilities and confirmed that each explicitly delegates sanitization to its dependents.

6.5 Conclusion

We examined whether developers of npm packages respond appropriately to security warnings documented by their dependencies. To do so, we conducted a large-scale analysis of 31,117 packages that depend on another package with a published security warning, leveraging state-of-the-art tools for detecting and confirming arbitrary code injection and arbitrary code execution vulnerabilities. This study uncovered 23 previously unknown vulnerabilities in packages that fail to sanitize user inputs. The results highlight a significant security concern: at least two

widely used npm packages, each with more than 10,000 dependents, delegate input sanitization responsibilities to their dependents, and not all dependents handle this responsibility correctly.

Chapter 7

Conclusion

This chapter summarizes the contributions of this thesis (Section 7.1), outlines future research directions (Section 7.2), and concludes with final remarks (Section 7.3).

7.1 Summary

In this thesis, we have shown that a substantial number of JavaScript-based applications continue to suffer from code-injection vulnerabilities, including arbitrary command injection (ACI), arbitrary code execution (ACE), and the latter's client-side counterpart, DOM-based cross-site scripting (DOM-XSS). These vulnerabilities are among the most severe classes of software flaws, potentially leading to full remote compromise when exploited. Their impact is compounded by the interconnected nature of both the web and the npm ecosystem, e.g., the security of a package may depend on the integrity of a large and evolving dependency chain.

The primary contributions of this thesis address three challenges: (1) measuring the prevalence of DOM-XSS vulnerabilities on the web and automatically detecting DOM-XSS even when they require user interactions or specific GET parameters; (2) detecting and confirming ACI and ACE vulnerabilities in npm packages, including cases requiring complex input structures; and (3) identifying features of program paths that correlate with automatic exploitability, enabling more efficient vulnerability confirmation.

Chapter 3 introduced a new DOM-XSS detection infrastructure featuring a fuzzing component that simulates user interactions to trigger event-driven code. This design addresses a key limitation of prior work, which largely overlooked vulnerabilities gated behind interactions. In a recent large-scale experiment, this system identified 15% more vulnerabilities than prior approaches. Through a comprehensive comparison and careful replication of prior studies, we further showed that improvements in modern browser URL-encoding mechanisms significantly influence the exploitability of DOM-XSS flows, even though that feature was not introduced for security purposes.

Chapter 4 shifted focus to server-side analysis. We enhanced NODEMEDIC, a state-of-the-art dynamic analysis tool for identifying and confirming ACI and ACE vulnerabilities, by integrating a fuzzer capable of generating complex structured inputs. We also developed techniques for

synthesizing valid JavaScript payloads required for confirming many ACE vulnerabilities. These enhancements led to a 65% increase in confirmed flows over a version of NODEMEDIC without our fuzzer. In particular, our ACE-confirmation method enabled the discovery of 21% more confirmed ACE flows.

Chapter 5 examined what features of program paths most strongly indicate automatic exploitability and used these insights to guide NODEMEDIC-FINE's fuzzer toward high-value paths. This guidance preserved the number of potential flows and yielded a small increase in confirmed flows (1%), but more importantly reduced confirmation time: In long-timeout experiments, the exploitability-guided approach is 28% faster than the baseline at confirming the same set of flows.

Vulnerabilities in npm packages can affect any dependent that calls a vulnerable entry point with unsanitized inputs. In Chapter 6, we examined developer practices around security warnings. Using an LLM-based pre-filtering step followed by manual inspection, we collected 301 packages whose documentation includes explicit security warnings, and analyzed their 31,117 dependents with NODEMEDIC-FINE, FAST, and Explode.js. This effort uncovered 23 previously unknown vulnerabilities stemming from dependents that failed to respect the documented warnings.

7.2 Future Directions

Autonomous vulnerability-analysis tools have advanced substantially, yet their current capabilities leave multiple important research problems open across scalability, coverage and exploit synthesis.

Better package drivers. NODEMEDIC-FINE does not currently synthesize exploits that require multiple entry point invocations. Explode.js [80] has advanced this capability by supporting exploit chains composed of linear chains of calls (i.e., where each element in the chain calls a function obtained from the previous call). However, it still cannot reason about arbitrary inter-entry point interactions. Supporting such interactions is difficult because identifying all meaningful combinations of calls is combinatorially explosive, and often unbounded. One direction is to analyze how dependent packages use the target package in practice, leveraging developer-authored usage patterns. While this would not cover all interactions, it may capture those most likely to matter in real-world exploitation.

Broader package support. Chapter 5 exposed fundamental limitations in the coverage of current state-of-the-art tools FAST, Explode.js, and NODEMEDIC-FINE. From the Random120k dataset, comprised of 120,000 packages, NODEMEDIC-FINE was able to initiate analysis for only 21%. FAST and Explode.js could analyze a larger fraction due to their static nature (75% and 58% of Random120k), but successful exploit validation still requires packages to be installable and runnable. Improving the ability of tools to automatically install, initialize, and execute a wider range of npm packages could have a direct and substantial impact on ecosystem security by enabling broader coverage.

Developer study on packages with security warnings. In Chapter 5, we identified 23 packages that unsafely ignore security warnings documented by their dependencies. Because some maintainers choose to document security expectations rather than implement sanitization internally, it is important to understand how developers perceive such warnings and how they expect to receive security-relevant information. A systematic study of developer attitudes and practices could inform better documentation, tooling, or automated feedback mechanisms aimed at reducing misuse.

7.3 Concluding Thoughts

Automated detection and confirmation of code-injection vulnerabilities provide powerful capabilities for both offensive and defensive security. Although this thesis foregrounds an attacker-oriented view, its results directly support defensive efforts. Our analysis uncovered numerous real vulnerabilities, several of which have already been patched. By improving the ability to identify and confirm vulnerabilities in both client-side and server-side JavaScript, we believe this work was meaningful, as it contributes to strengthening the security of the broader JavaScript-based ecosystem and the users who rely on it.

Chapter 8

Supplementar material

8.1 NODEMEDIC-FINE Supplementar Material

8.1.1 Supported Sinks

In Figure 8.1 we describe all sinks that are supported by NODEMEDIC-FINE, categorized by vulnerability type (ACI versus ACE). Different sinks have different argument that need to be tainted in order for a flow to be exploitable. While this is trivial for sinks like `eval` or `exec`, which only receive one argument, it is still an important nuance for the `Function` constructor for example. In the `Function` constructor ACE sinks, the last argument is the actual code, while the initial arguments are simply argument names that can be used in the function body. We include sinks from the `vm` module since the documentation warns users not to run untrusted code with its methods [95].

Sink Type	Sink Name	Sink Description
ACE	`eval`	Evaluates a given string argument as JavaScript code.
ACE	`Function` constructor	Create a new function from given string arguments.
ACE	`runInNewContext`	Run given JavaScript code. Only safe in a new process.
ACE	`runInThisContext`	Run given JavaScript code. Still has access to `global` object.
ACE	`runInContext`	Run given JavaScript code in a given context.
ACI	`execSync`	Synchronously spawn a shell command given as a string.
ACI	`exec`	Asynchronously spawn a shell command given as a string.
ACI	`spawn`	Spawns a new process using a given command. `shell=true` needs to be set for exploitability.
ACI	`spawnSync`	Synchronously spawn a new process using a given command. `shell=true` needs to be set for exploitability

Figure 8.1: Core ACE and ACI sinks supported by NODEMEDIC-FINE.

```
return new Function("x", "with (x) {  return " + user_input + " }  ")
```

(a) Code showing the sink call.

```
with (x) {  return
```

(b) Prefix

```
[[  <payload>, <literal: '} '> ]]
```

(c) Completion template given by the Enumerator.

```
global.CTF()}  //
```

(d) Exploit that was synthesized using the template provided by the Enumerator.

Figure 8.2: Prefix, completion, and exploit synthesized for a real-world prefix.

8.1.2 Example Enumerator Completion

In Figure 8.2 we illustrate an example from one of the 27 cases that the Enumerator successfully completed, as well as the accompanying proof-of-concept exploit that was built with its help. The vulnerable code is a simple call to the `Function` constructor ACE sink, which concatenates a fixed prefix containing part of a `with` statement with the user input, and a terminating character delineating the body of the `with` statement. Our Enumerator came up with a completion that represented the fact that any arbitrary statement could be placed on the `<payload>` part, followed by a closing bracket to close the `with` body. Regardless of the completion, NODEMEDIC-FINE's synthesis algorithm attempts to put a comment at the end of the attacker-controlled portion, so that whatever comes next is ignored. The final input is then synthesized taking into account the template offered by the Enumerator.

8.1.3 Vulnerability Characteristics

Figure 8.3 sheds light on how the proportion of packages with potential and confirmed flows, as detected by NODEMEDIC-FINE, varies in relation to their popularity. Furthermore, Figure 8.4 correlates the code size of these packages with the presence of flows. The influence of the overall package size on the likelihood of identifying flows is depicted in Figure 8.5. Figure 8.6 examines how the depth of the dependency tree is associated with flows while Figure 8.7 shows the effect of the number of unique dependencies on the detection of a flow.

These figures collectively indicate that the population of packages where we find sinks is reasonably diverse in terms of these properties. We observe that while popularity does not seem to significantly impact the performance of our tool, metrics that measure package complexity more directly do. We are still able to analyze very large packages, as our fuzzer is designed to analyze each public entry point separately. Although vulnerabilities can be missed if they require more complex interaction with the package API besides calling a single entry point with

a specific payload.

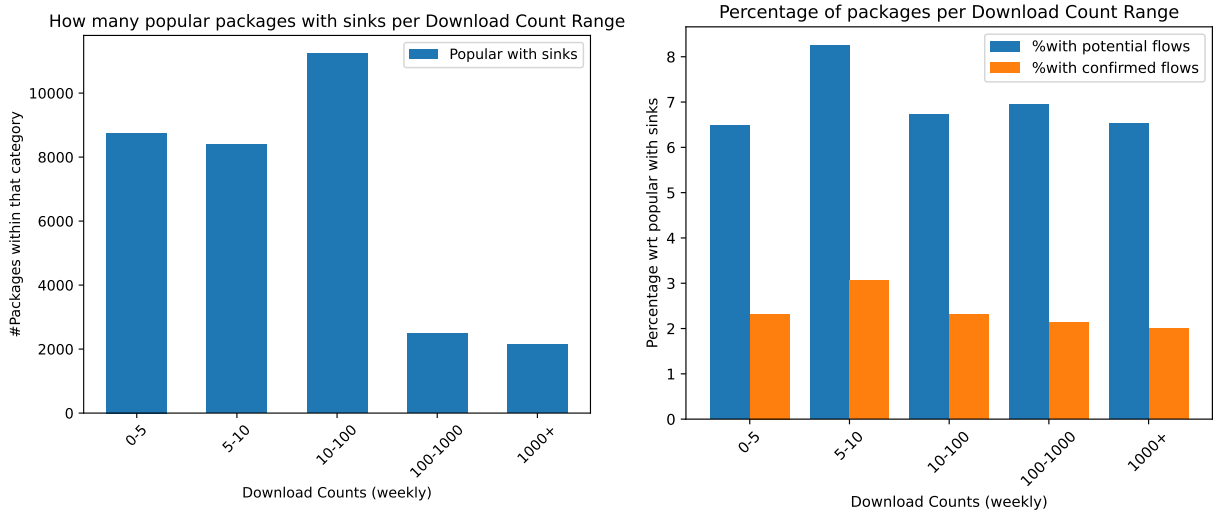

Figure 8.3: Frequency of packages within ranges of download counts, split into "with sinks", "with potential flows" and "with confirmed flows".

