ExplodeQ.js: A Library of Queries to Detect Injection Vulnerabilities in Node.js Applications

Miguel Alexandre Figueiredo Monteiro

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor(s): Prof. José Faustino Fragoso Femenin dos Santos
Prof. Nuno Miguel Carvalho Santos

Examination Committee

Chairperson: Prof. António Paulo Teles de Menezes Correia Leitão
Supervisor: Prof. José Faustino Fragoso Femenin dos Santos
Member of the Committee: Prof. Limin Jia

November 2023

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

To my parents,

Acknowledgments

I want to express my gratitude to my dissertation supervisors, Prof. José Santos and Prof. Nuno Santos, whose guidance, expertise and insights have been invaluable throughout my academic journey.

I would also like to extend my appreciation to all the dedicated members of the DIVINA and Explode.js projects. Collaborating with you has been a rewarding experience, and I am grateful for the opportunity to contribute to and learn from these remarkable initiatives.

To my friends and family, your encouragement and understanding have been a continuous source of strength. I want to express my heartfelt thanks to my parents for their friendship and unwavering belief in my abilities. Your love and support have been the driving force behind my academic achievements.

Lastly, I want to express my deep appreciation to Ana for her endless support, love, and for being a constant source of motivation throughout this journey. Your caring and uplifting presence, along with your kindness, have meant the world to me.

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) via the DIVINA project (ref.CMU/TIC/0053/2021), and the SmartRetail project (ref. C6632206063-00466847) financed by IAPMEI.

Resumo

O Node.js é um ambiente de servidor de código aberto que executa código JavaScript. Com o Node.js no seu fulcro, o npm permite que programadores partilhem pacotes de código. Em aplicações Node.js, é bastante comum haver um elevado número de dependências em pacotes de terceiros, o que torna difícil para os programdores de analisar cada um deles. Este facto, combinado com as propriedades dinâmicas do JavaScript, pode facilitar a introdução de vulnerabilidades de segurança no código. Recentemente, análise estática tem sido utilizada para detetar vulnerabilidades. Contudo, algumas das suas limitações, como a elevada taxa de falsos positivos, realçam a importância do desenvolvimento de novos métodos automatizados. Esta tese faz contribuições significativas para o desenvolvimento do Explode.js, a primeira ferramenta a combinar análise estática e execução simbólica para identificar automaticamente vulnerabilidades de injeção em aplicações Node.js e confirmar estas vulnerabilidades ao produzir um exploit funcional. Apresentamos o ExplodeQ.js: uma biblioteca de pesquisas (*queries*) para detetar vulnerabilidades de injeção e reconstruir dados controlados por atacantes em aplicações Node.js. Integradas dentro do Explode.js, as pesquisas são realizadas contra uma representação gráfica do código da aplicação analisada e extraem as informações necessárias para permitir execução simbólica e criar um exploit. O ExplodeQ.js foi avaliado como um componente integral do Explode.js. O Explode.js foi avaliado tendo como base dois conjuntos de dados compostos por pacotes npm do mundo real com vulnerabilidades confirmadas. Além disso, o Explode.js foi executado em 430 pacotes npm, onde detetou 31 vulnerabilidades zero-day, com 1 CVE atribuído até ao momento.

Palavras-chave: Análise Estática, Consulta de Grafos, Deteção de Vulnerabilidades, Node.js

Abstract

Node.js is an open-source server environment that executes JavaScript code. Pivoted around Node.js, the Node Package Manager (npm) allows developers to create and share code packages. Node.js applications typically have a high number of dependencies on third-party packages, which makes it hard for maintainers to examine every single one they directly or indirectly include. This fact, combined with JavaScript's dynamic properties, can make it rather easy for Node.js developers to introduce security vulnerabilities into their code. Recently, static analysis has been employed to detect vulnerabilities. However, some of its limitations, like the high false-positive rate, increase the importance of developing new and improved automated vulnerability analysis methods. This thesis makes significant contributions to the development of Explode.js, the first tool to combine static analysis and symbolic execution to automatically identify injection vulnerabilities in Node.js applications and confirm them by producing a functional exploit. We present ExplodeQ.js: a library of queries to detect injection vulnerabilities and reconstruct attacker-controlled data in Node.js applications. Integrated within Explode.js, the queries are run against a code property graph representation of the analyzed application and extract the information needed to enable symbolic execution and generate an exploit. ExplodeQ.js was thoroughly evaluated as an integral component of Explode.js. Explode.js was evaluated against two datasets composed of hundreds of real-world npm packages with confirmed vulnerabilities. Furthermore, Explode.js was executed against 430 real-world new npm packages, where it detected 31 zero-day vulnerabilities with 1 CVE being assigned so far as of the time of the writing of this thesis.

Keywords: Static Analysis, Graph Queries, Vulnerability Detection, Node.js

Contents

List of Tables

List of Figures

List of Listings

Acronyms

AST	Abstract Syntax Tree
CFG	Control Flow Graph
CPG	Code Property Graph
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DDG	Data Dependency Graph
HPG	Hybrid Property Graph
ODG	Object Dependence Graph
ODG	Object Property Graph
PDG	Program Dependence Graph
VIE	Vulnerability Identification Engine
VCE	Vulnerability Confirmation Engine

Chapter 1

Introduction

1.1 Motivation

JavaScript is one of the core technologies of the Web, alongside HTML and CSS. According to the latest Stack Overflow Developer Survey [1], 2023 marks JavaScript's eleventh year in a row as the most commonly used programming language. In 2023, it is being used by 98.7% of websites on the client-side [2] for enhancing web page functionality in the browser. Over the years, JavaScript has become one of the most popular programming languages for implementing server-side web applications as well.

A driving factor in this trend of leveraging JavaScript for server-side web development has been the emergence of Node.js [3], the most popular web framework for code learners and second most popular for professional developers in 2023 [1]. Node.js is an open-source server environment that executes JavaScript code. Due to its asynchronous event-driven architecture, Node.js is employed to build scalable web applications. Pivoted around Node.js, there is also an ecosystem of third-party packages managed by the Node Package Manager (npm). As of August 2023, there are over 2.4 million [4] packages that developers can easily import into their code for either writing web applications or their own packages.

However, researchers and security analysts have found an increasing number of diverse and impactful security vulnerabilities in these third-party packages. Npm packages are known to be susceptible to a variety of vulnerabilities, such as *command injection* [5, 6], *path traversal* [6], *prototype pollution* [6–8], and *cross-site scripting* [6]. The increasing number of npm package versions published each month contributes to new vulnerabilities being introduced into these packages, leading to the rise of reported vulnerabilities in advisory databases over time. According to WhiteSource [9], an average of 32,000 new npm package versions are published every month, and according to the GitHub Advisory Database [10], an average of 40 advisories reporting vulnerabilities in npm packages are published per month.

The proliferation of vulnerabilities within the Node.js ecosystem can be largely attributed to the characteristics of JavaScript itself. In particular, JavaScript's subtle language-specific behavior and dynamic properties, such as prototype based inheritance, combined with Node.js's single-threaded event loop architecture often lead inexperienced developers to introduce security vulnerabilities into their code. Incorporating vulnerability analysis tools into Continuous Integration/Continuous Deployment (CI/CD) pipelines provides an effective approach to prevent these vulnerabilities from making their way into production. These tools enable developers to swiftly find and fix potential security flaws early in the code development process. Unfortunately, the large number of packages being uploaded to npm coupled with the limitations of existing analysis tools [11] make it hard for developers to examine every single third-party package they directly or indirectly include. This fact underscores the increasing need for effective and efficient automated methods to detect and prevent critical security vulnerabilities in Node.js applications.

1.2 Goals

Given the motivation presented above, this thesis aims to make significant contributions to the development of Explode.js, a new JavaScript analysis tool being developed at INESC-ID. Explode.js is the first tool to combine static analysis and symbolic execution [12] to automatically identify injection vulnerabilities in Node.js applications and confirm them by producing functional exploits. In essence, Explode.js is divided into two engines, working together in complementary stages of a pipelined architecture: (*i*) the *Vulnerability Identification Engine (VIE)* leverages static analysis, more specifically code property graphs, to detect vulnerabilities; and (*ii*) the *Vulnerability Confirmation Engine (VCE)* uses symbolic execution [13] to automatically generate an exploit. Importantly, the VIE depends on a set of *queries* that will be responsible for traversing the code property graph of the analysed program, looking for patterns that are characteristic of specific code vulnerabilities. If such patterns are identified, then a potential vulnerability has been located and must be confirmed by the VCE.

Given the central role performed by these queries within Explode.js and their non-trivial nature, the main goal of this thesis is to develop a set of queries to efficiently detect injection vulnerabilities in Node.js applications as part of the VIE. This task can be quite challenging as vulnerabilities can manifest in various ways, and attackers are continually coming up with new attacking techniques. As a result, creating queries that can accurately identify a wide range of vulnerability patterns without generating false positives is a delicate balancing act.

To achieve this objective, a secondary goal must be addressed. Specifically, we require a dataset of Node.js vulnerabilities to test Explode.js, which, as of the beginning of this thesis, did not exist. Therefore, creating such a dataset becomes an integral part of this work, and we propose to curate it by analyzing advisory reports on vulnerabilities found in npm packages. Given that many reports can be incorrect or incomplete, we have to avoid potential mischaracterization of existing JavaScript vulnerabilities. This issue required every advisory to be manually inspected and corrected if necessary.

1.3 Contributions

As mentioned above, this thesis was performed within the overarching context of the development of Explode.js, the first tool that combines static analysis and symbolic execution to identify injection vulnerabilities in Node.js applications and confirm their existence by creating functional exploits. Our specific contributions are as follows:

1. The creation of the VulcaN dataset [11], the largest known dataset of Node.js code vulnerabilities. To carry out the evaluation of Explode.js, we constructed the VulcaN dataset, the largest known curated dataset of Node.js code vulnerabilities. VulcaN is composed of 957 npm packages with confirmed security vulnerabilities. Each vulnerable package in the dataset includes an advisory report with annotations that specify the precise location of the reported code vulnerabilities. This enhances the dataset usability and makes it an effective benchmark for assessing the performance and effectiveness of vulnerability detection tools. Using VulcaN, we conducted the first empirical study of static code analysis tools for detecting vulnerabilities in Node.js packages [11]. The study evaluates the effectiveness of nine static vulnerability detection tools against the VulcaN dataset. We found that the evaluated tools fail to detect many vulnerabilities and exhibit high false positive rates. Additionally, we show that many important vulnerabilities appearing in the OWASP Top10 [14] are not detected by any evaluated tool or even when using the combination of all tools.

2. The creation of ExplodeQ.js, a library of queries to detect injection vulnerabilities and reconstruct attacker-controlled data. ExplodeQ.js is a fundamental component of Explode.js, consisting of a library of graph processing queries to detect injection vulnerabilities and reconstruct attacker-controlled data in Node.js applications. ExplodeQ.js contains a total of 18 queries written in Cypher [15], i.e., the graph query language of Neo4j [16], where Explode.js stores the code property graphs to be analysed. These queries are divided into two major categories: (*i*) the vulnerability detection queries, and (*ii*) the data reconstruction queries. The vulnerability detection queries are two: (*a*) the *Injection Query*, which identifies taint-style vulnerabilities including command injection, code injection, and path traversal, and (*b*) the *Prototype Pollution Query*, which detects prototype pollution vulnerabilities. The data reconstruction queries are 16: two *Parameter Structure Queries* are responsible for reconstructing the structure of JavaScript complex data types under attacker control, including objects and arrays; the remaining 14 *type queries* are designed to assign a specific JavaScript type to variables manipulated by an attacker. Integrated into Explode.js's VIE, these queries are run against a CPG representation of the analyzed application and extract the information the VCE needs to successfully apply symbolic execution and generate an exploit. This CPG representation is stored in a Neo4j database.

3. The evaluation of Explode.js against two curated datasets and the discovery of dozens of zero-day vulnerabilities. We extensively evaluated ExplodeQ.js as an integral component of Explode.js's Vulnerability Identification Engine. We tested Explode.js against two datasets composed of real-world npm packages with confirmed vulnerabilities: our VulcaN dataset, and the SecBench.js dataset [17]. We then compared the results obtained for Explode.js against the ones obtained for ODGen [6], the state-of-the-art tool for static vulnerability detection in Node.js packages. Explode.js outperformed ODGen by identifying vulnerabilities at a rate 3.31 times greater across both datasets. Although both tools displayed similar precision rates across the taint-style vulnerabilities, ODGen stood out for its enhanced precision in detecting prototype pollution. We also tested Explode.js in the wild, analyzing 430 real-world npm packages, and successfully detected 31 zero-day vulnerabilities with 1 CVE [18] identifier being assigned so far as of the time of the writing of this thesis.

Publications: The scientific contributions of our VulcaN dataset, described in Chapter 4, were presented in the following article recently published in a top journal (Scimago: Q1):

> *Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages* [11]. Tiago Brito, Mafalda Ferreira, Miguel Monteiro, Pedro Lopes, Miguel Barros, José Fragoso Santos, Nuno Santos. IEEE Transactions on Reliability 2023.

The scientific contributions of ExplodeQ.js (presented in Chapter 5 and 6) are currently being incorporated into a second article that will be submitted to PLDI'24 (Core: A*) in November 2023.

1.4 Thesis Outline

We structure the remainder of this thesis as follows. Chapter 2 presents some background on Node.js security vulnerabilities and code property graphs. Chapter 3 covers relevant related work such as Node.js security, static analysis using CPGs and vulnerability detection in Node.js applications. Chapter 4 describes the construction of the VulcaN dataset, and evaluates the effectiveness of JavaScript static analysis tools when executed against it. Chapter 5 provides a comprehensive overview of ExplodeQ.js, detailing the library of queries. In Chapter 6, we evaluate Explode.js. Chapter 7 concludes the thesis, summarizing our contributions and pointing out potential avenues for future research.

Chapter 2

Background

This chapter provides some necessary background on Node.js and compares server-side JavaScript security to client-side JavaScript security (Section 2.1). Then, it covers some of the most common Node.js injection vulnerabilities and how npm can be a leading factor in introducing them into the packages (Section 2.2). Finally, we present code property graphs, one of the most common program representations used to create static analysis tools (Section 2.3).

2.1 Node.js Overview

JavaScript has established itself as one of the most popular programming languages among developers. While it was traditionally used for front-end development in the browser, it has evolved into a full-stack development language, mainly due to the adoption of JavaScript engines into fully-featured server-side runtimes like Node.js [3].

Node.js is a cross-platform, back-end runtime environment that executes JavaScript code. Due to its asynchronous event-driven runtime, Node.js is employed to build scalable network applications. In particular, it runs the V8 JavaScript engine in a stand-alone process. Instead of using an interpreter, V8 uses just-in-time compilation techniques to translate JavaScript into more efficient machine code, allowing Node.js to be highly performant.

Node.js also allows developers to use, implement, and share third-party libraries via the Node Package Manager (npm). By reusing code that has already been written and tested, developers can focus their time on writing code to fulfill the application's specific behavior. In fact, prior research [19–21] demonstrates that code reuse can shorten time to market, improve software quality, and increase overall productivity. The evidence from earlier work [22–24] shows that, in contrast to other ecosystems, npm presents a high number of transitive dependencies and that vulnerable packages are extremely popular [5]. As a result, many npm packages rely directly or indirectly on vulnerable packages.

2.2 Security Vulnerabilities in Node.js Applications

Vulnerabilities in Node.js applications can cause serious security breaches. Just like client-side code, server-side JavaScript code running on Node.js servers can access the standard JavaScript API. However, in contrast to client-side JavaScript applications that run in the browser, Node.js allows for direct interaction with crucial components like the file system, operating system, databases, and networking services, yet it does not provide any security mechanisms like sandboxing of untrusted code. In fact, vulnerabilities like path traversal and SQL injection that frequently affect Node.js are not present in client-side JavaScript code because the browser acts like a sandbox. Additionally, vulnerabilities that occur on both ends show that server-side vulnerabilities can have a much higher impact on the system's security.

Staicu et al. [5] conducted a study to better understand and automatically prevent injection attacks on Node.js, and they found thousands of modules vulnerable to command injection attacks because unsanitized attacker-controlled data reached dangerous API calls like `eval` and `exec`. As previously mentioned, unlike client-side JavaScript code, where execution is limited by the browser sandbox, Node.js servers run in a privileged context, making it possible for an attacker to take full control of the system.

The root cause for the spread of vulnerabilities in Node.js packages is the fact that the npm ecosystem is open by design, which allows arbitrary users to create and import third-party vulnerable packages. As a result, vulnerabilities are introduced in their own packages and spread to the dependent ones. This has been confirmed by the over 3,000 GitHub advisories [10] reported because of vulnerabilities found in npm packages. Abdalkareem et al. [25] have shown that developers often perceive npm packages as being well-implemented and tested. Nevertheless, a lot of third-party packages are not consistently maintained, which results in vulnerabilities persisting in the code for a long time even after being reported. Additionally, Zimmermann et al. [22] found that a single vulnerable package can poison the dependency chain and affect thousands of other packages and projects. However, the openness of npm is one of the main reasons for Node.js' success, providing developers with every imaginable package, ranging from trivial packages [25] to complex web frameworks. This allows developers to deliver code faster due to a higher level of abstraction.

Some of the most commonly reported vulnerabilities in Node.js applications include injections and prototype pollution, which we briefly review next.

Injections: For an injection to occur, an unsanitized attacker-controlled flow of information originating from an untrusted source must reach a sink. Users' inputs and inputs from other modules are regarded as unreliable sources, whereas a sink is typically an unsafe API call. Listing 2.1 shows a module that is vulnerable to code injection (i.e., one of the specific injection types enumerated below). An attacker controls the input parameters `x` and `y`, which means there is a dangerous data flow potentially controlled by an attacker between line 1 (source) and line 4 a dangerous sink call (`eval`). Some of the most well-known injection vulnerabilities follow this pattern. They are distinguished from each other because they have different sinks, which impact different components of the system.

- *Code Injection*: Code injection consists of injecting code, which is later executed by the server. It differs from command injection—which is presented next—because it is constrained by the

```javascript
module.exports = function f(x, y) {
  if (y > 0) {
    arg = x * 2;
    eval(arg);
  }
}
```

Listing 2.1: Code injection example.

```javascript
module.exports = function f(key, subKey, val) {
  obj = {"name": "john", "age": 30, "role": "user"};
  obj[key][subKey] = val;
}
```

Listing 2.2: Prototype pollution example.

language's capabilities. Nevertheless, in Node.js it is rather easy to escalate a code injection vulnerability into a command injection vulnerability by calling `exec` in the injected code. Notable sinks: `eval`, `Function()`.

- *Command Injection*: Allows executing arbitrary commands in the server's operating system. Unlike `eval` or `Function()`, which are traditional JavaScript APIs, the `exec` API function has been introduced by Node.js and can only be called after including its parent module, `child_process`. Notable sinks: `exec`, `child_process.spawn`, `child_process.execFile`.

- *SQL Injection*: A SQL injection attack occurs when a user injects malicious SQL statements into a database query. This can result in the attacker reading, modifying, or even deleting sensitive data. Node.js default API does not provide modules to interact with databases. However, APIs for interacting with well-known database engines like MySQL can be installed using npm. Notable sinks: `mysql.connection.query`.

- *Path Traversal*: A path traversal attack occurs when an attacker attempts to access restricted files on the server by injecting malicious input. Node.js includes a module for interacting with the file system (`fs`), which can be exploited to conduct these attacks if the input is not carefully validated. Notable sinks: `fs.readFile`, `fs.createReadStream`.

Prototype Pollution: JavaScript is a prototype-based language, so any property lookup first traverses the prototype chain—a collection of prototypical objects—until it reaches an object with null as its prototype, before returning a definition. By traversing an object's prototype chain, we eventually reach `Object.prototype`, the base object from which all objects inherit, and that contains the built-in functions, such as `hasOwnProperty()`, `toString()` and `valueOf()`. For instance, Figure 2.1 shows the prototype chain of the object `obj` created in Listing 2.2.

Any link in the prototype chain can also be changed, including the built-in functions, which can be redefined. The combination of these two dynamic features can result in prototype pollution, an object-related vulnerability. More specifically, it can lead to the pollution of the base object, which can easily escalate into arbitrary code execution or *denial-of-service*. In Listing 2.2, we have an example of a vulnerable module where an attacker controls `key`, `subKey` and `val`. An attacker can, for example, redefine

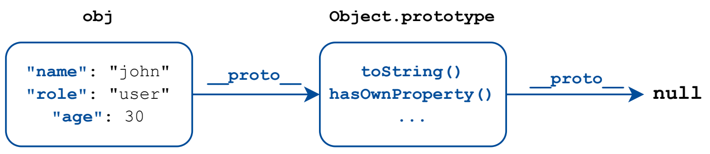


Figure 2.1: Prototype chain of object `obj` of Listing 2.2.

the built-in `toString` function by manipulating the server to execute the following code:

```
function newToString() { return "Polluted!" };
obj["__proto__"]["toString"] = newToString;
```

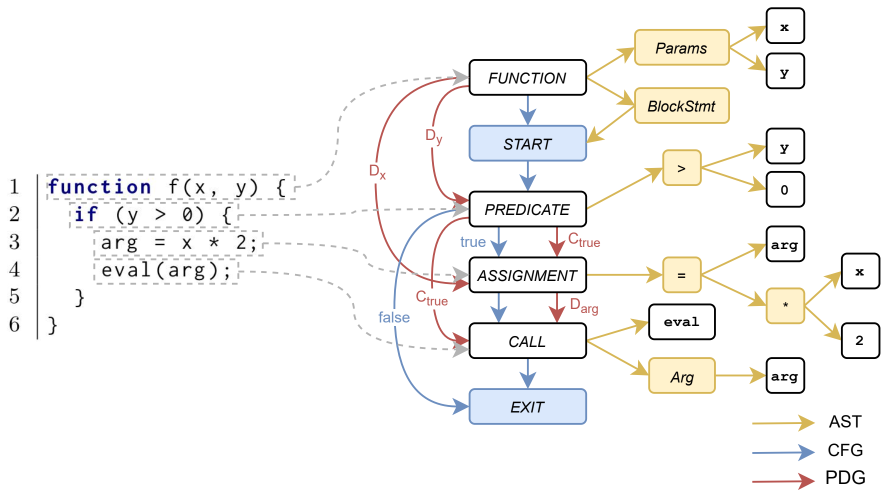
Now, every time an object calls `toString()`, instead of executing the default JavaScript function, it will return the string `"Polluted!"`.

2.3 Code Property Graph

A *Code Property Graph (CPG)* is a graph-based representation of a program initially introduced by Yamaguchi et al. [26] to identify security vulnerabilities in C and C++ code. With CPGs, one can detect security flaws through the use of graph traversals that specifically search for vulnerabilities' code patterns. Graph traversals can be encoded as graph queries and discharged to a general purpose graph database, such as Neo4j [16]. CPGs were proven effective in detecting code injections, *XSS*, and *CSRF injections* in PHP applications [27, 28]. For JavaScript, they were employed to detect client-side [29] and server-side vulnerabilities [6]. More recently, code property graphs were also effectively used to detect vulnerabilities in WebAssembley [30], a low-level language.

