Detecting Multi-file Vulnerabilities Using Code Property Graphs

PIC2 - Master in Computer Science and Engineering
Instituto Superior Técnico, Universidade de Lisboa

Guilherme Figueira da Silva Gonçalves — 95585*
guilherme.silva.goncalves@tecnico.ulisboa.pt

Advisors: José Santos & Pedro Adão

Abstract The World Wide Web, initially conceived for facilitating information sharing among CERN researchers, has evolved into a vital force driving social transformation and economic impact in our daily lives. JavaScript, a key programming language in web development, allows for dynamic and interactive content in browsers. Node.js further revolutionizes web development by extending JavaScript's influence across the entire development stack, facilitating the creation of scalable websites. However, Node.js faces security challenges due to JavaScript's language-specific behavior, leading to vulnerabilities often overlooked by developers. Additionally, the Node Package Manager (NPM) introduces risks, as its vast repository of community-managed packages may contain vulnerabilities. To address these issues, static analysis tools employing graph-based approaches, such as Graph.js and ODGen, have proven effective. This work focuses on enhancing Graph.js, acknowledging its limitations in modular reasoning, particularly regarding the handling of external modules. We propose two strategies, Complex Graph with Simple Queries and Simple Graph with Complex Queries, aiming to improve Graph.js's accuracy by reducing false positives. The upgraded Graph.js version will be evaluated on SecBench and Vulcan datasets, comparing results with competitors ODGen and CodeQL. The anticipated outcome is an improved Graph.js, offering better support for modular reasoning and enhanced reliability in static analysis for Node.js applications. The goal is to strengthen Graph.js, making it a more reliable tool for static analysis in Node.js applications that can be integrated in the CI/CD pipelines. The document provides background information in Section 2, outlines related work in Section 3, details the proposed solution in Section 4, presents the evaluation and planning methods in Section 5, and concludes with a summary and remarks in Section 6.

Keywords — Static Analysis, Node.js, Modularity, Vulnerability Detection, Code Property Graph

*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 (https://nape.tecnico.ulisboa.pt/en/apoio-ao-estudante/documentos-importantes/regulamentos-da-universidade-de-lisboa/).

Contents

1 Introduction

In our day-to-day lives, the web plays a crucial role, evident in activities such as online communication through platforms like email and social media, accessing information via search engines, and participating in the digital economy through e-commerce platforms.

JavaScript stands out as a fundamental element in web development, being one of the most widely used programming languages for executing code in web browsers. Its adaptability is crucial in creating dynamic and interactive web content, thereby enhancing user experiences across the World Wide Web.

Node.js [1] has revolutionized web development by allowing the use of JavaScript across the entire development spectrum. This extends the influence of JavaScript beyond the browser, empowering the creation of scalable websites. Node.js offers an event-driven architecture and a non-blocking I/O model, contributing to the efficiency and performance of web applications.

In spite of its advantages, Node.js struggles with its security. The language-specific behavior and dynamic properties of JavaScript, including prototype-based inheritance, often lead less experienced developers to inadvertently introduce security vulnerabilities into their code. Developers can easily miss subtle security relevant issues, such as improper input validation or unhandled exceptions, because they are hard to detect manually. Instances like school attacks [2], where exploited vulnerabilities in web applications pose substantial threats to essential systems and public safety, illustrate the severity of these concerns. For that reason, there is an urge to automatically detect and mitigate vulnearbilities in Node.js.

In addition to the challenges outlined in the previous paragraph, the platform's default package manager, Node Package Manager (NPM) [3], also introduces vulnerabilities in Node.js applications. NPM hosts a repository with milions of packages [4] that are community-managed and come with their unique dependencies. Within this repository, developers are responsible for managing vulnerabilities in their respective packages. Consequently, many NPM packages are known to have vulnerabilities. For that reason, choosing a package becomes a daunting task when aiming to develop secure code, as even well-intentioned developers may unintentionally introduce packages with vulnerabilities. These vulnerable packages can serve as potential entry points for exploits, underscoring the challenges in ensuring the security of Node.js applications.

To address the challenges associated with manual vulnerability detection, static analysis emerges as a crucial strategy for automatically identifying and mitigating vulnerabilities. Although there are many approaches to statically analyse JavaScript code, graph-based approaches, employed by tools such as Graph.js [5] and ODGen [6], have proved to be effective at detecting a variety of vulnerabilities. These tools construct a Code Property Graph (CPG) that represents the program and execute queries on it to detect vulnerabilities. Graph.js exhibits fewer false positives and is more efficient than the second best tool, ODGen, in two benchmarks: SechBench [7] and Vulcan [8]. For that reason, we believe, to the best of our knowledge, that Graph.js leads the field.

Despite its good results, Graph.js has some limitations. The tool struggles to address usage of modules in code effectively. Graph.js categorizes the return values of function calls as unsafe, regardless of the attacker's control over them. This strategy leads to an increased number of false positives, reducing the tool's overall effectiveness. To this end, we will implement two different strategies for modular analysis of taint flows in Node.js applications and their respective queries. We focus on injection vulnerabilities, given their significant appearance in Node.js applications. More concretely, our work makes the following contributions:

1. ***Complex Graph with Simple Queries Strategy:*** This strategy involves consolidating the graphs from different modules into a unified graph and establishing connections between the nodes in both graphs. To this end, we propose adding new nodes and edges to the graphs in order to ensure that graph models the source code's module features correctly.

2. ***Simple Graph with Complex Queries Strategy:*** This strategy involves consolidating graphs from various modules into a unified representation of the entire application. Similar to the previous approach, we propose the addition of new nodes and edges to the graphs to accurately capture the module features of the source code. The proposed new nodes remain consistent across both strategies, while the new edges differ. Unlike the previous strategy, we maintain the separation of the graphs, delegating the responsibility of establishing connections between them to the queries. Furthermore, function call nodes now incorporate information about the specific function in the external module that could be called. With this added information, function call nodes facilitate the queries' job.

3. ***Queries for Vulnerability Detection in both strategies:*** This involves developing the new set of queries capable of identifying vulnerabilities represented by the updated graphs. For the Complex Graph with Simple Queries strategy, minimal adjustments to existing Graph.js queries suffice. Conversely, implementing the Simple Graph with Complex Queries strategy requires the creation of two entirely new queries.

To evaluate our new version of Graph.js, we will run it on SecBench and Vulcan datasets, comparing its results with our main competitors, ODGen and CodeQL [9]. We expect to improve Graph.js by reducing the number of false positives.

The main contribution of this work is an upgraded version of Graph.js with improved support for modular reasoning. This new version addresses the limitations of Graph.js, by reducing false positives and enhancing accuracy. The goal is to strengthen Graph.js, making it a more reliable tool for static analysis in Node.js applications that can be integrated in the CI/CD pipelines.

This document is organized as follows. Section 2 provides the necessary background for this work. More concretely, it overviews the Node.js security model, module usage in Node.js and JavaScript and introduces Graph.js, the focus of this work. Section 3 overviews the most important reasearch efforts that are related to this work. Section 4 describes the proposed solution for the problem. Section 5 addresses the evaluation methods necessary and how the work will be carried out throughout the semester. Finally, section 6 brings the document to a close, offering a recap of this work and some conclusion remarks.

2 Background

This section covers the basics of module usage in JavaScript and Node.js (Section 2.1), the Node.js security model (Section 2.2), and introduces the Graph.js (Section 2.3). Understanding Graph.js is key here, since our work aims to improve upon it.

2.1 Modularity in JavaScript and Node.js

In this subsection, we define modularity as the practice of segmenting one's application into modules. These modules can exist in one or multiple source files. A module is essentially anything that encapsulates units of code, contributing to the overall organization and structure of a program. The benefits of having organized and well-structured code are improved readability, reusability and abstraction. Listings 1 and 2 showcase the usage of modules to organize an application. The only difference between the listings lies in how modules are handled by both JavaScript and Node.js. The key differences will be addressed in the following subsections

2.1.1 Modularity in JavaScript

In the evolution of JavaScript, the concept of module usage changed from pre-ECMAScript 6 (ES5) to the post-ECMAScript 6 era. Prior to ES6, creating modules relied on conventions, patterns, and third-party libraries. With the introduction of ES6, JavaScript offered native support for module usage through the `import` and `export` keywords. These keywords standardized and simplified the process of defining and using modules within JavaScript code. In Listing 1, we illustrate how `import` and `export` work, by showcasing the exportation of a constant string (the string `greeting`) and a function (the `sayHello` function). In this example, Module 1 (lines 1 and 2) demonstrates the usage of the export keyword, while Module 2 (line 2) illustrates the usage of the import keyword.