8.1.4 Fuzzing Timeout

In our NODEMEDIC-FINE, we needed to set a timeout that was sufficiently short to scalably analyze all packages in the NPM-DATASET dataset (33,011 packages) but long enough that we would not miss a significant number of potential flows. In Figure 8.8 we plot the accumulated number of flows found as time progresses during fuzzing. Overall, we start seeing diminishing returns around the 30 seconds mark. We chose a timeout of 2 minutes for fuzzing.

8.1.5 LLM Signaled Sentences in Packages Confirmed to have Warnings

Below follows a table with all packages that were confirmed to have security-relevant warnings in their documentations, along with the exact warning sentence that was signaled by the LLM. Note that some of the signaled sentences are ambiguous and missing relevant context from other content in the actual documentation.

Package (name@version)	Warning sentence
@akepinski/remark-math@5.1.1-cjs	Always be wary of user input and use rehype-sanitize.'

Continued on next page

Package (name@version)	Warning sentence
gunsafe@2.6.0	The ability to run stored code adds some additional possibilities to the versatility of Gunsafe, but use with care! The ‘–run –global‘ argument uses ‘eval()‘, for example.
mongo-escape@2.0.6	’Don’t rely on this module for escaping the [mapReduce] command or [$where] operator as these commands parse and execute their values as JavaScript.’
@fiad/twig-addons@1.3.0	[dangersign] Since this filter involves the usage of *eval()*, it’s recommended to use it for static sites generation purposes only, so that the stringified *JavaScript* execution will be limited to the development environment. Look [here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval#never_use_eval!) to learn more.
node-lambda-babel-template@2.4.0	Piping anything unknown through ‘bash‘ can be dangerous!
fs-dot@5.0.0	’whatever you pass as arguments will passed through to the fs commands without validation’
jse-eval@1.5.2	IMPORTANT: As mentioned under [Security](#security) below, this library does not attempt to provide a secure sandbox for evaluation. Evaluation involving user inputs (expressions or values) may lead to unsafe behavior. If your project requires a secure sandbox, consider alternatives such as [vm2](https://www.npmjs.com/package/vm2).
@bootcamp-ra/mocha-as-promised@1.0.7	This library is **UNSAFE** as it uses ‘eval‘ to load the code and the test code (check out Usage to understand why).’
crystalgazer@0.9.0	’We’re not doing any validation, just passing your input to the command.’
hst-virtual-dom@0.4.5	We use ‘Function‘ which use ‘eval‘ to implement this, so there may be some security problem.
exec-string@1.0.2	Make sure you undestand why JavaScript’s eval function is no longer used. Please refer to [this link](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval#never_use_eval!) for more information.
j2j@0.1.2	’The JavaScript you use as input is evaluated which means your code gets executed.’
cgijs@1.2.0	’Please consider your ‘security risks based on your executable’s/ script’s security risks‘.’
stenoread-nodejs@1.0.1	I know that passing on parameters to a shell command might be a bad idea ;-).

Continued on next page

Package (name@version)	Warning sentence
jsx-to-json-brick@0.0.9	注意：eval如果您不控制输入，不仅使用危险，而且的结果jsx2json可能不是纯JSON。
@olzie-12/ftp-deploy@1.2.49	'be sure to escape quotes and spaces'
shelljs-exec-proxy@0.2.1	'Current versions of ShellJS export the '.exec()' method, which if not used carefully, could introduce command injection Vulnerabilities to your module.'
preschool@0.2.1	'Specifications make use of JavaScript's eval. While using eval is typically considered to be bad form, when calling another function the performance does not appear suffer in V8.'
atlas-stddev@1.0.0	'The caller is expected to sanitize input.'
radspec@1.12.1	'You either need to write your own JavaScript VM or use eval (unsafe!) from inside JavaScript'
minimount@1.1.1	As you know, eval is evil, so use this carefully.
javawebstart@1.1.0	It is up to you to validate if the JNLP and jar file you wish to run come from a **trusted source**
@ra-bootcamp/mocha@1.3.0	'This library is **UNSAFE** as it uses 'eval' to load the code and the test code (check out Usage to understand why).'
comaas@0.3.3	'!!! Do not come with security system, you have to implement your own security layer. !!!'
@aws-cdk/aws-s3-deployment@1.204.0	**IMPORTANT** The 'aws-s3-deployment' module is only intended to be used with zip files from trusted sources. Directories bundled by the CDK CLI (by using 'Source.asset()' on a directory) are safe. If you are using 'Source.asset()' or 'Source.bucket()' to reference an existing zip file, make sure you trust the file you are referencing. Zips from untrusted sources might be able to execute arbitrary code in the Lambda Function used by this module, and use its permissions to read or write unexpected files in the S3 bucket.'
cholesky-solve@0.2.1	NOTE the module does no sanity checking on the input arguments. It is assumed that the user knows what he/she is doing!
markdown-it-template-literals@1.0.0	This package uses evil eval, be aware of injection attacks!
construct-from-spec@0.1.0	'Please note, that this can lead to security vulnerabilities, cause config author can do an arbitrary code execution.'
hydroplane@1.0.7	I built it in like 2 hours and it literally uses eval().

Continued on next page

Package (name@version)	Warning sentence
jsontype@2.1.0	Caution: Do not accept validation schemas from the user because they could harm you.
expression-eval@5.0.1	'Evaluation involving user inputs (expressions or values) may lead to unsafe behavior.'
coroutiner@0.3.3	Just be careful what you run this over.
git-revision-webpack-plugin@5.0.0	'This configuration is not not meant to accept arbitrary user input and it is executed by the plugin without any sanitization.'
@hagana/hagana@1.0.0	[dangersign] Something that I still need to think about is the fact that using the 'commands startsWith' approach is it opens a hole that allows an attacker to run 'node –version && cat /.ssh/id_rsa' which is clearly a problem.
mobx-modules@0.0.2	By carefull not to introduce security issues in your app through this module (uses: new Function)
@flk/parser@0.1.1	Again we emphasize to only use this when feeding jsdom code you know is safe. If you use it on arbitrary user-supplied code, or code from the Internet, you are effectively running untrusted Node.js code, and your machine could be compromised.
es6format@1.0.1	es6format makes use of Function(), so it should never be called on format strings that are not known to be safe.
json-rule-processor@1.2.3	By default, 'eval' of JavaScript in strings is turned **off** for security reasons, but can be activated with this flag.
static-eval@2.1.1	'static-eval is like 'eval'. It is intended for use in build scripts and code transformations, doing some evaluation at build time—it is **NOT** suitable for handling arbitrary untrusted user input. Malicious user input _can_ execute arbitrary code.'
dhl-nolp@1.0.1	'This library uses JSDOM's JavaScript execution machanism which might potentially result in code injection vulnerabilities, given the DHL page contains malicious JavaScript.'
asyncme@0.1.0	'The task functions are serialized and then executed on the child process using the [Function constructor] which will practically 'eval()' the function, so keep that in mind a'
@caijs/eval@1.0.3	It is NOT suitable for handling arbitrary untrusted user input. Malicious user input can execute arbitrary code.'
ptolemy@0.0.3	'always be careful when using eval'
@axway/api-builder-plugin-fn-javascript@4.0.2	[dangersign] **Warning**: You should never use this flow-node to execute code coming from untrusted sources.

Continued on next page

Package (name@version)	Warning sentence
frontmatter@0.0.3	For untrusted source, the ‘safeLoad‘ option should be used:
node-command-line@2.0.2	’I’ve added a basic sanitization step that removes characters commonly associated with shell command injection attacks. This helps prevent unwanted characters from being executed within the shell command.’
lazaretto@4.0.2	Lazaretto should not be relied on for completely safe isolation, the file system and so forth can still be accessed so you still need containers/vms for safe isolation of user code
blobs@2.3.0-beta.2	’_Options are **not** [sanitized](https://en.wikipedia.org/wiki/HTML_sanitization). Never trust raw user-submitted values in the options._’
burly-bouncer@1.0.3	’Never trust the end-user to supply you with valid information.’
react-editable-json-tree@2.3.0	’This library was previously affected by an ‘eval‘ security vulnerability.’
yamler@1.0.1	’If you do not trust or control the contents of the yml file, use true.’
@cyyynthia/jscert@0.1.2	**WARNING**: For the time being, you shouldn’t pass untrusted data to the lib or unexpected things might happen to your cat. While the lib should be working for *valid* data, it lacks on proper hardening of the parsing bits and proper data validation/error handling, so maliciously crafted bits of data may cause some damage.
html2image@2.0.0	If the url address contain **&**, you should use **encodeURIComponent** to encode the url address, then pass to the html2image command
@egoist/devalue@2.0.3	When using ‘eval‘, ensure that you call it *indirectly* so that the evaluated code doesn’t have access to the surrounding scope:
@iac-factory/cli-utilities@0.7.26	Having the ability to issue os.exec or interface stdin always makes the application dangerous.
pomodoro-cli@0.2.1	WARNING: The command is passed directly to a shell with the same user permissions this program runs under – use with caution!
improv@1.0.0	’Improv does absolutely no validation or security checking of anyt’
js-solver@0.0.2	The equations are evaled, so be aware of that. Also I used with(Math) to get the math stuff to eval... yeah...
@artit91/exec@1.0.0	User input should be escaped!
knorke@1.0.2	Most config options are used verbatim when executing shell commands. Do _not_ pass unsanitized user input to this script. You have been warned!