CPGs can be obtained by merging a program's *Abstract Syntax Tree (AST)*, *Control Flow Graph (CFG)*, and *Program Dependence Graph (PDG)*. To illustrate how these graphs are assembled together into a consistent CPG data structure, Figure 2.2 provides a code injection example and its respective code property graph. Next, we present each of these subgraphs in more detail and explain how the CPG can be used to identify the vulnerability in said code example.

Abstract Syntax Tree (AST): When we write code we want it to be as concise as possible, but tools like compilers need a lot more information to understand it fully. To provide that information, a compiler needs to perform lexical analysis to tokenize different parts of the code like variables and functions. It then parses these tokens into a tree that represents the structure of the code: the abstract syntax tree. The AST can then be safely used to translate code, e.g., converting C code to machine code. Each node of the AST denotes a construct occurring in the text and can be inserted into one of two categories: *inner nodes* and *leaf nodes*. Inner nodes represent operators such as assignments and function calls. Leaf nodes represent operands such as constants or identifiers. In Figure 2.2 the *CALL* node is an inner node as it represents a function call, whereas the `eval` and `arg` nodes are leaf nodes, as they represent the called function and the function argument, respectively. Yamaguchi et al. [26] have shown that abstract syntax trees are well-suited to model vulnerabilities. Still, they do not contain semantic information such as the



```
1 function f(x, y) {
2   if (y > 0) {
3     arg = x * 2;
4     eval(arg);
5   }
6 }
```

Figure 2.2: Code injection example and its code property graph.

program's control or data flow. As a result, they cannot be used solely to identify an attacker-controlled program flow. Consequently, there is a need for additional structures such as CFG and PDG.

Control Flow Graph (CFG): The control flow graph represents a program's flow, i.e., all the paths that might be traversed through a program during its execution. It explicitly describes the order in which statements are executed as well as the predicates necessary for a certain execution path to be taken.

The CFG of a function contains a starting node, a node for each statement and predicate contained in the function, and an end node. Nodes are connected by directed edges that indicate control flow. Edges originating from predicate nodes are labeled as either `true` or `false` denoting the boolean value the predicate must evaluate to for the control to be transferred to the destination node. In Figure 2.2 the assignment in line 3 is represented by an assign statement, *ASSIGNMENT* node in the CFG, whereas the if in line 2 is represented by a predicate, *PREDICATE* node in the CFG, and follows either the edge labeled as `true` or `false` depending on the predicate value.

Program Dependence Graph (PDG): Program dependence graphs were originally introduced by Ferrante et al. [31] to execute program slicing. PDGs represent dependencies between statements and predicates, making it possible through static analysis to track down the data flow in a program, in particular the propagation of attacker-controlled data.

Just like control flow graphs, program dependence graphs' nodes include the statements and predicates of a function and its edges are of one of the following types: *Data dependence edges* indicate that a variable defined at the source statement is used at the destination statement. *Control dependence edges* indicate that a predicate has an influence on the execution of a statement. In Figure 2.2, the *FUNCTION* node has a data dependence edge, labeled as D_x, pointing to the *ASSIGNMENT* node because the value of `arg` depends on the value of `x`. The same logic applies for the data dependency between the *ASSIGNMENT* node and the *CALL* node. The control dependence edge between the *PREDICATE* node and the *CALL* node shows that *CALL* is only executed if the predicate value is `true`.

Vulnerability identification: In the paragraphs above, we explained the structure of CPGs, let us now examine how one can leverage them to identify security vulnerabilities. Listing 2.3 shows the pseudocode

```
FIND FLOW
    (source:CFG)
        -[data_dep_edge+:PDG]
            ->(call:CFG_CALL)
                -[ast_edge:AST]
                    ->(sink:AST)
WHERE
  sink in ["eval", "exec", "readFile",...]
```

Listing 2.3: Query pseudo-code to find an injection vulnerability.

of a query we can execute over a code property graph, including the one presented in Figure 2.2, to detect an injection vulnerability. The query starts by finding a `source` (CFG node), which represents an attacker-controlled flow entering the application, e.g, a *FUNCTION* statement. There must be one or more data dependency edges (`data_dep_edge+`) linking the `source` to a `call` statement, also part of the CFG. The `call` statement needs to be connected by an AST edge (`ast_edge`) to a `sink`, part of the AST, where the called function is a dangerous API like `eval` or `exec`.

Summary

This chapter provided essential background for the remaining of this thesis. First, we introduced the Node.js platform and the Node Package Manager (npm), which allows developers to easily share and manage third-party code packages for Node.js applications. Then, we discussed how the Node.js architecture is prone to impactful security vulnerabilities, such as command injection, path traversal, and prototype pollution. We finished with an explanation of code property graphs (CPG), a graph-based representation of programs that combines abstract syntax trees (AST), control flow graphs (CFG), and program dependence graphs (PDG), and that can be leveraged to identify security vulnerabilities. In the next chapter, we overview the most relevant research work on Node.js security, static analysis using CPGs, and vulnerability detection tools for Node.js applications.

Chapter 3

Related Work

There is a rich body of prior research work on security analysis for JavaScript, e.g., information flow monitoring techniques [32–34], taint analysis [5, 7, 35, 36], symbolic execution engines [37–40], among others [41–43]. In this chapter, we focus mainly on: the Node.js platform and ecosystem security (Section 3.1); code property graphs and how they were successfully employed to detect vulnerabilities in several programming languages (Section 3.2); and vulnerability detection tools for Node.js applications (Section 3.3).

3.1 Node.js Security

In the previous chapter, we described some of the most common Node.js security flaws and how npm's architecture can be a driving factor in spreading vulnerabilities across shared packages. In this section, we examine some of the most popular empirical studies on security aspects related to the Node.js platform (Section 3.1.1); tools for managing third-party package inclusion (Section 3.1.2); and datasets that can be used to evaluate Node.js vulnerability detection tools (Section 3.1.3).

3.1.1 Empirical Studies

There is a wide variety of empirical studies on the security of the Node.js platform and JavaScript web servers. In the following, we highlight the primary studies on this subject along with their corresponding conclusions. For each paper, we pinpoint the central question it aims to address, structuring our discussion around that question.

How prevalent are injection vulnerabilities in npm packages? As we already mentioned, injection attacks can be extremely harmful because Node.js servers operate in a privileged context. Staicu et al. [5] conducted a study of injection vulnerabilities in 235,850 npm packages. The authors demonstrate that a significant amount of the tested packages make use of potentially vulnerable APIs, such as `eval` and `exec`, with 15,604 using at least one injection API and 20% depending directly or indirectly on at least one injection API. Furthermore, they manually identify recurrent usage patterns in 150 instances of injection APIs and find that over 20% of such patterns cannot be easily removed. Among these, 58% are exploitable, meaning that attacker-controlled data could potentially reach the unsafe sinks. Notably, 90% of vulnerable call sites do not employ any mitigation techniques, 9% use regular expressions, and none utilizes third-party sanitization packages. Additionally, the authors demonstrate that it is unlikely that developers will take action to voluntarily fix potential vulnerabilities, even when provided with a description of the problem and an example attack. Out of the 20 previously unknown injection vulnerabilities discovered in npm packages and reported by the authors, only 3 have been fixed over the course of several months. Based on the previous observations, Staicu et al. [5] created *Synode*, an automatic hybrid approach to identify potential injection vulnerabilities and prevent injection attacks. We present a thorough explanation of Synode in the following section.

The study clearly demonstrates the prevalence of injection vulnerabilities in npm packages due to Node.js developers frequently using injection APIs without proper data sanitization.

What are the security pitfalls to be aware of when using the Node.js platform and server-side JavaScript? Ojamaa and Düüna [44] were the first to describe systematically the security pitfalls of the Node.js platform and server-side JavaScript. Drawing from practical experience, as well as an extensive review of literature, web resources, and developer forums, they identify the following key security concerns:

1. Server-Side JavaScript: Node.js runs in a privileged environment (non-sandboxed) and any corruption of the server's state can be critical. Additionally, dynamic JavaScript features, such as prototype-based inheritance, make it easier for developers to introduce security vulnerabilities.

2. Misleading Examples: misleading and insecure code examples are common in the web and literature.

3. Handling Computation Intensive Requests: due to Node.js' single-threaded event loop architecture, time-consuming operations like image scaling can block the main thread, preventing the server from accepting new connections.

4. Malicious Installation Scripts in Packages: besides deliberately distributing a package containing malware, a malicious developer can leverage the `scripts` property of the package metainfo to run malicious scripts with `root` privileges during installation.

5. Error Handling: due to the single-threaded event loop based architecture, an unexpected exception will terminate the server causing subsequent requests to fail.

6. The `eval` function: the `eval` function and its relatives are often cited as unsafe Node.js APIs as they support dynamic execution of program text. Still, the authors identify 10,557 instances of static references to these functions in 1,510 npm packages, highlighting their widespread use.

7. Server Poisoning: because Node.js applications run in a single thread, if a request can corrupt the global state, it can potentially disrupt the behavior of the entire server, affecting all subsequent requests, not just the one causing the issue.

The authors also suggest mitigating techniques to address certain security concerns. For instance, the use of packages like `child_process` to facilitate creating and managing jobs for 3), and the creation of a trust based reputation and peer review system to assist developers in determining which packages to trust for 4).

How prevalent are trivial packages in Node.js applications? The Node Package Manager (npm) allows developers to share third-party packages. A trivial package contains code that a developer can easily write, raising questions about the need for adding an extra dependency instead of writing the code manually. Abdalkareem et al. [25] studied the prevalence of trivial packages in the npm ecosystem by analyzing over 230,000 of them. They show that trivial packages are prevalent and growing in popularity, constituting 16.8% of the npm packages studied. Additionally, out of 38,807 Node.js applications in their dataset, 10.9% depend on at least one trivial package. To better understand the advantages and drawbacks of using trivial packages, the authors also conducted a user study with 88 developers, analyzing that more than half of the developers perceive trivial packages as being well tested and implemented. However, less than half of the studied packages even have tests. Furthermore, only 8% of developers consider using trivial packages can open a door for security vulnerabilities. Yet, 11.5% of the analyzed trivial packages have more than 20 dependencies; the higher the number of dependencies, the higher the attack surface.

The authors clearly show that even tough developers typically trust trivial packages, these seemingly harmless dependencies can often introduce severe security vulnerabilities into their applications.

How prevalent are regular expression denial of service (ReDoS) vulnerabilities in Node.js web servers? Regular expressions are extensively employed in software applications, being well-known for their susceptibility to errors and security threats like bypassing checks. However the performance aspect of regular expressions is often overlooked. Attackers can overwhelm servers by submitting hard-to-match inputs, leading to denial of service attacks known as ReDoS. To better understand the impact of this security issue, Staicu and Pradel [45] conducted a study on ReDoS vulnerabilities in Node.js web servers. They download the 10,000 most popular npm packages and after removing regular expressions that are immune to algorithmic complexity attacks, the authors obtain a total of 138,123 expressions, with mean 37.93 and median 4.00 per package. Then, through a semi-automatic search for vulnerable regular expression patterns, they find 25 previously unknown ReDoS vulnerabilities in the analyzed packages. Furthermore, the authors employ the crafted exploits in 8 out of the 25 vulnerable packages to measure how the input size affects the time it takes for vulnerable expressions to match. They determine that, for most of the exploits, the input dependency is quadratic, reaching a one second matching time within 20,000 and 40,000 characters. Next, they create a simple Express [46] application that calls one of the vulnerable packages to measure the matching time impact on the availability of a web server. They show that payloads larger than 4,000 characters, sent by an attacker to the server, can delay up to eight requests of a regular user with a maximum delay of 20 seconds per request. Lastly, the authors create payloads based on the 8 tested exploits and send them to 2,846 real websites running an Express [46] web server, observing that 339 are vulnerable to ReDoS.

This study demonstrates the prevalence of ReDoS vulnerabilities within npm packages and how attackers can exploit them to temporarily disrupt vulnerable websites.

The results of the aforementioned studies show how impactful and prevalent certain vulnerabilities are in Node.js applications, particularly within npm packages. This highlights the urgency of addressing the current limitations of existing JavaScript analysis tools and the necessity of developing more precise and effective techniques for automated vulnerability detection/prevention.

3.1.2 Managing Third-Party Package Inclusion

As we mentioned in Section 2.1, one of the main reasons for Node.js' success is npm, which enables developers to incorporate third-party packages into their code, allowing them to focus on implementing the specific functionalities of their applications. Nonetheless, developers often underestimate the security implications of introducing third-party packages into their applications. In doing so, they may inadvertently introduce unnecessary dependencies into their codebase, increasing the attack surface.

Here, we describe four tools designed to address the security concerns associated with the indiscriminate adoption of third-party packages by Node.js developers in their applications. The tools employ three techniques: (1) reduce the attack surface of applications by removing unused code parts and packages; (2) assign permissions to packages to prevent malicious package updates; and (3) enforce security policies at runtime to guarantee the safe usage of vulnerable modules.

Mininode: Koishybayev and Kapravelos [42] show that for 1,055,131 npm packages, on average, only 6.8% of the code is written by the developer, whereas 93.2% is developed by third parties. To mitigate the increased attack surface caused by a higher number of dependencies, the authors present *Mininode*, a tool that can be incorporated into the production process of Node.js applications to minimize the source code size while maintaining the same functionality. It achieves this by employing static analysis to remove unused code parts and packages (possibly vulnerable) from the application. The static analysis has at its core a representation of the analyzed code as a dependency graph. Out of the 1,055,131 packages analyzed, Mininode successfully reduces 672,242 and finds 119,433 of them to have at least one vulnerable dependency. Additionally, the tool is capable of completely eliminating all vulnerable dependencies from 2,861 packages and partially from 10,618 others.

Lightweight Permission System: A malicious package update occurs when an attacker obtains control of a developer's account and publishes a new version of the package that contains malicious code. Malicious package updates can be a potential problem in Node.js applications as they often depend on many third-party packages. Additionally, these packages perform simple computations and do not need access to critical components like the file system. Ferreira et al. [47] present a lightweight permission system that protects Node.js applications against malicious updates. To reduce the attack surface, the system sandboxes packages and enforces a least-privileged design at the package level during runtime. In particular, the package permissions work as follows: (1) developers declare permissions from a set of common permissions for their packages, (2) the system enforces strict adherence to the declared permissions, and (3) developers are required to accept the package permission's at installation and when permissions change in subsequent updates. Furthermore, the changes made to the Node.js runtime, such as mediating the `require` function, aid in dynamically enforcing the permissions. With minimal runtime overhead, the system was able to protect 31.9% of the 703,457 analyzed packages.

Npm Dependency Guardian: The permission system introduced by Ferreira et al. [47] requires developers to grant the package permissions at installation time and again when permissions are modified in an update. However, current research suggests that users often disregard requests for specific permissions and may not comprehend their implications [48]. Consequently, Ohm et al. [49] present a similar approach that employs fine-grained permissions and a system to automatically deduce and enforce policies, without the need for a user to define the permissions of the application manually. This method infers capabilities based on a trusted package version, by statically analysing the source code of the package and its dependencies. Following the package update, their modified Node.js interpreter enforces the established capabilities at runtime. Mininode was tested against 10 historical malicious package update attacks, being capable of stopping 9.

Synode: Due to the prevalence and impact of injection vulnerabilities in Node.js applications, Staicu et al. [5] created Synode. Synode combines static analysis and runtime enforcement of security policies to identify potential injection vulnerabilities and use vulnerable modules in a safe way. This combination of techniques establishes a secure mode for third-party modules during installation via Node.js installation hooks. The analysis process involves studying call sites of certain APIs. It gathers potential input values, organizing them into a tree structure. Through static analysis, it derives templates representing known and unknown parts of possible string values. Using these templates, the analysis determines if an injection API call site is safe or requires a runtime check to block malicious inputs. In cases where input values cannot be statically determined to be safe, a dynamic check is performed at runtime to guarantee that no malicious inputs reach the injection APIs. The static technique was run against 15,604 npm modules that contain calls to the unsafe sinks (`eval` and `exec`) and it found 36.66% of injection APIs to be statically safe. The runtime mechanism was applied to 24 vulnerable modules and prevented all attempted injections.

While the main goal of the presented tools is to prevent the occurrence of potential attacks when in the presence of vulnerable code, Explode.js follows a different approach. It focuses on identifying and confirming security vulnerabilities during the development process, ensuring that the code is secure upon execution.

3.1.3 Evaluation Datasets

There is a wide variety of vulnerability detection tools for Node.js applications (see Section 3.3). These tools employ different analysis techniques, such as static and dynamic analysis, each with its own set of advantages and limitations. Consequently, it is crucial to be able to compare them. As of the beginning of this work, there was no rigorous benchmark for measuring the effectiveness of vulnerability detection tools in Node.js. Therefore, in the context of this thesis, we develop the VulcaN dataset (see Chapter 4) to address this gap. In the meantime, another dataset with the same objective, SecBench.js [17], was

Language **Tool** **Graph Database**	**C/C++** Joern TinkerPop	**C/C++** Joern-IntraProc TinkerPop	**WebAssembly** Wasmati Neo4j	**PHP** Joern-PHP TinkerPop	**JavaScript** JAW Neo4j	**JavaScript** ODGen NetworkX
Format Strings	✓	✓	✓	*N/A*	*N/A*	*N/A*
Buffer Overflow	✓	✓	✓	*N/A*	*N/A*	*N/A*
Memory Disclosure	✓	✓	✗	*N/A*	*N/A*	*N/A*
Resource Leaks	✓	✓	✗	*N/A*	*N/A*	*N/A*
Null Point Reference	✓	✓	✗	*N/A*	*N/A*	*N/A*
Use After Free	✓	✓	✓	*N/A*	*N/A*	*N/A*
Double Free	✗	✗	✓	*N/A*	*N/A*	*N/A*
SQL injection	✓	✓	✓	✓	✗	✓
Code Injection	✓	✓	✓	✓	✗	✓
Command Injection	✓	✓	✓	✓	✗	✓
Path Traversal	✓	✓	✓	✓	✗	✓
XSS	*N/A*	*N/A*	*N/A*	✓	✗	✓
Client-Side CSRF	*N/A*	*N/A*	*N/A*	✗	✓	✗
Server-Side CSRF	*N/A*	*N/A*	*N/A*	✗	✗	✓
SSRF	*N/A*	*N/A*	*N/A*	✗	✗	✓
Prototype Pollution	*N/A*	*N/A*	*N/A*	*N/A*	✗	✓

Table 3.1: Vulnerability coverage for tools that employ CPGs. *N/A*: not applicable; ✓: Detects the vulnerability; ✗: Does not detect the vulnerability.

published. While not exactly identical to the methodology we follow to construct VulcaN, there are significant similarities between the two datasets.

SecBench.js: Bhuiyan et al. [17] consider a total of 2,809 Node.js vulnerabilities found in npm packages and reported in advisory databases. Among the top seven reported vulnerability types in these databases, the authors exclude the top two: malicious packages, i.e, packages with intentionally embedded malicious code, and cross-site scripting (XSS). They consider the remaining five types to construct the SecBench.js dataset, which comprises a total of 600 vulnerabilities, each associated with one npm package version. Specifically, the dataset includes: 192 cases of prototype pollution, 98 instances of ReDoS, 101 command injection vulnerabilities, 169 path traversal vulnerabilities, and 40 arbitrary code injection vulnerabilities. SecBench.js includes executable exploits that trigger each vulnerability and an oracle for assessing the exploit's effectiveness. Furthermore, the dataset presents fixed versions of the vulnerable code for 48% of the vulnerabilities. This aids in measuring false positive rates and delves into how vulnerabilities are addressed and fixed, offering insights into the entire vulnerability life cycle. Additionally, the dataset is thoroughly vetted, with each vulnerability validated to confirm its existence (by creating an exploit), attack class, and metadata.

When compared against VulcaN, SecBench.js offers the advantage of including exploit annotations for all of its vulnerabilities, ensuring their confirmation. In contrast, VulcaN provides exploits only when they are included in the advisory report. On the other hand, one disadvantage of SecBench.js compared to VulcaN is that it lacks certain prevalent and impactful vulnerability types found in Node.js applications, such as Cross-Site Scripting.

3.2 Static Analysis Using Code Property Graphs

As mentioned in Section 2.3, Yamaguchi et al. [26] presented a method to effectively mine large amounts of source code for vulnerabilities based on code property graphs. Table 3.1 presents some of the most impactful vulnerabilities, the tools that were successful in identifying them by employing code property graphs, and the graph databases used by the tools to query the graph. A check mark (✓) indicates that the tool was either used to identify that kind of vulnerability or that its specifications would permit it. For instance, Wasmati [30] and Joern [26] are able to detect taint-style flaws, so they can detect vulnerabilities like code injection and path traversal. We use *N/A* (not applicable) to indicate that the language is not susceptible to that type of vulnerability. For instance, prototype pollution is only possible in JavaScript.

Even though code property graphs have already been employed to identify security vulnerabilities in Node.js applications [6], we are the first to apply them to synthesise driver symbolic programs to verify the existence of potential vulnerabilities.

In the following, we present some of the most popular tools that incorporate code property graphs into their static analysis methodologies for detecting security vulnerabilities in a variety of programming languages. Specifically, we describe how code property graphs have evolved to accommodate the unique characteristics of these languages.

Joern: Yamaguchi et al. [26] employed CPGs to model templates for common vulnerabilities using graph traversals. In order to search for specific patterns that represent vulnerabilities in CPGs, graph traversals can be encoded as graph-database queries. To test this new structure, the authors implement a static code analysis tool (Joern) based on the idea of code property graphs, to mine vulnerabilities in the Linux kernel and more generally in C/C++ source code. The tool is able to successfully detect vulnerabilities such as *buffer overflows*, *null pointer dereferences*, and *resource leaks*, among others. The authors import the code property graph to the Apache TinkerPop [50] graph database and use *Gremlin* [51] as the graph traversal language.

Joern-Interprocedural: In their original concept, CPGs do not support inter-procedural analysis. However, a year later, the authors expanded on their earlier work and provide support for it [52] by using a representation similar to the *System Dependence Graph* [53]. They extend the CPG by defining a data flow between call sites and their callees, introducing edges between arguments and parameters of the respective callees and from return statements back to the call sites. Additionally, the authors also include *post-dominator trees* [54], a program representation derivable from the control flow graph, which is able to detect modifications made by functions to their arguments and the effects these have as data is passed back along call chains. The new inter-procedural code property graph was applied to large C/C++ code bases and is able to discover taint-style vulnerabilities that lead to buffer overflows, buffer overreads, and null pointer dereference vulnerabilities. This approach extends Joern, consequently it uses the same graph database and query language.

Joern-PHP: Backes et al. [27] applied and extended the concept of CPGs to model and discover vulnerabilities in PHP code. Analogously to Yamaguchi et al. [52], to support inter-procedural analysis, they include call graphs in their CPG representation, connecting call nodes to the root node of the corresponding function definition. The resulting graph structure can be leveraged to examine control and data flows between functions. The authors employ the new graph to discover vulnerabilities such as SQL injections, code injections, and command injections on the server-side and cross-site scripting and *session fixation* on the client-side. This tool is integrated within Joern, using the same graph database and query language.