2.1.2 Modularity in Node.js

In the context of Node.js, the creation of modules has been facilitated through the CommonJS module system. This system employs the `require` function for importing modules and the `module.exports` object for exporting modules. Listing 2 illustrates the same modules from Listing 1, now adapted to the CommonJS system in Node.js. This example demonstrates how modules are imported using the `require` function (line 2 of Module 2) and exported using the `module.exports` object (line 2 of Module 1) in Node.js.

```js
// module1.js
export const greeting = "Hello";
export function sayHello(name) {
    return `${greeting}, ${name}!`;
}
```

```js
// module2.js
import { greeting, sayHello } from
↪ './module1';
console.log(greeting); // Output: Hello
const message = sayHello("Alice");
console.log(message); // Output: Hello,
↪ Alice!
```

Listing 1: Modularity in ECMAScript 6 example

Additionally, Node.js version 12 added support for the ES6-style modules. It is important to note that, despite this support for ES6-style modules, the underlying module loader behavior differs based on the syntax used. Specifically, invoking `require` function always utilizes the CommonJS module loader. On the other hand, using the `import` keyword relies on the ECMAScript module loader.

```js
// module1.js
module.exports = {
    greeting : "Hello",
    sayHello: function(name) {
    return `${greeting}, ${name}!`;
}
}
```

```js
// module2.js
const module1 = require('./module1.js')
console.log(module1.greeting); // Output:
↪ Hello
const message = module1.sayHello("Alice");
console.log(message); // Output: Hello,
↪ Alice!
```

Listing 2: Modularity in Node.js

Node Package manager: Besides user-defined modules, Node.js offers the developer the ability to integrate third-party modules as packages into their projects, through its default package manager, Node Package Manager (NPM) [3]. NPM serves as a central hub for sharing and obtaining packages in the Node.js ecosystem, facilitating the distribution of reusable code components.

2.2 Node.js Security Model

Despite module usage offering numerous benefits, it also introduces vulnerabilities in Node.js applications. The client-side of a Node.js application operates in a sandboxed environment and with limited privileges. On the other hand, the server-side does not run in a sandboxed environment and often operates with elevated privileges. For that reason, vulnerabilities exploited in the server-side of a Node.js application can compromise the whole machine. In this subsection, we introduce the Node.js security model, offering an overview of the common vulnerabilities found in Node.js applications.

2.2.1 Taint-Style Vulnerabilities

Taint-style vulnerabilities are type of vulnerabilities that often appear in Node.js applications. These vulnerabilities involve data flowing from untrusted sources to sensitive sinks. Sources and sinks act like delimiters to taint-style vulnerabilities, showing where unsafe data flows start and end.

- ***Sources:*** Sources refer to locations in the application where untrusted values enter the system. Sources typically include the program entry points, such as web forms, query parameters, request bodies, files, databases, and more. In the context of a module, we consider its parameters as sources if the function is exported by the module and that module can be imported by unsafe code. In Listing 3, the source is the argument `b`
- ***Sinks:*** Sinks are a function calls that trigger security-sensitive behavior. Data handled by sinks can influence or modify a program's behavior, making it crucial to validate and sanitize data flows from a source to a sink. In Listing 3, the sink is the call to the `eval` function.

Exploring a type of taint-style vulnerabilities in detail, we now focus on injection vulnerabilities, which are common in Node.js applications and the focus of this work.

- ***Code Injection:*** Code injection vulnerabilities occur when an attacker-controlled string is passed to a runtime evaluation API without proper sanitization. Notable sinks: `eval` and `Function`.
- ***OS Command Injection:*** OS command injection vulnerabilities occur when an attacker is able to directly impact the commands executed by the operating system. Notable sinks: `child_process.exec`, `child_process.spawn`, and `child_process.execFile`.
- ***SQL Injection:*** SQL Injection vulnerabilities occur when attacker is able to manipulate a web application's SQL query by injecting malicious SQL code. Notable sinks: `mysql.connection.query`.
- ***Path Traversal:*** Path Traversal vulnerabilities occur when an attacker is able to access files or directories outside the application's scope. Notable sinks: `fs.readFile` and `fs.createReadStream`.

To illustrate injection vulnerabilities, Listing 3 offers an example of code injection. In this case, the attacker-controlled variable `b` (source) reaches the `eval` call (sink) without proper sanitization. This lack of sanitization creates a vulnerability, allowing the attacker to potentially execute arbitrary code.

For instance, if the attacker provides the string `"process.exit(0)"` as input, the program could be terminated, because this input constitutes valid JavaScript code. For that reason, the subsequent `eval` call executes it, terminating the program. Another analogous scenario involves providing the string `"require('child_process').spawn('cat /etc/passwd')"` as input. This input executes the command `"cat /etc/passwd"`, which, in Unix systems, reads the file `/etc/passwd`. For that reason, this input enables the attacker to read the contents of the file containing user account and password information. Consequently, this data can be extracted and potentially exploited to gain unauthorized access to the compromised system.

Both exploits underscore the substantial risks linked to code injection vulnerabilities. For that reason, addressing these vulnerabilities is crucial for ensuring the integrity and safety of the application. This emphasizes the importance of implementing robust validation and sanitization mechanisms.

```
1  module.exports = function(b) {
2      if(b){
3          eval(b);
4      }
5  }
```

Listing 3: Code injection Example

2.2.2 Other types of vulnerabilities

Beyond the previously mentioned vulnerabilities, Node.js applications may exhibit additional types of vulnerabilities. Below, we list four other vulnerability types that will not be addressed in this work.

- ***Prototype Pollution:*** Prototype Pollution vulnerabilities occur when an attacker is able to update a built-in JavaScript property through the object's prototype chain. This update alters the behavior of all JavaScript object of the corresponding type.
- ***Cross-Site Request Forgery (CSRF):*** CSRF vulnerabilities occur when an authenticated end user unintentionally executes unwanted actions on a web application. With the aid of social engineering, an attacker may deceive the victim into carrying out actions chosen by the attacker. This could include state-changing operations like transferring funds or altering their email, for regular users. For users with administrative permissions, it can lead to compromising the entire web application.
- ***Denial of Service (Dos):*** A Denial of Service vulnerabilities occur when an attacker is capable of temporarily or permanently disrupting a website or service, rendering it unavailable to users.

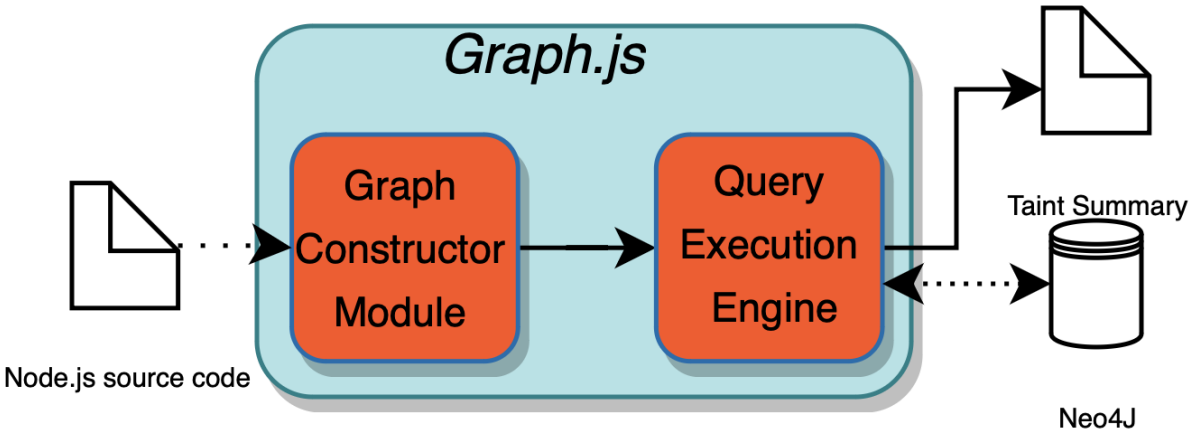

Figure 1: Graph.js Architecture Overview

- ***Regular expression denial of service (ReDoS):*** ReDoS vulnerabilities occurs when a malicious input string causes a regular expression to execute slowly or stall, potentially resulting in a denial of service. These vulnerabilities are a type of Denial of Sevice vulnerabilities
- ***Server-side request forgery (SSRF):*** Server-side request forgery (SSRF) vulnerabilities occur when an attacker is able to manipulate a web application into making unintended requests to internal or external resources. This can potentially expose sensitive information or enable attacks on other systems.