Continued on next page

Package (name@version)	Warning sentence
nicedice@1.0.2	If you manage a JSFuck-style arbitrary code execution attack using only the characters mentioned above, digits, and/or parentheses, please tell me!!!
eval-expression@1.0.0	**Be warned:** Take [all precausions which apply to using ‘eval‘][eval-precautions]. ‘eval-expression‘ is no safer, no more performant and no easier to debug.
@devjoyvn/autoloader-ts@2.2.2	DO NOT misuse this.
stahp@0.9.0	’If handling untrusted code it is recommended to use this alongside a module like VM2.’
jsonpickle-port@1.0.0	Only load data you have personally produced if you want to be safe.
hydro.pdf@1.0.4	’否则有 XSS 风险。’
vime@0.0.1	’Important note: this module is currently not designed for executing untrusted code (”sandboxing.”) If that is an important use case for you, please file a bug report.’
homebridge-wyze-robovac@1.5.1	Be careful to check that the generated keys do not contain any ’shell’ special characters like ’*’ or ’—’ (vertical bar).
react-google-tag-manager@2.2.1	As ‘eval‘ can be used to do harm, make sure that you are understanding what you are doing here and read through the scrip
flatpad@0.2.2	Sandbox realized by **with** and **eval** is not suitable for esm.
@momsfriendlydevco/eval@1.0.0	Eval is massivly unsafe and unless used exactly right can cause major harm.
pursuit-core@0.0.1	’Caveat: It does use new Function to compile the generated code into functional code, so take great precautions with what you trust it with.’
email@0.2.6	’Some protection against injection attacks is enabled. Use at your own risk. Or better yet, fork it and submit something better!’
ktxml@1.0.6	’##### 库中使用到了 eval. 十分危险, 谨防注入’
ez-fasta@0.1.0	В н и м а н и е ! Э т а ф у н к ц и я н е д е л а е т н и к а к и х п р е д в а р и т е л ь н ы х п р о в е р о к н а в а л и д н о с т ь в х о д я щ и х ф а й л о в и ч и т а е т в с ё п о д р я д .
sami.js@2.0.1	’아무 처리도 하지 않은 데이터이니 XSS에 주의하세요!’
curling@1.1.0	’The following characters are not allowed to be provided as part of a command to prevent possible command injections:’

Continued on next page

Package (name@version)	Warning sentence
page-evaluate@1.1.0	'Note: Sanitize any input, but this is safe - "fresh copies of all the JavaScript spec-provided globals [are] installed on window" - [jsdom readme]'
@typicalninja21/urlrequest@2.0.1	Input is not safe, and it is not meant to be
limelightdb@3.1.5	'Because the filter is just a JavaScript function, it could lead to remote code execution.'
nbk@0.1.31	Do not run the Web UI on a port open to public traffic! Doing so would allow remote code execution on your machine.
cupjs@1.3.1	采用正则替换生成函数，函数再生成HTML字符串方式，并未加上安全性的措施，请用于可信任的内容生成。
js-string-format@0.1.0	'this library does not sanitize the input other than simple type checking'
easyimage-tmpfix@0.1.4	'when you want to call a custom command to ImageMagick, you will need to take care of escaping special characters etc'
compilers@2.0.0	Specifications make use of JavaScript's 'eval'. While using 'eval' is typically considered to be bad form, when calling another function the performance does not appear suffer in V8. See http://jsperf.com/eval-function-call.
@voidptr9/service@1.2.0	**Warning:** If you're not fond of 'new Function' and language-semantics-breaking designs, then Service is not what you should be using.
meval@1.1.0	This is **not** a "safe JavaScript eval"!
js-string-to-value@1.2.0	However if for some reason the parser is reporting the variable is not valid but is, you can still run unsafeParseJs() which will skip validation (it may also be faster, however only use this if you know what and where your data is and is coming from)
ra-mocha-as-promised@1.1.10	This library is **UNSAFE** as it uses 'eval' to load the code and the test code (check out Usage to understand why).
canvas-sequencer@3.1.0	'This is error-prone and risky however, and exposes you to all the incumbent problems of the eval() function.'
babel-plugin-codegen-dynimport@1.0.0	'All code run by 'codegen' is _not_ run in a sandboxed environment'
@phase2/outline-include@0.1.5	Allows included scripts to be executed. You must ensure the content you're including is trusted, otherwise this option can lead to XSS vulnerabilities in your app!

Continued on next page

Package (name@version)	Warning sentence
decompress-zip@0.3.3	Setting to false has significant security implications if you are extracting untrusted data.
corporate-punk@1.2.0	'This app is just passing query parameters directly to an 'imagemagick' wrapper, without any kind of sanity check, **bad shit can happen**.'
farse@0.2.2	'In general, I recommend you default to using farse.inverse.inexact over farse.inverse.exact. The .inexact version, though less precise, is slightly safer because it uses the Function'
@caijs/python-eval@1.0.1	It is NOT suitable for handling arbitrary untrusted user input.
pppipe@0.0.3	'Never use this in production.'
@payid-org/payid-cli@1.0.4	'when passing a PayID as an argument in non-interactive mode, the PayID must be escaped or quoted to avoid the '$' being interpolated as a variable by the shell.'
picotemplate@0.0.2	This module heavily uses 'eval()'. You should audit it before use and not pass it user provided strings.
gbkuai-shadowsocksconfig@0.0.8	'No sanitization is performed for these fields. Client code is responsible for sanitizing these values when received from untrusted input.'
run-on-server@3.2.0	Out of the box, this effectively gives the client serverside 'eval'.
@mischback/imp@2.1.0	You should not expose ImP publicly, at least not without some wrapper that does perform sanitarization of any user input.'
qtools-parse-command-line@1.0.9	'For historical reasons, it defaults to evaluating incoming Javascript. This can, of course, be dangerous. It should be called with the 'noFunctions:true' argument unless you need function evaluation.'
apple-java-script@1.0.1	AppleJavaScript direct call could be dangerous because it uses 'eval' behind the scenes to parse returned AppleScript value.
justcurl@1.0.3	'WARNING: Make sure to sanitize any user input. Some precautions are already taken, but it's in no way perfect.'
arepl-backend@3.0.5	this should ONLY be used to execute trusted code. It does not have any security features whatsoever.
pursuit@0.3.1	'It does use 'new Function' to compile the generated code into functional code, so take great precautions with what you trust it with. Think twice before using it to generate code on the client-side.'
@debonet/es6format@1.0.1	es6format makes use of Function(), so it should never be called on format strings that are not known to be safe.

Continued on next page

Package (name@version)	Warning sentence
reshape-code-gen@2.0.0	'However, if any type of outside user input is accepted and evaluated as a ‘code‘ node, or if a malicious plugin is being used, you have a serious security vulnerability on your hands.'
@ramumb/strip-tags@0.1.3	'Lastly, while stripTags is good enough for most purposes, it shouldn't be relied upon for security purposes.'
cli-input@0.2.0	Caution: the above example executes commands via the shell, be careful.
json-to-styled@0.1.2	## Warning: uses Function()/eval() so don't use it on a server or a production environment
blender-compiler@0.1.1	**Precaution** - Eval is used to compile the shader.
gulp-shell@0.8.0	**WARNING**: [Using command templates can be extremely dangerous](https://github.com/sun-zheng-an/gulp-shell/issues/83). Don't shoot yourself in the foot by passing arguments like $(rm -rf $HOME).
graphql-playground-html@1.6.30	You must sanitize any and all user input values to ‘renderPlaygroundPage()‘ values.
safe-eval@0.4.1	'Be careful about the objects you are passing to the context API, because they can completely defeat the purpose of safe-eval.'
browser-redirect@1.0.2	:warning: Please don't install this package. This package allows [OS command injection](https://portswigger.net/web-security/os-command-injection)
@phase2/outline-icon@0.1.5	'WARNING: Be sure you trust the content you are including as it will be executed as code and can result in XSS attacks.'
m2m-supervisor@0.1.1	Support remote submission of local 'shell' commands – _**user caution is advised!**_
run-script@0.1.1	Please note: this is a simple wrapper for creating new dynamic functions, similar to eval, which has performance and security issues.
@innotrade/enapso-sparql-client@1.1.9	It's inadvisable to concatenate strings in order to write a query, especially if data is coming from untrusted sources.
@entan.gl/vsce@1.79.6	**Warning:** When using vsce as a library be sure to sanitize any user input used in API calls, as a security measure.
codex-function@1.0.1	'THIS WILL DOWNLOAD CODE FROM OPENAI AND EVAL IT'
jsx2json@1.0.1	'*Note:* Not only is using ‘eval‘ dangerous if you aren't controlling the input, but the result of ‘jsx2json‘ may not be pure JSON.'

Continued on next page

Package (name@version)	Warning sentence
peertube-plugin-simplelogo@0.0.6	There is no sanitarization for your inputs (neither url or width). We assume that administrators are not evil, and don't do XSS and co.
@58fe/hammer-security@0.1.1	远程命令执行漏洞，用户通过浏览器提交执行命令，由于服务器端没有针对执行函数做过滤，导致可以执行命令，通常可导致入侵服务器。
alicatejs@0.1.1	All templating engines allow some level of arbitrary expression execution, which could rely on something like 'eval', or have a built in parser for such evaluations.
muffin-js@1.1.2	Remember: EVAL IS EVIL!
@nexssp/expression-parser@1.0.5	**Important:** Uses JavaScript evaluation - only use with trusted input sources.
un-eval@1.2.0	'You should avoid using un_eval any untrustable objects (maybe from user input) then eval it. NEVER use eval unless you know what will happen.'
@ardatan/fast-json-stringify@0.0.6	Treat the schema definition as application code, it is not safe to use user-provided schemas.
@cjs-mifi-test/execa@6.0.0	We recommend against using this option since it is: - not cross-platform, encouraging shell-specific syntax. - slower, because of the additional shell interpretation. - unsafe, potentially allowing command injection.
textops@0.0.2	That being said, you should still only run text operations on text from trusted sources or in a secure sandbox.
module-cache@1.0.4	DO NOT USE THIS MODULE TO LOAD UNTRUSTED MODULES!
graphql-playground-html-patched@1.9.5	'You must sanitize any and all user input values to 'renderPlaygroundPage()' values.'
nbob@2.1.0	'The version of Handlebars (v3.0.3) that is used by nBob has a vulnerability that allows remote attackers to conduct cross-site scripting (XSS) attacks by leveraging a template with an attribute that is not quoted.'
node-retrieve-globals@6.0.1	[dangersign] The 'node:vm' module is not a security mechanism. Do not use it to run untrusted code.
funjson@2.1.0	'Implemented using 'eval', don't parse untrusted JSON!'
express-template-to-pdf@1.0.5	'This is one way to enable running Puppeteer in Docker but may be a security issue if you are loading untrusted content, in which case you should override these defaults.'

