TÉCNICO LISBOA

Detecting Multi-file Vulnerabilities Using Code Property Graphs

Guilherme Figueira da Silva Gonçalves

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor(s): Prof. José Faustino Fragoso Femenin dos Santos
Prof. Pedro Miguel dos Santos Alves Madeira Adão

Examination Committee

Chairperson: Prof. Manuel Fernando Cabido Peres Lopes
Supervisor: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Member of the Committee: Prof. Rodrigo Fraga Barcelos Paulus Bruno

November 2024

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 family and my closest friends,

Acknowledgments

I would like to express my deepest gratitude to my dissertation supervisors, Prof. José Santos and Prof. Pedro Adão. Their guidance, expertise, and encouragement have been instrumental in shaping this work. Their invaluable insights and feedback have consistently pushed me to improve and refine my research.

I would also like to express my appreciation the members working on the Graph.js and Explode.js projects. Their collaborative spirit, innovative ideas, and dedication have greatly influenced the progress and success of my research.

Finally, I am deeply thankful to my family for their unwavering support and encouragement. Their love, patience, and understanding have been a constant source of strength and motivation throughout this journey.

Resumo

Num mundo cada vez mais digital, a necessidade de aplicações web seguras é crucial, especialmente com o crescente uso de *JavaScript* e *Node.js* [1] no desenvolvimento web. Apesar das suas vantagens, o Node.js enfrenta desafios de segurança significativos, em grande parte devido a vulnerabilidades introduzidas pela sua natureza dinâmica e pelo *Node Package Manager* (NPM) [2]. Para abordar estas questões, melhoramos a ferramenta de deteção de vulnerabilidades *Graph.js* [3], que tem provado ser eficaz, mas é limitada pela sua falta de suporte para análise interprocedimental e de múltiplos ficheiros, resultando em falsos positivos.

Para resolver as limitações mencionadas em cima, nós propomos modificações ao *Graph.js* ao desenvolver um *Extended Multi-version Dependency Graph* (EMDG) que melhora a análise interprocedimental e unifica gráficos de vários módulos. Além disso, introduzimos três novos algoritmos de deteção: os algoritmos *Top-Down*, *Bottom-Up com Pre-processing* e *Bottom-Up Greedy*. Para além dos algoritmos, também propomos um nova definção de objecto controlado pelo atacante, com o objetivo de fazer com que a tool detete vulnerabilidades que não detetava.

Nós avaliámos o nosso trabalho em dois conjuntos de dados (*datasets*), o *VulcaN* [4] e o *SecBench* [5]. A avaliação demonstra que a abordagem *Bottom-Up Greedy* atinge um *recall* de 82% e uma precisão (*precision*) de 85%, superando o *Graph.js* original e o *ODGen* [6] ao reduzir falsos positivos. Além disso, testes realizados num conjunto de dados de pacotes NPM do mundo real revelam que a abordagem *Bottom-Up Greedy* reporta 83% menos vulnerabilidades e é, em média, 3 segundos mais rápida do que a versão atual do *Graph.js*. A precisão e o *recall* para este conjunto de dados são estimadas em 83% e 81%, respetivamente. Utilizando a nova definição de objeto controlado por atacante, o número de vulnerabilidades reportada aumenta em 30%, mas a precisão (*precision*) estimada mantém-se.

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

Abstract

In our increasingly digital world, the need for secure web applications is crucial, especially with the growing use of JavaScript and Node.js [1] in web development. Despite its advantages, Node.js faces significant security challenges, largely due to vulnerabilities introduced through its dynamic nature and the Node Package Manager (NPM) [2]. To address these issues, we enhance the Graph.js [3] vulnerability detection tool, which has proven effective but is limited by its lack of support for inter-procedural and multi-file analysis, resulting in false positives.

To address those limitations, we propose modifications to Graph.js by developing an Extended Multi-version Dependency Graph (EMDG) that improves inter-procedural analysis and unifies graphs from various modules. Additionally, we introduce three new detection algorithms: the Top-Down, Bottom-Up with Pre-processing, and Bottom-Up Greedy algorithms, each designed to effectively identify vulnerabilities. Besides the algorithms, we also propose a new definition for an attacker-controlled object, aiming to detect vulnerabilities that the tool did not report.

We evaluated our new approaches in two ground truth datasets, the VulcaN [4] and SecBench [5]. The evaluation demonstrates that the Bottom-Up Greedy approach achieves an 82% recall and 85% precision, outperforming the original Graph.js and ODGen [6] by reducing false positives. Furthermore, testing on a dataset of real-world NPM packages reveals that the Bottom-Up Greedy approach reports 83% fewer vulnerabilities and is 3 seconds faster on average than the current version of Graph.js. Precision and recall for this dataset are estimated at 83% and 81%, respectively.Notably, with the new attacker-controlled object definition, the number of vulnerabilities reported increases by 30%, yet the estimated precision remains the same.

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

Contents

Acknowledgments vii
Resumo ix
Abstract xi
List of Tables xv
List of Figures xvii
List of Listings xix
Acronyms xxi

1 Introduction **1**
1.1 Motivation 1
1.2 Goals 2
1.3 Contributions 2
1.4 Thesis Outline 3

2 Background **5**
2.1 Modularity in JavaScript and Node.js 5
2.2 Node.js Security Model 6
2.3 Graph.js 9

3 Related Work **15**
3.1 Node.js Security 15
3.2 Vulnerability Detection in Node.js applications 18

4 Extended MDGs **21**
4.1 Inter-procedural Analysis Support 21
4.2 Multi-file Support 23

5 Vulnerability Detection in the EMDGs **29**
5.1 Cypher Language Queries 29
5.2 Top-Down Vulnerability Detection 30
5.3 Bottom-Up Vulnerability Detection 32
5.4 Soudness Considerations 36