JAW: Khodayari et al. [29] created JAW, a framework that extends CPGs to study client-side *cross-site request forgery (CSRF)* vulnerabilities. The original concept of CPG does not support JavaScript's peculiarities, such as asynchronous events or the execution environment. Therefore, the authors create a *Hybrid Property Graph (HPG)* by merging the CPG with a JavaScript specific *Inter-Procedural Call Graph*, which allows inter-procedural static analysis for JavaScript programs, and an *Event Registration, Dispatch and Depedency Graph*, which models the event-driven execution paradigm of JavaScript and dependencies between event handlers. To help model JavaScript's dynamic behavior, they augment the HPG to include data values collected at runtime. JAW successfully detects client-side CSRF vulnerabilities, which may lead to the execution of arbitrary state-changing operations on the server-side or enable XSS and SQL injection. JAW uses Neo4j [16] as its graph database.

ODGen: Li et al. [6] created a static analysis tool that employs code property graphs to automatically detect Node.js vulnerabilities. The authors use NetworkX [55], a Python package for manipulating complex networks, as their graph database. We provide a more in-depth description of this work in the following section, where we specifically discuss vulnerability detection tools for Node.js applications.

Wasmati: Brito et al. [30] employed and extended code property graphs to create Wasmati, a static vulnerability scanner for WebAssembly. Similar to the work of Backes et al. [27], Wasmati merges the CPG with a call graph to support inter-procedural analysis. The original PDG definition (see Section 2.3) is not ideal for WebAssembly as keeping track of control dependence edges can result in graphs 65 times larger. Additionally, the data dependence edges in the original PDG do not allow reasoning on different variable scopes, return values from function calls, and propagation of constant values. Therefore, the authors introduce the *Data Dependency Graph (DDG)*, which does not include control dependence edges and uses a custom dependency analysis that combines reaching definitions and constant/function propagation. Wasmati can successfully detect security flaws such as *format strings*, taint-style vulnerabilities, and buffer overflow. The authors import the graph into the Neo4j [16] graph database.

3.3 Vulnerability Detection in Node.js Applications

In the previous section, we discussed the use of code property graphs to find vulnerabilities in several programming languages, including JavaScript. In this section, we analyse some of the most popular tools [6, 7, 35, 56–58] capable of automatically detecting and exploiting vulnerabilities in Node.js applications. Additionally, we explore the trade-offs associated with using static, dynamic, and hybrid analysis techniques for vulnerability detection in JavaScript, assessing their respective advantages and drawbacks.

Table 3.2 presents a list of some of the most popular Node.js detection and exploitation tools grouped by analysis technique. The table shows the types of injection-style vulnerabilities each tool is capable of detecting and whether they are able to generate the corresponding exploit. It is important to note that while some of these tools can identify other Node.js vulnerabilities like Cross-Site Request Forgery

Technique	**Tool**	**Vulnerability Detection**				**Exploit**
		CWE-22	CWE-78	CWE-94	CWE-471	**Generation**
Static	Nodest	✗	✓	✓	✗	✗
	ObjLupAnsys	✗	✗	✗	✓	✗
	CodeQL	✓	✓	✓	✓	✗
	ODGen	✓	✓	✓	✓	✗
	FAST	✓	✓	✓	✗	✓
Dynamic	NodeMedic	✗	✓	✓	✗	✓
Hybrid	**Explode.js**	✓	✓	✓	✓	✓

Table 3.2: Coverage of vulnerability detection and exploit generation in Node.js security tools. ✓: The tool detects the vulnerability/generates an exploit for the detected vulnerabilities. ✗: The tool does not detect the vulnerability/generate an exploit for the detected vulnerabilities. The CWEs are: Path Traversal (CWE-22), OS Command Injection (CWE-78), Code Injection (CWE-94), Modification of Assumed-Immutable Data (CWE-471).

	Coverage	Performance	Proof of Vulnerability	False Positive Rate
Static	✓	✓	✗	high
Dynamic	✗	✗	✓	low
Hybrid	✓	✓	✓	low

Table 3.3: Advantages and disadvantages of the three types of analysis tools considered.

(CSRF), we choose to focus on injection-style vulnerabilities as they are the ones relevant for our work. From the analysis of the table, we conclude that:

- Only three tools (ODGen, FAST, and Explode.js) detect all the considered vulnerabilities.
- FAST is the only static tool that generates exploits.
- Explode.js is the only tool that detects and exploits both taint-style and prototype tampering vulnerabilities.

Table 3.3 summarizes the advantages and disadvantages of the three types of analysis tools considered. Different types of tools offer different trade-offs. Static tools guarantee full analysis coverage by checking every line of code that can potentially be executed during runtime. However, they typically have a high false positive rate, since they have a harder time generating exploits that prove the existence of the reported vulnerability. On the other hand, dynamic tools have low false positive rates as they typically work by finding concrete inputs that trigger the vulnerable behaviour. However, they usually do not offer coverage guarantees, which may result in not analyzing potentially vulnerable code. In this regard, they are as effective as backend fuzzers used to generate driving inputs. Finally, hybrid tools combine static and dynamic analysis to ensure both full coverage guarantees and proof of vulnerability.

There are several tools that leverage different analysis techniques to detect and exploit vulnerabilities in Node.js applications. Nonetheless, to the best of our knowledge, there is still no solution that uses an hybrid approach, thus guaranteeing full coverage and proof of vulnerability, to autonomously detect both taint-style and prototype pollution vulnerabilities, and subsequently validate their presence by creating a functional exploit. Explode.js' approach is particularly characterized by the use of code property graphs (static analysis) to enable symbolic execution (dynamic analysis).

In the following, we describe the vulnerability detection tools for Node.js applications displayed in Table 3.3. For each tool, we explain the analysis technique it uses, the types of vulnerabilities that is capable of detecting, and its effectiveness in identifying security vulnerabilities.

Nodest: Nodest [35] is a static tool for detecting code and command injection vulnerabilities in Node.js applications. It uses an iterative feedback-driven static analysis based on abstract interpretation. More concretely, the analysis at the core of Nodest implements an iterative procedure with each iteration encompassing a larger subset of the codebase of the application to be analyzed. At the beginning, the analysis only considers the packages directly included in the application itself and, with each new iteration, it applies a set of well-tailored heuristics to include or delete packages from the analysis set. Third-party packages that are not in the analysis set are not analyzed. Nodest was executed against 11 npm packages, detecting a total of 63 vulnerabilities, including 2 that were previously unknown.

ObjLupAnsys: Li et al. [7] created ObjLupAnsys, a static analysis tool for detecting prototype pollution vulnerabilities in Node.js applications. To identify the vulnerabilities, the tool executes queries that look for vulnerable object lookups over a new graph structure called *Object Property Graph (OPG)*. In particular, the OPG represents JavaScript objects, including details like variable names and properties. Objects are represented as nodes, and their relationships, such as serving as a property of another, are depicted trough edges. ObjLupAnsys was tested against 48,162 npm packages, where it detected 61 zero-day vulnerabilities with 11 CVE identifiers being assigned.

CodeQL: CodeQL [56] is an industrial static analysis tool capable of detecting some of the most common Node.js vulnerabilities like command injection, path traversal and prototype pollution. The tool creates a graph database that represents the entire code base of a project. Developers can craft custom queries to search the database for specific patterns, including security vulnerabilities. As part of this work, we evaluated CodeQL against the VulcaN dataset, demonstrating that it detects 300 out of 957 vulnerabilities present in dataset.

ODGen: Li et al. [6] developed ODGen, a static analysis tool that detects both taint-style and prototype pollution vulnerabilities in Node.js applications. The tool executes queries over an improved representation of a code property graph (CPG) to identify the vulnerabilities. In particular, the new CPG is created by merging the abstract syntax tree (AST) with a new graph named *Object Dependence Graph (ODG)*. The ODG represents objects, variables, and scopes as nodes and their relations as edges. These edges represent relations between ODG object nodes and AST nodes, such as object definition and object lookup, as well as relations between ODG object nodes and ODG variable nodes, such as property lookups. ODGen was employed to analyze a dataset consisting of 300,000 npm packages, detecting 180 zero-day vulnerabilities with 70 CVE identifiers being assigned.

FAST: Kang et al. [57] created FAST, a static analysis tool that uses top-down abstract interpretation techniques to scale the discovery of code injection, command injection, and path traversal vulnerabilities in large npm packages. The bottom-up abstract interpretation module constructs a control flow graph to identify paths between source and sink functions. The top-down abstract interpretation module generates precise and informative data flow paths by tracing the control flow path identified during the bottom-up interpretation. In particular, FAST conducts a data flow path search between sources and sinks to detect potential vulnerabilities. After identifying vulnerable paths, FAST verifies exploitability through symbolic constraint solving, generating human-verifiable exploits. The tool was executed against a total of 100,399 npm packages where it detected 242 zero-day vulnerabilities and exploited 182 of them. Furthermore, it obtained 21 CVE identifiers.

NodeMedic: Cassel et al. [58] developed *NodeMedic*, a dynamic analysis tool capable of detecting and exploiting code injection and command injection vulnerabilities in Node.js packages. First, NodeMedic employs source-to-source rewriting to instrument packages with taint provenance tracking mechanisms. To address the need for precise JavaScript analysis, it implements propagation policies, while scalability is tackled through an algorithm that adjusts propagation precision based on dependencies. The resulting taint provenance analysis yields a data structure storing tainted value operations, serving as the foundation for post-detection analyses. Finally, the tool uses a constraint-based synthesis algorithm to automatically generate exploits from provenance graphs, and it measures flow exploitability by connecting these graphs to attack-defense trees [59]. NodeMedic was executed on 10,000 npm packages, resulting in the identification of 152 zero-day vulnerabilities, out of which 108 were subsequently verified through the automatic generation of exploits. Additionally, it obtained 4 CVE identifiers.

Summary

In this chapter, we presented the related work. First, we highlighted the primary empirical studies on Node.js security [5, 25, 44, 45]. This was followed by an overview of tools designed for managing third-party packages [42, 47, 49], and an introduction to evaluation datasets [17] that can be used to assess Node.js vulnerability detection tools. Subsequently, we presented static analysis tools that employ code property graphs (CPGs) to detect security vulnerabilities [6, 26, 27, 29, 30, 52] in a variety of programming languages, including JavaScript, C/C++, and WebAssembly. Finally, we analysed some of the most popular tools for automatically detecting and exploiting vulnerabilities in Node.js applications [6, 7, 35, 56–58]. The next chapter introduces VulcaN, a curated dataset of Node.js packages with well-characterized security vulnerabilities, used to conduct the first empirical study that evaluates JavaScript vulnerability detection tools.

Chapter 4

The VulcaN Dataset

In this chapter we present the VulcaN dataset, which is essential to the evaluation of Explode.js, and illustrate the effectiveness of JavaScript static analysis tools when run against this dataset. We developed these contributions in the form of two research questions (RQs):

- **RQ1: How to obtain an annotated dataset of vulnerabilities in npm packages?** We address this research question in Section 4.1, where we explain the creation of the VulcaN dataset, the largest known curated dataset of Node.js packages with well-characterized security vulnerabilities.
- **RQ2: How effective are available detection tools in uncovering vulnerabilities in JavaScript code?** We used the dataset to conduct the first empirical study [11] aimed at evaluating existing JavaScript vulnerability detection tools on Node.js packages. Section 4.2 illustrates the results obtained for each evaluated tool and detection technique across the entire dataset.

Lastly, in Section 4.3, we present the most important threats to validity of our study.

4.1 RQ1: How to obtain an annotated dataset of vulnerabilities in npm packages?

To evaluate JavaScript vulnerability detection tools, we require an annotated vulnerability dataset to compare the output of a given tool against ground truth data. The npm repository provides an excellent source for retrieving both (i) an extensive collection of vulnerable Node.js applications (i.e., vulnerable npm package versions), and (ii) information about real-world vulnerabilities (documented by the advisory database). Unfortunately, this information cannot be used as-is from existing advisories. First, advisories often lack relevant information about the reported vulnerability (e.g., the exact code location of the vulnerability within the package). Second, in many cases, most of the explanations regarding the reported vulnerability are given in external references, where information tends to be inconsistent and unstructured. Third, some advisories may be incorrect in places (e.g., the classification of the vulnerability type), which may lead to the mischaracterization of existing JavaScript vulnerabilities. These obstacles preclude an automated advisory analysis approach.

Exclusions & Inconsistencies	**# of Advisories**
Malware Packages	416
Missing Package Code	31
Missing JavaScript Code	22
Incorrect Vulnerable Version	42
Missing External References	291
Imprecise CWE	101
Lack of Analysis Information	402

Table 4.1: Number of advisories excluded or inconsistent.

4.1.1 Selection and Validation of Reports

To create our dataset, we collected a snapshot of the existing npm advisories until the end of June 2021. Then, through manual analysis, we excluded some advisories and fixed inconsistencies in the remaining ones (see Table 4.1). Out of the 1828 advisories from the original snapshot, we excluded 469, keeping 1359 for further analysis. Next, we present our exclusion criteria and discuss the detected inconsistencies.

Excluded advisories. As of June 30th 2021, there were 1828 advisories published in npm. Of these 1828 advisories, 416 are categorized by npm as Embedded Malicious Code (CWE-506): these are packages designed with malicious intent, named very similarly to real legitimate packages so as to deceive developers into installing them. These packages are not relevant for our study, which focuses only on unintentional vulnerabilities. From the remaining 1412 advisories, we excluded 31 for lacking available code. Lastly, out of the resulting 1381 vulnerable package versions, 22 were excluded for not including JavaScript code; instead, they had pre-transpiled variants such as CoffeeScript [60] and TypeScript [61], which prevented us from analyzing their source code directly. Consequently, in the end, VulcaN successfully collected 1359 advisories and their corresponding package versions for further manual analysis.

Detected inconsistencies. During the manual analysis, we noticed several inconsistencies in the collected advisories. Most notably, only a small minority of advisories comes with the exact code locations that trigger their corresponding vulnerability. Furthermore, some advisories provide an *incorrect vulnerable package version*, i.e., the advisory metadata points to a package version that does not contain the described vulnerability. When the advisory does not come with additional external references, which is the case for 21% of the analyzed advisories, correcting the incorrect vulnerable package version anomaly can be quite challenging, as the advisory metadata alone is generally insufficient for pinpointing the correct package version. Another detected anomaly is the *imprecise classification of vulnerability type/category*. Most of the times this imprecision is subjective, as a Common Weakness Enumeration (CWE) [62] class can be a subcategory (child) of another more general CWE. This is particularly common for Path Traversals (CWE-22) and Code Injections (CWE-94), to which more precise classes can be attributed; in particular, CWE-23 (Relative Path Traversal) and CWE-24 (another specific Path Traversal variant) to CWE-22 and CWE-95 (Eval Injection) to CWE-94. Some vulnerabilities are simply miscategorised; for instance, sometimes Code Injection (CWE-94) vulnerabilities are categorized as Cross-Site Scripting (CWE-79). During the analysis, we detected 63 cases of vulnerability miscategorization (different CWE)

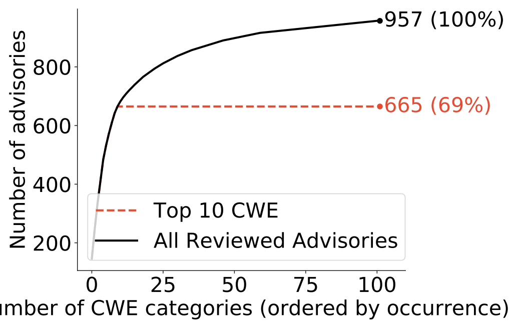


Figure 4.1: CDF of # of reviewed advisories ranked by CWE.

and 21 cases of incorrect vulnerable package version referenced in the advisory. The remaining cases lack the CWE categorization.

4.1.2 Analysis of Reported Vulnerabilities

We analyzed the vulnerabilities in the selected 1359 npm advisories. Our goal was to characterize the vulnerability landscape of the npm package ecosystem by studying the distribution of existing vulnerabilities according to their category and assess the potential security risks posed by the affected packages. From the 1359 advisories manually analyzed, we managed to verify the vulnerability for 957 advisories: these are the ones included in our dataset and characterized in this study. The remaining advisories (402) did not include sufficient information to successfully verify the vulnerability.

Figure 4.1 displays the cumulative distribution function (CDF) of the number of vulnerabilities of our dataset ranked by their CWE category. This distribution is heavily skewed toward a relatively small number of CWE categories, i.e., a large fraction of vulnerabilities pertains to a restricted set of categories. In particular, the top-10 CWEs cover 665 advisories, i.e., 69% of the total number of verified vulnerabilities.

To estimate the potential security risks of such vulnerabilities, we mapped each of the top-10 CWE categories to the latest OWASP ranking (from 2021). OWASP is an organization that works to raise awareness about web security and ultimately improve it. The OWASP Top 10 [14] list is a popular document representing a broad consensus about the most critical security risks to web applications. Updated every few years, this document describes risks, such as injection attacks, broken authentication, and known vulnerable dependencies. As shown in Table 4.2, most vulnerability types can be mapped to a top-10 web security risk. This means that npm packages have well-known risks that security professionals are familiar with. Most notably, 9 out of 10 CWE categories (i.e., 576 advisories) appear in the top-3 OWASP list. This translates to about 60% of the total number of analyzed vulnerabilities in our dataset (957), i.e., many reported vulnerabilities can introduce serious security flaws in web applications.

4.1.3 Our Curated Dataset

Based on the selected advisories, we created a curated dataset aimed at providing a baseline for assessing the effectiveness of vulnerability detection tools. It comprises: (i) the code for the vulnerable version of the npm package indicated in the advisory, and (ii) a corresponding *review* file. This file

#	Vulnerability Type	OWASP Security Risk	# Occurrences
1	Path Traversal	1. Broken Access Control	146
2	Cross-Site Scripting	3. Injection	99
3	Uncontrolled Resource Consumption	-	89
4	OS Command Injection	3. Injection	75
5	Insufficient Transport Layer Protection	2. Cryptographic Failures	75
6	Modification of Assumed-Immutable Data	3. Injection	48
7	Improper Input Validation	3. Injection	41
8	Prototype Pollution	3. Injection	36
9	Code Injection	3. Injection	33
10	Command Injection	3. Injection	23

Table 4.2: Possible mapping of most occurring vulnerabilities in dataset with OWASP Top 10 Web Security Risks (2021) [64]: Path Traversal (CWE-22), Cross-site Scripting (CWE-79), Resource Exhaustion (CWE-400), Insufficient Transport Layer Protection (CWE-818), OS Command Injection (CWE-78), Modification of Assumed-Immutable Data (CWE-471), Improper Input Validation (CWE-20), Improperly Controlled Modification of Object Prototype Attributes (CWE-1321), Code Injection (CWE-94), and Improper Neutralization of Special Elements used in a Command (CWE-77).

```
function search (opts) {
  if (!opts.filter && opts.collection) {
    if (typeof opts.collection === 'string') {
      opts.filter = "function filter (doc) { return doc.type === '" + opts.collection + "'}";
    } else { ... }
    eval(opts.filter);
    opts.filter = filter;
  }
}
```

Listing 4.1: Code injection vulnerability (npm advisory 315)

```
{
  "advisory": { "id": 315, "cwe": "CWE-94" },
  "package_link": "registry.npmjs.org/summit/-/summit-0.1.22.tgz",
  "vulnerability": [
    {
      "source": {
        "file": "lib/drivers/search/pouch.js",
        "lineno": 4,
        "code": "return function search (opts) {"
      },
      "sink": {
        "file": "lib/drivers/search/pouch.js",
        "lineno": 20,
        "code": "eval(opts.filter);"
      }
    }
  ]
}
```

Listing 4.2: Example of VulcaN review file for advisory 315 of Listing 4.1

contains ground truth information that allows us to validate the output of a given tool when analyzing that specific package version.

Review example: Listing 4.2 shows the created review file for advisory 315 [63]. This advisory reports the presence of a code injection vulnerability in package *summit-0.1.22*. A review is a JSON object that contains several fields that describe: i) the advisory identifier (`id`), ii) the vulnerability type as per the CWE taxonomy (`cwe`), iii) the affected package version (`package_link`), and iv) the vulnerability location expressed as a `source`/`sink` pair. A source/sink is specified as a JSON object with fields denoting: the file name (`file`); the line number (`lineno`); and the corresponding line of code (`code`).

Vulnerability location: The location fields are used to determine if the output of a given vulnerability detection tool is correct. Besides using (one or multiple) `source`/`sink` pairs to locate a vulnerability, we can also employ (one or multiple) `block` patterns, which indicate contiguous code regions in which the flaw exists. This form of representation is necessary for vulnerability types that cannot be expressed as source/sink pairs, e.g., usage of HTTP instead of HTTPS which allows for MITM attacks (CWE-818).

Interestingly, we found that, in most cases (78%), the exact location of a vulnerability is very clear, e.g., a call to `eval` in a code injection. These cases can be represented by source/sink pairs, while the remaining cases (22%) can be represented using code blocks that cover all vulnerability-relevant code. Although the block size depends on the vulnerability, in our dataset the average block size is six lines-of-code. Moreover, since the review files can be processed automatically, we believe that our curated dataset will be useful for benchmarking purposes beyond the scope of our work.

Methodology: Vulnerability locations were identified manually. To account for their possible mislabeling, each advisory was analyzed by two authors at separate times and their results cross-checked. Two authors performing cross-validation disagreed in 84 reviews (8.8% of 957 reviews). Most inconsistencies were differences in source/sink pairs. These cases were resolved by selecting the correct source/sink pair or specifying a superset of source/sink pairs. In rare cases, one author failed to locate the vulnerability. These cases were handled by jointly reviewing the location identified by the other author.

Dataset size: From the 1359 analyzed advisories, we were able to manually verify 957 review files (70%) at the time of the paper submission. For each review file we confirmed the exact location of the reported vulnerability. For this reason, we are confident to include these reviews in our curated dataset.

4.2 RQ2: How effective are available detection tools in uncovering vulnerabilities in JavaScript code?

In this section, we determine and characterize how precise the publicly available vulnerability detection tools are at identifying vulnerabilities in known vulnerable JavaScript code. First, we assess which static vulnerability detection tools are available. We distinguish between a broader set of code analysis tools which can serve many different purposes (e.g., detecting programming malpractices) from those that are specifically oriented toward the detection of vulnerabilities. Then, to perform a quantitative and qualitative assessment of the selected tools, we specify the evaluation metrics and methodology we used to rank the tools. Finally, we present our findings, relying on the result of running the selected tools across all 957 advisories of our curated dataset.

4.2.1 Tool Selection Criteria and Selection Process

We focus specifically on fully automatic tools for analysis of npm packages. In particular, to select a given tool, it must:

- **C0. Depend only on the package source code:** The tool requires only the source code of the package to analyze. This excludes tools that require a test suite to guide the analysis.
- **C1. Be available and transparent:** The tool is publicly available and implements a technique that is non-proprietary. Its source code does not need to be open as long as the tool's code analysis techniques can be clearly characterized, e.g., through available documentation, rule set, and usage examples.