Continued on next page

Package (name@version)	Warning sentence
youtube-ext@1.1.25	[dangersign] YouTube stream data is decoded by evaluating arbitrary JavaScript code. By default, youtube-ext uses ‘eval‘ or ‘node:vm‘. Please install [isolated-vm](https://www.npmjs.com/package/isolated-vm) or [@ohmyvm/vm](https://www.npmjs.com/package/@ohmyvm/vm) to prevent security issues.
@nomad-xyz/contracts-da-bridge@1.0.0	You should never run ‘forge –ffi‘ without knowing what exactly are the shell commands that will be executed, as the testing suite could be malicious and execute malicious commands.
poppins-exec@0.1.0	’Be careful how you use this, or you’ll subject yourself to [shell injection](http://en.wikipedia.org/wiki/Code_injection#Shell_injection).’
webpack-Minimount-starter@0.1.3	As you know, eval is evil, so use this carefully.
ying@2.3.0	’This library uses ‘new Function‘ (which uses ‘eval‘) to compile template code. You should sanitize the data when dealing with user input.’
discord-eval.ts@1.1.3	Think of securing access because a malicious Eval can be devastating for your PC!
electron-toolkit@1.0.24	Override and disable eval , which allows strings to be executed as code
@tilastokeskus/cross-spawn@5.0.1	’You must manually escape the command and arguments which is very error prone, specially when passing user input’
nodepub@3.2.1	’This is a utility module, not a user-facing one. In other words it is assumed that the caller has already validated the inputs. Only basic sanity checks are performed.’
@iac-factory/git-clone@0.4.9	Having the ability to issue ‘os.exec‘ or interface ‘stdin‘ always makes the application dangerous.’
comment-regex@2.0.0	Do not use it with untrusted user input.
firestore-serializers@1.0.3	’However, keep in mind the old approach may be susceptible to injection attacks!’
@lpezet/etl-js@3.1.2	Commands, scripts and more can be executed as part of the Mods defined in the template. Therefore, you should make sure to use only the Mods you trust in your ETL Template.’

Continued on next page

Package (name@version)	Warning sentence
intershop-lazy@1.0.1	'Usage of ‘create_lazy_producer()‘ is inherently unsafe; therefore, no untrusted data (such as coming from web form as data source) should be used to call this function (although the function that ‘create_lazy_producer()‘ creates is itself deemed safe).'
beardfondle@0.1.6	DO NOT USE THIS LIBRARY IN PRODUCTION. ‘eval()‘ IS ALMOST NEVER SAFE.
gist-init@1.0.1	'be careful, don't use this on anyone elses work but your own.'
commandbox_remote@0.1.4	'*This essentially starts a network-accessible shell, so it should be used with extreme caution!*'
expr2fn@1.0.0	'The context in which ‘eval‘ and ‘Function‘ were invoked can be modified by expressions, especially when they are provided by the user.'
gulp-math@1.0.0	Please note, ‘gulp-math‘ uses the ‘eval‘ function for evaluating expressions.
node-shred@1.1.0	Defenses against [command-injection attacks][1] have been put in place, except with regards to the ‘shredPath‘ parameter, which cannot be defended.
@matteodisabatino/typed-env@2.0.0	For conversion into Functions the built-in object eval() is used, however eval() is known to be problematic since statement is directly executed and this exposes the application to security risks. So, please, use this conversion with caution.
bollireact@1.2.0	'*NOTE*: I basically ‘child_process.exec‘ a bunch of shit so use with caution!'
mock-globals@0.1.5	'**WARNING**: this is not a secure sandbox and is not intended for running untrusted code! The "protection" it provides is only proof against *accidental* global modifications, and can be trivially bypassed in several ways that I can easily think of, and probably hundreds of less-trivial ways. It is intended only for running tests, with *no thought given to any actual security*.'
usher-cli@2.15.1	'For the ‘shell‘ command, beware that the properties defined in the usher file can overwrite those used by [child_process.exec](https://nodejs.org/api/child_process.html) module.'
@gzzhanghao/jsdom@11.0.1	Again we emphasize to only use this when feeding jsdom code you know is safe. If you use it on arbitrary user-supplied code, or code from the Internet, you are effectively running untrusted Node.js code, and your machine could be compromised.

Continued on next page

Package (name@version)	Warning sentence
node-tezzeract@0.0.3	- consider sanitizing inputs to ‘exec‘ such that they do not contain shell meta-characters such as ‘$()‘
@zent/codemods@1.0.1	[dangersign] Codemods are not guaranteed to be safe for all inputs, you must review the output
code-execution-engine@0.4.6	[redexclamationsign]This package is not secure by default. Visit [Security](#security) for production projects.[redexclamationsign]
k8s-resource-parser@0.2.2	’Only use when you expect correct strings (e.g. coming directly from the K8s API), as the parser isn’t designed to handle maliciously-crafted inputs.’
tdp-glob-file-copier@0.1.2-alpha	’* **This module is somewhat insecure currently** e.g. some config params are used in shell commands.’
js-exec@1.2.4	’Executing an inputted string, as JS code can be **Extremely** risky.’
fis3-deploy-ala-http-push@2.0.11	’**此代码存在很大的安全隐患，没有做任何安全考虑，请不要部署到线上服务。**’
as-run-js@0.0.2	’This module use ‘eval()‘.’
@zvenigora/jse-eval@1.10.0	Evaluation involving user inputs (expressions or values) may lead to unsafe behavior.
brackets@0.5.8	’Since this imposes very serious security and stability risks, Brackets Server will not load nor execute domains from user extensions, unless ‘-d‘ option is specified.’
@hitsuji_no_shippo/self-referenced-object@4.0.0	self-referenced-object evaluates any expressions inside template literals by calling ‘pass:[Function(’”use strict”; return ‘’ + expression + ‘;”)()]‘ which is a marginally safer version of ‘eval‘ (ie still incredibly unsafe), so you should avoid passing any untrusted data into an object evaluated by ‘resolveReferencesInObject‘ (or at least don’t self-reference untrusted data in an ‘resolveReferencesInObject‘ evaluated object).
@bricehabib/react-lottie-player@1.4.2	The default lottie player uses ‘eval‘. If you don’t want eval to be used in your code base, you can instead import ‘react-lottie-player/dist/LottiePlayerLight‘.
firedev-vsce@0.0.4	When using vsce as a library be sure to sanitize any user input used in API calls, as a security measurement.
object-string-parser@0.1.0	[dangersign][dangersign][dangersign] actually the source code is only one sentence, and not secure, so don’t use it
@invertase/next-mdx-remote@1.0.0	Do not pass user input into ‘¡MDXRemote /¿‘.

Continued on next page

Package (name@version)	Warning sentence
@leichtgewicht/rexreplace@6.0.16	Purposefully implemented the most insecure way possible to remove _any_ incentive to consider running code from an untrusted part.’
@alex.garcia/unofficial-observablehq-compiler@0.6.0-alpha.9	Keep in mind, there is no sandboxing done, so it has the same security implications as ‘eval()‘
node-package-api@1.0.0	’Do not expose this package to the web without any security to prevent unauthorised access to your system.’
expand-template-literal@1.0.3	DO NOT use where malicious templates could cause harm.
mongoson@0.2.0	You do NOT want to use this unattended.
@copha/execa@0.0.1	We recommend against using this option since it is: - not cross-platform, encouraging shell-specific syntax. - slower, because of the additional shell interpretation. - unsafe, potentially allowing command injection.
any-serialize@1.4.12	’(Which is in principle similar to unsafe eval.)’
cholesky-tools@0.1.7	For performance reasons there is no input validation. It is up to user to insure valid input.
object-graph-as-json@2.0.0	’UnsafeDecoder allows the input to run arbitrary code at decode time, as demonstrated in src/index.test.js, so it is not suitable for use in most situations without modification.’
at-bindings@0.3.0	‘at-bindings‘ doesn’t sanitize any inputs (e.g. in the ‘schedule‘ function).
mongo-express-patch@0.21.1	’JSON documents are parsed through a javascript virtual machine, so **the web interface can be used for executing malicious javascript on a server**.’
surrial@2.0.2	CAUTION! Evaluates the string in the current javascript engine (‘eval‘ or one of its friends). Be sure the ‘serializedThing‘ comes from a trusted source!
@nuxtjs/devalue@1.2.3	When using eval, ensure that you call it *indirectly* so that the evaluated code doesn’t have access to the surrounding scope:
babelon@1.0.5	’‘babelon‘ uses ‘vm‘, ‘Function‘, and/or ‘eval‘, so only use it where you trust the input.’
JSONPath@0.11.2	’Although JavaScript evaluation expressions are allowed by default, for security reasons (if one is operating on untrusted user input, for example), one may wish to set this option to ‘true‘ to throw exceptions when these expressions are attempted.’

Continued on next page

Package (name@version)	Warning sentence
funcster@0.0.5	This package performs the equivalent of ‘eval‘, and thus should only be used to deserialize functions delivered from trusted sources.
yaml-parser@3.5.3	Use with care with untrusted sources
check-dependencies@2.0.0	’Do not pass untrusted input here.’
@readme/markdown@10.2.10	’This is essentially a wrapper around [‘mdx.run‘](https://mdxjs.com/packages/mdx/#runcode-options) and **it will ‘eval‘ the compiled MDX**. Make sure you only call ‘run‘ on safe syntax from a trusted’
loopback-component-explorer@6.5.1	Versions of swagger-ui prior to 3.0.13 are vulnerable to Cross-Site Scripting (XSS). The package fails to sanitize YAML files imported from URLs or copied-pasted. This may allow attackers
better-eval@1.3.0	Remember: **never use better-eval blindly with user code.** These checks are precautions for your own usage, but any user with malicious intent could find a way to get through them. Thus, use this package with caution.
@dschulmeis/ls-utils@1.3.0	Beware, that this could potentialy execute JavaScript code in the context of the current document, if the HTML code is coming from an untrusted source!
ramda-suggest@1.3.5	’The easiest way allow inputs of things other that just primitive JavaScript data types (i.e. Arrays, Objects, Functions) was to use eval.’
callable-extractor@0.1.1	The library is built on top of [babel-parser] and uses eval under the hood - so be extremely careful and use this functionality judiciously, in an isolated testing context.
@nbarray/execa@1.0.0	Prefer execa() whenever possible, as it’s both faster and safer.
squish-squash@1.0.6	Use with caution!
js-sanitizer@1.0.15	Using eval() or the newer new Function() is a known security risk and it is generally a bad idea to use them.
gulp-less-templates@0.0.6	’You probably dont want to use this’
ipfs-webpack-plugin@0.1.0	IPFSWebpackPlugin does make use of ‘eval‘ which executes the JavaScript received from IPFS.
exceptionable@0.0.5	’This should be used with care, if you really need something like this it can be useful, but if you don\’t it can also be a footgun.’