6 Evaluation **41**

6.1 Experimental Setup 41

6.2 RQ1: Which of our three proposed algorithms is most effective for vulnerability detection? 44

6.3 RQ2: How much does our best algorithm improve detection over state-of-the-art tools? . 46

6.4 RQ3: What is the impact of our new attacker-controlled object definition on the detection? 51

7 Conclusions and Future Work **55**

7.1 Conclusions 55

7.2 Future Work 56

Bibliography **59**

List of Tables

6.1 Summary of the vulnerabilities considered in each dataset 42
6.2 Detection results using Regular Expressions in the Combined dataset 43
6.3 Detection results in the Combined dataset 44
6.4 Comparison of ODGen and Bottom-Up greedy approach on the Combined dataset 47
6.5 Comparison of Graph.js and Bottom-Up greedy approach Combined dataset 47
6.6 Results of the evaluation on the Collected dataset 48
6.7 Sampling results in the collected dataset 49
6.8 Sample detection metrics in the collected dataset 49
6.9 Results of the evaluation on the new definition on the Collected dataset. 52
6.10 Precision with the new attacker-controlled object definition in the Collected dataset 52

List of Figures

2.1 Prototype Chain for the `obj` variable in Listing 2.2 8
2.2 Graph.js Architecture Overview 10
2.3 Main module's CPG 11
2.4 Bar module's CPG 11

4.1 Extended MDG example 22
4.2 Dependency DAG Example (Main imports Bar and Foo, and Bar imports Foo) 24
4.3 Export summaries for the running example 25
4.4 Multi-file MDG for the running example 26

5.1 MDG to illustrate the Top-Down approach 31
5.2 EMDG to illustrate the Bottom-Up approach 35
5.3 Transposed Call Graph Generated for the graph in Figure 5.2 35
5.4 Corresponding MDG of Listing 5.5 37

6.1 Comparison of the vulnerabilities detected in the Combined dataset 45
6.2 Comparison of the vulnerable packages reported in the Collected dataset 48
6.3 Code snippet from the the *cpuprofile-webpack-plugin-1.10.3* package 50
6.4 Code snippet from the *picture-tuber-2.0.0* package 50

List of Listings

1 Main Module in ES6 . . . 6
2 Exported module in ES6 . . . 6
3 Main Module in CommonJs . . . 6
4 Exported module in CommonJs . . . 6
2.1 Code Injection Example . . . 8
2.2 Prototype pollution example . . . 8
2.3 Main module's source code . . . 11
2.4 Bar module's source code . . . 11
4.1 Main module . . . 24
4.2 Bar module . . . 24
4.3 Foo module . . . 24
5.1 Example Query in Cypher Query Language (Neo4j) . . . 30
5.2 Top Down Taint Propagation Query in Cypher Query Language (Neo4j) . . . 31
5.3 Bottom Up Taint Propagation Query in Cypher Query Language (Neo4j) . . . 34
5.4 Build Call graph in Cypher Query Language (Neo4j) . . . 34
5.5 Overlooked Vulnerability Example . . . 37
6.1 Code snippet from the package *ungit-0.8.4* to illustrate a False positive . . . 45
6.2 Code snippet from the package *mixin-deep-1.3.0* to illustrate a False Negative . . . 45
6.3 *Index.js* . . . 50
6.4 *Profile.js* . . . 50
6.5 *Index.js* . . . 50
6.6 *Tube.js* . . . 50
6.7 Code snippet package *node-watch-0.7.4* . . . 51

Acronyms

AST	Abstract Syntax Tree
CFG	Control Flow Graph
ODG	Object Dependency Graph
CPG	Code Property Graph
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DAG	Directed Acyclic Graph
MDG	Multiversion Dependency Graph
EMDG	Extended Multi-version Dependency Graph

Chapter 1

Introduction

1.1 Motivation

In our day-to-day lives, the web plays a crucial role, evident in activities such as online communication through 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 leads 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 [7], 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 vulnerabilities in Node.js.

In addition to the challenges outlined in the previous paragraph, the platform's default package manager, Node Package Manager (NPM) [2], also introduces vulnerabilities in Node.js applications. NPM hosts a repository with milions of packages [8] 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 be vulnerable. 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 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 [3] 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 [5] and VulcaN [4]. 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: it lacks support for inter-procedural and multi-file analysis. On one hand, Graph.js labels function call return values as unsafe, regardless of attacker control. On the other hand, its inability to handle file dependencies forces a file-by-file analysis, overlooking module interactions. Both problems lead to an increased number of false positives, reducing the tool's overall effectiveness.

1.2 Goals

Given the motivation presented above, this thesis aims to enhance Graph.js by improving its detection capabilities and reduce the number of false detections. To achieve this objective, we propose modifications to Graph.js' core modules.

Firstly, our goal is to improve the functionality of Graph.js' Graph Constructor Module, which builds the Multi-version Dependency Graph (MDG), a new type of CPG proposed by the authors of Graph.js, from the source file. To achieve this, we extended the Graph Constructor Module to create an Extended Multi-version Dependency Graph (EMDG). To enhance inter-procedural analysis, we add new nodes, including call and return nodes, and new edges, such as argument and return edges. To address the multi-file issue, the EMDGs unify the graphs from different modules into a single graph, forming a graph with sub-graphs.

In addition to developing the EMDGs, we need to create new detection algorithms to identify the vulnerabilities that Graph.js previously detected, as Graph.js cannot connect sub-graphs in the EMDGs. Therefore, we introduce three new detection algorithms that can connect these sub-graphs and detect vulnerabilities. These queries use either a Top-Down or Bottom-Up approach to traverse the graph.

In its essence, our main goal is to provide a more reliable and efficient tool for identifying vulnerabilities within Node.js applications.

1.3 Contributions

As mentioned above, this thesis aims at enhancing Graph.js detecting capabilities. More concretely, our work makes the following contributions:

1. ***Extended MDGs***: To accomplish this, we expanded the Graph Constructor Module to generate an Extended Multi-version Dependency Graph (EMDG). We improved inter-procedural analysis by introducing additional nodes, such as call and return nodes, and by adding new edges, like argument and return edges. To handle the multi-file challenge, the EMDGs combine graphs from various modules into one unified graph, which consists of interconnected sub-graphs.