Technique	Included Tools (& Version)	Only Package Source Code (C0)	Not Available or Proprietary (C1)	Not Scriptable Interface (C2)	Not Security Oriented (C3)	Other Exclusionary Reasons
Graph-based Analysis (GBA)	CodeQL (2.2.6) [56] ✷ ODGen (-) [6] ✷	SyNode [5] NodeSec [65] Affogato [36]	Beyond Security [66] Checkmarx [67] Fortify [68] Veracode [69] Kiuwan [70] CodeSonar [71] Thunderscan [72] WhiteHat [73]	JsPrime [74] ✷ Codeburner [75] ✷ SonarQube [76] ✷ CodeWarrior [77] ✷	WALA [78] PMD [79] Aether [80] Coala [81] JsHint [82] ✷ EsComplex [83] Coverty Scan [84] DeepScan [85] AppInspector [86] ✷ TAJS [87] SAFE [88]	ESLint SP [89] ✷ Mozilla ScanJs [90] ✷ SemGrep [91] JAW [29] ✷ Joern [26, 92] ✷
Syntax-based Analysis (SBA)	NodeJsScan (0.2.8) [93] ✷ ESLint SSC (**) [94] ✷					
Keyword-based Analysis (KBA)	Graudit (2.8) [95] ✷ InsiderSec (2.0.5) [96] ✷ MS DevSkim (0.4.109) [97] ✷ Mosca (0.8) [98] ✷ Drek (1.0.3) [99] ✷					

Table 4.3: Tools included/excluded. Excluded for reasons beyond C0, C1, C2, & C3: *ESLint Security Plugin* and *Mozilla ScanJs* use *ESLint* rules subsumed by *ESLint SSC*'s; *SemGrep* requires special rules for security purposes and is the backbone of NodeJsScan; JAW only implements rules for detecting client-side CSRF; Joern supports JavaScript inspection, but does not support default rules for detection. Some tools excluded due to C1 were tested using free trials but failed to comply with additional criteria. **ESLint SSC was used with eslint@7.32.0.

C2. **Have a scriptable interface:** The tool must support a command-line interface (CLI), or similar interaction, allowing it to be executed and its output analyzed via a script. This facilitates the scalability and automation of the analysis.

C3. **Be security-oriented:** The tool must identify vulnerabilities or security bad practices in JavaScript. This excludes tools that only construct artifacts, such as control-flow graphs, produce warnings about coding styles and conventions, or produce statistical information about the code, such as code metrics, that might be irrelevant from a security standpoint.

Based on the above criteria, we ended up selecting nine tools for our testing purposes. We started by examining the academic literature [5, 6, 26, 29, 36, 65] and searching the Internet, including OWASP lists [100, 101], repository collections [102–104] and other websites [105–107], for suitable tools for vulnerability detection in npm package code. Most tools we screened were developed by the industry and the open-source community. In total, we first collected 40 JavaScript analysis tools. This full list is presented in Table 4.3.

After inspecting all 40 analysis tools, we found that we needed to manually test 19 tools, which are annotated with the symbol ✷ in Table 4.3. These 19 tools were tested against the Damn Vulnerable Node Application (DVNA) [108], a web application written in JavaScript that was purposely built with a range of vulnerabilities matching the OWASP Top 10 Web Security Risks. After executing each of the 19 tools against the vulnerable application, we excluded those that do not allow the analysis to be automated via a script and also those that fail to show security-oriented results.

Out of the remaining 19 tools, we selected 14 tools that are available, transparent, can be automated, and show security-oriented results. Out of these 14 tools, we also excluded *ESLint Security Plugin*, *Mozilla ScanJs*, *SemGrep*, *JAW*, and *Joern* as explained in the caption of Table 4.3. Consequently, we ended up with 9 distinct candidates that represent proper fully automatic vulnerability detection tools for Node.js applications.

```
<report_mosca>
<Path>/src/lib/drivers/search/pouch.js</Path>
<Title>Possible code injection</Title>
<Description>
    Command injection is an attack in which
    the goal is execution of arbitrary
    commands on the host operating system
    via a vulnerable application
</Description>
<Level>High</Level>
<Match>eval\s?\(|setTimeout|setInterval</Match>
<Result>Line: 20 - eval(opts.filter);</Result>
</report_mosca>
```

Listing 4.3: Snippet of Mosca output classified with Score A for adviosry 315.

```
/src/lib/drivers/search/pouch.js-19- }
/src/lib/drivers/search/pouch.js:20: eval(opts.filter);
/src/lib/drivers/search/pouch.js-21- opts.filter = filter;
```

Listing 4.4: Snippet of Graudit output classified with Score B for adviosry 315.

4.2.2 Evaluation Methodology

Tool evaluation metrics: To evaluate the selected tools, we use two main metrics: *true positive rate* (TPR) and *precision* (P). The TPR represents the proportion of the total vulnerabilities that are correctly detected by a given tool, i.e. the true positives (TP): $TPR = TP/|vulnerabilities|$. The TPR is useful to assess the raw detection rate of a tool without considering the influence of false positives (FP), i.e., its results that do not match the reported advisory. Precision represents the proportion of correctly classified positive cases: $P = TP/(TP + FP)$. This metric is useful to assess if a tool produces too many false positives that can unnecessarily consume analysts' resources.

Tool classification score: To compute the evaluation metrics for a given tool, we need to analyze the output that it generates when applied to analyzing a specific vulnerability. Given that each tool outputs the vulnerability analysis results in its own specific, unstandardized format, we characterize a tool's output according to a common discrete classification score:

- **Score A**: The tool correctly detects and classifies the vulnerability reported in the advisory (*true positive*).
- **Score B**: The tool shows a warning for the vulnerable code, but does not explicitly classify the finding as a vulnerability (*true positive*).
- **Score C**: The tool only shows results that do not match the vulnerability in the advisory report (*false positives*).
- **Score D**: The tool produces no output (*false negative*).

We split the TP results according to two distinct classes: A means an *explicit vulnerability notification*, and B a *security warning notification*. The tools ranked with score A provide a richer output to the user and, thus, more information about the detected vulnerability. As an example, consider the output of two tested tools, Mosca and Graudit, with regards to advisory 315 shown in Listing 4.3 and 4.4, respectively.

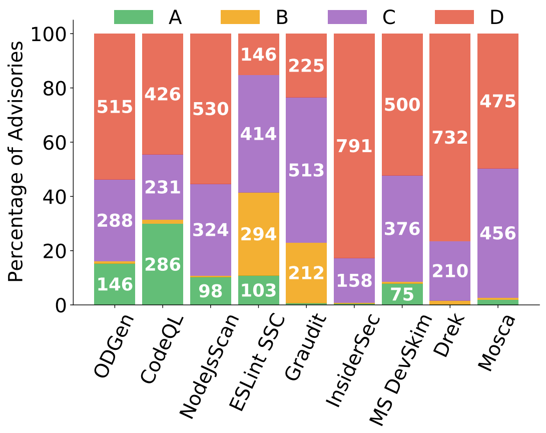


Figure 4.2: Score distribution for each tool.

Although both outputs flag the vulnerable *eval* call reported by the review file of Listing 4.1, Mosca's output clearly identifies a possible code injection, provides a description, a severity level, and the line of code containing the vulnerability. On the other hand, Graudit only shows the vulnerable line of code without explaining how or why it flags that particular snippet. For this reason, the output of Mosca is classified with *Score A* while the output of Graudit is classified with *Score B*.

This discrete classification is also important to account for tools that might flag for the vulnerability at a place that is only close to it (textually, or on the AST). Considering this, we require that tools must clearly identify the vulnerable statement for some vulnerabilities, e.g., code injections and others that can typically be pinpointed to a single statement, while for other vulnerability types multiple lines-of-code are acceptable. Two authors performed a cross-check of all tools' outputs to guarantee fairness of the tool classification in these cases.

4.2.3 Results Across the Entire Dataset

Figure 4.2 displays the score distribution for each tool across our entire dataset, and Table 4.4 shows the evaluation metrics for each tool. Globally, the tested tools perform rather poorly. We can draw the following main observations:

1. Some tools have very low TPR: Counting A and B scores as successful detections, we see that InsiderSec, Drek and Mosca only detect 7 (0.7%), 15 (1.6%) and 25 (2.6%) vulnerabilities, respectively. Hence, these tools fail to detect most vulnerabilities of the dataset.

2. The tools with best TPR have very low precision: The tools that have higher TPR are: ODGen, Graudit, ESLint SSC, and CodeQL. Unfortunately, Graudit and ESLint SSC also have a considerable number of false positives, which tends to erode the confidence of application developers in vulnerability detection tools. Graudit detects 219 vulnerabilities (22.9%), but it also reports over 109k FPs, giving it an overall precision of just 0.2%. A higher number of FPs is expected from a keyword-based tool like Graudit, as many of its string signatures often match non-vulnerable code snippets. ESLint SSC has the highest TPR (41.5%). However, it is also the tool with the highest number of reported FPs (over 389k) and, consequently, the lowest precision (0.1%). This is because ESLint SSC includes many rules from

Scope	ODGen		CodeQL		NodeJsScan		ESLint SSC		Graudit		InsiderSec		MS DevSkim		Drek		Mosca	
	TP (%)	FP (P%)	TP (%)	FP (P%)	TP (%)	FP (P%)	TP (%)	FP (P%)	TP (%)	FP (P%)	TP (%)	FP (P%)	TP (%)	FP (P%)	TP (%)	FP (P%)	TP (%)	FP (P%)
CWE-22	70 (47.9)	136 (34.0)	104 (71.2)	416 (20.0)	56 (38.4)	257 (17.9)	110 (75.3)	25467 (0.4)	122 (83.6)	3101 (3.8)	2 (1.4)	401 (0.5)	0 (0.0)	368 (0.0)	0 (0.0)	1057 (0.0)	0 (0.0)	241 (0.0)
CWE-79	1 (1.0)	33 (2.9)	26 (26.3)	843 (3.0)	6 (6.1)	924 (0.6)	29 (29.3)	123477 (0.0)	11 (11.1)	23634 (0.0)	0 (0.0)	28 (0.0)	0 (0.0)	3990 (0.0)	0 (0.0)	6353 (0.0)	1 (1.0)	1664 (0.1)
CWE-400	4 (4.5)	40 (9.1)	13 (14.6)	212 (5.8)	2 (2.2)	374 (0.5)	39 (43.8)	21936 (0.2)	4 (4.5)	2795 (0.1)	0 (0.0)	22 (0.0)	0 (0.0)	1026 (0.0)	0 (0.0)	74 (0.0)	1 (1.1)	271 (0.4)
CWE-78	22 (29.3)	40 (35.5)	43 (57.3)	416 (9.4)	2 (2.7)	121 (1.6)	29 (38.7)	7405 (0.4)	4 (5.3)	1567 (0.3)	0 (0.0)	15 (0.0)	0 (0.0)	269 (0.0)	3 (4.0)	380 (0.8)	3 (4.0)	190 (1.6)
CWE-818	0 (0.0)	11 (0.0)	16 (21.3)	57 (21.9)	0 (0.0)	97 (0.0)	1 (1.3)	6990 (0.0)	3 (4.0)	1431 (0.2)	1 (1.3)	8 (11.1)	64 (85.3)	1093 (5.5)	0 (0.0)	393 (0.0)	0 (0.0)	150 (0.0)
CWE-471	11 (22.9)	19 (36.7)	13 (27.1)	54 (19.4)	0 (0.0)	295 (0.0)	38 (79.2)	7466 (0.5)	0 (0.0)	2632 (0.0)	0 (0.0)	0 (0.0)	0 (0.0)	249 (0.0)	0 (0.0)	220 (0.0)	0 (0.0)	438 (0.0)
CWE-20	5 (12.2)	19 (20.8)	4 (9.8)	25 (13.8)	0 (0.0)	116 (0.0)	19 (46.3)	7947 (0.2)	4 (9.8)	911 (0.4)	0 (0.0)	13 (0.0)	0 (0.0)	181 (0.0)	1 (2.4)	26 (3.7)	2 (4.9)	294 (0.7)
CWE-1321	3 (8.3)	12 (20.0)	5 (13.9)	78 (6.0)	0 (0.0)	92 (0.0)	31 (86.1)	18130 (0.2)	0 (0.0)	4468 (0.0)	0 (0.0)	9 (0.0)	0 (0.0)	0 (0.0)	0 (0.0)	301 (0.0)	0 (0.0)	465 (0.0)
CWE-94	4 (12.1)	18 (18.2)	5 (15.2)	79 (6.0)	2 (6.1)	159 (1.2)	16 (48.5)	17930 (0.1)	10 (30.3)	7159 (0.1)	1 (3.0)	23 (4.2)	0 (0.0)	358 (0.0)	8 (24.2)	858 (0.9)	8 (24.2)	199 (3.9)
CWE-77	8 (34.8)	11 (42.1)	11 (47.8)	90 (10.9)	1 (4.3)	32 (3.0)	9 (39.1)	5390 (0.2)	0 (0.0)	892 (0.0)	0 (0.0)	1 (0.0)	0 (0.0)	1 (0.0)	0 (0.0)	52 (0.0)	0 (0.0)	97 (0.0)
Other CWE	26 (8.9)	154 (14.4)	60 (20.5)	1283 (4.5)	34 (11.6)	2548 (1.3)	76 (26.0)	147552 (0.1)	61 (20.9)	60895 (0.1)	3 (1.0)	104 (2.8)	17 (5.8)	7930 (0.2)	3 (1.0)	9159 (0.0)	10 (3.4)	2868 (0.3)
Dataset	154 (16.1)	493 (23.8)	300 (31.3)	3553 (7.8)	103 (10.8)	5015 (2.0)	397 (41.5)	389690 (0.1)	219 (22.9)	109485 (0.2)	7 (0.7)	624 (1.1)	81 (8.5)	15465 (0.5)	15 (1.6)	18873 (0.1)	25 (2.6)	6877 (0.4)

Table 4.4: TP, TPR (%), FP and Precision (P%) for each tool by CWE. TPR highlights: green (TPR $\geq$ 50%) or yellow (50% > TPR $\geq$ 15%). When TPR is highlighted we also highlight the FP and P columns: yellow (50% > P $\geq$ 15%), light red (15% > P $\geq$ 2.5%) and dark red (P < 2.5%). The CWEs are: Path Traversal (CWE-22), Cross-site Scripting (CWE-79), Resource Exhaustion (CWE-400), Insufficient Transport Layer Protection (CWE-818), OS Command Injection (CWE-78), Modification of Assumed-Immutable Data (CWE-471), Improper Input Validation (CWE-20), Code Injection (CWE-94), Improper Neutralization of Special Elements used in a Command (CWE-77), and Improperly Controlled Modification of Object Prototype Attributes (CWE-1321).

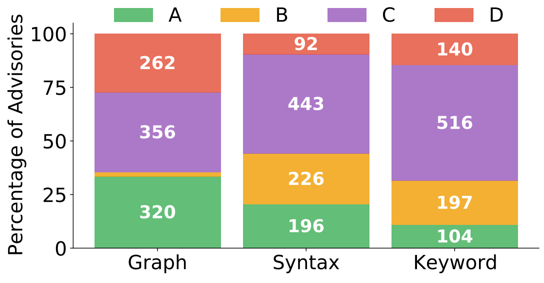


Figure 4.3: Score distribution for each detection technique.

different ESLint plugins, some of which are simple matches (akin to keyword-based analysis) with greedy behaviour, leading to a higher number of FPs.

3. Graph-based analysis has the best detection capability: Figure 4.3 shows the scores according to a particular detection technique. These results show that graph-based analysis reports a significantly larger number of results with score A (explicit vulnerability notifications). Syntax and keyword-based analysis look fairly similar, with reasonable detection rates, but also a high number of reports containing only false positive results.

When considering the results of both tools in this category, i.e., ODGen and CodeQL, they strike a better balance between true positives and precision. CodeQL detects 300 vulnerabilities (31.3%) and has a significantly higher precision (7.8%), when compared to most other tools, while ODGen detects 154 vulnerabilities (16.1%). This number is significantly lower than CodeQL's, but it represents a much higher precision (23.8%) than any other tool tested. Although both these tools do not have the highest TPR, most of their detected vulnerabilities were classified with the A score, meaning that the reported information is richer and more meaningful to the user. Consequently, CodeQL and ODGen are the most balanced tools, achieving a reasonable detection rate (TPR) and less FPs, when compared to other tools with similar TPR. We also note that both these tools have the potential for being further improved by extending them with additional rules.

4. Combining multiple tools increases TPR, but also lowers the overall precision: The combination of the two best tools (CodeQL and ESLint SSC) detects 508 vulnerabilities (53.1%), albeit with only 0.12% precision. If we add the third best tool (Graudit), we detect more vulnerabilities (551/57.6%), but the precision further decreases to 0.11%. Finally, combining both graph-based tools, CodeQL and ODGen, allows for the detection of 339 vulnerabilities (35.4%) with a precision of 7.7%. This shows that combining the best tools can increase the TPR, but at the cost of also increasing the number FPs, which limits the advantage of such an approach.

4.3 Threads to Validity

Although our dataset contains real known-vulnerable npm packages, there may be an implicit bias towards vulnerabilities that are easier to analyze and more common across different programming languages (i.e., not specific to JavaScript code). Thus, since our dataset may not be fully representative of all vulnerabilities in Node.js packages, a tool that can detect all vulnerabilities of our dataset may still miss yet unknown ones.

We may have missed some relevant tool, failed to evaluate an analyzer that excels above all tested tools in our study, or overlooked third-party detection rules that produce better results. To reduce this risk, we will promote the reproducibility of our evaluation by providing our curated dataset.

Both the labelling of vulnerable packages and identification of their vulnerable code snippets were performed manually. Given the challenges of manual code inspection, these annotations could be mislabeled. To mitigate this risk, all vulnerabilities were analysed by at least two authors at separate times and we will make our dataset available for public scrutiny.

A potential concern is whether our study is susceptible to survivor bias. For instance, assuming hypothetically that all the packages that we analyze had already been analyzed using CodeQL during the code development phase, and that the vulnerabilities reported by CodeQL had been accordingly fixed by the developer prior to package release on npm, then the number of vulnerabilities effectively detected by CodeQL could be higher than those reported in our study. This would misleadingly suggest that the quality of CodeQL is worse than what it is in reality. Note, however, that such a comprehensive characterization of each tool is beyond the scope of this work. In our study, we concentrate on evaluating tools' ability to detect, not all possible vulnerabilities, but only those that have been officially reported in npm packages already in production.

We consider any vulnerability reported by a tool that does not match the known vulnerable code in the dataset as a false positive, which may not hold true for all the reports. Confirming these results would require manual validation of every package in the dataset, which is not feasible given the high number of reported warnings (several thousand overall).

Although our study gives a comprehensive picture of vulnerabilities in npm packages, our dataset may not be representative of vulnerabilities in Node.js applications. Node.js applications not only depend on multiple npm packages, but contain application-specific code stitching together function calls from imported packets. This code is not included in our curated dataset. As a result, we cannot extrapolate that all package vulnerabilities have an impact on a Node.js application. Nevertheless, our study gives important insights to improve the detection of vulnerabilities in npm packages, which has been the focus of several research works [5, 6, 22, 25, 45, 57, 58].

Summary

In this chapter, we presented the VulcaN dataset. First, we explained how we obtained an annotated dataset of vulnerabilities in npm packages by analyzing publicly available security reports. Subsequently, we conducted a study on the effectiveness of JavaScript security vulnerability detection tools by executing them against the VulcaN dataset. Finally, we highlighted the significant threats to validity of our study, acknowledging potential biases, human errors, and limitations. The next chapter introduces ExplodeQ.js, our library of queries designed to detect injection vulnerabilities and reconstruct attacker-controlled data in Node.js applications.

Chapter 5

Vulnerability Detection

In this chapter, we introduce our solution for detecting injection vulnerabilities in Node.js applications. We begin by giving an overview of Explode.js (Section 5.1). Subsequently, we present ExplodeQ.js, a library of queries to detect injection vulnerabilities and reconstruct attacker-controlled data in Node.js applications represented as Explode.js code property graphs. We divide our discussion of ExplodeQ.js into two sections. In Section 5.2, we describe the queries for identifying taint-style and prototype tampering vulnerabilities. In Section 5.3, we detail the queries for reconstructing attacker-controlled data capable of reassembling complex data structures and computing potential JavaScript types for tainted parameters.

5.1 Explode.js

ExplodeQ.js is part of a larger project, named Explode.js. Explode.js is the first tool to combine static analysis and symbolic execution to identify injection vulnerabilities in Node.js applications and confirm their existence by automatically producing a functional exploit. In this section, we present the architecture of Explode.js, explaining its main components. Then, we present our running example, a command injection vulnerability found in a npm package. Lastly, we use this example to explain the structure of Explode.js code property graphs.

5.1.1 Architecture

In Figure 5.1, we present the architecture of Explode.js. Explode.js is composed of two engines: the Vulnerability Identification Engine (VIE), and the Vulnerability Confirmation Engine (VCE).

The VIE employs code property graphs (CPGs) to model and discover vulnerabilities and consists of two modules as illustrated in the figure. The CPG Constructor Module receives as input a valid Node.js application and encodes it as a code property graph data structure. The resulting CPG is then fed into the the Query Execution Module. This module imports the CPG into a Neo4j [16] database and executes Cypher [15] queries against it to identify attacker-controlled data flows that may result in injection vulnerabilities and reconstruct the structure of the objects and variables under the attacker's control. The output of the VIE consists of a taint summary file, which is then forwarded to the second main engine

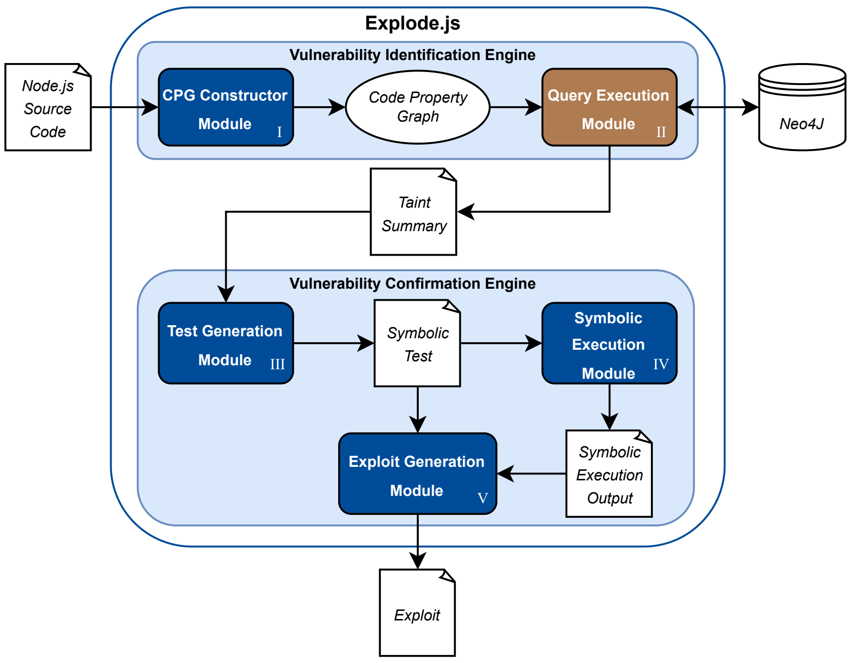

Explode.js
Vulnerability Identification Engine
Node.js Source Code
CPG Constructor Module I
Code Property Graph
Query Execution Module II
Neo4J
Taint Summary
Vulnerability Confirmation Engine
Test Generation Module III
Symbolic Test
Symbolic Execution Module IV
Symbolic Execution Output
Exploit Generation Module V
Exploit

Figure 5.1: Explode.js architecture.

of Explode.js' pipeline. Then, the Vulnerability Confirmation Engine (VCE), uses a symbolic execution tool to create a working exploit that confirms the existence of the vulnerability identified by the VIE. This engine consists of three modules (also shown in Figure 5.1). The Test Generation Module uses the data extracted from the CPG by the Query Execution Module (i.e., the taint summary file) to generate a symbolic test. This test serves as input to the Symbolic Execution Module, which receives the generated test and executes it to find the concrete inputs that trigger the vulnerability. Lastly, the Exploit Generation Module receives the symbolic test and the vulnerability triggering inputs and generates a functional exploit that proves the existence of the vulnerability detected by the VIE.