2.3 Graph.js

In this section we present *Graph.js* [5] [10]. Graph.js is a novel tool designed statically analyse Node.js applications using Code Property Graphs (CPGs). We selected Graph.js because it is the leading static analysis tool for vulnerability detection in Node.js applications.

An illustration of its architecture can be found in Figure 1. The two modules that compose Graph.js are as follows:

- ***Graph Constructor Module:*** This module takes a Node.js application as input and models the program in a Multiversion Dependency Graph (MDG), a novel type of CPG introduced by the authors of Graph.js. Then, the graph is forwarded to the Query Execution Engine to identify the vulnerabilities modeled by it.
- ***Query Execution Engine:*** This module imports the previously generated Multiversion Dependency Graph (MDG) into a *Neo4j* [11] database and executes queries to detect vulnerabilities. The results of this process are captured in a file named *taint_summary.json*, offering information on identified vulnerabilities, including their locations within the source files.

Graph.js is designed to identify injection vulnerabilities and prototype pollution vulnerabilities (CWE 1321 [12]). In the particular case of injection vulnerabilities, it is designed to detect the following types:

- ***OS command injection*** (CWE-78 [13])
- ***Abitrary Code Execution*** (CWE-94 [14])
- ***Path Traversal*** (CWE-22 [15])

Before diving into the specifics of each module that constitute Graph.js, we first introduce the running example. This example will support our discussion by demonstrating the MDGs generated by Graph.js and how the available queries work on them.

```
1  const bar = require('./bar.js');
2  function f(x,y){
3     if(x > 0){
4          var a = bar.f("foo",y);
5          eval(a);
6     }
7     else{
8           var b = bar.g(x,0);
9           eval(b);
10    }
11 }
12 module.exports = f;
```

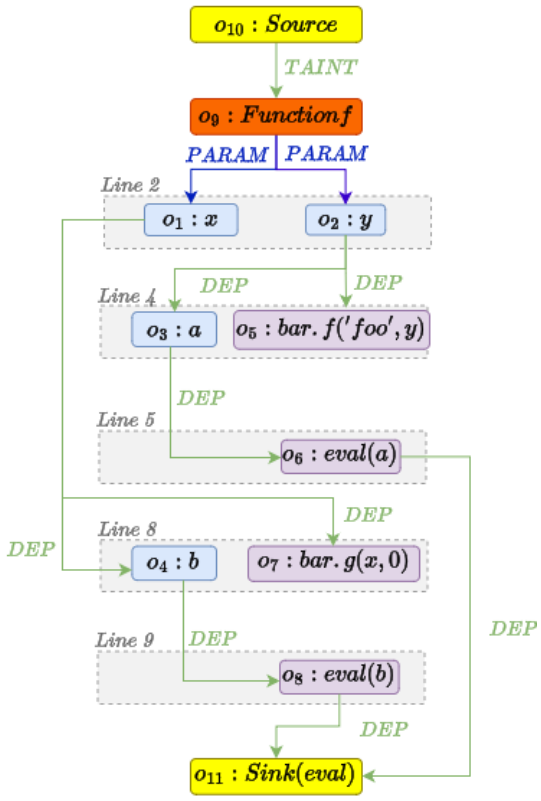

Figure 2: Main module (left side) and correspoding MDG (right side)

2.3.1 Running Example

Figures 2 and 3 illustrate the example that will used throughout this report. The example consists of two modules: the `Main` module and the `Bar` module. Initially, the `Main` module includes the `Bar` module and subsequently calls either function `f` or function `g` from the `Bar` module, depending on whether `x` is greater than `0`. In the call to `f`, the Main code provides the constant string `"foo"` and the variable `y`. Turning to the call to `g`, it supplies the variable `x` and the numeric value `0`. In both conditional branches, the `Main` module also invokes `eval` with the return value of the corresponding function from the `Bar` module.

As for the the *Bar* module, it declares the previously mentioned functions. Both functions start by dynamically evaluating the variable `a` through the use of the `eval` function. Then, function `f` returns its `b` argument, while function `g` returns the number `0`.

2.3.2 Graph Constructor Module

The *Graph Constructor Module* is responsible for converting the source code into a graph structure called MDG. The MDG integrates details about a program's structure with information regarding the dependencies between the objects it manipulates. Furthermore, it also stores information on how the objects evolve throughout the program's execution. This represents a distinctive innovation introduced by Graph.js when compared to previous graph-based approaches.

MDG Nodes: Exploring the structure of Multiversion Dependency Graphs (MDGs) using the provided running example, we find nodes of the following types:

- ***Tainted Source Nodes:*** Tainted Source nodes represent any data within the application whose safety cannot be assured. This data may originate from various parts of the program, with user input being the

```
1  const foo = 4;
2  function f(a,b){
3      eval(a);
4      return b;
5  }
6  function g(a,b){
7      eval(a);
8      return 0;
9  }
10 module.exports = {f, g};
```

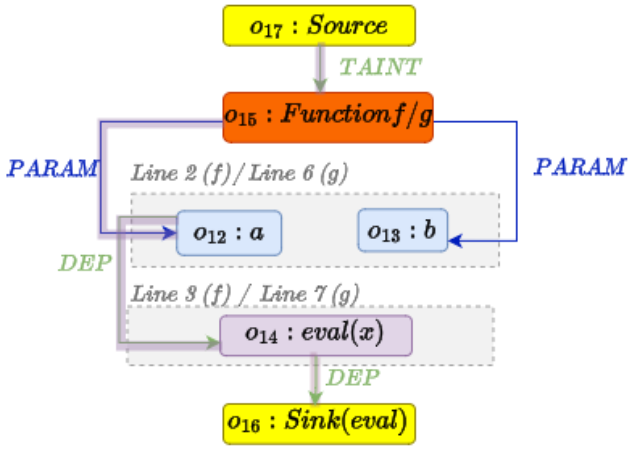

Figure 3: Bar module (left side) and correspoding MDG (right side)

most frequent source. This node type is demontrated by o_{10} and 0_{17}.

- ***Unsafe Sinks Nodes:*** Unsafe Sink nodes represent calls to risky functions and/or APIs. This node type is demonstrated by o_{11} and o_{16}.
- ***Value Nodes:*** Value Nodes represent objects and primitive values generated during the program's execution. In both examples, these nodes correspond to the variables used by the program. More concretely, o_1 to o_4 in Figure 2 and o_{12} to o_{13} in Figure 3, demonstrate these nodes.
- ***Call Nodes***: Call nodes represent the calling of functions within the program. This node type is demonstrated by o_5 and o_7 in Figure 2 and o_{14} in Figure 3.
- ***Function Nodes:*** Function nodes represent the functions declared throughout the program. This node type is demonstrated by o_9 and o_{15}.

MDG Edges: Although the nodes are important, the crucial information is encoded in the edges, since they capture the relationships between the objects. The edges are as follows:

- ***Property Edges:*** Property edges represent an object's structure. The edge $n_1 \xrightarrow{prop(p)} n_2$ indicates that the object represented by node n_1 has a property named p, whose value is represented by n_2.
- ***New Version:*** Whenever an object represented by node n_1 is modified, a new value node is generated to represent that object with the updated property. The edge $n_1 \xrightarrow{NV(p)} n_2$ signifies that n_2 is the new version of the object n_1, resulting from an update to the property p.
- ***Dependency Edges:*** Dependency edges illustrate relationships involving data between variables, objects, sources, and sinks. The edge $n_1 \xrightarrow{DEP} n_2$ indicates that the value represented by n_2 is computed using the value represented by n_1. In Figures 2 and 3, these edges are visually distinguished in green. For instance, in Figure 2, the edge $o_4 \xrightarrow{DEP} o_8$ signifies that the call to `eval` (o_8) depends on the variable `b` (o_4).
- ***Parameter Edges:*** Parameter Edges connect a function node to the nodes representing its parameters. An edge $n1 \xrightarrow{PARAM} n2$ signifies that the function represented by n_1 has a parameter represented by n_2. In Figures 2 and 3, these edges are visually distinguished in blue. For instance, in Figure 2, the edges $o_9 \xrightarrow{PARAM} o_1$ and $o_9 \xrightarrow{PARAM} o_2$ signify that the function represented by o_9 (function `f`) has two parameters, o_1 (`x`) and o_2 (`y`).