Continued on next page

Package (name@version)	Warning sentence
cidr-tools@11.0.3	'It is expected that the passed CIDRs and IPs are validated as the module's own input validation is rudimentary. You are encouraged to use modules like [is-cidr](https://github.com/silverwind/is-cidr) and [is-ip](https://github.com/sindresorhus/is-ip) to validate before passing to this module.'
@ector/cli@1.1.0	**Warning**: it's safer to quote the entry using double quotes '".
@agoric/evaluate@2.2.6	The evaluated code will have full access to the globals, which is usually far more authority than you really want to give that code
JASON@0.1.3	Warning: unlike JSON, JASON is *unsafe*. You should only use it in contexts where you have strong guarantees that the strings that you pass to the JASON parser have been produced by a JASON formatter from a trusted source.
webpack-plugin-react-to-html@2.2.2	'*Warning! This plugin executes your code in a Node context after each compilation.*'
mkpi@1.1.6	The [exec] and [macro] directives can run arbitrary commands and execute arbitrary javascript if your input is untrusted set the 'safe' option and these directives are no longer recognised.
@iooxa/runtime@0.2.7	The functions provided are strings and their evaluation can be dangerous if you do not trust the source.
node-red-contrib-builder@0.1.0	'This node have a little code injection checking to prevent malicious usage injection'
bdwm-orion@1.0.13	'** 注意: ** 如果JSON内容是由不受信任的用户直接编写的，则一定要进行Base64或额外的一层JSON编码包装以防止错误或潜在的攻击。'
@afterburner-js/afterburner-js@2.0.1	You should only ever run this application against a host you fully trust because of the proxy and the ability to execute system commands. However, this behavior can be disabled or modified. See 'middleware/proxy.js' and 'middleware/shelly.js'.
safe-eval-2@0.4.2	User-submitted data should not be run through safe-eval.
github-webhook@2.0.2	'you will generally want to quote the rule to prevent shell trickery.'
graphql-playground-middleware-express-patched@1.10.7	'All versions of 'graphql-playground-express' until '1.7.16' or later have a security vulnerability when unsanitized user input is used while invoking 'expressPlayground()'.'

Continued on next page

Package (name@version)	Warning sentence
@quarto/external-alex-garcia-unofficial-observablehq-compiler@0.0.6	Keep in mind, there is no sandboxing done, so it has the same security implications as ‘eval()‘
gulp-minify-inline@1.1.0	’Please note that the plugin defaults ‘js.output.inline_script‘ to ‘true‘ in order to combat XSS (contributed by @TimothyGu). This is quite useful in general but you might want to re-set it to ‘false‘ explicitly in (an extremely rare) case it breaks things for you’
a-sandbox@0.0.0-alpha.2	[dangersign] 进行定制时，**不能**将外部的对象泄漏到沙箱中，防止沙箱中的代码通过**原型链**对外部进行攻击。
babel-plugin-inline-constants@5.0.0	[dangersign] **Danger**: modules to be inlined are evaluated with Node, so only use this plugin if you completely trust your code.
peertube-plugin-bittube-logo-favicon@1.0.5	’There is no sanitarization for your inputs (neither url or width). We assume that administrators are not evil, and don’t do XSS and co.’
@breaktherules/kewlpackage@0.8.3	ShellJS is not safe for untrusted input. Do not use shelljs.exec with unsanitized input.
jsonpickle@1.2.0	Only load data you have personally produced if you want to be safe
gradleshim@1.0.2	’You should not manipulate these files flippantly, do not accept user input as it could result in injection. Static, safe, and tested changes are key to making this an effective solution.’
nearley-there@1.0.0	*Warning:* Uses ‘eval()‘. Don’t use this other than for testing.
with-with@1.1.2	This is not a sandbox do not use it as a sandbox do not try to make it a sandbox
aws-lambda@1.0.7	Versions prior to 1.0.5 suffer from ’Command Injection’ vulnerability
compile-template@0.3.1	’The node vm module is not a security mechanism. Do not use untrusted code in templates.’
nrsc@0.2.4	’USE WITH EXTREME CARE!’
babelon7@2.0.0	‘babelon‘ uses ‘vm‘, ‘Function‘, and/or ‘eval‘, so only use it where you trust the input.
433mhz@2.0.0	Be aware of command injection: the binary code you pass to ‘transmitCode‘ is passed directly into ‘exec‘ (no sanitization)!
atlas-interactive-shell@1.0.2	Don’t pass unsanitized user input into this function.
crossvm@0.0.9-beta	’CrossVM is not a security mechanism. Do not use it to run untrusted code.’

Continued on next page

Package (name@version)	Warning sentence
gyp-reader@0.0.2	'Do not run this on gyp files that you are not sure of.'
babel-codemod@2.1.2	'This feature should only be used as a convenience to load code that you or someone you trust wrote. It will run with your full user privileges, so please exercise caution!'
@spongex/docbuilder@2.1.1	'__Performs command injection, use at your own risk! Please read documentation before use!__'
estime@1.3.0	支持'eval'的Javascript环境，但是又担心eval的安全性问题。
@agoric/insecure-evaluate@0.1.1	INSECURE three-argument evaluate function.
jsoncomma@1.0.0	parseUnsafe is called that for a reason so only feed it trusted data
ctx-loader@1.0.3	'This is experimental, and it wraps a private method in NodeJS, which can be dangerous.'
uneval@0.1.2	'Do **not** use this unless you know all the things that could possibly go wrong with this.'
jsex@1.0.32	'Basically you can just 'eval' the string if you trust the source. However if you don't, use 'String.prototype.parseJsex(forbiddenMethods)' instead.'
open@10.2.0	'This package does not make any security guarantees. If you pass in untrusted input, it's up to you to properly sanitize it.'
ops-per-sec@3.0.0	'When using the cli your string will be interpreted using Node's 'vm' in a sandboxed context.'
docker-wrap@2.0.5	'This does not currently do any input sanitization and forwards your inputs to the shell, so be sure to not use user inputs for the arguments without sanitizing them first.'
nstal@0.1.20	Warning! This command can be dangerous!
indomitable@4.1.0	'Not recommended as every broadcastEval uses eval() internally'
shell-source@1.1.0	Since sourcing a shell script allows it to execute arbitrary code, you should be 100% sure its contents are not malicious!
kaiser@0.0.4	The resulting serializer, when used on unsanitized data, will be as safe as the most unsafe object in the whitelist, so be careful:
@ukstv/ses@0.10.3	'Still under development: do not use for production systems yet, there are known security holes that need to be closed.'
ejs@3.1.10	'If you run the EJS render method without checking the inputs yourself, you are responsible for the results.'
interpolate-parameters@3.0.1	'Do not use this module on untrusted strings (eg. user input or where user could manipulate the string somehow).'

Continued on next page

Package (name@version)	Warning sentence
@marchyang/execa@6.0.1	We recommend against using this option since it is: - not cross-platform, encouraging shell-specific syntax. - slower, because of the additional shell interpretation. - unsafe, potentially allowing command injection.
das-sdk@1.9.3	'Developers should valid the validity of the value before using them.'
fis3-deploy-http-push@2.0.8	'此代码存在很大的安全隐患，没有做任何安全考虑，请不要部署到线上服务。'
self-referenced-object@2.0.0	self-referenced-object evaluates any expressions inside template literals by calling "Function('"use strict"; return '' + expression + "';")()" which is a marginally safer version of 'eval' (ie still incredibly unsafe), so you should avoid passing any untrusted data into an object evaluated by sro (or at least don't self-reference untrusted data in an sro evaluated object).
@seung_h/node-sh@1.2.1	'**Caution**: This function uses the [child process](https://nodejs.org/api/child_process.html) module to execute commands directly.'
commitr@1.1.5	'Giving this kind of control to a file you downloaded can be dangerous.'
gulp-fest-hardcore@3.0.0	'надо быть внимательным: формируемый шаблон условно делится на выражения и возвращаемое значения.'
@ministryofjustice/express-template-to-pdf@2.1.0	'This is one way to enable running Puppeteer in Docker but may be a security issue if you are loading untrusted content, in which case you should override these defaults.'
krtek@0.1.1	'**WARNING** Krtek can be evil :japanese_goblin:, because 'eval()' is evil.'
jsx0@1.0.1	'This is dangerous to use with user-generated content and in web applications.'
jstojson@1.0.1	'jstojson' currently utilizes 'eval()' method which means if you pass in a malicious piece of code, it could potentially harm your system. So use it at your own risk.
qone@2.0.0	Eval can retain context information, the disadvantage is that the execution code contains compiler code, and it is unsafe, and so on.
pupbot-plugin-jsconsole@1.0.1	目前没有对任何JS函数进行限制，所以不要在公共的群中开启此插件！
@sporeball/node-crush@0.1.0	'as mentioned above, it also requires an eval for decompression; make sure you trust the code you're passing to it.'

Continued on next page

Package (name@version)	Warning sentence
es6-dynamic-template@2.0.0	'Version 1 used 'eval', requiring you to sanitise user input before use.'
@plugin.land/run-command@1.2.1	'Avoid using child_process.exec, and never use it if the command contains any input that changes based on user input.'
git-kit@1.0.2	These scripts are mostly simple wrappers around existing git commands, and not a lot of validation is going on so handle them with as much care as you would every other shell command.
@iac-factory/tty-testing@0.1.9	CLI utilities can be incredibly dangerous.
safer-eval@1.3.6	'**Warning:** The 'saferEval' function is harmful - so you are warned!'
php-escape-shell@1.0.0	'This function should be used to make sure that any data coming from user input is escaped before this data is passed to the exec() functions.'
@viperhq/exec@1.0.1	## User input should be escaped!
neat@2.1.0	'[dangersign] As a general rule (not just for Neat), you should never execute a remote file without prior verification because it could have been tampered with malicious code.'
gulp-htmlincluder@2.2.4	I have flagged these features below, but please use caution when using them so that you don't have any untrusted data that isn't unsanitized going into the system.
atocha@2.0.0	'Note: Do **not** pass unsanitized input since there's no filtering going on. See [execa](https://github.com/sindresorhus/execa) for that.'
node-stringify@0.2.1	'Please be aware that eval may be insecure.'
@curvenote/runtime@0.2.9	The functions provided are strings and their evaluation can be dangerous if you do not trust the source.
panic-server@1.1.1	Loading the client software into a browser or Node.js process exposes the mother of all XSS vulnerabilities. Connect it to the coordinator, then it'll have full control over your process.
baset-vm@0.14.4	it's not designed for running untrusted code - normal code won't affect host's environment in most cases but there are NO PROGRAM RESTRICTIONS to do it, so if you know how this context is built and which parts are actually shared between host and child you are able to affect host from child

Continued on next page

Package (name@version)	Warning sentence
@nuxt/devalue@2.0.2	While ‘devalue‘ prevents the XSS vulnerability shown above, meaning you can use it to send data from server to client, **you should not send user data from client to server** using the same method.
accord-papandreou@0.20.0-patch1	but be careful.
obj-to-json-cli@0.0.4	Uses ‘eval‘, so only use it with trusted input.
coffeeson@0.1.0	’So don’t accept and parse Coffeeson from untrusted sources, that’s what JSON is for.’
johnny-tools-react-native@1.0.3	Please make sure you sanetise the values. (no spaces in the card number, correct format for month and year)
@bishal-9/video-to-mp3-converter@1.0.0	**NOTE:** Do not pass any song name with a space inside it.
travisci-webhook@2.0.0	you will generally want to quote the rule to prevent shell trickery.
@jsenv/uneval@1.6.0	However JSON.stringify is way faster and is safe (it cannot execute arbitrary code). So prefer JSON.stringify + JSON.parse over uneval + eval when you can.
@eklingen/vinyl-stream-gears@4.0.5	Optional argument objects are not sanitized.
@commonify/execa@6.0.0	unsafe, potentially allowing command injection.
@drfrost/xum@1.0.0-alpha.10	’Use with caution since your command won’t be validated.’
o-command-line@1.0.3	Executing bash commands from a program is a high security risk
eval-serializer@0.3.2	Big disclaimer: Even though it tries its best to sandbox the eval context, it’s always possible that there’s a security risk I missed to address. Only use this library when the configuration cannot be modified by 3rd party.
worker-proof@1.0.1	**WARNING:** This library works by serializing functions and evaluating strings as code on the other side. Generally, this is considered inadvisable due to potential security risks involved with any variation of ‘eval‘ of use of ‘Function‘ constructor, as is done here. This should probably not be used in a real production app. Get professional security review if you’re considering it

Continued on next page

Package (name@version)	Warning sentence
@inikulin/jsdom-only-external-scripts@11.1.0	Again we emphasize to only use this when feeding jsdom code you know is safe. If you use it on arbitrary user-supplied code, or code from the Internet, you are effectively running untrusted Node.js code, and your machine could be compromised.