The primary focus of this work is to create a library of queries for detecting injection vulnerabilities and reconstructing attacker-controlled data in Node.js applications. This library is integrated within the Explode.js Vulnerability Identification Engine, more specifically in the Query Execution Module. Hence, this thesis will not delve into the specifics of the Vulnerability Confirmation Engine.

5.1.2 Command Injection Example

Figure 5.2 shows a code snippet inspired in the npm package *@stoqey/gnuplot v0.0.3* [109], which is used to draw charts using `gnuplot` and convert the resulting plot to a specific format, such as `pdf`. This package is susceptible to a command injection vulnerability (CVE-2021-29369 [110]), allowing an attacker to execute arbitrary OS commands. For instance, by calling the exported function `plot` (line 3) with the payload:

```
options = { filename: 'file; cat /etc/passwd', data: []}
```

an attacker is able to list the system accounts.

The provided code snippet consists of three functions: `plot` (lines 3-18), `getFilePath` (lines 20-27),

```
1   const cp = require("child_process");
2
3   function plot(options) {
4     if (!options.data) throw new Error("Missing
          ↪ data property");
5     if (!options.style) options.style = "lines";
6
7     const cmd = getCmd(options)
8     getFilePath(options)
9
10    let gnuplot;
11    if (options.format)
12      gnuplot = cp.exec(`gnuplot | ${cmd} - ${
          ↪ options.filePath}`, {},
          ↪ post_gnuplot_processing)
13    else
14      gnuplot = cp.exec(`gnuplot > ${options.
          ↪ filePath}`, {},
          ↪ post_gnuplot_processing)
15
16    setup_gnuplot(gnuplot, options);
17      ...
18  }
19
20  function getFilePath(options) {
21    if (options.safe && (options.format === "pdf
          ↪ " || options.format === "eps") {
22      options.filePath = `${options.filename}.{
          ↪ options.format}`
23      cp.exec(`touch ${options.filePath}`);
24    }
25    else
26      options.filePath = options.filename
27  }
28
29  function getCmd(options) {
30    if (options.format === "pdf") return "ps2pdf
          ↪ ";
31    else if (options.format === "eps") return "
          ↪ ps2eps";
32    return null;
33  }
```

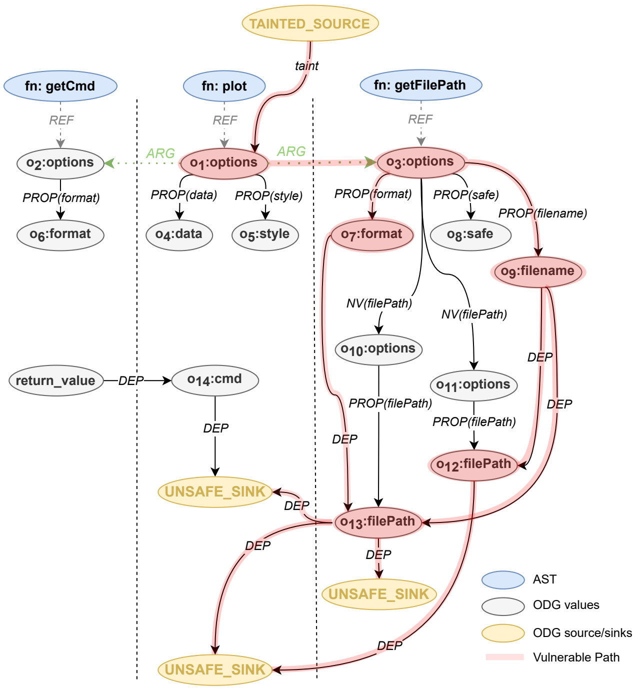


Figure 5.2: Command injection vulnerability (CVE-2021-29369) and object dependence graph of the corresponding program.

and `getCmd` (lines 29-33). The main function `plot` receives as input an object that contains a set of plotting options (`options`) and executes the `gnuplot` command to generate the required chart (line 12 or 14). This function leverages two auxiliary functions: `getFilePath`, which creates the output file (line 23) before executing `gnuplot` if the program is running in safe mode; and `getCmd`, which decides which converting tool to use (`ps2pdf`, `ps2eps` or none) based on the format specified by the user (`options.format`) .

This package contains three dangerous `exec` function calls (lines 12, 14, and 23) as they are reachable by unsanitized attacker-controlled data. For instance, in the `getFilePath` function, if the safe mode is on and the `options` object contains a valid format, the program executes a `touch` command, using the value of the attacker-controlled parameter `options.filename`, to create the output file. To execute the `touch` command, the program makes an unsafe call to `exec` (line 23), allowing the attacker to execute arbitrary OS commands.

5.1.3 Graph Construction

Before delving into the mechanisms used by Explode.js to detect injection vulnerabilities and reconstruct attacker-controlled data, we must first understand the structure of its underlying code property graphs (CPGs). Explode.js CPGs are obtained by merging the abstract syntax tree (AST), control flow graph (CFG), and object dependence graph (ODG) of a program. The AST and CFG graphs used in Explode.js are standard, following the same definition as the original CPGs introduced by Yamaguchi et al. [26] (see Section 2.3). However, in contrast to the original concept, instead of using a program dependence graph (PDG), Explode.js CPGs model data dependencies using a new type of graphs, called object dependence graphs, which represent the structure and evolution of objects and variable values during the execution of the analyzed program. In the following, we focus on Explode.js ODGs, explaining their structure and semantics by appealing to the object dependence graph representation of the running example illustrated in Figure 5.2. Object dependence graphs contain the following three types of nodes:

- **Tainted Source nodes:** *Tainted source* nodes represent untrusted data entering the application context, e.g., user input or input from other modules. For instance, in Figure 5.2, untrusted inputs, such as the parameter `options` of function `plot` (line 3), represented in the graph by $o_1\!:\!options$, are connected by a special *taint* parameter edge to a global `TAINTED_SOURCE` node used to represent attacker-controlled data.

- **Unsafe Sink nodes:** *Unsafe sink* nodes represent calls to dangerous APIs, such as the `exec` function. For instance, in Figure 5.2, each call to the `exec` API (lines 12, 14, and 23) is represented in the graph as an individual `UNSAFE_SINK` node.

- **Value nodes:** *Value* nodes represent objects and primitive values computed during the execution of the program. For example, in the running example, $o_1\!:\!options$ is a value node that represents the object `options` with properties `data` and `style`, each of them associated with their respective value node, $o_4\!:\!data$ and $o_5\!:\!style$.

The crucial information in a ODG is represented by the relationships between its nodes. The ODG has five main types of edges:

- **Property edges:** *Property* edges represent the structure of objects. A property edge $n_1 \mapsto_{PROP(p)} n_2$ signifies that the object represented by n_1 has a property named p, represented by the value node n_2. For instance, in Figure 5.2, the edge $o_1\!:\!options \mapsto_{PROP(data)} o_4\!:\!data$ means that the object `options` contains a property named `data` (line 4), with value $o_4\!:\!data$. Additionally, an arbitrary property is represented by an asterisk: "*". For example, a property edge $n_1 \mapsto_{PROP(*)} n_2$ signifies that the object represented by n_1 has an arbitrary property, represented by n_2.

- **Dependency edges:** *Dependency* edges represent data dependencies between variables, objects, sources, and sinks. A dependency edge $n_1 \mapsto_{DEP} n_2$ means that the value/object represented by n_2 is computed using the value/object represented by n_1. For instance, in Figure 5.2, the edge $o_9 : filename \mapsto_{DEP} o_{12} : filePath$ signifies that `options.filePath` is computed using the value of `options.filename` (line 22). Additionally, we use dependency edges to connect unsafe sinks to values/objects used to computed their arguments. For example, in the running example, the edge $o_{13} : filePath \mapsto_{DEP} UNSAFE\_SINK$ means that `options.filePath` is used to compute an argument of the `exec` function call (line 23).

- **Argument edges:** *Argument* edges connect the formal parameters of a given function to the values they may assume. An argument edge $n_1 \mapsto_{ARG} n_2$ signifies that the parameter represented by n_2 may take on the value represented by n_1. For example, in the running example, the edge $o_1 : options \mapsto_{ARG} o_3 : options$ means that the function parameter `options` in line 20, represented by $o_3 : options$, may take on the value represented by $o_1 : options$, corresponding to the function call on line 8.

- **New Version edges:** Explode.js CPGs keep track of the entire lifetime of objects. To achieve this, whenever an object is modified, Explode.js creates a new object with the updated property. This is represented in the graph by connecting the node representing the previous version of the object to the node representing its current version, via a *new version* edge. For instance, in the running example, the edge $o_3 : options \mapsto_{NV(filePath)} o_{10} : options$ means that a new version of `options` (o_{10}) is created by modifying the property `options.filePath` (line 22). Additionally, modifying an arbitrary property is represented by an asterisk: "*". For example, a new version edge $n_1 \mapsto_{NV(*)} n_2$ signifies that a new version of n_1, represented by n_2, is created by modifying an arbitrary property.

- **Reference edges:** *Reference* edges connect the abstract syntax tree (AST) and the object dependence graph (ODG). A reference edge $n_1 \mapsto_{REF} n_2$ connects the AST node n_1 to the ODG node n_2 that represents the values/objects created or modified within the AST element represented by n_1. For example, in Figure 5.2, the edge $fn : plot \mapsto_{REF} o1 : options$ connects the AST node that represents the declaration of the function `plot` (line 3) to the ODG node that represents its parameter `options`. Additionally, a reference edge $CallExpression : exec(arg) \mapsto_{REF} UNSAFE\_SINK$ connects the AST node that represents a call to an unsafe function like `exec` to the corresponding `UNSAFE_SINK` node in the ODG.

5.2 Queries for Vulnerability Detection

In this section, we describe the vulnerability detection queries, which are executed against an Explode.js CPG representation of the analyzed application to detect some of the most common Node.js injection vulnerabilities. First, we detail the *Injection Query*, capable of detecting taint-style vulnerabilities (Section 5.2.1). Subsequently, we describe the *Prototype Pollution Query* (Section 5.2.2), designed to uncover prototype pollution vulnerabilities. For both queries, we recall the type of vulnerability they detect, followed by a formal description and a demonstration of its functionality by example.

```
 1  MATCH
 2      // Tainted path sub-query
 3      (source:TAINTED_SOURCE)
 4          -[taint:TAINT_EDGE]
 5            ->(param:ODG_VALUE_NODE)
 6               -[tainted_path:ODG_EDGE*1..]
 7                   ->(sink:UNSAFE_SINK),
 8      // AST source sub-query
 9      (source_ast:AST_NODE)
10          -[source_ref:REF_EDGE]
11              ->(param),
12      // AST sink sub-query
13      (sink_ast:AST_NODE)
14          -[sink_ref:REF_EDGE]
15              ->(sink)
16  WHERE
17      ALL(edge IN tainted_path WHERE
18          edge.RelationType = "PROP" OR
19          edge.RelationType = "ARG" OR
20          edge.RelationType = "DEP")
21  RETURN *
```
Listing 5.1: Injection Query in Cypher Query Language (Neo4j).

5.2.1 Injection Query

A program is said to contain an injection vulnerability if data controlled by the attacker can reach an unsafe sink without being previously sanitised. With Explode.js CPGs, we can detect such scenarios by searching for paths connecting the tainted source to an unsafe sink in the ODG of the analyzed application.

The Injection Query, illustrated in Listing 5.1, can be understood as a combination of three sub-queries:

- **Tainted path sub-query** (lines 3-7 and 17-20): The goal of this query is to search for paths that connect the tainted source to unsafe sinks (lines 3-7). The query specifies that the paths must start with a `taint` edge (lines 3-5), where `source` represents malicious data entering the application (e.g., attacker-controlled function) and `param` represents an attacker-controlled parameter. Next, the query looks for tainted paths (lines 5-7) that connect `param` to an unsafe sink, restricting the type of edges that can be used in `tainted_path` (lines 17-20). In particular, `tainted_path` can include edges of type property, argument, and dependency (lines 17-20). Notably, tainted paths cannot include new version edges. The reason is simple: the fact that a given value depends on a new version of an object does not necessarily require it to depend on the previous version.

- **AST source sub-query** (lines 9-11): This query searches for the AST node that represents the function corresponding to the attacker-controlled parameter, `param`, whose value reaches a sensitive sink. It does so by looking for the reference edge (line 10) that connects the AST node to the corresponding ODG node. The AST node provides valuable information, including the vulnerable function's name and code line.

- **AST sink sub-query** (lines 13-15): This query searches for the AST node that represents the call to the unsafe sink `sink`. It does so by looking for the reference edge (line 14) that connects the AST node to the corresponding ODG node. The AST node provides valuable information, including the sink's name and code line.

```json
{
    "vuln_type": "command-injection",
    "source": "plot",
    "source_lineno": 3,
    "sink": "exec",
    "sink_lineno": 12,
    "tainted_params": [
        "options"
    ],
    "params_types": {
        "options": {
            "data": "any",
            "style": "any",
            "safe": "any",
            "format": "string",
            "filename": "string"
        }
    }
}
```

Listing 5.2: Taint summary for the command injection example of Figure 5.2.

Explode.js' Vulnerability Identitication Engine can detect code injection, command injection, and path traversal vulnerabilities. Furthermore, while maintaining the same Injection Query, it could be expanded to identify other taint-style vulnerabilities like SQL injection. This extension would involve supplying common sinks like `mysql.connection.query` to the CPG Constructor Module, which would represent the new sinks as unsafe sink nodes. Additionally, the query could be easily extended to ignore data dependency paths that go through certain program-specific sanitisation functions, keeping the ad-hoc aspects of the analysis within the vulnerability detection queries without the need to modify the structure of code property graphs and their construction algorithm.

When executed against the CPG presented in Figure 5.2, the Injection Query detects three command injection vulnerabilities, represented in the ODG by the paths highlighted in red that connect the tainted source node to three unsafe sinks nodes. More concretely, the attacker-controlled object `options` (o_1`:options`) can reach three dangerous sinks (lines 12, 14 and 23), represented in the graph by the `UNSAFE_SINK` nodes.

The information extracted by the query is unified in a taint summary. Listing 5.2 presents a simplified taint summary for the command injection vulnerability presented in Figure 5.2 (the complete taint summary contains a vulnerability report for each vulnerable sink, i.e., lines 12, 14, and 23). The taint summary is composed by the following properties:

- `vuln_type`: The type of vulnerability detected. For instance, in the case of the running example, the query identifies a command injection.
- `source` and `source_lineno`: The name of the vulnerable function and the corresponding code line. For example, in Figure 5.2, the attacker controls the `plot` function defined on line 3.
- `sink` and `sink_lineno`: The name of the unsafe sink reachable by attacker-controlled data and the corresponding code line. For instance, within the context of Figure 5.2, the attacker-controlled data reaches the dangerous sink `exec` located at line 12.
- `tainted_params`: The parameter names of the vulnerable function. For example, in the running

example, the vulnerable function `plot` contains one parameter: `options`.

- `param_types`: The types of the tainted parameters. For instance, in Listing 5.2, the attacker-controlled object `options` contains the sub property `filename`, typed as a string. The keyword `any` is used to represent any data type, including number, string, object, among others. The separator `'|'` is used to express types disjunctions; for instance, in Listing 5.5 the attacker-controlled parameter `obj` can be an array or an object.

5.2.2 Prototype Pollution Query

A program is said to contain a prototype pollution vulnerability if an attacker can modify the standard JavaScript built-in objects without proper sanitization. In this work, we focus specifically on the object `Object.prototype`, which is the topmost object on every prototype chain, serving as the base prototype for all objects created in JavaScript. Therefore, by altering `Object.prototype`, an attacker can potentially change the behaviour of all the other objects in the program. With Explode.js CPGs, we can detect prototype pollution vulnerabilities by searching for a first object lookup, where the attacker controls the property, followed by an object assignment over the result of the initial lookup, where the attacker controls both the set property and the value assigned. The prototype pollution base vulnerability pattern is illustrated in the ODG of Figure 5.3. By controlling the first lookup (grey and green paths) an attacker can obtain a pointer to `Object.prototype`. In the base vulnerability pattern of Figure 5.3, `obj_2` is set to `Object.prototype` if `key` is set to `__proto__`. Then, by controlling the set property (yellow path) and the value assigned to the object resulting from the first lookup (blue and red paths), an attacker can redefine JavaScript built-in methods like `toString`. In the base vulnerability pattern, it suffices to set `subKey` to `toString` and value to `maliciousFunction`.

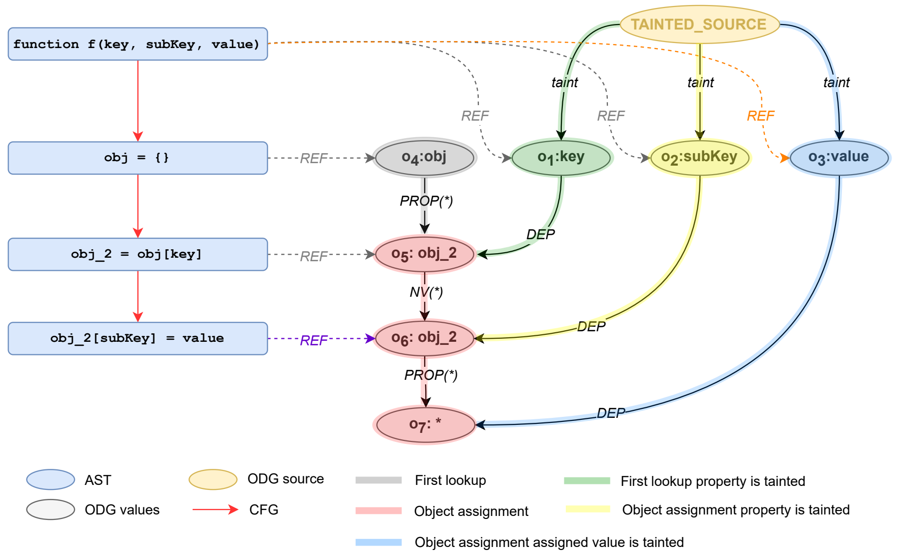


Figure 5.3: Object dependency graph of the prototype pollution base vulnerability pattern.

The Prototype Pollution Query, illustrated in Listing 5.3, can be understood as a combination of the following seven sub-queries:

```
1  MATCH
2      // First lookup sub-query
3      (obj:ODG_VALUE_NODE)
4          -[first_lookup:PROP {property: "*"}]
5              ->(sub_obj:ODG_VALUE_NODE),
6      // Object assignment sub-query
7      (sub_obj:ODG_VALUE_NODE)
8          -[nv:NV {property: "*"}]
9              ->(nv_sub_obj:ODG_VALUE_NODE)
10                 -[second_lookup:PROP {property: "*"}]
11                     ->(property:ODG_VALUE_NODE),
12     // First lookup property is tainted sub-query
13     (source:TAINTED_SOURCE)
14         -[key_taint:TAINT]
15             ->(key:ODG_VALUE_NODE)
16                 -[tainted_key_path:ODG_EDGE*1..]
17                     ->(sub_obj),
18     // Object assignment property is tainted sub-query
19     (source)
20         -[subKey_taint:TAINT]
21             ->(subKey:ODG_VALUE_NODE)
22                 -[tainted_subKey_path:ODG_EDGE*1..]
23                     ->(nv_sub_obj),
24     // Object assignment assigned value is tainted sub-query
25     (source)
26         -[value_taint:TAINT]
27             ->(value:ODG_VALUE_NODE)
28                 -[tainted_value_path:ODG_EDGE*1..]
29                     ->(property),
30     // AST source sub-query
31     (source_ast:AST_NODE)
32         -[source_ref:REF_EDGE]
33             ->(value),
34     // AST object assignment sub-query
35     (assignment_ast:AST_NODE)
36         -[assignment_ref:REF_EDGE]
37             ->(nv_sub_obj)
38 WHERE
39     ALL(edge IN tainted_key_path WHERE
40         edge.RelationType = "PROP" OR
41         edge.RelationType = "ARG" OR
42         edge.RelationType = "DEP") AND
43     ALL(edge IN tainted_subKey_path WHERE
44         edge.RelationType = "PROP" OR
45         edge.RelationType = "ARG" OR
46         edge.RelationType = "DEP") AND
47     ALL(edge IN tainted_value_path WHERE
48         edge.RelationType = "PROP" OR
49         edge.RelationType = "ARG" OR
50         edge.RelationType = "DEP")
51 RETURN *
```

Listing 5.3: Prototype Pollution Query in Cypher Query Language (Neo4j).

- **First lookup sub-query** (lines 3-5): This query searches for the first object lookup. The query specifies that the path must start with a `first_lookup` property edge (line 3-5) that connects an object to a corresponding arbitrary sub-object, `sub_obj`. In the base vulnerability pattern, this corresponds to the grey path connecting o_4 to o_5.

- **Object assignment sub-query** (lines 7-11): This query searches for the object assignment over the result of the initial lookup. The query specifies that the path must start with a `nv` new version edge (line 7-9) that connects the sub-object, resulting from the first lookup, to a corresponding new version, `nv_sub_obj`, generated by modifying an arbitrary property. Next, the query looks for a `second_lookup` property edge (line 9-11) that connects the new version of the sub-object to a corresponding arbitrary property, property. In Figure 5.3, this corresponds to the red path connecting o_5 to o_6 and o_6 to o_7.