Within the context of this work, our primary emphasis is on the *dependency edges* (DEP) and *parameter edges* (PARAM), as the modular constructs have more direct impact on the dependecy part of the analysis. The dynamics of module usage significantly influence the determination of dependencies. As a result, our attention is directed towards understanding and thoroughly examining this dimension of the graph construction.

2.3.3 Query Execution Engine

Leveraging all the information encoded within the MDG, the *Query Execution Engine* can identify vulnerable paths by executing a series of queries. In the following paragraphs, we focus on explaining only the injection vulnerability queries since that's the vulnerability type addressed by this work.

In order to identify injection vulnerabilities, the queries look for a paths from a tainted source to a sensitive sink that go through DEP and PARAM edges. In the running example, the injection present in the `Bar` module (line 3), can be detected by finding the path:

$$o_{17} \xrightarrow{TAINT} o_{15} \xrightarrow{PARAM} o_{12} \xrightarrow{DEP} o_{14} \xrightarrow{DEP} o_{16}$$

This path is highlighted in the figure in light purple.

2.3.4 Limitations

Although Graph.js can accurately detect injection vulnerabilities, it also has some limitations. Graph.js goes through each module separately in its analysis. This approach leads to false positives because it treats all function parameters as potentially unsafe. For instance, consider the call to `Bar.f` (line 4). This call has the string `"foo"` as its first argument. Later, in the `Bar` module (line 3), that same string is used as the input for the call to `eval`. This call, therefore, corresponds to `eval("foo")`, which is actually safe. However, since Graph.js analyses each function separately and there is a taint path connecting o_{17} to o_{16}, Graph.js will report the `eval` call as potentially vulnerable, which is a false positive vulnerability report.

Another issue regarding the analysis of Graph.js is that it does not model information regarding external modules. Hence, when dealing with those calls, the analysis has the following options:

- ***Consider return values always tainted:*** This strategy introduces false positives by design. Following this approach, the return value of the call to `Bar.g` (line 8) is marked as tainted. Consequently, a vulnerability is flagged in line 9. However, it is important to note that this identified vulnerability is a false positive, as the tainted value does not pose an actual security risk or vulnerability in the context of the program's intended functionality.
- ***Consider return values always untainted:*** This strategy introduces false negatives by design. Following this approach, the return value of the call to `Bar.f` (line 4) is labeled as untainted. Consequently, a vulnerability in line 5 goes undetected. However, it is important to note that this unidentified vulnerability is a false negative, as the untainted value may indeed pose an actual security risk or vulnerability within the intended functionality of the program.

In the specific context of Graph.js, a function's return value is deemed tainted if any of its arguments are tainted, representing a combination of the approaches mentioned earlier. However, this approach is susceptible to false positives, as the return value might not necessarily depend on its arguments. For instance, Graph.js identifies the return value of `Bar.g` (line 8) as tainted, based on the fact that the attacker-controlled `x` variable is passed into that function call. This particular scenario results in a false positive, as elaborated above.

3 Related Work

Scientific research has sought techniques to enhance the security of Node.js applications. In this section, we focus on Node.js security (Section 3.1) and vulnerability detection tools for Node.js applications (Section 3.2).

3.1 Node.js Security

In Section 2.2, we highlighted that Node.js struggles with its security and that NPM further aggravates those struggles. In this subsection, we overview two tools for managing third-party package inclusion (Section 3.1.1) and datasets that can be used to evaluate Node.js vulnerability detection tools (Section 3.1.2). On one hand, controlling third-party package inclusion enhances Node.js security by aiming to ensure the use of only secure packages. In the case of insecure packages, this control seeks to ensure that only secure inputs reach these packages. On the other hand, datasets aid in comparing and evaluating tools, enabling developers to choose the best tools for evaluating their application.

3.1.1 Managing Third-Party Package Inclusion

Developers often underestimate the security impact of introducing npm packages in their application. These packages may introduce some of the vulnerabilities explained in Section 2.2. Here, we introduce two tools, *Mininode* and *Synode*, designed to address the inclusion of packages in Node.js applications. These tools rely on two techniques: reduce the attack surface by reducing the application's functionalities to the minimum necessary and enforce security policies at runtime to ensure the safe usage of the modules.

Mininode: Node.js applications heavily depend on incorporating third-party libraries. To address this dependency, I. Igibek *et al.* [16] conducted a study to explore how the extensive integration of third-party libraries could contribute to the attack surface of Node.js applications. The study revealed that, on average, only 6.8% of the code in analyzed applications was original and 11.3% relied on potentially vulnerable third-party packages. To address and mitigate the risks of a vast attack surface, the authors proposed *Mininode*. Minode aims at reducing the attack surface by reducing the application's functionalities to the minimum necessary for its intended purpose, thereby mitigating vulnerabilities introduced by unused packages.

Mininode takes a Node.js application as input and initiates the analysis with the generation of its Abstract Syntax Tree (AST). Subsequently, it constructs a file-level dependency graph by resorting to all available information regarding exports and calls to `require` in the AST. At this stage, AST nodes are marked as either used or unused, distinguishing between those considered essential and those deemed unnecessary and eligible for removal. In the final step, nodes identified as unused are pruned from the AST, and the updated AST is then employed to generate the corresponding code for each module.

Its evaluation demonstrated that Mininode removed vulnerabilities across all categories in 13.8% of cases and succeeded in completely eliminating all vulnerabilities in 13.65% of cases.

Synode: Similar to the work of I. Igibek *et al.*, C. Staicu *et al.* [17] also assessed the landscape of utilized APIs. However, C. Staicu *et al.* focused on the susceptibility of APIs to injection vulnerabilities. They conducted an extensive analysis of 235,850 NPM packages with the aim of understanding the vulnerability of these packages. Their findings shwocased that 15,604 modules employed APIs vulnerable to injection. Furthermore, the research found that patches to those vulnerabilities take a long time to be developed and sometimes are insufficient, predominantly relying on regular expression sanitization that fails to cover all possible dangerous inputs.

Given the widespread utilization of these modules in applications, the authors introduced Synode. Synode employs a combination of static analysis and runtime enforcement of security policies to identify potential injection vulnerabilities and ensure secure usage of vulnerable modules. Using static analysis, the tool derives user input templates representing possible input values. These templates enable the tool to assess whether an injection API call site is secure or requires runtime checks to block malicious inputs. If the template cannot be statically defined, dynamic checks are implemented to prevent potentially harmful inputs from reaching vulnerable APIs. The authors' analysis demonstrated that their approach is efficient, incuring sub-millisecond runtime overhead, and provides robust protection against attacks on vulnerable modules with minimal false positives.

The two tools mentioned in the previous paragraphs focus on protecting the modules used in an application by removing them. This approach differs from that used by Graph.js since it only identifies vulnerabilities, leaving the task of removing them to the developer.

3.1.2 Benchmarks and Empirical Studies

To facilitate a fair comparison between two distinct static analysis tools, it is imperative to leverage datasets containing known vulnerabilities. Here, we introduce two datasets that can serve as ground truth for evaluating static analysis tools.

VulcaN: T. Brito *et al.* [8], performed an assessment of fully automated JavaScript static analysis tools capable of seamless integration into the CI/CD pipeline. Their focus was on the examination of server-side JavaScript, particularly NPM packages.