2. ***New Detection Algorithms***: We developed three new algorithms: the Top-Down algorithm, the Bottom-Up with Pre-processing algorithm and the Bottom-Up Greedy algorithm. Each algorithm finds paths from the program's sources to sensitive sinks, differing only in how they navigate the graph to identify these paths.

3. ***New Attacker-Controlled Object Definition***: While our primary goal is to reduce the false positives reported by Graph.js, we also propose a new attacker-controlled object definition. This definition is designed to enable the tool to identify additional vulnerabilities that it previously missed, by accounting for various methods of taint introduction that were overlooked before.

4. ***Evaluation on the Combined dataset***: We evaluated on the combination of two ground truth datasets: VulcaN [4] and SecBench [5]. The Bottom-Up approaches outperformed the Top-Down approach, with the Bottom-Up Greedy approach being the best, achieving an 82% recall and 85% precision. Compared to the current version of Graph.js, the Bottom-Up Greedy approach improved upon it by improving precision by 2%, worsening recall by 1%, and reports 15% fewer false positives. Against ODGen, it showed a 40% improve in recall, a 18% improve in precision, and about 30% fewer false positives, which is an improvement over ODGen.

5. ***Evaluation on the Collected dataset***: Additionally, we evaluated the tool using a dataset of real-world NPM packages. In this dataset, the Bottom-Up Greedy approach reported 83% fewer vulnerabilities compared to the current version of Graph.js, while also being 3 seconds faster on average in package analysis time. We estimate that the precision and recall for this dataset are 83% and 81%, respectively. Notably, with the new attacker-controlled object definition, the number of vulnerabilities reported increases by 30%, yet the precision remains the same.

1.4 Thesis Outline

This document is organized as follows. Chapter 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. Chapter 3 covers relevant related work, such as Node.js security and vulnerability detection in Node.js applications. Chapters 4 and 5 introduces the EMDGs and the detection of vulnerabilities in the EMDGs, respectively. Chapter 6 addresses the evaluation on this work. Finally, Chapter 7 brings the document to a close, offering a recap of this work and some conclusion remarks.

Chapter 2

Background

This chapter provides the necessary background for understanding our work. First, we cover module usage in JavaScript and Node.js in Section 2.1. Then, in Section 2.2, we overview the Node.js security model, including some of the most common vulnerabilities present in Node.js applications. Finally, we introduce Graph.js in Section 2.3. Understanding Graph.js is crucial here, as our work builds upon it to enhance its functionality and address security concerns.

2.1 Modularity in JavaScript and Node.js

In this section, 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, 2, 3 and 4 showcase the usage of modules to organize an application. The 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 (ES6) 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 Listings 1 and 2, 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 (Listing 2 lines 2 and 3) demonstrates the usage of the export keyword, while Module 2 (Listing 1 line 2) illustrates the usage of the import keyword.

```
1 // main module
2 import { greeting, sayHello } from './module1';
3 console.log(greeting); // Output: Hello
4 const message = sayHello("Alice");
5 console.log(message); // Output: Hello, Alice!
```

Listing 1: Main Module in ES6

```
1 // exported module
2 export const greeting = "Hello";
3 export function sayHello(name) {
4     return `${greeting}, ${name}!`;
5 }
```

Listing 2: Exported module in ES6

```
1 // main module
2 const module1 = require('./module1.js')
3 console.log(module1.greeting); // Output: Hello
4 const message = module1.sayHello("Alice");
5 console.log(message); // Output: Hello, Alice!
```

Listing 3: Main Module in CommonJs

```
1 // exported module
2 module.exports = {
3     greeting : "Hello",
4     sayHello: function(name) {
5     return `${greeting}, ${name}!`;}}
```

Listing 4: Exported module in CommonJs

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. Listings 3 and 4 illustrate the same modules from Listings 1 and 2, 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.

Starting from Node.js version 12, it 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.

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) [2]. 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 section, 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 2.1, 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 2.1, 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.

- ***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 2.1 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/shadow\")"` as input. This input executes the command `"cat /etc/shadowd"`, which, in Unix systems, reads the file `/etc/shadow`. 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.

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

Listing 2.1: Code Injection Example

```
1 module.exports = function pp(x,y,z) {
2     let obj = {"foo":3};
3     let aux = obj[x]
4     aux[y] = z
5 }
```

Listing 2.2: Prototype pollution example

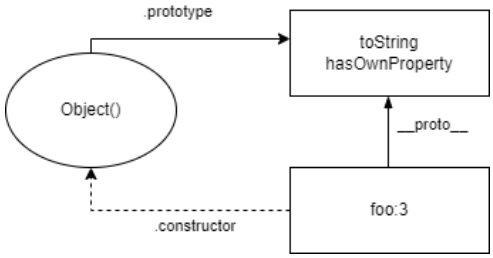

Figure 2.1: Prototype Chain for the `obj` variable in Listing 2.2

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.

2.2.2 Prototype Pollution Vulnerabilities

Prototype Pollution is another common vulnerability in Node.js applications. It occurs when an attacker can alter a built-in JavaScript property by exploiting the object's prototype chain. In JavaScript, each property lookup traverses the object's prototype chain until it locates the property. Consequently, an attacker can manipulate `Object.prototype` through any object, as it serves as the base object from which all objects inherit. This object contains essential built-in functions such as `hasOwnProperty()`, `toString()`, and `valueOf()`. Exploiting this vulnerability can lead to severe consequences, including remote code execution or denial of service.

For example consider Listing 2.2, which exemplifies prototype pollution. In this example, we define a function `pp`. This function creates an object `obj` with a property `"foo"` set to 3. Then, it assigns the value of `obj[x]` to a variable `aux`. Finally, the code attempts to modify the property `y` of `aux` to the value `z`.