- **First lookup property is tainted sub-query** (lines 13-17 and 39-42): This query checks if the property of the first lookup is controlled by an attacker, i.e., it searches for a path that connects the tainted source to the node that represents the sub-object resulting from the first lookup. The query specifies that the path must start with a key_taint edge (lines 13-15), where source represents malicious data entering the application and key a parameter controlled by an attacker. Next, the query looks for a tainted_key_path (lines 15-17) that connects key to the sub-object, sub_obj, restricting the type of edges that can be used in tainted_key_path (lines 39-42). In the base vulnerability pattern, this corresponds to the green path connecting TAINTED_SOURCE to o_5.
- **Object assignment property is tainted sub-query** (lines 19-23 and 43-46): This query checks if the property of the object assignment is controlled by an attacker, i.e., it searches for a path that connects the tainted source to the node that represents the new object version resulting from the object assignment. The query specifies that the path must start with a subKey_taint edge (lines 19-21), where source represents malicious data entering the application and subKey a parameter controlled by an attacker. Next, the query looks for a tainted_subKey_path (lines 21-23) that connects subKey to the new version object, nv_sub_obj, restricting the type of edges that can be used in tainted_subKey_path (lines 43-46). In the base vulnerability pattern, this corresponds to the yellow path connecting TAINTED_SOURCE to o_6.
- **Object assignment assigned value is tainted sub-query** (lines 25-29 and 47-50): This query checks if the assigned value of the object assignment is controlled by an attacker, i.e., it searches for a path that connects the tainted source to the node that represents the property resulting from the object assignment. The query specifies that the path must start with a value_taint edge (lines 25-27), where source represents malicious data entering the application and value a parameter controlled by an attacker. Next, the query looks for a tainted_value_path (lines 27-29) that connects value to the property resulting from the object assignment, property, restricting the type of edges that can be used in tainted_value_path (lines 47-51). In Figure 5.3, this corresponds to the blue path connecting TAINTED_SOURCE to o_7.
- **AST source sub-query** (lines 31-33): This query searches for the AST node that represents the function corresponding to the attacker-controlled parameters, key, subKey, and value, whose values are used in the first lookup and in the object assignment. It does so by looking for the reference edge (line 32) that connects the AST node to the corresponding ODG node. In the base vulnerability pattern, this reference edge corresponds to the orange dotted path. The AST node provides valuable information, including the vulnerable function's name and code line.
- **AST object assignment sub-query** (lines 35-37): This query searches for the AST node that represents the object assignment that leads to the pollution of Object.prototype. It does so by looking for the reference edge (line 36) that connects the AST node to the corresponding ODG

```
 1 function setValue(obj, path, value) {
 2     var dotPath = path.split(".");
 3     for (let i = 0; i < dotPath.length; i++) {
 4         const key = dotPath[i];
 5
 6         if (i === dotPath.length - 1) {
 7             obj[key] = value;
 8         }
 9
10         obj = obj[key];
11     }
12 }
```
Listing 5.4: Prototype pollution vulnerability (CVE-2021-23440)

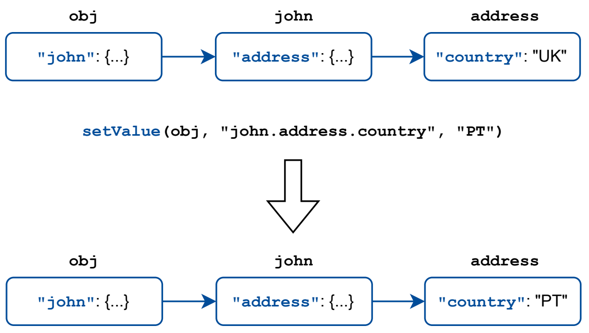


Figure 5.4: Correct usage of the package *set-value v3.0.0*, whose code is illustrated in Listing 5.4.

node. In the base vulnerability pattern, this reference edge corresponds to the purple dotted path. The AST node provides valuable information, including the assignment code line.

Listing 5.4 presents a code snippet slightly adapted from the npm package *set-value v3.0.0* [111], which is used to set nested properties on an object using dot notation. The exported function, `setValue` (line 1), receives as input the object to set the `value` on (`obj`), a string representing the path of the property to set (`path`), and the value to be set (`value`). The function splits the `path` string into an array, `dotPath`, that represents the levels of nesting to access the desired property (line 2). Subsequently, it traverses the nested properties in the `dotPath` array (line 3) by updating the `obj` reference at each iteration (line 10). When it reaches the last element in the path, it assigns `value` to that property, effectively modifying the object (line 7). Figure 5.4 illustrates a correct use of the package, where the value of the nested property `john.address.country` is changed to `"PT"`. The *set-value v3.0.0* package is susceptible to a prototype pollution vulnerability (CVE-2021-23440 [112]), allowing an attacker to pollute `Object.prototype`. For instance, by using the payload:

```
obj = {}, path = '__proto__.toString', value = function() {return 'Polluted!'}
```

an attacker is able to redefine the JavaScript built-in method `toString`. After the first iteration, `obj` is a reference to the top-level prototype `Object.prototype`. During the second iteration, the `toString` method is redefined for all JavaScript objects that rely on the native `Object.prototype.toString` (line 10).

The ODG of the prototype pollution vulnerability presented in Listing 5.4 is illustrated in Figure 5.5. When executed against the ODG, the Prototype Pollution Query detects one prototype pollution vulnerability, represented in the graph by the grey, red, green, blue and yellow highlighted paths. More concretely, the attacker controls the property, `key`, of the first lookup (line 10), represented in the graph by the green and grey paths. Additionally, the attacker also controls the set property, `key`, and the value assigned, `value`, of the object assignment over the result of the first lookup (line 7), represented in the ODG by the yellow, blue and red paths. Listing 5.5 presents the taint summary for the prototype pollution vulnerability of Listing 5.4.

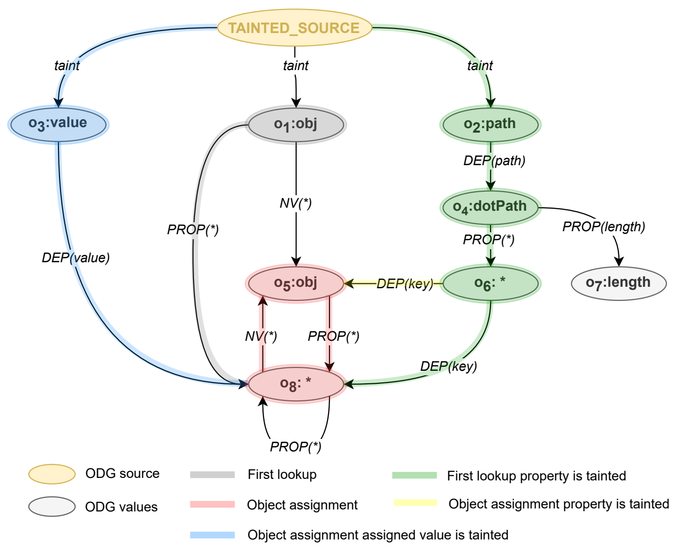


Figure 5.5: Object dependency graph of program given in Listing 5.4.

```
[
    {
        "vuln_type": "prototype-pollution",
        "source": "setValue",
        "source_lineno": 1,
        "sink": "obj[key] = value;",
        "sink_lineno": 7,
        "tainted_params": [
            "obj", "path", "value"
        ],
        "params_types": {
            "obj": "array | object",
            "path": "string",
            "value": "any"
        }
    }
]
```

Listing 5.5: Taint summary of the prototype pollution vulnerabilty of Listing 5.4

```
MATCH
    (source:AST_NODE_FunctionDeclaration)
        -[ref:REF_EDGE]
            ->(param:ODG_VALUE_NODE)
                -[struct_path:ODG_EDGE*0..]
                    ->(property:ODG_VALUE_NODE)
WHERE
    source.Id = $source_id AND
    ALL(edge IN struct_path WHERE
        edge.RelationType = "PROP" OR
        edge.RelationType = "ARG")
RETURN *
```

Listing 5.6: Parameter Structure Query in Cypher Query Language (Neo4j).

5.3 Queries for Reconstructing Attacker-Controlled Data

After identifying potential vulnerabilities and their corresponding paths, our goal is to confirm their existence. To this end, we must generate inputs that match the structure of attacker-controlled data and execute the vulnerable functions on those inputs. The parameters of the vulnerable functions can be primitive types, such as strings and numbers, or object types; in which case, they typically have a fixed structure. In this section, we describe our queries for reconstructing attacker-controlled parameters. We divide them into two categories: *parameter structure queries* that reconstruct the structure of tainted parameter objects and *type queries* that compute a potential JavaScript type for attacker-controlled parameters and properties whose structure was not determined by the parameter structure queries.

5.3.1 Parameter Structure Queries

In this subsection, we present the *parameter structure queries*, giving a formal description of the main query and demonstrating its functionality by example. With Explode.js CPGs, we can reconstruct object types by traversing down the dependence chain of the object dependence graph, examining the property and argument edges that flow from the parameter nodes identified as tainted.

```
1  const spawn = require("child_process").spawn;
2
3  function genArgs(spawn, condition) {
4      if (condition) {
5          var argString = spawn.args;
6          return argString.split(" ");
7      }
8  }
9
10 module.exports = function spawn_cmd(obj) {
11     var arguments = genArgs(obj.spawn, obj.cond);
12     var command = obj.spawn.cmd;
13     spawn(command, arguments)
14 }
```

Listing 5.7: Command injection vulnerability example for reconstructing attacker-controlled data.

```
{
    "vuln_type": "code-injection",
    "source": "spawn_cmd",
    "source_lineno": 10,
    "sink": "spawn",
    "sink_lineno": 13,
    "tainted_params": [
        "obj"
    ],
    "params_types": {
        "obj": {
            "spawn":{
                "args": "string",
                "cmd": "string"
            },
            "cond": "any"
        }
    }
}
```

Listing 5.8: Taint summary of the command injection vulnerability of Listing 5.7

The Parameter Structure Query is illustrated in Listing 5.6. The query must be executed once for each vulnerable function, which is identified by the constant `$source_id`. The goal of this query is to collect paths that connect attacker-controlled parameters to their deepest property. The query specifies that the paths must start with a `ref` edge (lines 2-4), connecting the vulnerable function to one of its parameters. Next, the query looks for structure paths (lines 4-6) that connect `param` to its deepest property, `property`. We specify that the structure paths consist only of property and argument edges (lines 9-11), as those are the ones that represent the structure of the parameter. Notably, structure paths do not include new version edges. The reason is simple: a property that is present in a new version of a parameter is not necessarily present in the parameter provided by the attacker. Finally, given that the query is only meant to work for the function with id `$source_id`, we restrict `source.Id` to match `$source_id` (line 8).

Listing 5.7 shows a command injection vulnerability, where the value of the attacker-controlled property `obj.spawn.cmd` (line 12), reaches an unsafe sink, `spawn` (line 13), without being previously sanitised. Figure 5.6 illustrates the ODG of the vulnerable program, where tainted paths are highlighted in red. By executing the Parameter Structure Query against the ODG of the source function `spawn_cmd`, we obtain the following list of property paths for the attacker-controlled parameter `obj`:

$$[[(o_1,\ \texttt{spawn},\ o_2),\ (o_2,\ \texttt{cmd},\ o_9)],$$
$$[(o_1,\ \texttt{spawn},\ o_2),\ (o_4,\ \texttt{args},\ o_6)],$$
$$[(o_1,\ \texttt{cond},\ o_3)]]$$

For each attacker-controlled parameter, the list of property paths extracted by the query go trough a post-processing phase to reconstruct the structure of the object. More concretely, the structure reconstruction algorithm builds a tree-like representation of the collected paths, creating each object/property at their corresponding nesting level. For example, the list of property paths above, can be easily merged into the tree of Figure 5.7 that represents the structure of the parameter `obj`. From this tree-like representation it is straight forward to compute the taint summary representing the structure of the tainted object as given in Listing 5.8. Finally, tainted parameters and object properties that are not objects are temporarily typed as any. For example, the properties `obj.cond`, `obj.spawn.cmd`, and `obj.spawn.args` are typed as any. In

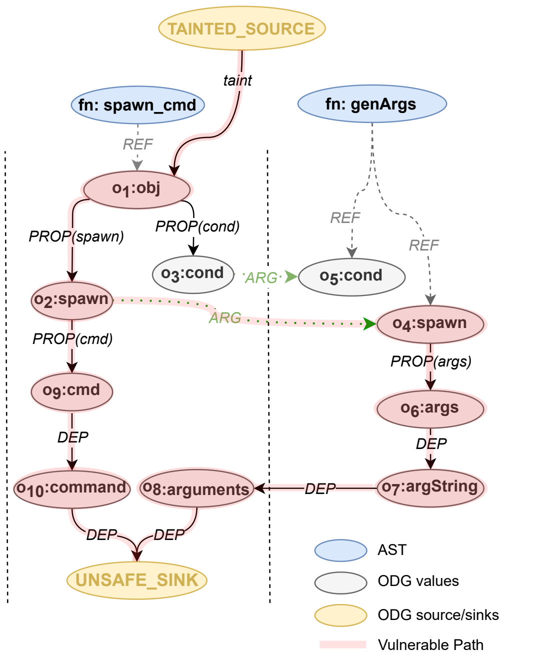


Figure 5.6: Object dependence graph of the program given in Listing 5.7

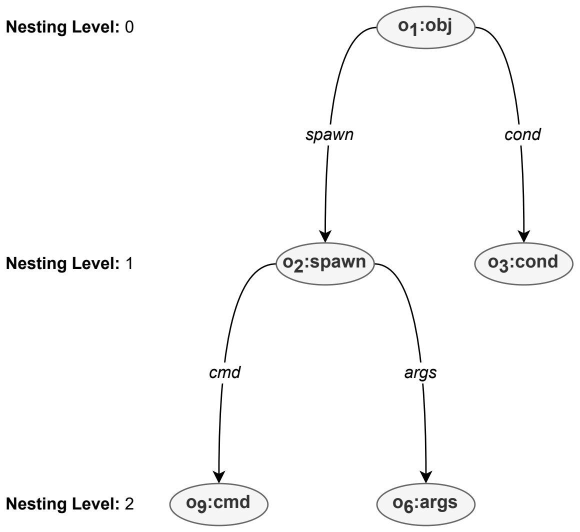


Figure 5.7: Tree representation of the structure of the attacker-controlled parameter `obj` from Listing 5.7

a posterior stage, the type queries are executed to determine more accurate types for these parameters and properties.

5.3.2 Type Queries

In this subsection, we present the *type queries*, explaining how they compute potential JavaScript types for each attacker-controlled parameter/property that does not correspond to an object. Then, we give a formal description of one of the queries and demonstrate its functionality by example. For simplification reasons, in the following paragraphs, when we mention attacker-controlled parameters, we are also including attacker-controlled object properties.

Parameter-Dependent Set Query: To enhance the accuracy of type computation, we must compute, for each attacker-controlled parameter, the set of objects and/or values that depend on it; onward *parameter-dependent set*. To compute this set, we use the *Parameter-Dependent Set Query* given in Listing 5.9. The query must be executed once for each attacker-controlled parameter and simply collects all paths that connect the parameter to nodes representing values computed using its value. The query searches for paths, `dep_path`, that connect the parameter to its dependent nodes (lines 2-4), restricting the type of edges that can be used in `dep_path` to dependency edges (lines 7-8). Finally, given that the query is only meant to work for the parameter with id `$param_id`, we restrict `param.Id` to match `$param_id` (line 6).

By executing the Parameter-Dependent Set Query against the ODG of Figure 5.6, for the object properties `obj.spawn.cmd`, `obj.spawn.args` and `obj.cond`, we obtain the following dependency paths:

```
MATCH
  (param:ODG_VALUE_NODE)
      -[dep_path:ODG_EDGE*1..]
          ->(variables:ODG_VALUE_NODE)
WHERE
    param.Id = $param_id AND
    ALL(edge IN dep_path WHERE
        edge.RelationType = "DEP")
RETURN *
```

Listing 5.9: Dependent Variables Query in Cypher Query Language (Neo4j).

```
MATCH
    // AST object sub-query
    (call_exp:AST_NODE_CallExpression)
        -[callee:AST_EDGE {RelationType: "callee"
             ↪ }]
            ->(mem_exp:AST_NODE_MemberExpression)
                -[object_edge:AST_EDGE {
                     ↪ RelationType: "object"}]
                    ->(object:AST_NODE_Identifier),
    // AST property sub-query
    (mem_exp)
        -[property_edge:AST_EDGE {RelationType: "
             ↪ property"}]
            ->(property:AST_NODE_Identifier)
WHERE
    object.IdentifierName in $dependent_values AND
    property.IdentifierName in $prototypes
RETURN true
```

Listing 5.10: Prototype Method Call Query in Cypher Query Language (Neo4j).

obj.spawn.cmd: [(o_9:cmd, DEP, o_{10}:$command$)]

obj.spawn.args: [(o_6:$args$, DEP, o_7:$argString$),

(o_7:$argString$, DEP, o_8:$arguments$)]

obj.cond: []

These dependency paths can be easily converted into the following parameter-dependent sets:

obj.spawn.cmd: {cmd, command}

obj.spawn.args: {args, argString, arguments}

obj.cond: {}

Note that the properties `obj.spawn.cmd`, `obj.spawn.args` and `obj.cond` are exactly those for which the Parameter Structure Query was unable to determine the structure.

Main Type Queries: Having computed the nodes that depend on each tainted parameter, we are now in position to explain the main type queries. Table 5.1 contains the description of the queries, along with the possible JavaScript type(s) checked by each query. For each parameter, the queries search for specific abstract syntax tree (AST) patterns that contain nodes representing the tainted parameter value or values computed using that value, which are in the parameter-dependent set. The queries use the parameter-dependent set because computing a type based solely on the parameter can be challenging as a parameter is often manipulated and assigned to other variables. Therefore, considering variables that depend on the parameter provides a more accurate type computation. Each time a query computes a different type, this type is appended to the list of potential types for the parameter. After executing all the type queries, the parameter possesses a list of potential JavaScript types it can assume. If none of the queries detect a pattern, the parameter is typed as any. In the following, we explain one of the queries given in Table 5.1. The other main type queries are analogous.

Prototype Method Call Query: This query is used to check whether a tainted variable may have array or string type and is given in Listing 5.10. The query searches for JavaScript array and string prototype

Query Name	**Description –** The query looks for...	**Type(s)**
Built-in Function Argument	Node.js built-in function calls where the argument type is known and the parameter is an argument of the function, e.g. `path.join(param)` implies that `param` has type string.	*all*
Typeof	conditional statements that check whether the type of the parameter is equal to a specific type, e.g. `if (typeof param === "object")`.	*all*
Static Methods	Node.js built-in static method calls where the parameter is an argument of the function, e.g. `Object.entries(param)` implies that param has type object.	array, number, object
Prototype Method Call	Node.js prototype method calls where the parameter is the object of the member expression, e.g. `param.join("")`.	array, string
For Of	for of statements where the parameter is the iterable array, e.g. `for (const i of param)`.	array
Boolean Binary Expression	conditional statements that check whether the parameter is equal to a boolean, e.g. `if (param === true)`.	boolean
Function Call	function calls where the parameter is the called function, e.g. `param()`.	function
Number Binary Expression	binary expressions where the operands are the parameter and a number literal, e.g. `param + 20`.	number
Number Operators	binary expressions where the operator can only be applied between two numbers and the parameter is an operand, e.g. `param ** num`.	number
String Concatenation	binary expressions where the operands are the parameter and a string literal, and the operator is "+" e.g. `param + "string"`.	string
Template Literal	template literals where the parameter is an embedded expression, e.g. `‘This is a string ${param}‘`.	string
Sink Call Argument	sink calls where the parameter is the first argument, e.g. `eval(param)`.	string
Computed Property	computed member expressions where the parameter is the property being accessed, e.g. `obj[param]`.	string

Table 5.1: Main Type Queries. *all*: array, boolean, function, number, object, string.

method calls of the form `object.method()`, where `method` is either an array or string prototype and `object` is the tainted parameter or a value that depends on it. For example, in `param.join("")`, `param` is the object and `join` is a known array prototype method call. The Prototype Method Call Query is executed twice for each parameter: one where `$prototypes` (line 15) is a list of JavaScript string prototype methods; and other where `$prototypes` is a list of JavaScript array prototype methods. The query can be understood as two sub queries:

- **AST object sub-query** (lines 3-7 and 13): This query searches for an AST call expression node, where the callee is a member expression of form *object.property* (lines 3-7), with `object` being a parameter-dependent value (line 13).

- **AST property sub-query** (lines 9-11 and 14): This query checks if the property node, `property`, is either a string or an array prototype method (line 14).

Suppose the Prototype Method Call Query is executed to check the type of the attacker-controlled property `obj.spawn.args` of the code snippet given in Listing 5.7. The relevant part of the code property graph is given below:

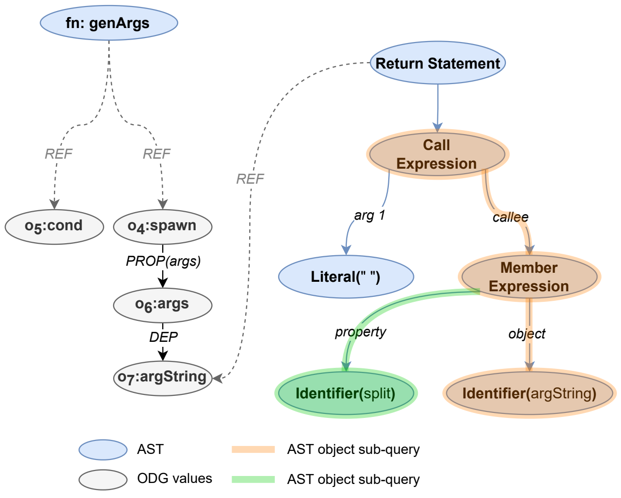


Figure 5.8: Partial object dependence graph and partial abstract syntax tree of the program given in Listing 5.7

The path identified by the AST object sub-query is highlighted in orange, while the path identified by the AST property sub-query is highlighted in green. Essentially, the query finds a call expression whose callee is a member expression with an object that depends on the tainted parameter (`argString`) and a property corresponding to a `String.prototype` method (`split`). This allows us to identify the property `obj.spawn.args` as potentially being of type string.

Summary

In this chapter, we introduced ExplodeQ.js, our library of queries for detecting injection vulnerabilities and reconstructing attacker-controlled data in Node.js applications. First, we presented Explode.js, the first tool to combine static analysis and symbolic execution to identify and confirm injection vulnerabilities in Node.js applications. Subsequently, we detailed two vulnerability detection queries: the Injection Query and the Prototype Pollution Query. Finally, we described the parameter structure queries and the type queries, responsible for reconstructing attacker-controlled data. In the next chapter, we evaluate ExplodeQ.js as an integral component of Explode.js's Vulnerability Identification Engine.

Chapter 6

Evaluation

In this chapter, we assess the accuracy of ExplodeQ.js in detecting injection vulnerabilities and reconstructing attacker-controlled data. The ExplodeQ.js library is integrated within the Explode.js Vulnerability Identification Engine (VIE). Hence, our evaluation covers the entire Explode.js VIE pipeline. We begin the chapter by describing our experimental setup. Then, we answer the following research questions (RQs):

- **RQ1:** What is Explode.js's effectiveness in detecting vulnerabilities?
- **RQ2:** What is Explode.js's precision in detecting vulnerabilities?
- **RQ3:** Does Explode.js find zero-day security vulnerabilities among real-world npm packages?
- **RQ4:** What is Explode.js's effectiveness in reconstructing attacker-controlled data?
- **RQ5:** What is the performance overhead of Explode.js's Vulnerability Identification Engine?

6.1 Experimental Setup

This section presents our experimental setup. We begin by describing the three benchmark datasets against which Explode.js was executed (Section 6.1.1). Then, we present the baseline detector used for comparison with Explode.js (Section 6.1.2), followed by an explanation of the experimental procedure employed to execute both Explode.js and the baseline detector against the benchmark datasets (Section 6.1.3). Finally, we present our experimental environment (Section 6.1.4).