Prior to assessing the tools, the authors built an annotated dataset of real-world vulnerabilities, which was nonexistent at the time of publication. To construct this dataset, the authors gathered a snapshot of NPM advisories until the end of June 2021. From the packages included in the snapshot, they excluded those marked as malicious, those lacking source code, and those that were not in plain JavaScript (i.e, used TypeScript [18]). The authors were able to manually verify 957 packages by the time of publication, thus these are the packages included in the dataset. Examples of the vulnerability types present in the dataset are: Path Traversal (CWE-22 [15]), OS Command Injection (CWE-78 [13]), and Code Injection (CWE-94 [14]).

The assessed tools had to meet the following requisites: rely only on the package's source code, being open-source, having a command-line interface and having a security oriented approach. In the end, they were left with 9 tools, which included *CodeQL* [9] and *ODGen* [6].

The evaluation of the selected tools exposed a trade-off between the true positive rate and precision. Specifically, tools that struck a better balance between true positives and precision were those employing graph-based techniques, namely ODGen and CodeQL. These tools were capable of detecting 31.3% and 16.1% of vulnerabilities, respectively. Furthermore, the combination of the best tools, with CodeQL among them, only identified 53.1% of vulnerabilities in the dataset. According to the authors, the undetected vulnerabilities may stem from challenges in handling the dynamic nature of JavaScript and due to incomplete sink sets.

SecBench: M. Pradel *et al.* [7] also constructed dataset of real-world vulnerabilities. The main contribution from this dataset when compared to the Vulcan dataset is that this dataset is executable. In other words, the dataset includes inputs that allows to trigger the vulnerabilities.

This dataset selected vulnerable packages from diverse sources, including Snyk, Github Advisories, and Hunter.dev. Particularly, it focused on specific vulnerability types, namely Prototype Pollution (CWE-1321 [12]), ReDoS (CWE-1333 [19]), Code Injection (CWE-94 [14]), OS Command Injection (CWE-78 [13]) and Path Traversal (CWE-22 [15]). They prioritized packages that could be successfully installed and that had vulnerabilities that could be replicated, while excluding those causing compatibility issues with the authors' setup or marked as unstable. To establish fixed versions of the packages, the authors derived either from a commit directly addressing the vulnerability listed in the advisory or through a detailed analysis of a failed exploit in a newer version. Their work resulted in a dataset with 600 vulnerable packages.

In comparison to VulcaN, SecBench.js has the benefit of including exploit annotations for all its vulnerabilities. However, it falls short in addressing some common and impactful vulnerability types in Node.js applications, such as Cross-Site Scripting, which are present in VulcaN.

3.2 Vulnerability Detection in Node.js applications

This subsection will introduce four tools: *ODGen*, *FAST*, *Nodest* and *CodeQL*. Similarly to Graph.js, these tools aim to detect vulnerabilities using static analysis methods but employ distinct approaches in their detection strategies.

ODGen: Prior efforts in the realms of C/C++ and PHP have introduced static analysis techniques that use graph query approaches to model program information and detect vulnerabilities. However, when applied to JavaScript, these approaches fall short in capturing essential elements, such as the object's prototype chain. This translates into some vulnerabilities being missed by automatic analysis methods.

S. Li *et al.* [6] addressed this gap by introducing a novel graph structure called the *Object Dependency Graph* (ODG). The ODG captures relationships between objects by representing them as nodes in a graph and their

interactions as edges. To preserve object lookups and definitions, the ODG integrates the Abstract Syntax Tree (AST) of the code within the graph. ODG employs a two-phased analysis, facilitating offline graph queries aimed at detecting a diverse range of vulnerabilities, such as Prototype Pollution (CWE-1321 [12]), OS Command Injection (CWE-78 [13]) and Path Traversal (CWE-22 [15])

In the paper, the authors indicate how to query the ODG in order to identify vulnerabilities in Node.js applications. For instance, for injection vulnerabilities, the query revolves around identifying a backward taint-flow from a sensitive sink to an attacker-controlled source. Conversely, for prototype pollution vulnerabilities, the query focuses on locating object assignments where the attacker has control over both the property being assigned and the corresponding value.

During the evaluation, ODGen demonstrated its capability to effectively identify various vulnerabilities, including injection and prototype pollution. ODGen's performance surpassed that of other tools in the field, outperforming, for example, both JSJoern [20] and JSTap-vul [21], with fewer false positives and false negatives, showcasing its enhanced accuracy and reliability. More specifically, ODGen exhibited a false positive rate of 32%, whereas JSJoern and JSTap-vul reported false positive rates of 75% and 80%, respectively.

FAST: In their work, M. Kang *et al.* [22] recognized that abstract interpretation techniques, exemplified by approaches like ODGen, encounter scalability challenges when dealing with code exceeding a certain threshold of lines. This scalability issue prevents the identification of many vulnerabilities, as the exploration of paths increases exponentially. This leads to situations where vulnerable sinks and/or sources remain unexplored. Additionally, they observed that numerous taint-flow style tools struggle to handle `Promise` calls due to their asynchronous nature.

To address these challenges, the authors introduced a novel approach to taint-style analysis named *FAST* (Fast Abstract Interpretation for Scalability). FAST employs a combined methodology, integrating a bottom-up abstract interpretation method to identify pathways from entry points to sink functions, along with a top-down approach to construct a data-flow graph. The top-down approach not only extracts source-sink paths but also ensures that only the instructions directly dependent on the sink are analyzed. Therefore, the top-down approach improves precision by disregarding unrelated instructions. To further enhance accuracy and reduce false positives, FAST attempts to generate a working exploit for a given path using solvers like Z3. It only considers a vulnerability exploitable when a successful exploit is generated. This approach helps FAST achieve a more accurate and reliable identification of exploitable vulnerabilities.

In evaluations against ODGen and CodeQL, using datasets featuring real-world vulnerable Node.js packages and a dedicated benchmark for scalability assessment, FAST demonstrated superior performance. The tool exhibited a false positive rate of 11.8%, which is better than that presented by ODGen and CodeQL, with rates of 23.3% and 27.8%, respectively. Additionally, it had a false negative rate of 16.6%, surpassing the rates of the other tools, which were 43.7% and 35.3%, respectively. The type of vulnerabilities that FAST was able to detect include code injection, command injection, and path traversal vulnerabilities. In terms of scalability, FAST was able to detect 14 vulnerabilities in its dedicated dataset, while ODGen detected none.

Nodest: Building on a similar motivation as in FAST, which addresses the scalability challenges of static analysis tools, B. Nielsen *et al.* [23] introduced a technique aimed at analyzing only the essential modules within a package. Their approach involves dynamic decision-making at runtime to determine which packages should be analyzed and which ones can be safely ignored based on feedback. To assess the effectiveness of their approach in detecting injection vulnerabilities in Node.js applications, they implemented it in a tool called *Nodest*.

The core concept behind Nodest is the recognition that not all modules require exhaustive analysis. To accommodate this idea, Nodest dynamically maintains two working sets: *MSp* (modules to be analyzed) and *MSb* (modules not to be analyzed). Utilizing a set of tags and predicates, Nodest assigns tags to each module, enabling it to determine whether a module should be included in *MSp*. For example, if the predicate *isInTaintFlow(M)*, when applied to module *M*, returns true, M is added to *MSp* because a taint flow reaches that module. This predicate indicates that a taint flow reaches module *M*, therefore it needs to be analyzed.

While modules that will not be analyzed can be known beforehand, Nodest doesn't necessarily require the user to initialize that set. Instead, Nodest dynamically populates this set during its analysis. For that reason, it showcases adaptability and flexibility in identifying which modules are crucial for analysis and which can be

safely excluded. Nodest identified 63 vulnerabilities, including 2 previously unknown, across 11 npm packages during execution.

CodeQL: CodeQL is a static analysis tool used for identifying security vulnerabilities and bugs in software code. Developed by GitHub, CodeQL employs a semantic code analysis approach, treating code as a database to query and explore. It allows developers to create queries to detect patterns and potential issues within a codebase. In particular, CodeQL is capable of detecting some of the most common vulnerabilities present in Node.js applications, such as command injection, path traversal and prototype pollution. By leveraging CodeQL, developers can perform in-depth analyses, tracing data flows and uncovering security vulnerabilities, even in large and complex code repositories.