Table 8.1: Packages with Warnings.

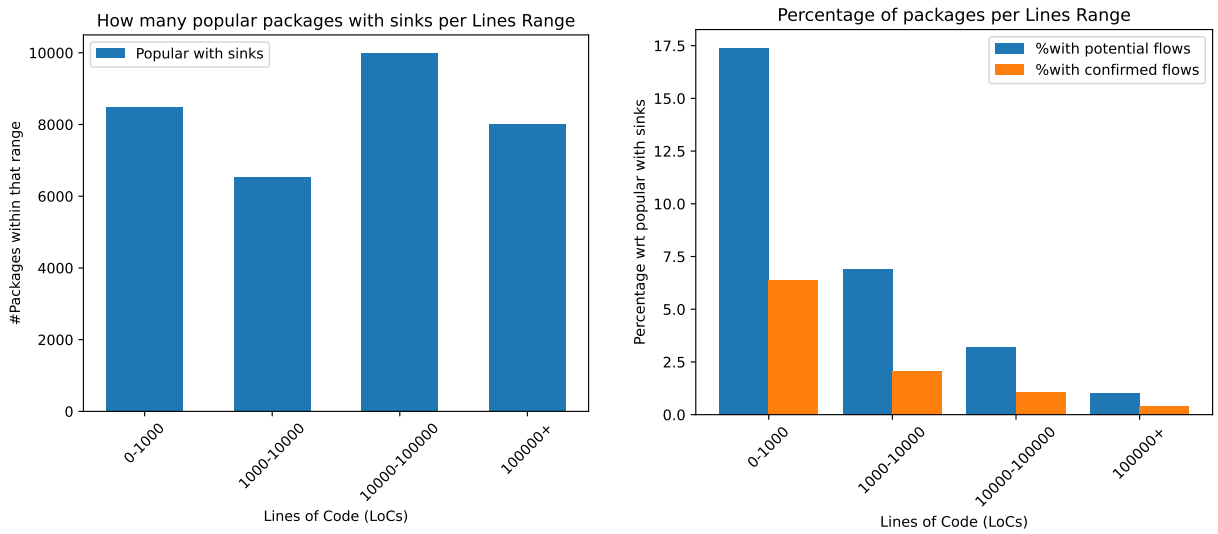

Figure 8.4: Frequency of packages within ranges of lines of code counts, split into ”with sinks”, ”with potential flows” and with ”confirmed flows”.

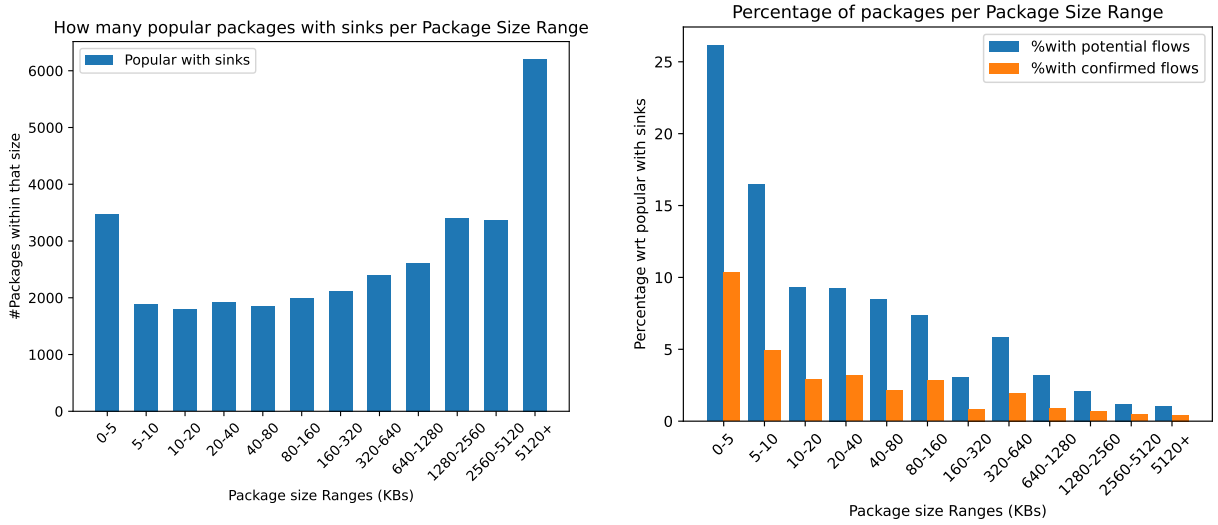

Figure 8.5: Frequency of packages within ranges of package size, split into ”with sinks”, ”with potential flows” and with ”confirmed flows”.

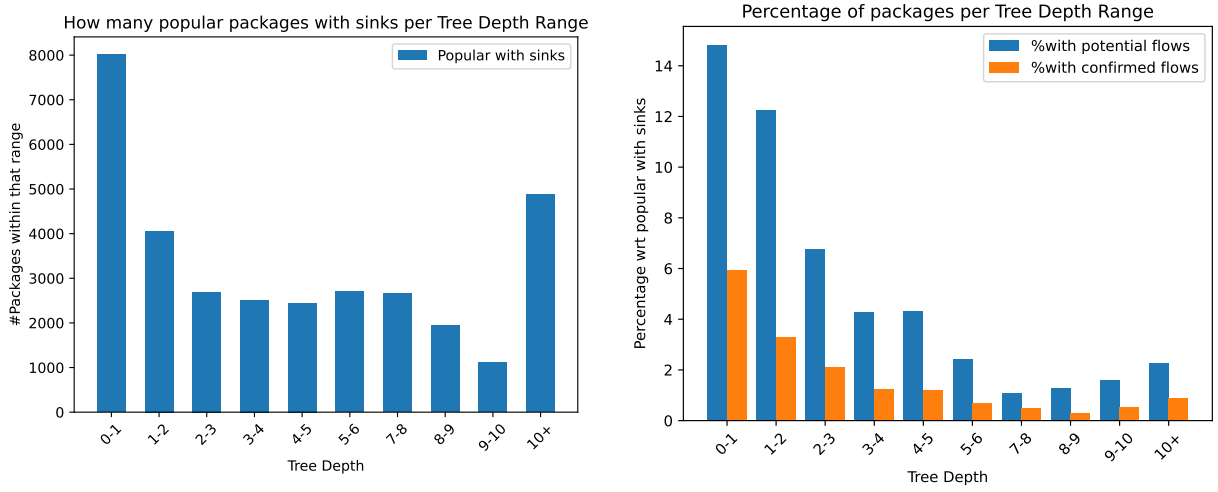

Figure 8.6: Frequency of packages within ranges of tree depth size, split into ”with sinks”, ”with potential flows” and with ”confirmed flows”.

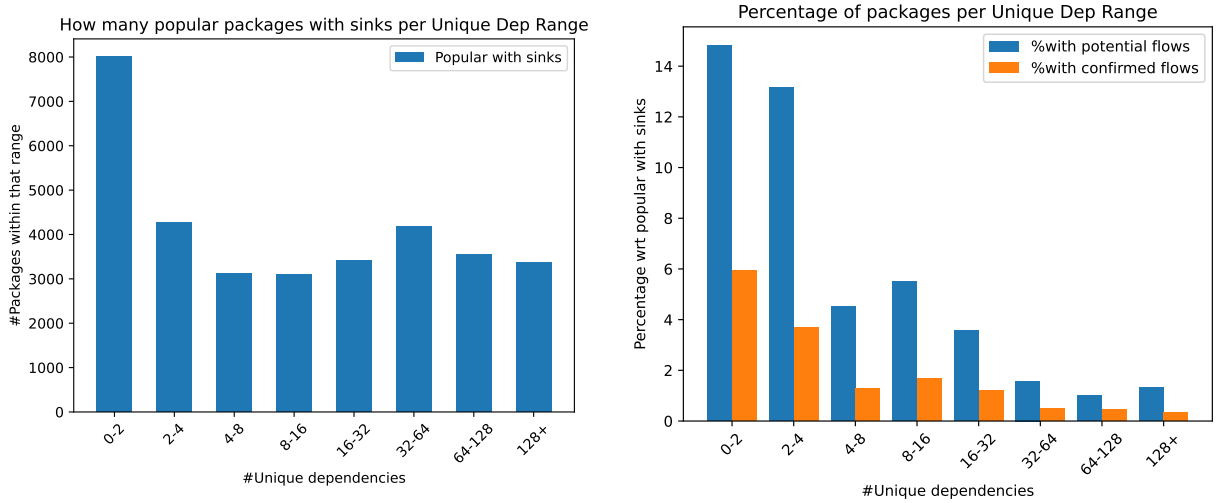

Figure 8.7: Frequency of packages within ranges of unique dependency numbers, split into ”with sinks”, ”with potential flows” and with ”confirmed flows”.