The vulnerability in this code arises from the fact that `x`, `y` and `z` can be controlled by an attacker, thus can be manipulated to reference properties in `obj`'s prototype chain (Figure 2.1) and assign an attacker controlled value, leading to prototype pollution. For instance, consider that `pp` is called with the following arguments:

```
function exploited() { return "BAM!!" };
pp("__proto__","toString",exploited)
```

When an attacker calls `pp` with those arguments, he overwrites the `toString` function offered by the `Object.prototype` with the function exploited.. Now, every time an object calls `toString()`, instead of executing the default JavaScript function, it will execute the function `exploited`, enabling the attacker to execute arbitrary code.

2.2.3 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 are common in Node.js applications.

- ***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.
- ***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* [3, 9]. Graph.js is a novel tool designed to 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 2.2. 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 Multi-version 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 Multi-version Dependency Graph (MDG) into a *Neo4j* [10] 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 detect injection vulnerabilities like OS command injection (CWE-78 [11]), Arbitrary Code Execution (CWE-94 [12]), Path Traversal (CWE-22 [13]), and prototype pollution vulnerabilities (CWE-1321 [14]).

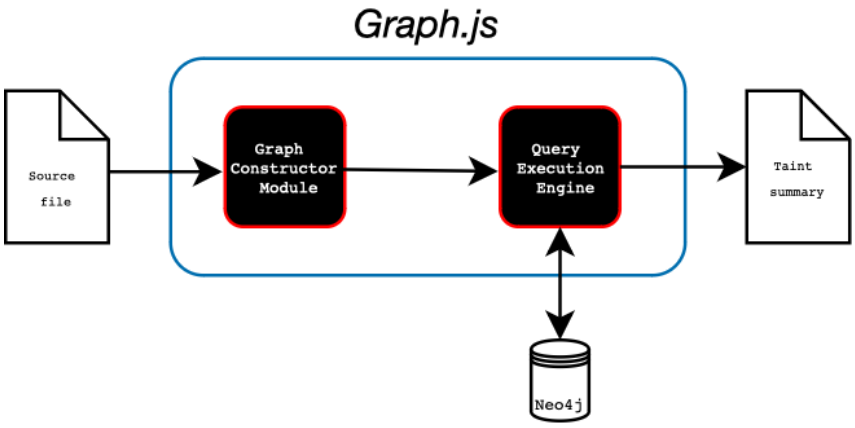

Figure 2.2: Graph.js Architecture Overview

Before diving into the specifics of each module that constitutes 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.

2.3.1 Running Example

Listings 2.3 and 2.4 illustrate the example that will be used throughout this section, while Figures 2.3 and 2.4 ilustrated their corresponding CPGs. 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`. When calling `g`, it provides 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 Multi-version Dependency Graphs (MDGs) using the provided running example, we find nodes of the following types:

```
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;
```

Listing 2.3: Main module's source code

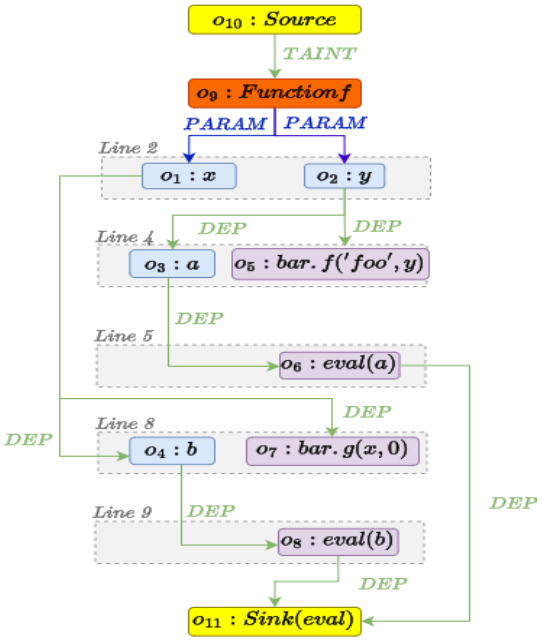

Figure 2.3: Main module's CPG

```
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};
```

Listing 2.4: Bar module's source code

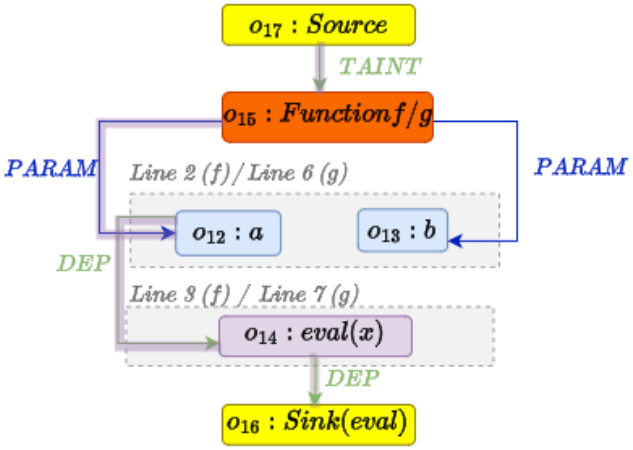

Figure 2.4: Bar module's CPG

- ***Tainted Source Nodes*** (yellow): 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 most frequent source. This node type is demontrated by o_{10} and 0_{17}.
- ***Unsafe Sinks Nodes*** (yellow): Unsafe Sink nodes represent calls to risky functions and/or APIs. This node type is demonstrated by o_{11} and o_{16}.
- ***Value Nodes*** (blue): 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 2.3 and o_{12} to o_{13} in Figure 2.4, demonstrate these nodes.
- ***Call Nodes*** (purple): Call nodes represent the calling of functions within the program. This node

type is demonstrated by o_5 and o_7 in Figure 2.3 and o_{14} in Figure 2.4.

- ***Function Nodes***(orange): 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. For instance, the code snippet `n1 = "p": n2`, creates a node to represent the sub-object `p`, connecting it to `n1` through a property edge.

- ***New Version Edges***: 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 $n_1 \xrightarrow{\text{NV(p)}} n_2$ signifies that n_2 is the new version of the object n_1, resulting from an update to the property p. For instance, the code snippet `a.x = 2`, creates a new version of a, a', connecting a and a' with a new version edge.