The tools presented in the previous paragraphs compete with Graph.js. These tools can detect vulnerabilities in Node.js applications, even in the presence of modules, which Graph.js fails to do it accurately. More concretely, FAST and Nodest allow for a more scalable and efficient analysis of applications than ODGen, CodeQL and Graph.js. Additionally, FAST is the only static analysis tool that generates exploits for the vulnerabilities it detects, reducing the number of reported false positives. However, these tools still exhibit false positives and false negatives that impact their performance. Moreover, none of these tools can detect all the vulnerabilities that Graph.js can. For instance, both are unable to detect prototype pollution vulnerabilities.

4 Proposed Solution

To address the limitations discussed in Section 2.3.4, we will develop a new version of Graph.js with support for reasoning about modules. Specifically, two new strategies, called *Complex Graph with Simple Queries* and *Simple Graph with Complex Queries*, will be developed and incorporated into Graph.js, creating Graph.jsV1. In this section, we describe the strategies Complex Graph with Simple Queries (Section 4.1) and Simple Graph with Complex Queries (Section 4.2), while also providing a comparison between both them (Section 4.3).

4.1 Complex Graph with Simple Queries

The idea of the strategy Complex Graph with Simple Queries is to merge the MDGs from various modules into a single MDG that models the application as a whole. To do this, it is not enough to simply concatenate the graphs of the different modules into a single graph. It is also necessary to connect the arguments of function calls to their corresponding parameters and the return values of function calls to the values returned inside the corresponding functions. More specifically, this strategy will require the creation of the following types of nodes and edges:

- ***Return Value Node***: Return Value Nodes represent a function's return value. Figure 4 shows two return nodes: the return node of `Bar.f` (o_7) and the return node of `Bar.g` (o_{10})
- ***Return Edges***: Return edges, RET, connect the node representing a function's return value and the node representing the object that captures that value in the caller function. The edge $n_1 \xrightarrow{RET(\texttt{f})} n_2$ signifies that the object n_2 may correspond to the object n_1 returned by function `f`. In Figure 4, the edge $o_{10} \xrightarrow{RET(\texttt{Bar.g})} o_4$ signifies that object o_4 corresponds to the object o_{10} returned by function `Bar.g`.
- ***Argument Edges***: Argument edges, ARG, connect the arguments of a function call in the caller context with their corresponding parameters in the callee context. The edge $n_1 \xrightarrow{ARG(\texttt{f})} n_2$ signifies that n_2 represents a formal parameter of `f` and that that parameter receives the value represented by n_1. In Figure 4, the edge $o_2 \xrightarrow{ARG(\texttt{Bar.f})} o_6$ signifies that the the `Main` module calls `Bar.f` with o_2 as the formal parameter o_6.

To uncover the vulnerabilities modeled by the updated MDGs, modifications to the queries become imperative. While the queries continue to seek paths from a tainted source node to a sensitive sink through DEP and

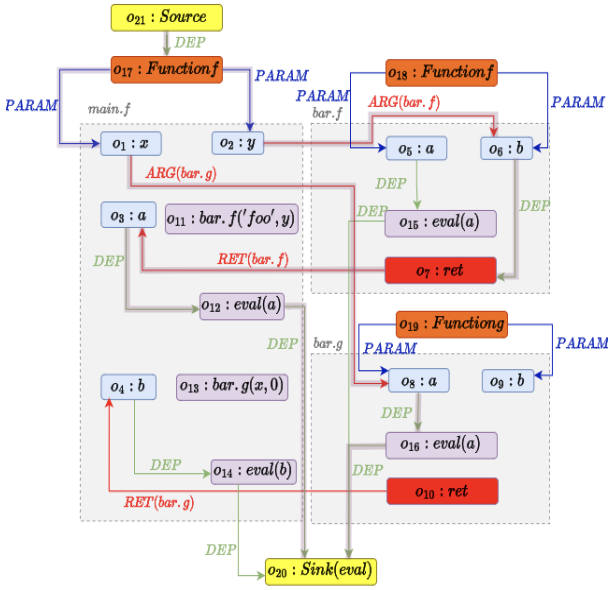

Figure 4: Complex Graph with Simple Queries technique applied to the graphs in Figures 2 and 3

PARAM edges, they now must also incorporate the new ARG and RET edges. In essence, the queries are designed to trace a path from the tainted source node to the sensitive sink that traverses DEP, PARAM, ARG, and RET edges. For instance, the detection of the injection vulnerability in the `Main.f` (line 5) involves executing a query that finds the path:

$$o_{21} \xrightarrow{DEP} o_{17} \xrightarrow{PARAM} o_2 \xrightarrow{ARG(\texttt{f})} o_6 \xrightarrow{DEP} o_7 \xrightarrow{RET(\texttt{f}} o_3 \xrightarrow{DEP} o_{12} \xrightarrow{DEP} o_{20}.$$

This path is highlighted in the figure in light purple.

4.2 Simple Graph with Complex Queries

The idea of the strategy Simple Graph with Complex Queries is also to merge the MDGs from various modules into a single MDG that models the application as a whole. Unlike the previous strategy, the graphs of different modules are not directly connected. Instead, we keep the separation between callers and callees, relying on queries to establish connections. To ensure that the queries work, function call nodes now contain information specifying which function in the external module could be called. This is illustrated by the orange boxes below o_{11} and o_{13} in Figure 5. In addition to annotating call nodes with identifiers of the functions being called, we expand the graph by introducing the following nodes and edges:

- ***Return Value Object Node***: Return Value Nodes represent a function's return value. Figure 4 shows two return nodes: the return node of `Bar.f` (o_7) and the return node of `Bar.g` (o_{10}). These are the same objects as those presented in the previous strategy.
- ***Return Edges***: RET edges connect function call nodes and the node representing the object that captures its return value in the caller function. The edge $n_1 \xrightarrow{RET(\texttt{f})} n_2$ signifies that the object n_2 represents the return value of `f` at the call site n_1. In Figure 5, the edge $o_{13} \xrightarrow{RET(\texttt{bar.g})} o_4$ represents the return value of `Bar.g` at the callsite represented by o_{13}.
- ***Argument Edges***: Argument edges, ARG, connect the objects used as arguments in function calls and their corresponding function call nodes. The edge $n_1 \xrightarrow{ARG(\texttt{f.x})} n_2$ signifies the function represented by

n_2 (`f`) has a parameter `x` and that parameter receives the value represented by n_1. In Figure 5, the edge $o_2 \xrightarrow{ARG(\texttt{bar.f.b})} o_6$ signifies that the `Main` module calls `Bar.f` (o_6), with o_2 as its formal parameter `b`. Note that these argument edges are distinct from those in the previous strategy. These edges connect arguments to call nodes, while the others connect arguments to parameters.

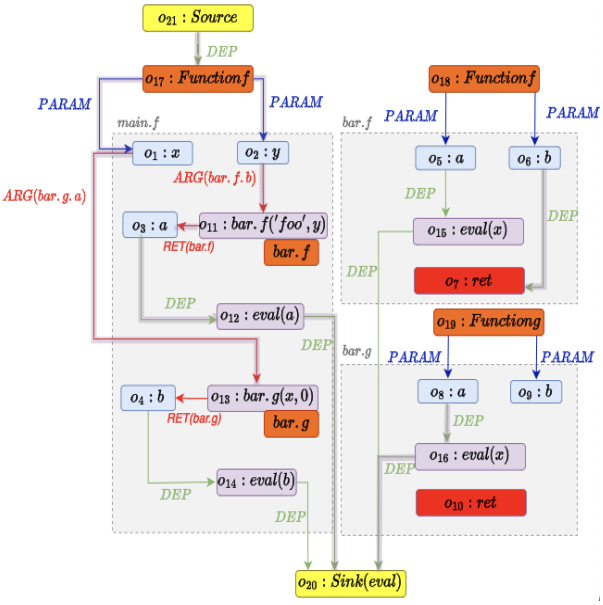

Figure 5: Simple Graph with Complex Queries technique applied to the graphs in Figures 2 and 3

To effectively report the vulnerabilities embedded within the updated MDGs, it becomes imperative to make modifications to the queries. Specifically, the combination of the following queries is necessary:

- ***Taint Propagation Query***: This query is designed to identify paths from the tainted source node to a taint sink node that go through DEP, PARAM, ARG, and RET edges. However sequences of argument and return edges do not necessarily propagate taint. For instance the sequence $n_1 \xrightarrow{ARG(\texttt{f.x})} n_2 \xrightarrow{RET(\texttt{f})} n_3$ only propagates taint from n_1 to n_3 if `f`'s return value depends on the `x` parameter. To detect this, we require the next query to be executed as an auxiliary query
- ***Parameter-Return Dependency Query***: This query aims to determine if the return value is influenced by the parameter, indicating the propagation of taint from the parameter to the return value. Similar to the previous query, it follows DEP, PARAM, ARG, and RET edges, but with distinct start and end nodes. Specifically, it initiates from the node that represents the parameter and concludes at the return value node. This query may recursively call itself, considering that applications can have nested function calls with data flows that need to be checked.