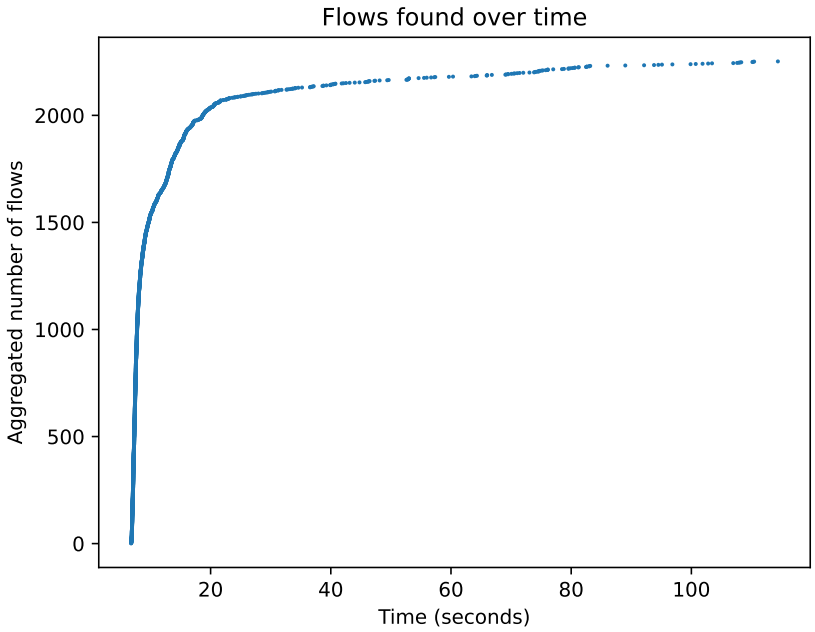

Figure 8.8: How many flows would be found (y-axis) if we set the fuzzing timeout to (x-axis in seconds).

Bibliography

[1] Chrome platform status. https://chromestatus.com/metrics/feature/timeline/popularity/3160. accessed 2025-07-21.

[2] Cloudflare workers node.js runtime apis. https://developers.cloudflare.com/workers/runtime-apis/nodejs/. Accessed: 2025-08-09.

[3] Visual studio code. https://github.com/microsoft/vscode. Accessed: 2025-08-09.

[4] Wayback machine, 1996–. https://web.archive.org.

[5] Wfuzz – the web fuzzer. https://github.com/xmendez/wfuzz, 2014.

[6] Npm passes the 1 millionth package milestone! What can we learn?, 2021. http://tinyurl.com/npm-1-millionth.

[7] WARC, web ARChive file format, 2022–. https://www.loc.gov/preservation/digital/formats/fdd/fdd000236.shtml.

[8] https://security.snyk.io/vuln/SNYK-JS-NETWORK-6184371, January 2024. Available from Snyk, Snyk-ID SNYK-JS-NETWORK-6184371.

[9] Network and distributed system security symposium (ndss) 2025. Internet Society, 2025.

[10] Network and distributed system security symposium (ndss) 2026. Internet Society, 2026.

[11] Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. Natisand: Native code sandboxing for javascript runtimes. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 639–653, 2023.

[12] Acunetix. Acunetix vulnerability report 2021. https://www.acunetix.com/white-papers/acunetix-web-application-vulnerability-report-2021/?#cross-site-scripting-xss, 2024. Accessed: 2024-09-03.

[13] AFLFuzzJS. afl-fuzz-js: A JavaScript Port of the American Fuzzy Lop Fuzzer, Year. Software available from URL.

[14] Scott G Ainsworth, Michael L Nelson, and Herbert Van de Sompel. Only one out of five archived web pages existed as presented. In *Proceedings of the 26th ACM Conference on Hypertext & Social Media*, pages 257–266, 2015.

[15] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 377–392, 2018.

[16] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[17] Souphiane Bensalim, David Klein, Thomas Barber, and Martin Johns. Talking about my generation: Targeted dom-based xss exploit generation using dynamic data flow analysis. In *Proceedings of the 14th European Workshop on Systems Security*, 2021.

[18] Tim Berners-Lee, Larry Masinter, and Mark McCahill. Uniform resource locators (url). RFC 1738, December 1994.

[19] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. Secbench. js: An executable security benchmark suite for server-side javascript. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1059–1070. IEEE, 2023.

[20] Daniel Borkmann, Jesper Dangaard Brouer, et al. ebpf: The extended berkeley packet filter. Linux Kernel Documentation, 2014. Introduced in Linux kernel 3.18.

[21] Tiago Brito, Mafalda Ferreira, Miguel Monteiro, Pedro Lopes, Miguel Barros, José Fragoso Santos, and Nuno Santos. Study of javascript static analysis tools for vulnerability detection in node. js packages. *IEEE Transactions on Reliability*, 72(4):1324–1339, 2023.

[22] Darion Cassel. *Practical End-to-End Analysis of Information Flow Security Policies*. PhD thesis, Carnegie Mellon University, 2023.

[23] Darion Cassel, Nuno Sabino, Min-Chien Hsu, Ruben Martins, and Limin Jia. Nodemedic-fine: Automatic detection and exploit synthesis for node. js vulnerabilities. In *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS'25). doi*, volume 10, 2025.

[24] Darion Cassel, Wai Tuck Wong, and Limin Jia. NodeMedic: End-to-end analysis of node.js vulnerabilities with provenance graphs. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 2023.

[25] The MITRE Corporation. CWE - CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection') (4.3), 2020–. https://cwe.mitre.org/data/definitions/77.html.

[26] The MITRE Corporation. CWE - CWE-94: Improper Control of Generation of Code ('Code Injection') (4.3), 2020–. https://cwe.mitre.org/data/definitions/94.html.

[27] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy. https://mitmproxy.org/, 2010. Version 9.0.

[28] Mickaël Courtes. Landlock: Unprivileged access control for linux. Linux Kernel Documentation, 2021. Introduced in Linux kernel 5.13.

[29] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and*

Analysis of Systems, 2008.

[30] Deno Land Inc. and contributors. Deno: A modern runtime for javascript, typescript and webassembly, 2025. Version 2.x (latest stable release as of November 2025).

[31] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. Rescan: A middleware framework for realistic and robust black-box web application scanning. In *Network and Distributed System Security (NDSS) Symposium*, 2023.

[32] EJS Team. Ejs, 2025. Embedded JavaScript templates.

[33] Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. Detecting hallucinations in large language models using semantic entropy. *Nature*, 630(8017):625–630, 2024.

[34] Mafalda Ferreira, Miguel Monteiro, Tiago Brito, Miguel E Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. Efficient static vulnerability analysis for javascript with multiversion dependency graphs. *Proceedings of the ACM on Programming Languages*, 8(PLDI):417–441, 2024.

[35] Yaw Frempong., Yates Snyder., Erfan Al-Hossami., Meera Sridhar., and Samira Shaikh. Hijax: Human intent javascript xss generator. In *Proceedings of the 18th International Conference on Security and Cryptography - SECRYPT,*, 2021.

[36] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484. IEEE, 2009.

[37] Behrad Garmany, Martin Stoffel, Robert Gawlik, Philipp Koppe, Tim Blazytko, and Thorsten Holz. Towards automated generation of exploitation primitives for web browsers. In *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[38] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. AFFOGATO: Runtime detection of injection attacks for Node.js. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018.

[39] Ayush Goel, Jingyuan Zhu, Ravi Netravali, and Harsha V. Madhyastha. Jawa: Web archival in the era of JavaScript. In *Proc. of ODSI*, 2022.

[40] Google. honggfuzz: a general-purpose, easy-to-use fuzzer with interesting analysis options. https://github.com/google/honggfuzz, 2015.

[41] Google. Expanding user protections on the web. https://blog.chromium.org/2017/11/expanding-user-protections-on-web.html, 2017.

[42] Google. Further protections from harmful ad experiences on the web. https://blog.chromium.org/2018/11/further-protections-from-harmful-ad.html, 2018.

[43] Google. A secure web is here to stay. https://security.googleblog.com/2018/02/a-secure-web-is-here-to-stay.html, 2018.

[44] Google. Under the hood: How Chrome's ad filtering works. https://blog.chromium.org/2018/02/how-chromes-ad-filtering-works.html, 2018.

[45] Google. Building a more private web: A path towards making third party cookies obso-

lete. https://blog.chromium.org/2020/01/building-more-private-web-path-towards.html, 2020.

[46] Google. Protecting against resource-heavy ads in Chrome. https://blog.chromium.org/2020/05/resource-heavy-ads-in-chrome.html, 2020.

[47] Google. Samesite cookie changes in february 2020: What you need to know. https://blog.chromium.org/2020/02/samesite-cookie-changes-in-february.html, 2020.

[48] J. Gruber and the V8 Project. Block code coverage. Google Docs, 2018. Design document for V8 code-coverage support including block-level instrumentation.

[49] Brij B Gupta, Aakanksha Tewari, Ankit Kumar Jain, and Dharma P Agrawal. Fighting against phishing attacks: state of the art and future challenges. *Neural Computing and Applications*, 28(12):3629–3654, 2017.

[50] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *Proceedings of the 22nd USENIX Security Symposium*, pages 49–64, 2013.

[51] Florian Hantke, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti, and Ben Stock. You call this archaeology? evaluating web archives for reproducible web security measurements. In *ACM Conference on Computer and Communications Security*, 2023.

[52] Joona Hoikkala. ffuf - Fuzz Faster U Fool. https://github.com/ffuf/ffuf, 2018. Accessed: 2025-04-17.

[53] Huli. Ejs vulnerabilities in ctf. https://blog.huli.tw/2023/06/22/en/ejs-render-vulnerability-ctf/, 2023. Accessed: 2025-11-13.

[54] IAB Technology Laboratory, Inc. Content taxonomy 3.0 and descriptive vectors, June 2022. GitHub repository.

[55] Internet Archive. About the internet archive. https://archive.org/about/, 2024. Accessed: 2024-09-03.

[56] JSFuzz. Jsfuzz. GitHub repository, 2020. Available at: https://github.com/fuzzitdev/jsfuzz.

[57] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[58] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, V. N. Venkatakrishnan, and Yinzhi Cao. Scaling JavaScript abstract interpretation to detect and exploit node.js taint-style vulnerability. In *IEEE Symposium on Security and Privacy*, 2023.

[59] Zifeng Kang, Song Li, and Yinzhi Cao. Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites. In *NDSS*, 2022.

[60] Rahul Kanyal and Smruti R Sarangi. Panoptichrome: A modern in-browser taint analysis framework. In *Proceedings of the ACM Web Conference 2024*, pages 1914–1922, 2024.

[61] R. Karim, F. Tip, A. Sochurkova, and K. Sen. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering*, 2018.

[62] Wenjun Ke, Yifan Zheng, Yining Li, Hengyuan Xu, Dong Nie, Peng Wang, and Yao He. Large language models in document intelligence: A comprehensive survey, recent advances, challenges and future trends. *ACM Transactions on Information Systems*, 2025.

[63] Soheil Khodayari, Thomas Barber, and Giancarlo Pellegrino. The great request robbery: An empirical study of client-side request hijacking vulnerabilities on the web. In *IEEE Symposium on Security and Privacy*, 2024.