- ***Dependency Edges*** (green): Dependency edges illustrate relationships involving data between variables, objects, sources, and sinks. The edge $n_1 \xrightarrow{\text{DEP}} n_2$ indicates that the value represented by n_2 is computed using the value represented by n_1. For instance, in Figure 2.3, 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*** (blue): Parameter Edges connect a function node to the nodes representing its parameters. An edge $n_1 \xrightarrow{\text{PARAM}} n_2$ signifies that the function represented by n_1 has a parameter represented by n_2. For instance, in Figure 2.3, the edges $o_9 \xrightarrow{\text{PARAM}} o_1$ and $o_9 \xrightarrow{\text{PARAM}} o_2$ signify that the function represented by o_9 (function `f`) has two parameters, o_1 (`x`) and o_2 (`y`).

- ***Taint Edges*** (green): Taint edges link the Taint Source to function nodes. The edge `TAINT_SOURCE` $\xrightarrow{\text{TAINT}}$ n_2 means the function n_2 is exported via `module.exports`, allowing an attacker to control its parameters. In Figure 4.1, the edge $o_{10} \xrightarrow{TAINT} o_9$ shows that the attacker controls the parameters of the function `f` because it is exported by the module.

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.

For example, in the case of injection vulnerabilities, the queries look for paths from a tainted source to a sensitive sink that go through dependency, new version, property and parameter edges edges. In the running example, the injection present in the `Bar` module (line 3), can be detected by finding the path:

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

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` (Listing 2.3 line 4). This call has the string `"foo"` as its first argument. Later, in the `Bar` module (Listing 2.4 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.

Another issue regarding the analysis of Graph.js is that it does not correctly classify functions' return values, since it has limitations regarding inter-procedural analysis. Hence, when dealing with function 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` (Listing 2.3 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 corresponds to the number `0`. For that reason, it 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` (Listing 2.3 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 attacker controls the untainted value. For that reason, it may pose as 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` (Listing 2.3 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.

Summary

This chapter provided essential background for the remaining of this thesis. Initially, we examined module usage in JavaScript and Node.js, highlighting its benefits such as code organization and reusability. Additionally, we explored into the Node.js package manager, NPM. Next, we addressed common vulnerabilities such as command injection, path traversal, and prototype pollution that may appear in Node.js applications. Finally, we introduce the central focus of this thesis, Graph.js, covering both its modules, the Graph Constructor Module and the Query Execution Engine. In the following chapter, we will provide an overview of the most relevant research on Node.js security and vulnerability detection tools for Node.js applications.

Chapter 3

Related Work

Scientific research has sought techniques to enhance the security of Node.js applications. In this chapter, 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 Chapter 2, we highlighted that Node.js struggles with its security. In this section, we overview some 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, the datasets aid in comparing and evaluating tools, enabling developers to choose the best tools for analyzing 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 some tools 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.* [15] 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.* [16] 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, the tool uses dynamic checks to prevent potentially harmful inputs from reaching vulnerable APIs. The authors' analysis demonstrated that their approach is efficient, incurring sub-millisecond runtime overhead, and provides robust protection against attacks on vulnerable modules with minimal false positives.

Lightweight Permission System: Numerous packages offered by NPM are susceptible to attacks through malicious updates pushed to their dependencies. Recognizing that many of these packages are straightforward and don't need access to security-sensitive tools like filesystem or network APIs, Ferreira et al. [17] proposed a solution to enforce a least-privilege design per package. This approach safeguards applications and their package dependencies from malicious updates.

Their permission system operates by running packages within a sandbox, ensuring runtime permission enforcement dynamically. Developers specify permissions from a common set for their packages, with the system strictly enforcing these permissions by regulating the require function. Additionally, developers must consent to package permissions during installation and subsequent updates. The system exhibited minimal runtime overhead and was able to protect 31.9% of the 703,457 analyzed packages.

Npm Dependency Guardian: Ohm *et al.* introduced a system known as *NPM Dependency Guardian*, which also implements a least-privilege design per package. What sets NPM Dependency Guardian apart from Ferreira *et al.*'s lightweight permission system is its ability to automatically infer the required privileges for packages, eliminating the need for manual user definition.

This method involves deducing capabilities from a trusted package version by conducting static analysis on the package's source code and its dependencies. Subsequently, the policy enforcement comprises two components. Firstly, the access to modules is restricted by modifying the `require` calls. This modification ensures that the application only requires modules listed in the policies' allow list, while blocked modules return dummy objects. Secondly, it also enforces restrictions on access to global objects. Each module exports a dummy object that contains references to objects permitted by the policies. In testing, NPM Dependency Guardian effectively prevented 9 out of 10 historical malicious package update attacks.

The tools mentioned in the previous attacks focus on preventing potential attacks even in the presence of insecure code. In contrast, Graph.js only detects vulnerabilities during development process to ensure that the code is secure upon execution

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 a baseline for evaluating static analysis tools.

VulcaN: T. Brito *et al.* [4], 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 [13]), OS Command Injection (CWE-78 [11]), and Code Injection (CWE-94 [12]).

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* [19] 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.* [5] 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 [14]), ReDoS (CWE-1333 [20]), Code Injection (CWE-94 [12]), OS Command Injection (CWE-78 [11]) and Path Traversal (CWE-22 [13]). 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 section 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 [14]), OS Command Injection (CWE-78 [11]) and Path Traversal (CWE-22 [13])

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 [21] and JSTap-vul [22], 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.*[23] 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.* [24] 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 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 accurately detect all the vulnerabilities that Graph.js can. For instance, FAST and Nodest are unable to detect prototype pollution vulnerabilities.

Summary

In this chapter, we provided an overview of the related work. We began with Node.js security, focusing on research aimed at minimizing the impact of vulnerabilities introduced through third-party code inclusion [15–17, 25]. Next, we introduced two benchmarks suitable for evaluating the performance of static analysis tools [4, 5]. Finally, we examined some of the most popular tools for static analysis in Node.js applications [6, 19, 23, 24].