The process of identifying the vulnerabilities begins by executing the taint propagation query, on the main module's subgraph, which holds the program's entry point. In graphs without function call nodes referencing external functions, this initial query suffices for detecting the targeted vulnerabilities. However, when a function call node referring to an external function is encountered during the first query, the second query must be executed as a subquery. If the subquery reveals that the return value is not tainted, that taint path is sensitive but not further explored. Otherwise, the first query resumes, aiming to check if this direct path reaches a sensitive sink.

For example, detecting the injection vulnerability in the `Main` module (line 5) involves executing a query that starts by finding the path:

$$o_{21} \xrightarrow{DEP} o_{17} \xrightarrow{PARAM} o_2 \xrightarrow{ARG(\texttt{bar.f.b})} o_{11} \xrightarrow{RET(\texttt{f}} o_3 \xrightarrow{DEP} o_{12} \xrightarrow{DEP} o_{20}.$$

The subquery triggered while identifying the mentioned path is as follows:

$$o_{18} \xrightarrow{PARAM} o_6 \xrightarrow{DEP} o_7.$$

Both paths are highlighted in the figure in light purple

4.3 Comparing both Strategies

We will implement both strategies to determine their effectiveness. While they are equivalent in expressiveness, their performance may vary. Therefore, it's crucial to implement and evaluate both to understand their behavior. The second strategy, with cached results of auxiliary queries, is expected to perform better. Once we identify a function propagating taint from a parameter to the return value, there's no need to traverse that path again. In contrast, the first strategy always seeks complete paths, requires traversing a function's body each time it's encountered, resulting in potential redundancy.

Both strategies present different implementation challenges. In the first strategy, the heavy lifting lies in graph construction: we need to add numerous connections to weave together the MDGs from various modules. In the second, the heavy lifting is in query implementation, since the main query must be structured to accommodate interspersed calls to the auxiliary query (confirming parameter-return taint flows).

5 Evaluation & Planning

In this section, we discuss the evaluation of the solution (Section 5.1) and detail how the work will unfold throughout the semester (Section 5.2).

5.1 Evaluation

This subsection provides an overview of the selected datasets and their modular features, along with the tools used to compare with Graph.jsV1 (Section 5.1.1). Furthermore, it also describes the evaluation metrics that will be employed (Section 5.1.2).

5.1.1 Dataset Characterization

The evaluation involves comparing Graph.jsV1 with Graph.js (the existing version), ODGen, and CodeQL, using the SecBench and VulcaN datasets. All tools will be run on both datasets. Specifically, Graph.jsV1 will be run twice, once for the Complex Graph with Simple Queries strategy and another for the Simple Graph with Complex Queries strategy. However, it is necessary to validate that the datasets demonstrate the modular features discussed in this work. To this end, we conducted a characterization of the datasets' modular features, focusing on four key aspects:

- **File Count**: corresponds to the number of source files in the package's directory.
- **Exported Functions**: corresponds to the number of functions exported from the main module of a Node.js package. We determined the main file by examining the *package.json* file and by looking for the file specified in the *main* attribute. If the *main* attribute was absent, the main file defaulted to index.js. Subsequently, we generated the Abstract Syntax Tree (AST) for the package and determined the number of exported functions by traversing it, counting the number of times that either an assignment to `module.exports` or `export` keyword appeared.
- **Maximum Inclusion Depth**: corresponds to the maximum level of nested dependencies within a package. To calculate this metric, we used the *Dependency Tree* NPM package [24] to create the package's dependency graph. Afterwards, we performed a depth-first search (DFS) on the graph to calculate the inclusion depths, identifying the longest path in the graph. This can be done with a DFS, because the dependency graph is acyclic.

- **Maximum Call Depth**: corresponds the maximum depth of function call chains within a Node.js package. To account for calls extending into imported modules, the characterization computed the call depth on a file-by-file basis while traversing the file dependency tree. This approach was necessary due to the Node.js ability to export functions with names different from their original declaration in the module. In the file-by-file analysis, we generated the call graph using the *JSCG* [25] NPM package. We then calculated the maximum call depth using a depth-first search (DFS) to determine the longest path in the graph. However, the call graph might contain loops, since the sequence of calls $a \xrightarrow{calls} b \xrightarrow{calls} a$ and recursive calls, introduce circular dependencies. To mitigate this, the Depth-First Search (DFS) algorithm was adjusted to keep track of the current path it is traversing. Consequently, if a call leads back to a node within the current path, the DFS ignores that node, preventing this issue.

Figures 6 and 7 display the normalized results using min-max normalization. As indicated by all the plots, the datasets show similar values across various characteristics, such as the number of exported functions, the maximum inclusion depth, and the number of files. These values range from 0 to 16 for SecBench and 0 to 20 for VulcaN in both datasets. However, there is a difference in the maximum call depth, where VulcaN's call depth significantly surpasses that of SecBench. This is consistent with our experience with these datasets, where we observed that VulcaN has more more complex pacackages than SecBench.

In sumarry, the Vulcan and SecBench datasets are suitable datasets for evaluating the techniques we will develop because they exhibit the modular features addressed by this work. Specifically, their packages are organized into modules, and these modules are structured across various files. Additionally, these modules interact with each other through function calls.

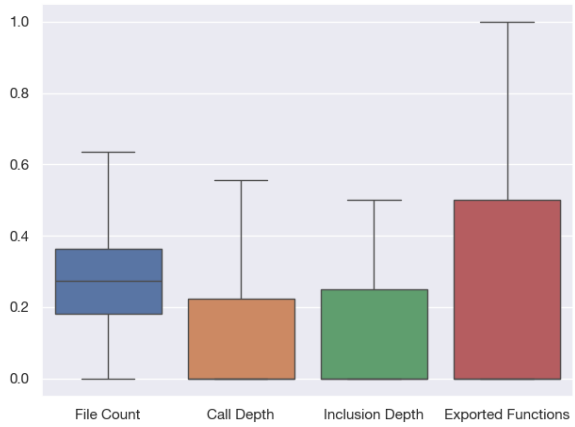

Figure 6: Characterization of the SecBench dataset

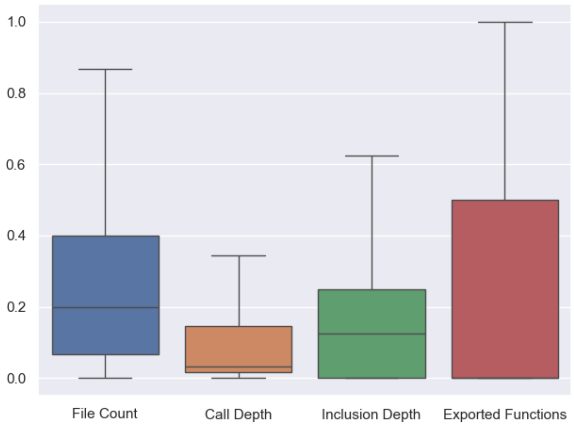

Figure 7: Characterization of the Vulcan dataset

5.1.2 Evaluation Metrics

In order to assess and draw conclusions about the tools' performance in the datasets, we need to compute some metrics. More concretely, the following metrics will be computed:

- ***Recall:*** Recall is the percentage of actual vulnerabilities correctly identified. It measures a tool's ability to capture and correctly flag vulnerabilities among all the vulnerabilities present in the dataset. We can compute a tool's recall in a dataset by using the formula bellow.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

- ***Precision:*** Precision is the percentage of reported vulnerabilities that are indeed true positives, out of the total number of reported vulnerabilities. It measures the accuracy of the tool in identifying and reporting

actual vulnerabilities without generating many false positives. We can compute a tool's precision in a dataset by using the formula bellow.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

- ***Efficiency:*** Evaluation is runtime performance of a tool during analysis. It measures the time taken by a tool to analyze a dataset. We can compute a tool's efficiency in a dataset by keeping track of the time a tool takes to evaluate a dataset.