6.1.1 Datasets

In the initial phase of the evaluation (RQ1 and RQ2), we measure Explode.js's effectiveness in detecting the target vulnerabilities. To achieve this, we evaluate Explode.js against two benchmark datasets, as this allows us to have a ground truth. In the subsequent phase of the evaluation (RQ3), we assess if Explode.js is able to find the target vulnerabilities in npm packages deployed in the wild. To achieve this, we gathered a set of 60K packages from the npm repository in September, 2023 and built a dataset containing real-world Node.js packages. In the third part of the evaluation (RQ4), we assess Explode.js's

CWE	Description	*VulcaNI*		*SecBench.js*		Total	Distribution (%)
		Total	**OK**	**Total**	**OK**		
CWE-22	Path Traversal (TS)	5	5	170	162	167	26.1%
CWE-78	Command Injection (TS)	92	87	101	95	182	28.4%
CWE-94	Code Injection (TS)	41	33	40	28	61	9.5%
CWE-471	Prototype Pollution	98	94	192	137	231	36%
Total	-	236	219	503	422	641	100%

Table 6.1: Summary of the benchmark datasets, according to the type of vulnerability. TS annotates **t**aint-**s**tyle vulnerabilities.

effectiveness in reconstructing attacker-controlled data by testing it against one benchmark dataset. Finally, in the last phase of the evaluation (RQ5), we assess the performance overhead of our tool by executing it against two benchmark datasets. In the following, we describe the three datasets.

VulcaN Injection (VulcaNI) Dataset: In Chapter 4, we thoroughly describe one of the main contributions of this thesis: the VulcaN Dataset. We recall that VulcaN is composed of 957 npm package versions that contain real-world Node.js vulnerabilities. Additionally, the review file of each package contains one or more reported vulnerabilities, and each vulnerability is annotated with the sink and source line number. Out of the 957 packages, we filtered 174 that report vulnerability types Explode.js is capable of detecting, i.e., code injection, command injection, path-traversal, and prototype pollution. Then, we excluded 15 out of this 174 packages due to wrong annotations (e.g., different types of vulnerability) and cases where the reported vulnerability was outside the package. VulcaNI is composed of the remaining 159 packages, encompassing a total of 219 vulnerabilities. Furthermore, to test Explode.js's effectiveness in reconstructing attacker-controlled data, we needed a ground truth. Hence, for each vulnerability in the dataset, we manually reconstructed the parameters of the corresponding vulnerable function (source). Table 6.1 presents the distribution of VulcaNI's vulnerability types.

SecBench.js Dataset: SecBench.js [17] is another vulnerability dataset, which comprises a total of 600 vulnerabilities, each associated with one npm package version. These vulnerabilities were reported in the GitHub Advisory Database [10], Snyk [113], and Huntr.dev [114]. SecBench.js includes the same vulnerability types as VulcaNI, along with Regular expression Denial of Service (ReDoS). Each package version includes only a single vulnerability, even if the package has multiple vulnerabilities. Each vulnerability is annotated with the sink line number. We filtered out 98 ReDoS vulnerabilities, which Explode.js cannot detect. We also excluded 80 vulnerabilities due to wrong annotations (e.g., non-existent or wrong sink line, no expected file) and cases where the package was unavailable or the file's type was not JavaScript, resulting in a total of 422 injection vulnerabilities. Table 6.1 presents the distribution of SecBench.js's vulnerability types, and the combined distribution of both benchmark datasets.

In the Wild Dataset: This dataset contains 60K real-world Node.js packages, extracted from the npm repository. From this set, we randomly executed Explode.js against 430 packages, provided that they were downloadable and importable.

6.1.2 Baseline Vulnerability Detection Tool

To offer a comparison of Explode.js with prior work, we also performed an evaluation of the latest state-of-the-art tool, ODGen [6], using the open-source implementation of the published work. We chose ODGen because it presents a similar approach to Explode.js, i.e., it creates a code property graph (CPG) representation of the analysed application and executes queries against it to detect Node.js injection vulnerabilities. Furthermore, in our empirical study for evaluating JavaScript vulnerability detection tools in Node.js packages (Section 4.2), ODGen demonstrated the most favorable trade-off between effectiveness and precision, ranking highest in precision and fourth in overall effectiveness among the assessed tools.

6.1.3 Benchmark Datasets Experimental Procedure

In this section, we explain the experimental procedures followed by Explode.js and ODGen to generate the results addressed in RQ1 (Section 6.2), RQ2 (Section 6.3), RQ4 (Section 6.5), and RQ5 (Section 6.6). We tested Explode.js and ODGen against every package included in the VulcaNI and SecBench.js benchmark datasets to detect the following vulnerabilities: path traversal (CWE-22), code injection (CWE-94), command injection (CWE-78), and prototype pollution (CWE-471). More concretely, for each package we executed the tools against the files that contain the reported vulnerabilities. Each individual run encompasses two phases: *(i)* graph construction, which converts the analysed file into its corresponding code property graph (CPG) representation; and *(ii)* detection, which executes the detection queries over the resulting CPG to detect Node.js injection vulnerabilities. Additionally, Explode.js contains a third phase: *(iii)* reconstruction, which executes the parameter reconstruction queries for each parameter of the vulnerable functions (sources) reported in phase *(ii)*. The phases *(i)* and *(ii)* are executed with a 5-minute timeout, while the reconstruction phase *(iii)* has an individual timeout of 2 minutes.

6.1.4 Experimental Environment

Our testbed consisted of 6 64-bit Ubuntu 22.04.3 servers with 64GB of RAM and 2x Intel(R) Xeon(R) Gold 5320 2.2GHz CPUs. All evaluations on the benchmark datasets were performed in one server, and the experiments on the In the Wild dataset used the six servers to collect and speed up the analysis.

6.2 RQ1: What is Explode.js's effectiveness in detecting vulnerabilities?

In this section, we assess Explode.js's effectiveness in detecting known injection vulnerabilities, and compare it with ODGen's. First, we introduce our evaluation methodology (Section 6.2.1). Then, we compare the results produced by both tools (Section 6.2.2).

CWE	VulcaNI							SecBench.js						
	Total	*Explode.js*		*ODGen*				Total	*Explode.js*		*ODGen*			
		TP	TPR	TP*	TPR*	TP	TPR		TP	TPR	TP*	TPR*	TP	TPR
CWE-22	5	5	1	2	0.4	1	0.2	162	156	0.96	132	0.81	9	0.06
CWE-78	87	81	0.93	49	0.56	31	0.36	95	91	0.96	72	0.76	41	0.43
CWE-94	33	30	0.91	17	0.52	17	0.52	28	22	0.79	11	0.39	11	0.39
CWE-471	94	37	0.39	13	0.14	13	0.14	137	74	0.54	28	0.2	27	0.2
Total	219	153	0.7	81	0.37	62	0.28	422	343	0.81	243	0.58	88	0.21

Table 6.2: Number of vulnerabilities detected in the VulcaNI and SecBench.js datasets. The incomplete detection of ODGen is represented with *.

6.2.1 Evaluation Methodology

We measured the number of true positives (TP) and the true positive rate (TPR) for both Explode.js and ODGen when executed against the VulcaNI and the SecBench.js . For VulcaNI, we consider a true positive if the vulnerability type, and both the source's and sink's line numbers reported by the tools match the dataset annotations, and for SecBench.js, we consider a true positive if the vulnerability type and the sink match, since SecBench.js does not annotate the source line number. We distinguish ODGen in two cases: *(i) complete*, when ODGen is able to detect the vulnerability type, the source code line, and the sink code line; and *(ii) incomplete*, when ODGen correctly detects the vulnerability type, but does not report the sink code line.

6.2.2 Results

Table 6.2 presents the number of vulnerabilities detected in VulcaNI and SecBench.js by Explode.js and ODGen. In the table, we distinguish ODGen's detection, where * refers to the incomplete detection. Explode.js detects 70% and 81% of the reported vulnerabilities, in the VulcaNI and SecBench.js datasets, outperforming ODGen by 1.89× and 1.40×, respectively. When taking into consideration the correct identification of the sink code lines, Explode.js detects 2.50× and 3.86× more vulnerabilities than ODGen, in the VulcaN and SecBench.js datasets, respectively. Notably, Explode.js outperforms ODGen in the detection of every vulnerability type targeted.

False Negative (FN) analysis: We manually inspected false negatives generated by Explode.js to understand their causes. The main reason for FNs in Explode.js is unimplemented JavaScript functionalities in the CPG Constructor Module, resulting in missing dependency edges in the graph. For instance, the CPG Constructor Module does not currently provide full support for the `arguments` and the `this` keywords. Explode.js exhibits a lower TPR in prototype pollution detection when compared to taint-style detection. This disparity arises from the nature of prototype pollution vulnerability patterns, which frequently involve third-party npm packages like *for-own* [115] and *for-in* [116] that ultimately lead to the vulnerability. However, since the code of these external packages is not represented in the graph, the Prototype Pollution Query fails to recognize the associated vulnerability pattern. Additionally, prototype pollution sources often use the `arguments` keyword, which, as previously explained, is not fully supported.

Vulnerability	Total	Explode.js			ODGen		
		TP	FP	Precision	TP	FP	Precision
Path Traversal	167	161	36	0.82	134	0	1
Command Injection	182	172	9	0.95	121	90	0.57
Code Injection	61	52	14	0.79	28	86	0.25
Prototype Pollution	231	111	76	0.59	41	13	0.76
Total	641	496	135	0.79	324	189	0.63

Table 6.3: Precision of Explode.js and ODGen (incomplete) for the VulcaNI and SecBench.js datasets combined.

6.3 RQ2: What is Explode.js's precision in detecting vulnerabilities?

In this section, we assess Explode.js's precision in detecting known injection vulnerabilities, and compare it with ODGen's. We begin this section by introducing our evaluation methodology (Section 6.3.1). Subsequently, we compare the results generated by both tools (Section 6.3.2).

6.3.1 Evaluation Methodology

We measured the number of false positives (FP), the number of true false positives (TFP), and the precision for both Explode.js and ODGen when executed against the VulcaNI and the SecBench.js datasets. We consider a false positive if the vulnerability detected by the tools is not annotated in the datasets. We consider a true false positive if the detected vulnerability does not correspond to a vulnerable path, i.e., we cannot generate an exploit that proves the existence of the vulnerability. We make this distinction because both tools often found true positives (i.e., vulnerabilities that correspond to exploitable paths) that are not annotated in the datasets. This happened because the datasets are not complete, e.g., SecBench.js only reports one tainted sink per package. However, vulnerable packages commonly contain more than one exploitable unsafe sink. For example, in the SecBench.js dataset, the authors only report one tainted sink for the npm package *ffmpegdotjs v0.0.4*, yet in total there are eleven described in the CVE. Additionally, in some packages we found zero-day vulnerabilities. For instance, in the VulcaNI dataset, we correctly report a code injection vulnerability present in the npm package *mosc v1.0.0* [117], but we also found a zero-day prototype pollution vulnerability that was not annotated. Based on these findings, we use the following formula to calculate the precision: $Precision = TP/(TP + TFP)$, where the TP number only includes the annotated vulnerabilities.

6.3.2 Results

Table 6.3 illustrates the precision of Explode.js and ODGen (incomplete) for the VulcaNI and SecBench.js datasets combined. Explode.js achieves 79% precision, reporting 135 true false positives, whereas ODGen achieves 63% precision, signaling 189 true false positives. Explode.js outperforms ODGen in precision by 1.25$\times$, also reporting 28% fewer true false positives. ODGen surpasses Explode.js in precision by 1.29$\times$ when detecting prototype pollution vulnerabilities.

```
 1  function merge_recursive(base, extend) {
 2    if (typeOf(base) !== 'object')
 3      return extend;
 4    for (var key in extend) {
 5      if (typeOf(base[key]) === 'object' && typeOf(
          ↪ extend[key]) === 'object') {
 6        base[key] = merge_recursive(base[key],
            ↪ extend[key]);
 7      } else {
 8        base[key] = extend[key];
 9      }
10    }
11    return base;
12  }
```

Listing 6.1: Prototype pollution vulnerability present in the npm package *merge v1.2.0* (CVE-2018-16469)

```
 1  [
 2      {
 3          "vuln_type": "prototype-pollution",
 4          "source": "merge",
 5          "source_lineno": 1,
 6          "sink": "base[key] = merge_recursive(base[
              ↪ key], extend[key]);",
 7          "sink_lineno": 6
 8      },
 9      {
10          "vuln_type": "prototype-pollution",
11          "source": "merge",
12          "source_lineno": 1,
13          "sink": "base[key] = extend[key];",
14          "sink_lineno": 8
15      }
16  ]
```

Listing 6.2: Taint summary for the prototype pollution vulnerability of Listing 6.1

False positive analysis: We manually inspected false positives produced by Explode.js to understand their causes. The primary cause of taint-style FPs is the mischaracterization of a particular vulnerability. We consider the Node.js function `require` as an unsafe sink, which means that if attacker-controlled data reaches this function, Explode.js will report a code injection vulnerability. However, this assumption is not always true. The `require` function leads to a code injection vulnerability only if, in addition to importing a package, an attacker can also execute an exported function of that package where the function's arguments are under its control. The main reason for prototype pollution false positives is the report of recursive object assignments as sinks. Although these assignments contribute to the existence of the vulnerability, they do not directly pollute `Object.prototype`. For instance, for the code snippet extracted from the vulnerable npm package *merge v1.2.0* [118] shown in Listing 6.1, Explode.js reports both lines 6 and 8 as sinks, as shown in the corresponding simplified taint summary of Listing 6.2. However, line 6 only iterates trough the objects `base` and `extend` recursively. Ultimately, the actual object assignment that pollutes the prototype always occurs on line 8, where `base` is a reference to `Object.prototype`, `key` the built-in method to be polluted (e.g. `toString`), and `extend[key]` the new assigned value.

6.4 RQ3: Does Explode.js find zero-day security vulnerabilities among real-world npm packages?

In this section, we assess Explode.js's effectiveness in detecting zero-day vulnerabilities in real-world npm packages. We begin by describing the experimental procedure for running Explode.js against the In the Wild dataset (Section 6.4.1). Subsequently, we introduce our evaluation methodology (Section 6.4.2). Finally, we analyse the results generated by Explode.js (Section 6.4.3).

Vulnerability Type	Packages	Files	TP	FP	Total
Path Traversal	5	13	1	13	14
Command Injection	24	29	18	11	29
Code Injection	1	1	1	0	1
Prototype Pollution	24	35	11	23	34
Total	54	78	31	47	78

Table 6.4: Number of zero-day security vulnerabilities detected by Explode.js in the In the Wild dataset.

6.4.1 Experimental Procedure

We executed Explode.js against a subset of the npm packages contained in the In the Wild dataset. The In the Wild dataset contains a total of 60K npm packages, from which we randomly filtrated 430. These 430 npm packages comprise a total of 3915 Node.js files. Explode.js was executed against each individual file with a timeout of 5 minutes. Analogously to the experimental procedure described in Section 6.1.3, for each tested file, Explode.js executed two phases: *(i)* graph construction, and *(ii)* detection. We did not execute the *(iii)* reconstruction phase against the In The Wild dataset because our focus in this section was solely detecting zero-day vulnerabilities.

6.4.2 Evaluation Methodology

We consider a detected vulnerability (true positive) as zero-day if we can manually confirms its existence with a functional exploit, and we do not find any information about the vulnerability online. Moreover, a vulnerability detected by Explode.js is considered a false positive if we cannot produce an exploit confirming its existence.

6.4.3 Results

Explode.js detected at least one security vulnerability in 184 of 3915 tested files. Due to time constraints, we randomly selected 30% of the signaled packages to analyse manually. Table 6.4 demonstrates the number of zero-day vulnerabilities detected by Explode.js, grouped by vulnerability type. Explode.js detects a total of 31 zero-day vulnerabilities across 54 npm packages, while signalling 47 false positives.

False positive (FP) analysis: We manually inspected false positives to identify their causes in Explode.js zero-day detection. The main reasons for FPs are: the presence of sanitization functions between tainted sources and unsafe sinks; and, in complex packages (comprising over 20 files), tainted data reaches unsafe sinks only under highly specific circumstances, making the creation of an exploit exceptionally challenging.

Responsible disclosure: In collaboration with snyk.io [113], we have responsibly reported all the zero-day vulnerabilities to their respective developers along with a proof of vulnerability. Among the 31 vulnerabilities, one CVE [18] identifier has been assigned for a command injection vulnerability found in the *find-exec v1.0.3* [119] npm package. Furthermore, we are currently awaiting response from the maintainers of a package (whose name we omit for security reasons) that contains a prototype pollution vulnerability. The remaining 29 vulnerabilities lack a CVE assignment for two reasons: the package is not popular enough (less than 2K weekly downloads), which means it does not present a significant impact in the open source community; or, the package maintainers consider the reported vulnerability a feature (e.g., packages extending the usage of the `exec` API).

6.5 RQ4: What is Explode.js's effectiveness in reconstructing data controlled by attackers?

In this section, we assess Explode.js's effectiveness in reconstructing attacker-controlled data. First, we introduce our evaluation methodology (Section 6.5.1). Subsequently, we analyse the results produced by Explode.js (Section 6.5.2).

6.5.1 Evaluation Methodology

We recall that Explode.js's *parameter reconstruction queries* reconstruct the attacker-controlled parameters of a vulnerable function (source). More concretely, there are two types types of reconstruction queries: *the parameter structure queries* (Section 5.3.1), which reconstruct the structure of parameters that correspond to objects, and the *type queries* (Section 5.3.2), which compute a potential JavaScript type for attacker-controlled parameters and properties whose structure was not determined by the parameter structure queries. We measured the number of vulnerable functions Explode.js is able to reconstruct when executed against the VulcaNI dataset. The reconstruction process of a vulnerable function involves the following three phases:

1. **Parameters:** Explode.js retrieves **all** the parameters of the vulnerable function.
2. **Structures:** Explode.js correctly reconstructs the structure of **all** the parameters of the vulnerable function that correspond to objects. A correct structure implies that all the properties and sub-objects of each object are created at their corresponding nesting level.
3. **Types:** For **all** the parameters/properties of the vulnerable function whose structure was not determined by the parameter structure queries, Explode.js computes at least one of the types they may assume during runtime.

Note that the reconstruction phases are interdependent. The correct reconstruction of structures (2) relies on the accurate retrieval of parameters (1). Similarly, the computation of types (3) is contingent upon both the accurate retrieval of parameters and the correct reconstruction of structures. Therefore, a vulnerable function is only fully reconstructed if it checks the three reconstruction phases.

6.5.2 Results

Table 6.5 displays the number of attacker-controlled vulnerable functions reconstructed by Explode.js for the vulnerabilities detected in the VulcaNI dataset, grouped by the reconstruction phases. Notably, each vulnerability detected by Explode.js contains a corresponding vulnerable function. Therefore, the

CWE	Vulnerable Functions	Reconstruction Phases		
		1. **Parameters**	2. **Structures**	3. **Types**
CWE-22	5	5	5	5
CWE-78	81	80	80	78
CWE-94	30	25	25	20
CWE-471	37	38	37	15
Total	153	148 (96.73%)	147 (96.08%)	118 (77.12%)

Table 6.5: Number of attacker-controlled vulnerable functions reconstructed by Explode.js for the vulnerabilities detected in the VulcaNI dataset.

number of vulnerable functions is equal to the number of vulnerabilities detected. Additionally, the table includes accuracy percentages for each reconstruction phase, calculated in comparison to the total number of vulnerable functions. Explode.js demonstrates high accuracy in retrieving vulnerable function parameters (96.73%) and reconstructing parameters that correspond to objects (96.08%). Furthermore, it successfully computes the types for 118 out of 147 vulnerable functions with correctly reconstructed object structures. In total, Explode.js is able to fully reconstruct 77.12% of the vulnerable functions detected in the VulcaNI dataset.

False negative (FN) analysis: We manually inspected false negatives to understand their causes. The main reason for Explode.js failing to retrieve the correct parameters of a vulnerable function (1), and consequently not reconstructing the ones that correspond to objects (2) is unimplemented JavaScript functionalities in the CPG Constructor Module, such as lack of support for object destructuring within function parameters. The only scenario in which the correct parameters are retrieved (1), but the structure reconstruction (2) is incorrect, is due to a bug in the CPG Constructor Module, that leads to the creation of a property that should not exist. The false negatives related to type computation (3) are expected. While an experienced developer can assume a parameter's potential type by manually analyzing its usage within a vulnerable function, Explode.js relies on specific AST patterns. Consequently, improving the accuracy of type computation can only be accomplished with the development of additional type queries.

6.6 RQ5: What is the performance overhead of Explode.js's Vulnerability Identification Engine?

In this section, we evaluate the performance overhead of Explode.js's Vulnerability Identification Engine (VIE) and ODGen when detecting known injection vulnerabilities. Additionally, we also assess the performance overhead of the VIE when reconstructing attacker-controlled data. We begin this section by describing our evaluation methodology (Section 6.6.1). Subsequently, we analyse and compare the results generated by both tools (Section 6.6.2).

6.6.1 Evaluation Methodology

We measured the average time taken by each phase during the execution of Explode.js and ODGen against the VulcaNI and the SecBench.js datasets, grouped by vulnerability type. We timed the *(i)* graph

CWE	Total	Graph Construction	Detection			Reconstruction	Total
			Injection	Prototype Pollution	Total		
CWE-22	167	3.52s	2.58s	0.31s	2.89s	4.38s	15.98s
CWE-78	182	3.71s	2.38s	0.32s	2.70s	4.68s	12.87s
CWE-94	61	5.83s	2.38s	0.57s	2.95s	6.07s	26.26s
CWE-471	231	4.18s	2.39s	0.86s	3.22s	1.34s	33.08s
Total	641	3.99s	2.44s	0.52s	2.95s	3.59s	21.81s

Table 6.6: Average time, in seconds, taken by each phase during the execution of Explode.js against the VulcaNI and the SecBench.js datasets, grouped by vulnerability type. The timeout duration is only considered in the total average time (last column).

CWE	Total	Graph Construction	Detection	Total
CWE-22	167	5.13s	0.80s	11.17s
CWE-78	182	8.34s	1.17s	43.80s
CWE-94	61	4.42s	0.72s	105.39s
CWE-471	231	17.25s	1.38s	199.58s
Total	641	8.39s	1.01s	86.61s

Table 6.7: Average time, in seconds, taken by each phase during the execution of ODGen against the VulcaNI and the SecBench.js datasets, grouped by vulnerability type. The timeout duration is only considered in the total average time (last column).

construction phase and the *(ii)* detection phase for both tools. For Explode.js, we individually measured the average execution time of the Injection Query and the Prototype Pollution Query, both of which are part of the *(ii)* detection phase. Additionally, for Explode.js, we also calculated the average duration of the *(iii)* reconstruction phase, which includes the total execution time of both the parameter structure queries and the type queries. We recall from Section 6.1.3 that phases *(i)* and *(ii)* were executed with a 5-minute (300s) timeout, while phase *(iii)* had an individual timeout of 2 minutes (120s).