Chapter 4

Extended MDGs

To address the limitations mentioned in Section 2.3.4, we need to extend Graph.js' Graph Constructor Module. This chapter details these modifications, starting with the new nodes and edges for improved inter-procedural analysis in Section 4.1. Next, in Section 4.2, we explain how to extend Graph.js for multi-file processing.

4.1 Inter-procedural Analysis Support

In this section, we introduce the new nodes and edges that enhance Graph.js' inter-procedural analysis. Section 4.1.1 describes the nodes that enhance the graph's functionality. Section 4.1.2 details the edges that connect these nodes.

4.1.1 Nodes

One problem highlighted in Section 2.3.4 is Graph.js's incorrect classification of function return values. To fix this and ensure accurate inter-procedural analysis, we introduce a new node type, the Return Node, and additionally make changes to the Call Nodes.

An Extended Multi-version Dependency Graph (EMDG) has the following nodes:

- ***Tainted Source Nodes*** (yellow): 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 most frequent source. In Figure 4.1, this node type is demonstrated by o_{21}
- ***Unsafe Sinks Nodes:*** (yellow): Unsafe Sink nodes represent calls to risky functions and/or APIs. In Figure 4.1, this node type is demonstrated by o_{20}.
- ***Value Nodes*** (blue): Value Nodes represent objects and primitive values generated during the program's execution. In Figure 4.1, this node type is demonstrated by, for example, o_1,o_2,o_5 and o_6.
- ***Call Nodes*** (purple): Call nodes represent function calls within the program and are a new node type. In the EMDGs, call nodes include the identifier of the called function at the call site. Figure 4.1

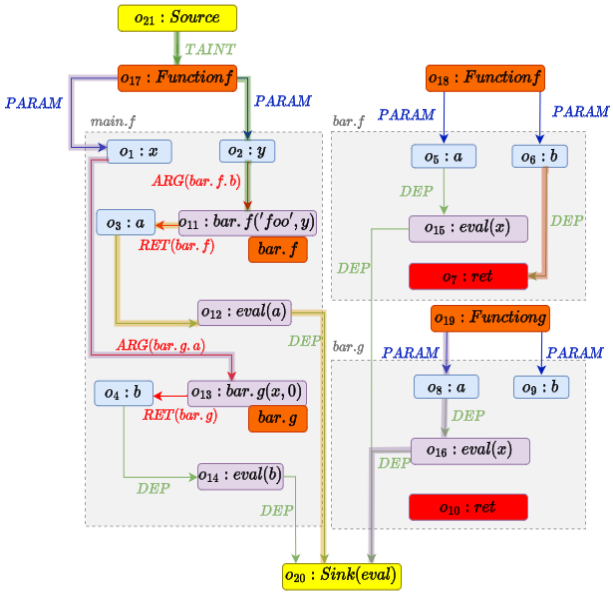

Figure 4.1: Extended MDG example

shows this with nodes o_{11} and o_{13}, where the function identifiers are displayed in the orange boxes attached to these nodes.

- ***Function Nodes*** (orange): Function nodes represent the functions declared throughout the program. This node type is demonstrated by o_{17}, o_{18} and 0_{19}.

- ***Return Value Node*** (red): Return Value Nodes represent a function's return value and are a new node type. Figure 4.1 shows two return nodes: the return node of `Bar.f` (o_7) and the return node of `Bar.g` (o_{10}).

4.1.2 Edges

After introducing the changes to the nodes in the previous section, the focus now shifts to discussing the introduction of two type edges in the graph: *Return Edges* and *Argument Edges*.

An EMDG has the following edges:

- ***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. For instance, the code snippet `n1 = "p": n2`, creates a node to represent the sub-object `p`, connecting it to `n1` through a property edge.

- ***New Version Edges***: 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 $n_1 \xrightarrow{\text{NV(p)}} n_2$ signifies that n_2 is the new version of the object n_1, resulting from an update to the property p. For instance, the code snippet `a.x = 2`, creates a new version of a, a', connecting a and a' with a new version edge.

- ***Dependency Edges*** (green): Dependency edges illustrate relationships involving data between variables, objects, sources, and sinks. The edge $n_1 \xrightarrow{\text{DEP}} n_2$ indicates that the value represented by n_2 is computed using the value represented by n_1. For instance, the edge $o_6 \xrightarrow{\text{DEP}} o_7$ signifies that the function `bar.f` returns an object (o_7) that depends on the object represented by o_6.

- ***Parameter Edges*** (blue): Parameter Edges connect a function node to the nodes representing its parameters. An edge $n_1 \xrightarrow{\text{PARAM}} n_2$ signifies that the function represented by n_1 has a parameter represented by n_2. For instance, in Figure 4.1, the edges $o_{17} \xrightarrow{\text{PARAM}} o_1$ and $o_{17} \xrightarrow{\text{PARAM}} o_2$ signify that the function represented by o_9 (function `f`) has two parameters, o_1 (`x`) and o_2 (`y`).

- ***Taint Edges*** (green): Taint edges link the Taint Source to function nodes. The edge `TAINT_SOURCE` $\xrightarrow{\text{TAINT}}$ n_2 means the function n_2 is exported via `module.exports`, allowing an attacker to control its parameters. In Figure 4.1, the edge $o_{21} \xrightarrow{TAINT} o_{17}$ shows that the attacker controls the parameters of the function `f` because it is exported by the module.

- ***Return Edges*** (red): RET edges connect function call nodes and the node representing the object that captures its return value in the caller function. The edge $n_1 \xrightarrow{\text{RET(f)}} n_2$ signifies that the object n_2 represents the return value of `f` at the callsite n_1. In Figure 4.1, the edge $o_{13} \xrightarrow{\text{RET(bar.g)}} o_4$ represents the return value of `Bar.g` at the callsite represented by o_{13}. This corresponds to the assignment `var a = bar.f("foo",y)` in the Main module".

- ***Argument Edges*** (red): ARG edges connect the objects used as arguments in function calls and their corresponding function call nodes. The edge $n_1 \xrightarrow{\text{ARG(f.x)}} n_2$ signifies the function represented by n_2 (`f`) has a parameter `x` and that parameter receives the value represented by n_1. In Figure 4.1, the edge $o_2 \xrightarrow{\text{ARG(bar.f.b)}} o_{11}$ signifies that the `Main` module calls `Bar.f` (o_{11}), with o_2 as its formal parameter `b`.