We expect Graph.jsV1 to have better precision than Graph.js, but it may potentially have lower recall. It will achieve better precision since, with the new strategies for handling modules, it can accurately analyze calls to external modules. However, these same strategies may result in the failure to detect some vulnerabilities, as we will conduct a more fine-grained analysis of modules, and there might be cases that escape this reasoning. Additionally, we expect Graph.jsV1 with the strategy Simple Graph with Complex Techniques to be more efficient than Graph.jsV1 with the strategy Complex Graph with Simple Strategies, for the reasons outlined in Section 4.3.

5.2 Planning

The work proposed by this report can be divided in 4 tasks. This subsection starts by describing these tasks. Then, Table 1 describes how the work will be done throughout the semester. More concretely, the tasks are as follows:

- ***Implement Complex Graph with Simple Queries Strategy:*** involves making modifications to the source code of Graph.js' graph constructor module. The goal is to integrate the new strategy into the existing functionality. Once completed, Graph.js will have the capability to generate the MDGs outlined in Section 4.1.
- ***Implement Simple Graph with Complex Queries Strategy:*** involves making modifications to the source code of Graph.js' graph constructor module to integrate the new strategy into its existing functionality. Once completed, Graph.js will have the capability to generate the MDGs outlined in Section 4.2.
- ***Implement the Queries for both strategies:*** involves creating the new queries necessary to detect the vulnerabilities modeled by both strategies, as described in Sections 4.1 and 4.2.
- ***Evaluation of the implementations:*** involves running both strategies along with the other tools in the datasets just as described in Section 5.1. Additionally, this task also encompasses computing all the evaluation metrics necessary.
- ***Writing of the Thesis:*** involves writing the thesis detailing all the work that will be done.

Period	***Work***
February - March	Implement Complex Graph with Simple Queries Strategy
April - May	Implement Simple Graph with Complex Queries Strategy
May - June	Implement the queries for both techniques
June - July	Evaluation of the implementations
July - September	Writing of the thesis

Table 1: Organization of the Work throughout the semester

6 Conclusion

In conclusion, this work addressed the challenges and vulnerabilities inherent in Node.js applications. Specifically, our work emphasizes the critical role of the World Wide Web in our daily lives and the impact of JavaScript and Node.js in web development. Despite their advantages, Node.js applications face security issues, introduced by the language-specific behaviors of JavaScript.

Static analysis provides a viable solution for identifying and mitigating vulnerabilities in Node.js applications. Graph-based approaches, exemplified by tools like Graph.js, have proven highly effective in this context. Graph.js is composed by two modules: the Graph Constructor Module, responsible for generating MDGs, and the Query Execution Engine, tasked with running queries in MDGs to detect vulnerabilities. To the best of our knowledge, Graph.js stands out as the leading static analysis tool for Node.js vulnerability detection. Graph.js exhibits fewer false positives and greater efficiency than its closest competitor, ODGen.

However, Graph.js exhibited limitations in modular reasoning, particularly in handling calls to external modules. This limitation, in particular, leads to an increased number of false positives. To addressed this issue, this work introduced two strategies designed to enhance Graph.js' accuracy and diminish the occurrence of false positives. More concretely, the contributions of this work are as follows:

1. ***Complex Graph with Simple Queries Strategy:*** involves merging the graphs from diverse modules into a unified graph, establishing connections between nodes in both graphs. The proposal includes introducing additional nodes and edges to ensure an accurate representation of the source code's modular features.

2. ***Simple Graph with Complex Queries Strategy:*** involves merging graphs from various modules into a unified representation of the entire application. Similar to the first approach, it suggests introducing new nodes and edges to correctly model the source code's modular features. The proposed nodes are consistent across both strategies, while the edges differ. Unlike the previous strategies, the graphs remain separate, and the responsibility of connecting them is delegated to the queries. Additionally, function call nodes now contain information about the specific external module function that could be called, assisting the queries' job.

3. ***Vulnerability Detection Queries for Both Strategies:*** involves developing a new set of queries capable of identifying vulnerabilities represented by the updated graphs. For the Complex Graph with Simple Queries strategy, minimal adjustments to existing Graph.js queries are needed. On the other hand, implementing the queries for the Simple Graph with Complex Queries strategy requires the creation of two entirely new queries.

The evaluation of the upgraded Graph.js version involved comparing results with competitors ODGen and CodeQL on SecBench and Vulcan datasets. The anticipated improvement lies in the reduction of false positives, enhancing the overall reliability of Graph.js. The outcomes of this research aim making Graph.js a more robust tool tool for static analysis in Node.js applications that can be integrated in the CI/CD pipelines.

Future Work: In summary, this work offered an enhanced version of Graph.js with improved modular reasoning, addressing its limitations and transforming it into a more robust tool for identifying vulnerabilities in Node.js applications. However, there is still room for further improvement in Graph.js. For instance, this study focuses only on injection vulnerabilities, but Graph.js likely encounters similar challenges in module usage when applied to other vulnerabilities, such as prototype pollution. Therefore, it is possible to extend the strategies presented in this report to account for other vulnerability types.

Bibliography

[1] "Node.js," https://nodejs.org/en, accessed: 2023-10-03.

[2] "Bbc: Highgate wood school closed following cyber attack," https://www.bbc.com/news/uk-england-london-66733964, accessed: 2023-10-03.

[3] "Node package manager," https://www.npmjs.com, accessed: 2023-10-03.

[4] "Npm passes the 1 millionth package milestone! what can we learn?" https://snyk.io/blog/npm-passes-the-1-millionth-package-milestone-what-can-we-learn/, accessed: 2023-10-03.

[5] Anonymous, "Efficient static vulnerability analysis for javascript with multiversion dependency graphs," paper under submission.

[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)*. Boston, MA: USENIX Association, Aug. 2022, pp. 143–160. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/li-song

[7] 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)*, 2023, pp. 1059–1070.

[8] 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," in *IEEE Transactions on Reliability*, 2023, pp. 1–16.

[9] "Codeql," https://github.com/github/codeql, accessed: 2023-10-29.

[10] M. Monteiro, "Explodeq.js: A library of queries to detect injection vulnerabilities in node.js applications," Master's thesis, Instituto Superior Técnico, 2023.

[11] "Cypher language query," https://neo4j.com, accessed: 2023-10-03.

[12] "CWE-1321: Improperly Controlled Modification of Object Prototype Attributes ('Prototype Pollution')," https://cwe.mitre.org/data/definitions/1321.html, The MITRE Corporation.

[13] "CWE-78: Improper Neutralization of Special Elements used in an OSCommand ('OS Command Injection')," https://cwe.mitre.org/data/definitions/78.html, The MITRE Corporation.

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

[15] "CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('PathTraversal')," https://cwe.mitre.org/data/definitions/22.html, The MITRE Corporation.

[16] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the Attack Surface of Node.js Applications," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Oct. 2020.

[17] C.-A. Staicu, M. Pradel, and B. Livshits, "Synode: Understanding and automatically preventing injection attacks on node.js," in *Network and Distributed System Security Symposium*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:51951699

[18] "Typescript," https://www.typescriptlang.org, accessed: 2024-1-9.

[19] "CWE-1333: Inefficient Regular Expression Complexity," https://cwe.mitre.org/data/definitions/1333.html, The MITRE Corporation.

[20] "Jsjoern," https://github.com/malteskoruppa/phpjoern, accessed: 2023-10-29.

[21] A. Fass, M. Backes, and B. Stock, "Jstap: A static pre-filter for malicious javascript detection," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 257–269. [Online]. Available: https://doi.org/10.1145/3359789.3359813

[22] 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)*, 2023, pp. 1059–1076.

[23] 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*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 455–465. [Online]. Available: https://doi.org/10.1145/3338906.3338933

[24] "Dependency tree," https://www.npmjs.com/package/dependency-tree, accessed: 2023-12-03.

[25] "Jscg," https://www.npmjs.com/package/jscg, accessed: 2023-12-03.