[64] Soheil Khodayari and Giancarlo Pellegrino. {JAW}: Studying client-side {CSRF} with hybrid property graphs and declarative traversals. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2525–2542, 2021.

[65] David Klein, Thomas Barber, Souphiane Bensalim, Ben Stock, and Martin Johns. Hand sanitizers in the wild: A large-scale study of custom JavaScript sanitizer functions. In *IEEE Symposium on Security and Privacy*, 2022.

[66] Maryna Kluban, Mohammad Mannan, and Amr Youssef. On detecting and measuring exploitable JavaScript functions in real-world applications. *ACM Transactions on Privacy and Security*, 2024.

[67] Igibek Koishybayev and Alexandros Kapravelos. Mininode: Reducing the attack surface of node.js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 121–134, San Sebastian, October 2020. USENIX Association.

[68] Dimitrios Kouzis-Loukas. *Learning Scrapy*. Packt Publishing Ltd, 2016.

[69] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *arXiv preprint arXiv:1811.00918*, 2018.

[70] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Network and Distributed System Security*, NDSS 2019, 2019.

[71] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1193–1204, 2013.

[72] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: Large-scale detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013.

[73] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR*, 163(4):845–848, 1965.

[74] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. *Detecting Node.Js Prototype Pollution Vulnerabilities via Object Lookup Analysis*. 2021.

[75] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node.js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[76] Blake Loring, Duncan Mitchell, and Johannes Kinder. Expose: practical symbolic exe-

cution of standalone javascript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 196–199, 2017.

[77] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven Node.js JavaScript applications. *ACM SIGPLAN Notices*, 2015.

[78] Valentin JM Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. Fuzzing: Art, science, and engineering. *arXiv preprint arXiv:1812.00140*, 2018.

[79] marmelab. Gremlins. https://github.com/marmelab/gremlins.js, May 2020. Accessed: 2024-10-09.

[80] Filipe Marques, Mafalda Ferreira, André Nascimento, Miguel E Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. Automated exploit generation for node.js packages. *Proceedings of the ACM on Programming Languages*, 9(PLDI):1341–1366, 2025.

[81] Matheos Mattsson. A comparison of ffuf and wfuzz for fuzz testing web applications. 2021.

[82] S McAllister, E Kirda, and C Kruegel. Expanding human interactions for in-depth testing of web applications. raid 2008. In *11th Symposium on Recent Advances in Intrusion Detection*.

[83] mde/ejs contributors. Unrestricted render option may lead to a rce vulnerability #451. https://github.com/mde/ejs/issues/451, 2019. Accessed: 2025-11-13.

[84] mde/ejs contributors. Mitigate prototype pollution effects (pull request #601). https://github.com/mde/ejs/pull/601, 2021. Accessed: 2025-11-13.

[85] mde/ejs contributors. [vulnerability] server side template injection leads to rce #663. https://github.com/mde/ejs/issues/663, 2022. Accessed: 2025-11-13.

[86] mde/ejs contributors. Ejs @ 3.1.9 has a server-side template injection vulnerability (unfixed) #735. https://github.com/mde/ejs/issues/735, 2023. Accessed: 2025-11-13.

[87] mde/ejs contributors. Ejs, server side template injection ejs@3.1.9 latest #720. https://github.com/mde/ejs/issues/720, 2023. Accessed: 2025-11-13.

[88] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *2018 Network and Distributed System Security Symposium (NDSS)*, 2018.

[89] Ali Mesbah, Arie Van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):1–30, 2012.

[90] Mozilla. Firefox 86 introduces total cookie protection. https://blog.mozilla.org/security/2021/02/23/total-cookie-protection/, 2021.

[91] Mozilla. Firefox 87 introduces smartblock for private browsing. https://blog.mozilla.org/security/2021/03/23/introducing-smartblock/, 2021.

[92] Mozilla. Firefox 90 introduces smartblock 2.0 for private browsing. https://blog.

mozilla.org/security/2021/07/13/smartblock-v2/, 2021.

[93] Mozilla. Firefox 93 features an improved smartblock and new referrer tracking protections. https://blog.mozilla.org/security/2021/10/05/firefox-93-features-an-improved-smartblock-and-.new-referrer-tracking-protections/, 2021.

[94] Mozilla. Firefox rolls out total cookie protection by default to more users worldwide. https://blog.mozilla.org/en/mozilla/firefox-rolls-out-total-cookie-protection-.by-default-to-all-users-worldwide/, 2022.

[95] Node.js Contributors. VM (executing JavaScript) – untrusted code warning. Node.js v24.5.0 API documentation, 2025. [Online; accessed 3-Aug-2025].

[96] Node.js Foundation. child_process.execfile — node.js v24.8.0 documentation. https://nodejs.org/api/child_process.html#child_processexecfilefile-args-options-callback, 2025. Accessed: 2025-09-20.

[97] OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025.

[98] OpenJS Foundation (Electron Project). Electron: Build cross-platform desktop apps with javascript, html, and css, 2025. Version 36.2.0 (latest stable release as of May 2025).

[99] Stack Overflow. Most popular technologies, 2024. Accessed: 2024-10-31.

[100] OWASP Foundation. OWASP zed attack proxy (ZAP). https://www.zaproxy.org/. Accessed: 2025-07-21.

[101] Brian S Pak. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *School of Computer Science Carnegie Mellon University*, 2012.

[102] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. Auto-patching DOM-based XSS at scale. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[103] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. DexterJS: Robust testing platform for DOM-based XSS vulnerabilities. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[104] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jäk: Using dynamic analysis to crawl and test modern web applications. In *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*, pages 295–316. Springer, 2015.

[105] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *2017 Network and Distributed System Security (NDSS) Symposium:[Proceedings]*, pages 1–14. Internet Society, 2017.

[106] Nuno Sabino, Darion Cassel, Rui Abreu, Pedro Adão, Lujo Bauer, and Limin Jia. SWIPE: DOM-XSS analysis infrastructure. https://doi.org/10.5281/zenodo.15883603, July 2025.

[107] Chris Saint-Amant. Scaling a/b testing on netflix.com with node.js. Netflix Technology Blog, August 2014. Originally published at techblog.netflix.com; accessed 2025-08-09.

[108] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic execution for javascript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, pages 1–14, 2018.

[109] Koushik Sen and Manu Sridharan. Jalangi2, 2014–. https://github.com/Samsung/jalangi2.

[110] Deniz Simsek, Aryaz Eghbali, and Michael Pradel. Pocgen: Generating proof-of-concept exploits for vulnerabilities in npm packages. *arXiv preprint arXiv:2506.04962*, 2025.

[111] Snyk. https://snyk.io/blog/how-much-do-we-really-know-about-how-packages-behave-on-the-npm-registry/, 2019. Accessed: 2025-07-01.

[112] Stephen Spender and Jonathan Corbet. Secure computing with filters (seccomp). Linux Kernel Documentation, 2005. Introduced in Linux kernel 2.6.12.

[113] Aleksei Stafeev, Tim Recktenwald, Gianluca De Stefano, Soheil Khodayari, and Giancarlo Pellegrino. Yurascanner: Leveraging llms for task-driven web app scanning. 2024.

[114] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel. Extracting Taint Specifications for JavaScript Libraries. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020.

[115] Cristian-Alexandru Staicu, M. Pradel, and B. Livshits. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS. In *NDSS*, 2018.

[116] Marius Steffens and Ben Stock. Pmforce: Systematically analyzing postmessage handlers at scale. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 493–505, 2020.

[117] Haiyang Sun, Andrea Rosà, Daniele Bonetta, and Walter Binder. Automatically assessing and extending code coverage for npm packages. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 40–49, 2021.

[118] Gemma Team. Gemma 3. 2025.

[119] Node.js Team. Permissions — node.js v24.3.0 documentation. https://nodejs.org/api/permissions.html, 2025. Accessed: 2025-11-07.

[120] HBLT Avgerinos Thanassis, Cha Sang Kil, and Brumley David. Aeg: Automatic exploit generation. In *ser. Network and Distributed System Security Symposium*, 2011.

[121] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. webfuzz: Grey-box fuzzing for web applications. In *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*, pages 152–172. Springer, 2021.

[122] W3Techs. Usage statistics of javascript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript, June 2025. Accessed: 2025-06-26.

[123] Wenhua Wang, Sreedevi Sampath, Yu Lei, and Raghu Kacker. An interaction-based test sequence generation approach for testing web applications. In *2008 11th IEEE High Assurance Systems Engineering Symposium*, pages 209–218. IEEE, 2008.

[124] Wenya Wang, Xingwei Lin, Jingyi Wang, Wang Gao, Dawu Gu, Wei Lv, and Jiashui Wang. Hodor: Shrinking attack surface on node. js via system call limitation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2800–2814, 2023.

[125] Zilun Wang, Wei Meng, and Michael R Lyu. Fine-grained data-centric content protection policy for web applications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2845–2859, 2023.

[126] Wapiti Scanner Project. Wapiti: Web-application vulnerability scanner. https://wapiti-scanner.github.io/, 2024. Version 3.2.0, accessed 2025-07-21.

[127] Sunny Wear. *Burp Suite Cookbook: Practical recipes to help you master web penetration testing with Burp Suite*. Packt Publishing Ltd, 2018.

[128] Web Application Security Working Group, W3C. Content security policy level 2. Technical Report W3C Recommendation 15 December 2016, World Wide Web Consortium (W3C), 2016. Recommendation version.

[129] Web Application Security Working Group, W3C. Trusted types: A browser api to prevent dom-based cross-site scripting. Technical Report Working Draft 3 November 2025, World Wide Web Consortium (W3C), 2025. Working Draft.

[130] Lukas Weichselbaum and Michele Spagnuolo. "csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy". In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, pages 973–985, 2016.

[131] Nico Weidmann, Thomas Barber, and Christian Wressnegger. Load-and-act: Increasing page coverage of web applications. In *International Conference on Information Security*, pages 163–182. Springer, 2023.

[132] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. Abusing hidden properties to attack the node.js ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021.

[133] Feng Xiao, Zheng Yang, Joey Allen, Guangliang Yang, Grant Williams, and Wenke Lee. Understanding and mitigating remote code execution vulnerabilities in cross-platform ecosystem. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2975–2988, 2022.

[134] Zheng Yang, Simon P. Chung, Jizhou Chen, Runze Zhang, Brendan Saltaformaggio, and Wenke Lee. Coindef: A comprehensive code injection defense for the electron framework. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3127–3144, 2025.

[135] Michal Zalewski. American Fuzzy Lop (AFL), 2024. Software available from http://lcamtuf.coredump.cx/afl/.

[136] Xinshi Zhou and Bin Wu. Web application vulnerability fuzzing based on improved ge-

netic algorithm. In *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, volume 1, pages 977–981. IEEE, 2020.

[137] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

[138] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security symposium (USENIX security 19)*, pages 995–1010, 2019.