4.2 Multi-file Support

This section explains how to extend the Graph Constructor Module to handle program dependencies. We start by creating a Directed Acyclic Graph (DAG) of module dependencies and then process it from sink modules (without dependents) to source modules (those not depended on by any other module). For each module, we generate its EMDG and summarize the objects it exports. When a module calls an external module's function, we add that function's graph using the previously generated summaries. This results in a unified MDG for the entire application, with each module's graph remaining separate. Section 4.2.1 introduces the example used throughout this section. Section 4.2.2 explains how to generate the dependency DAG. Section 4.2.3 covers the details of the exported object summary. Finally, Section 4.2.4 details the algorithm for generating the multi-file MDG.

```
1 // Main module
2 let bar = require('./bar.js');
3 let foo = require('./foo.js');
4 module.exports = function f(x){
5     bar.g(x);
6     foo.h(x)
7 }
```

Listing 4.1: Main module

```
1 // Bar module
2 let foo = require('./foo.js')
3 function g1(a){
4     foo.h(a);
5 }
6 module.exports = {
7     g:g1}
```

Listing 4.2: Bar module

```
1 // Foo module
2 function h1(b){
3     eval(b);
4 }
5 module.exports = {
6     h:h1
7 }
```

Listing 4.3: Foo module

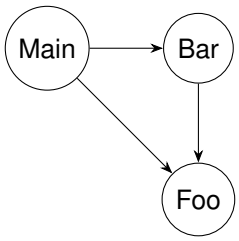

Figure 4.2: Dependency DAG Example (Main imports Bar and Foo, and Bar imports Foo)

4.2.1 Multi-file Example

Listings 4.1, 4.2, and 4.3 showcase a multi-file example that serves as the basis for illustrating multi-file support in the extended MDGs throughout this section. The example comprises three modules: `Main`, `Bar`, and `Foo`. In the Main module, both the `Foo` and `Bar` modules are imported using the `require` function. The `f(x)` function defined in the Main module invokes function `g` from the `Bar` module (`bar.g(x)`) and function `h` from the `Foo` module (`foo.h(x)`).

The `Bar` module imports the Foo module and defines function `g(a)`, which calls function `h` from the `Foo` module (`foo.h(a)`). It also exports function `g`.

The `Foo` module does not import any modules. It defines function `h(b)`, which uses the `eval` function. Function `h` is exported from the module.

4.2.2 Dependency DAG Construction

Before processing a file, we must generate the DAG of its dependencies. To achieve this, we use the NPM package *Dependency-tree* [26]. This package requires the following two inputs:

- ***Main File***: The primary file to be processed. This file serves as the entry point for the dependency analysis.
- ***Dependency Directory***: The directory in which to search for the package dependencies. This corresponds to the root directory of the package. We search for the dependencies recursively to include all nested dependencies within the directory.

After locating the dependencies, the *Dependency-Tree* package outputs the DAG, excluding Node.js built-in modules. These excluded packages typically correspond to sensitive sinks and are already represented as such in the graph, so they do not need to be included in the DAG. Additionally, the DAG includes all imported modules, whether they are used in the program or not. For example, Figure 4.2 depicts the DAG generated for the modules in the previous section.

Algorithm 1 `Get_Summary(MDG)`

```
summary ← Map()
for export ∈ MDG.exports do
    init ← export.init
    prop ← export.property
    match init do
        case Function
            summary[prop] ← init.name
        case Object
            summary[prop] ← BuildObj(init)
end for
return summary
```

Algorithm 2 `BuildObj(obj)`

```
object ← Map()
for prop ∈ obj.properties do
    init ← prop.init
    name ← prop.name
    match init do
        case Function
            object[name] ← init.name
        case Object
            object[name] ← BuildObj(init)
end for
return obj
```

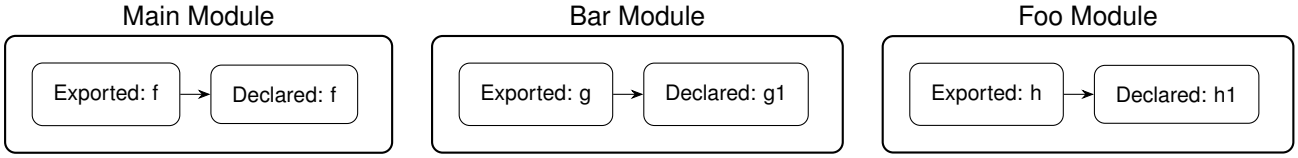

Figure 4.3: Export summaries for the running example

4.2.3 Export Summary

In addition to the dependency DAG, we need to summarize the object that each module exports, because a module might export a function with a different name that it was declared. To do this, we examine assignments to `module.exports`. For each assignment, we map the name under which the function was exported (the property name assigned to `module.exports`) to its original name. If the module exports an object, we recursively construct all properties of that object that are functions, creating the same mapping. The pseudo code for this algorithm can be found in Algorithm 1, which relies on Algorithm 2 to recursively construct the exported object.

For instance, Figure 4.3 illustrates the exported object summaries for the modules of the running example.

4.2.4 Multi-file MDG Generation

With the DAG and the export summaries, we can generate the Multi-file MDG using Algorithm 3. This algorithm uses two maps: *summaries* and *module_graphs*. The *summaries* map links modules to their corresponding exported object summaries, while the *module_graphs* map associates modules with their corresponding MDGs. Initially, these two objects are empty. Additionally, the algorithm requires the object *dag*, which corresponds to the dependency DAG explained in the previous section.

Figure 4.4 illustrates the Multi-file MDG generated for the running example. Its generation involves the following steps:

1. Start by generating the DAG illustrated in Figure 4.2.
2. Process the `Foo` module, as it is the sink file of the DAG. This step generates the MDG and the exported object summary depicted in Figure 4.3.