6.6.2 Results

Table 6.6 displays the average time taken by each phase during the execution of Explode.js against the VulcaNI and SecBench.js datasets, grouped by vulnerability type. Similarly, Table 6.7 illustrates the corresponding data for ODGen. Explode.js demonstrates superior efficiency in graph construction, operating at approximately 52.4% of the time taken by ODGen, which constructs graphs at a comparatively slower pace, with an average time of 8.39 seconds. Notably, ODGen takes on average 17.25 seconds to build the graph when analyzing files containing prototype pollution vulnerabilities. This occurs due to the substantial expansion in the size of the object dependency graph (ODG) caused by the patterns associated with prototype pollution vulnerabilities. Explode.js and ODGen exhibit an average detection time of 2.95 seconds and 1.01 seconds, respectively. It is important to note that the detection times were calculated based on files that completed analysis within the designated timeout. If timeouts during detection were taken into account, ODGen's average detection time would significantly increase, surpassing that of Explode.js. The reconstruction phase does not significantly impact the overall performance of Explode.js, presenting an average execution time of 3.59 seconds. Notably, files containing prototype pollution vulnerabilities show comparatively lower average reconstruction times. This is due to attacker-controlled objects of prototype pollution vulnerabilities typically having simpler structures, resulting in the Parameter Structure Query (Section 5.3.1) executing more quickly. Explode.js completes its analysis, on average,

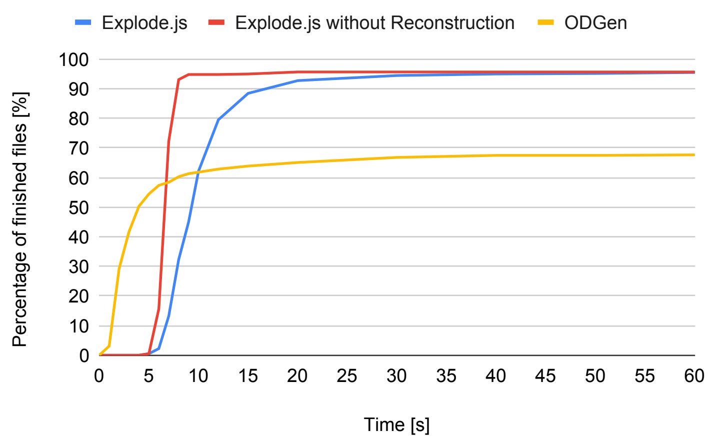


Figure 6.1: Cumulative distribution function (CDF) graph of total execution time to finish analysis.

within 21.81 seconds, making it 3.97× faster than ODGen, which has a total average analysis time of 86.61 seconds. It is important to notice that the total average time includes the timeout duration, significantly increasing the average analysis time for both tools.

Figure 6.1 shows a cumulative distribution function (CDF) graph of the percentage of files finished analysing within the first 60 seconds. The blue line represents the cumulative distribution for the full Explode.js pipeline, including the three phases, whereas the red line does not include the *(iii)* reconstruction phase. In total, both tools analysed 582 vulnerable files. Both Explode.js variants successfully analyse 95% of vulnerable files under 60 seconds, while ODGen only analyses 68%. Additionally, both Explode.js and Explode.js without reconstruction timed out in 23 (3.95%) analysed files, whereas ODGen experienced timeouts analysing 170 (29.21%).

Summary

In this chapter, we evaluated Explode.js's Vulnerability Identification Engine (VIE). We assessed both Explode.js's and ODGen's effectiveness, precision, and performance by executing them against two curated datasets of Node.js injection vulnerabilities: the VulcaNI and the SecBench.js datasets. We also executed Explode.js against real-world npm packages, where it detected several zero-day security vulnerabilities, with one CVE identifier being assigned so far. Additionally, we tested Explode.js's effectiveness in reconstructing attacker-controlled data by executing it against the VulcaNI dataset. The following chapter concludes this document by summarizing the achievements of this thesis and outlining avenues for future work.

Chapter 7

Conclusions and Future Work

In this chapter, we summarize the main findings and conclusions drawn from the research presented in this thesis (Section 7.1). Additionally, we identify promising directions for future research (Section 7.2).

7.1 Conclusions

Node.js has become one of the most popular choices among developers to build scalable and highly performant server-side web applications. However, because of JavaScript's dynamic behavior and the npm ecosystem's openness, developers may unintentionally introduce vulnerabilities into their code. Some of the most frequently reported vulnerabilities in Node.js applications include injections and prototype pollution, which can typically by exploited by attackers to gain complete control over a web server. To mitigate these risks, integrating vulnerability analysis tools into CI/CD pipelines has been a key prevention technique as they help identify and resolve security issues early in the development process. However, the complexity of Node.js applications, such as the large number of third-party package dependencies, combined with limitations of existing analysis tools [11] make it challenging for developers to perform reliable automated analyses. This highlights the growing demand for effective and efficient automated methods to detect and prevent impactful security vulnerabilities in Node.js applications.

Considering the aforementioned challenges, we make significant contributions to the development of Explode.js, a new JavaScript analysis tool being developed at INESC-ID. Explode.js is the first tool to combine static and dynamic analysis to identify injection vulnerabilities in Node.js applications and confirm them by producing a functional exploit. Explode.js is divided into two engines: *(i)* the Vulnerability Identification Engine (VIE) executes queries over a code property graph (CPG) representation of the analyzed application to detect injection vulnerabilities; and *(ii)* the Vulnerability Confirmation Engine (VCE) employs symbolic execution to automatically produce an exploit. The specific contributions of this thesis for the development of Explode.js are as follows:

1. **The creation of the VulcaN dataset.** To carry out the evaluation of Explode.js, we constructed the VulcaN dataset, the largest known curated dataset of Node.js code vulnerabilities. VulcaN is composed of 957 npm packages with confirmed security vulnerabilities. Each vulnerable package

in the dataset includes an advisory report with annotations that specify the location of the reported vulnerabilities, making VulcaN an effective benchmark for assessing the effectiveness of vulnerability detection tools. We used VulcaN to conduct the first empirical study of static code analysis tools for detecting vulnerabilities in Node.js packages, where we found that the assessed tools fail to detect many vulnerabilities and exhibit high false positive rates.

2. **The creation of ExplodeQ.js.** ExplodeQ.js is an integral component of Explode.js, consisting of a library of queries to detect injection vulnerabilities and reconstruct attacker-controlled data in Node.js applications. ExplodeQ.js contains two types of queries: *(i)* the vulnerability detection queries detect injection and prototype pollution vulnerabilities; and *(ii)* the data reconstruction queries reconstruct attacker-controlled data. Integrated within Explode.js's VIE, these queries are executed against a CPG representation of the analyzed application and extract the information the VCE needs to apply symbolic execution and produce a functional exploit.

3. **The evaluation of Explode.js.** We thoroughly evaluated ExplodeQ.js as a component of Explode.js's Vulnerability Identification Engine. We tested both Explode.js and ODGen, the state-of-the-art tool for static vulnerability detection in Node.js packages, against two benchmark datasets composed of npm packages with confirmed vulnerabilities: our VulcaN dataset, and the SecBench.js dataset. Explode.js significantly outperformed ODGen in overall effectiveness, precision and performance. Furthermore, Explode.js was tested on 430 real-world npm packages, leading to the detection of 31 zero-day vulnerabilities and the assignment of one CVE identifier so far as of the time of the writing of this thesis.

7.2 Future Work

The conducted research paves the way for future work in the field of static analysis and automatic vulnerability detection and exploitation. Section 7.2.1 delineates potential steps to enhance the current work within the scope of Explode.js. Section 7.2.2 suggests how Explode.js's contributions can be leveraged as a foundation for developing other code analysis tools.

7.2.1 Extending Explode.js

In the near future, we plan to expand the functionality of Explode.js' Vulnerability Identification Engine (VIE) to detect additional Node.js security vulnerabilities, like regular expressions denial of service (ReDoS) and cross-site scripting (XSS). This expansion will require extending our library of queries, ExplodeQ.js, by creating new detection queries capable of identifying new patterns associated with these vulnerabilities.

Furthermore, we plan to continue testing Explode.js against the In the Wild dataset. Although we have already executed our tool against a significant amount of real-world npm packages, uncovering several zero-day vulnerabilities, some of which received CVEs [18], our goal is to persist in the search for previously unknown vulnerabilities. This ongoing effort aims to establish Explode.js as a reliable tool for detecting real-world Node.js injection vulnerabilities.

Finally, we will keep developing Explode.js' Vulnerability Confirmation Engine to automatically generate exploits that confirm the existence of the vulnerabilities detected by the VIE.

7.2.2 Beyond Explode.js

We present two case studies that would benefit from the development of new vulnerability detection tools built upon the existing structure of Explode.js:

- **Security vulnerabilities in browser extensions:** Browser extensions enhance web user's experience but present significant security risks. Written in JavaScript, these extensions can read and write user data on web applications, accessing sensitive information such as browsing history and credentials. For instance, undetected code injection vulnerabilities allow malicious websites to exploit browser extension APIs, potentially leading to data theft and severe security breaches.
- **Denial of Wallet vulnerabilities in serverless applications:** Serverless computing offers automated scalability and availability in a pay-as-you-go manner, allowing developers to focus on application logic. JavaScript, a popular language for serverless apps, faces typical vulnerabilities like injection attacks, and specific serverless threats, like Denial of Wallet (DoW) attacks. In DoW attacks, attackers can exploit the scalability of serverless architectures to trigger excessive resource usage, causing financial harm to organizations

The existing Explode.js code property graphs (CPGs) can be extended to incorporate the unique characteristics of browser extensions and serverless applications. Following the methodology outlined in Explode.js, we can analyze the most critical vulnerability patterns associated with each case, such as code injections in browser extensions and DoW attacks in serverless applications. This analysis enables the development of customized graph queries designed to identify these vulnerabilities when executed against the new CPGs. Finally, we employ symbolic execution to synthesise exploits that confirm the existence of these vulnerabilities.

Furthermore, Explode.js can serve as a foundation for other use cases beyond automated vulnerability detection. For example, web developers commonly use full-stack frameworks like MERN (MongoDB [120], Express.js [46], React.js [121], and Node.js [3]) for web application development. Despite their widespread use, these frameworks lack native support for specifying and enforcing personal data protection policies. This gap can result in GDPR [122] compliance issues, such as processing data outside advertised purposes. We can extend Explode.js CPGs to represent these web applications, and execute graph queries over the resulting graph to detect potential GDPR violations, shifting the focus from detecting security vulnerabilities to ensuring compliance with data protection regulations.

Bibliography

[1] Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/.

[2] Usage statistics of JavaScript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript/.

[3] Node.js. https://nodejs.org.

[4] npm. https://www.npmjs.com/.

[5] C. Staicu, M. Pradel, and B. Livshits. SYNODE: understanding and automatically preventing injection attacks on NODE.JS. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. URL http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_07A-2_Staicu_paper.pdf.

[6] S. Li, M. Kang, J. Hou, and Y. Cao. Mining node.js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 143–160, Boston, MA, Aug. 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/li-song.

[7] Li, Song and Kang, Mingqing and Hou, Jianwei and Cao, Yinzhi. Detecting node.js prototype pollution vulnerabilities via object lookup analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 268–279, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3468542. URL https://doi.org/10.1145/3468264.3468542.

[8] O. Arteau. Prototype pollution attack in nodejs application. In *NorthSec*, 2018.

[9] Thousands of Malicious npm Packages Threaten Web Apps. https://threatpost.com/malicious-npm-packages-web-apps/178137/.

[10] Github advisory database. https://github.com/advisories.

[11] T. Brito, M. Ferreira, M. Monteiro, P. Lopes, M. Barros, J. F. Santos, and N. Santos. Study of javascript static analysis tools for vulnerability detection in node.js packages. *IEEE Transactions on Reliability*, pages 1–16, 2023. doi: 10.1109/TR.2023.3286301.

[12] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL https://doi.org/10.1145/360248.360252.

[13] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner. Javert 2.0: Compositional symbolic execution for javascript. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290379. URL https://doi.org/10.1145/3290379.

[14] OWASP Top 10 Web Security Risks. https://owasp.org/www-project-top-ten.

[15] https://neo4j.com/developer/cypher/.

[16] https://neo4j.com.

[17] M. H. M. Bhuiyan, A. S. Parthasarathy, N. Vasilakis, M. Pradel, and C.-A. 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, 2023. doi: 10.1109/ICSE48619.2023.00096.

[18] CVE-2023-40582. https://nvd.nist.gov/vuln/detail/CVE-2023-40582.

[19] V. R. Basili, L. C. Briand, and W. L. Melo. How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39:104–116, 1996.

[20] W. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30, 1994. doi: 10.1109/52.311048.

[21] P. Mohagheghi, R. Conradi, O. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. pages 282– 291, 06 2004. ISBN 0-7695-2163-0. doi: 10.1109/ICSE.2004.1317450.

[22] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 995–1010, USA, 2019. USENIX Association. ISBN 9781939133069.

[23] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 102–112, 2017. doi: 10.1109/MSR.2017.55.

[24] A. Decan, T. Mens, and M. Claes. An empirical comparison of dependency issues in oss packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 2–12, 2017. doi: 10.1109/SANER.2017.7884604.

[25] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 385–395, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. doi: 10.1145/3106237.3106267. URL https://doi.org/10.1145/3106237.3106267.

[26] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014. doi: 10.1109/SP.2014.44.

[27] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 334–349, 2017. doi: 10.1109/EuroSP.2017.14.

[28] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. Navex: Precise and scalable exploit generation for dynamic web applications. In *Proc. of USENIX Security*, 2018.

[29] S. Khodayari and G. Pellegrino. JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals. In *Proc. of USENIX Security*, 2021.

[30] T. Brito, P. Lopes, N. Santos, and J. F. Santos. Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security*, 118:102745, jul 2022. doi: 10.1016/j.cose.2022.102745. URL https://doi.org/10.1016%2Fj.cose.2022.102745.

[31] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981. ISBN 0897911466.

[32] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 3–18, 2012. doi: 10.1109/CSF.2012.19.

[33] J. Fragoso Santos and T. Rezk. An information flow monitor-inlining compiler for securing a core of javascript. volume 428, 06 2014. ISBN 978-3-642-55414-8. doi: 10.1007/978-3-642-55415-5_23.

[34] A. Chudnov and D. A. Naumann. Inlined information flow monitoring for javascript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 629–643, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338325. doi: 10.1145/2810103.2813684. URL https://doi.org/10.1145/2810103.2813684.

[35] B. B. Nielsen, B. Hassanshahi, and F. Gauthier. Nodest: Feedback-driven static analysis of node.js applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 455–465, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3338933. URL https://doi.org/10.1145/3338906.3338933.

[36] R. Karim, F. Tip, A. Sochůrková, and K. Sen. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering*, 46(12):1364–1379, 2020. doi: 10.1109/TSE.2018.2878020.

[37] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *2010 IEEE Symposium on Security and Privacy*, pages 513–528, 2010. doi: 10.1109/SP.2010.38.

[38] B. Loring, D. Mitchell, and J. Kinder. Expose: Practical symbolic execution of standalone javascript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, page 196–199, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350778. doi: 10.1145/3092282.3092295. URL https://doi.org/10.1145/3092282.3092295.

[39] G. Li, E. Andreasen, and I. Ghosh. Symjs: Automatic symbolic testing of javascript web applications. FSE 2014, page 449–459, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635913. URL https://doi.org/10.1145/2635868.2635913.

[40] E. Wittern, A. T. Ying, Y. Zheng, J. Dolby, and J. A. Laredo. Statically checking web api requests in javascript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 244–254, 2017. doi: 10.1109/ICSE.2017.30.

[41] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel. Extracting taint specifications for javascript libraries. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 198–209, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380390. URL https://doi.org/10.1145/3377811.3380390.

[42] I. Koishybayev and A. Kapravelos. Mininode: Reducing the attack surface of node.js applications. In *International Symposium on Recent Advances in Intrusion Detection*, 2020.

[43] M. Madsen, F. Tip, and O. Lhoták. Static analysis of event-driven node.js javascript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 505–519, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336895. doi: 10.1145/2814270.2814272. URL https://doi.org/10.1145/2814270.2814272.

[44] A. Ojamaa and K. Düüna. Assessing the security of node.js platform. In *2012 International Conference for Internet Technology and Secured Transactions*, pages 348–355, 2012.

[45] C.-A. Staicu and M. Pradel. Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 361–376, Baltimore, MD, Aug. 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/staicu.

[46] Express web framework. https://www.npmjs.com/package/express.

[47] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner. Containing malicious package updates in npm with a lightweight permission system. page 1334–1346, 2021. doi: 10.1109/ICSE43902.2021.001. URL https://doi.org/10.1109/ICSE43902.2021.00121.

[48] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315326. doi: 10.1145/2335356.2335360. URL `https://doi.org/10.1145/2335356.2335360`.

[49] M. Ohm, T. Pohl, and F. Boes. 2023.

[50] Apache TinkerPop. `https://tinkerpop.apache.org/`.

[51] Apache TinkerPop: Gremlin. `https://tinkerpop.apache.org/gremlin.html`.

[52] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812, 2015. doi: 10.1109/SP.2015.54.

[53] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 35–46, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912691. doi: 10.1145/53990.53994. URL `https://doi.org/10.1145/53990.53994`.

[54] T. Chen and Y. Cheung. Structural properties of post-dominator trees. In *Proceedings of Australian Software Engineering Conference ASWEC 97*, pages 158–165, 1997. doi: 10.1109/ASWEC.1997.623767.

[55] NetworkX. `https://networkx.org/`.

[56] CodeQL. `https://github.com/github/codeql`.

[57] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. N. Venkatakrishnan, and Y. Cao. Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1059–1076, Los Alamitos, CA, USA, may 2023. IEEE Computer Society. doi: 10.1109/SP46215.2023.10179352. URL `https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.10179352`.

[58] D. Cassel, W. T. Wong, and L. Jia. Nodemedic: End-to-end analysis of node.js vulnerabilities with provenance graphs. In *Proceedings of the IEEE European Symposium on Security and Privacy (Euro S&P)*, July 2023. doi: 10.1109/EuroSP57164.2023.00068. URL `https://www.andrew.cmu.edu/user/liminjia/research/papers/nodemedic-eurosp23.pdf`.

[59] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer. Foundations of attack–defense trees. In P. Degano, S. Etalle, and J. Guttman, editors, *Formal Aspects of Security and Trust*, pages 80–95, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19751-2.

[60] `http://coffeescript.org/`.

[61] Microsoft. TypeScript language specification Version 1.8. Technical report, Microsoft, 2016.

[62] Common Weakness Enumeration. https://cwe.mitre.org/.

[63] Advisory 315. https://www.npmjs.com/advisories/315.

[64] CWE to OWASP Top 10 (2021) mapping. https://cwe.mitre.org/data/definitions/1344.html.

[65] L. Gong. *Dynamic Analysis for JavaScript Code*. PhD thesis, University of California, Berkeley, 2018.

[66] Beyond Security. https://beyondsecurity.com/solutions/besource.html.

[67] Checkmarx SAST. https://www.checkmarx.com/products/static-application-security-testing.

[68] Fortify SAST. https://www.microfocus.com/en-us/products/static-code-analysis-sast.

[69] Veracode SAST. https://www.veracode.com/products/binary-static-analysis-sast.

[70] Kiuwan SAST. https://www.kiuwan.com/code-security-sast.

[71] CodeSonar. https://www.grammatech.com/products/source-code-analysis.

[72] Thunderscan. https://www.defensecode.com/thunderscan-sast.

[73] WhiteHat SAST. https://www.whitehatsec.com/platform/static-application-security-testing.

[74] JsPrime. https://github.com/dpnishant/jsprime.

[75] Codeburner. https://github.com/groupon/codeburner.

[76] SonarQube. https://github.com/SonarSource/sonarqube.

[77] CodeWarrior. https://github.com/CoolerVoid/codewarrior.

[78] WALA. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[79] PMD. https://github.com/pmd/pmd.

[80] Aether. http://aetherjs.com.

[81] Coala. https://github.com/coala/coala.

[82] JsHint. https://github.com/jshint/jshint.

[83] ESComplex. https://github.com/jared-stilwell/escomplex.

[84] Coverty Scan. https://scan.coverity.com.

[85] DeepScan. https://deepscan.io.

[86] AppInspector. https://github.com/microsoft/ApplicationInspector.

[87] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *Proc. of ISAS*, 2009.

[88] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu. Safe: Formal specification and implementation of a scalable analysis framework for ecmascript. In *Proc. of FOOL*, 2012.

[89] ESLint Security Plugin. https://github.com/nodesecurity/eslint-plugin-security.

[90] Mozilla ScanJS. https://github.com/mozilla/scanjs.

[91] SemGrep. https://semgrep.dev.

[92] Joern. https://github.com/joernio/joern.

[93] NodeJsScan. https://github.com/ajinabraham/njsscan.

[94] ESSC. https://github.com/Greenwolf/eslint-security-scanner-configs.

[95] Graudit. https://github.com/wireghoul/graudit.

[96] InsiderSec. https://github.com/insidersec/insider.

[97] MS DevSkim. https://github.com/microsoft/DevSkim.

[98] Mosca. https://github.com/CoolerVoid/Mosca.

[99] Drek. https://github.com/chrisallenlane/drek.

[100] OWASP Scanning Tools. https://owasp.org/www-community/Vulnerability_Scanning_Tools.

[101] OWASP App Security Tools. https://owasp.org/www-community/Free_for_Open_Source_Application_Security_Tools.

[102] https://github.com/dseguy/awesome-static-analysis.

[103] https://github.com/creinartz/awesome-static-analysis.

[104] https://github.com/codefactor-io/awesome-static-analysis.

[105] https://www.softwaresecured.com/top-sast-tools-for-developers/.

[106] https://medium.com/@manjula.aw/nodejs-security-tools-de0d0c937ec0.

[107] https://www.softwaretestinghelp.com/tools/top-40-static-code-analysis-tools/.

[108] DVNA. https://github.com/appsecco/dvna.

[109] Npm package @stoqey/gnuplot v0.0.3. https://www.npmjs.com/package/@stoqey/gnuplot/v/0.0.3.

[110] CVE-2021-29369. https://nvd.nist.gov/vuln/detail/CVE-2021-29369.

[111] Npm package set-value. https://www.npmjs.com/package/set-value/v/3.0.0.

[112] CVE-2021-23440. https://nvd.nist.gov/vuln/detail/CVE-2021-23440.

[113] Snyk. https://snyk.io/.

[114] Huntr.dev. https://huntr.dev/.

[115] Npm package for-own. https://www.npmjs.com/package/for-own.

[116] Npm package for-in. https://www.npmjs.com/package/for-in/.

[117] Npm package mosc v1.0.0. https://www.npmjs.com/package/mosc/v/1.0.0.

[118] Npm package merge v1.2.0. https://www.npmjs.com/package/merge/v/1.2.0.

[119] Npm package find-exec v0.0.3. https://www.npmjs.com/package/find-exec/v/1.0.3.

[120] MongoDB. https://www.mongodb.com/.

[121] React.js. https://react.dev/.

[122] GDPR - General Data Protection Regulation. https://gdpr-info.eu/.