Algorithm 3 `MF-MDG(dep,dag,summaries,graphs)`

```
1: mdg ← NULL
2: // Process dependencies first
3: for dep ∈ dag[main] do
4:     MF-MDG(dep,dag,summaries,graphs)
5: end for
6: // Process the file
7: mdg ← Generate_MDG(main)
8: for call ∈ mdg.external_calls do
9:     module ← call.module
10:    func ← summaries[module][call.func]
11:    graph ← graphs[module].Get_Func(func)
12:    mdg.Add_External_Func(graph)
13: end for
14: module_graphs[main] ← mdg
15: summaries[main] ← Get_Summary(mdg)
16: return mdg
```

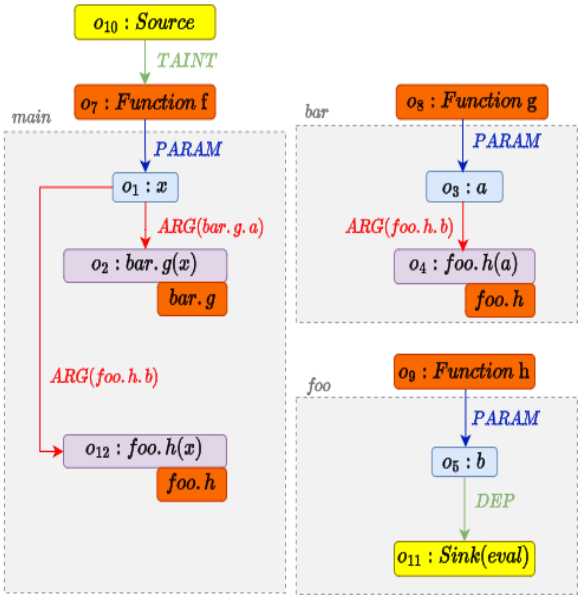

Figure 4.4: Multi-file MDG for the running example

3. Moving up the DAG, analyze the `Bar` module next. Since this module calls `Foo.h`, we include that function's MDG by looking for the mapping for `h` in `Foo`'s export object summary. Generate the MDG and the exported object summary presented in Figure 4.3.

4. Finally, process the `Main` module, which calls both `Foo.h` and `Bar.g`. Refer to the summaries of both `Foo` and `Bar` modules to include these functions' MDGs in the final MDG.

Summary

In this chapter we explained the changes required to allow Graph.js to support inter-procedural and multi-file support. First, we started by explaining the nodes and edges that compose the extended MDGs. Then, we outlined how the multi-file MDG is generated. In the next chapter, we will describe the new

detection queries to detect vulnerabilities in these extended MDGs

Chapter 5

Vulnerability Detection in the EMDGs

In the previous chapter, we discussed EMDGs and their effectiveness in modeling vulnerabilities. To detect vulnerabilities in these graphs, we need to identify taint paths that start from a *Taint Source* node and reach a sensitive sink. This requires connecting call chains across sub-graphs, which Graph.js's cannot do. Therefore, we introduce three new detection algorithms in this chapter that can connect these sub-graphs and detect vulnerabilities in the EMDGs. These algorithms use either a *Top-Down* or *Bottom-Up* approach to traverse the graph. Section 5.1 introduces the Cypher query language, which is essential to understand the queries mentioned throughout this chapter. Then, Section 5.2 covers the Top-Down approach and Section 5.3 presents the Bottom-Up approach. Finally, Section 5.4 discusses the soundness of these methods.

5.1 Cypher Language Queries

As mentioned in Section 2.3, after generating the EMDG, we import it into a Neo4j [10] database. Therefore the queries that we cover in the following sections are written in *Cypher* language and correspond to graph traversals. The key *Cypher* clauses are:

- ***MATCH***: The *MATCH* clause defines the pattern to search for in the graph, specifying nodes and relationships. When using multiple *MATCH* clauses or comma-separated patterns, all specified patterns must be matched for a result to be returned. In Listing 5.1, we define two patterns. The first pattern finds nodes of type *START_NODE* (referenced by *start*) connected to *END_NODE_1* nodes (referenced by *end_node1*) through *EDGE* relationships (with the list of edges stored in *edges1*). The second pattern checks if the same start node is also connected to *END_NODE_2* nodes (referenced by *end_node2*) through a possibly different set of *EDGE* relationships (stored in *edges2*).

- ***WHERE***: The *WHERE* clause filters results based on specified conditions. In Listing 5.1, we filter the start nodes to include only those whose *Id* contains the string *"Foo"*.

```
1  MATCH
2      (start:START_NODE)
3          -[edges1:EDGE*1..]
4              ->(end1:END_NODE_1),
5
6      (start)
7          -[edges2:EDGE*1..]
8              ->(end2:END_NODE_2)
9  WHERE
10     start.Id Contais "foo"
11 RETURN *
```

Listing 5.1: Example Query in Cypher Query Language (Neo4j)

- ***RETURN***: The *RETURN* clause specifies the data to retrieve. Use * to return all matched elements. In Listing 5.1, we return all possible results.

5.2 Top-Down Vulnerability Detection

In this section, we explain how to detect vulnerabilities in EMDGs using a *Top-Down* approach. This approach traces call chains from callers to callees, starting at the *Taint Source* node and moving toward sensitive sinks. To implement this, we use the *Top-Down Taint Query*, explained in Section 5.2.1, and Algorithm 4, explained in Section 5.2.2.

5.2.1 Top-Down Taint Query

The Top-Down Taint Query, as depicted in Listing 5.2, finds taint paths in the graphs that start in the node *start*, identified by its *Id*, to the node *sink* that can be of the following types:

- *Call Node*: indicates the need to connect two sub-graphs, as this node represents a function calling another.
- *Return Node*: indicates that the taint propagation has reached the return value of a function, which needs to be reported to correctly follow taint flows into call chains
- *Sink Node*: indicates the discovery of a valid path that must be reported.

For instance, in Figure 5.1, if the starting node corresponds to o_8 (Taint Source node), one of the paths returned by this query is the green path.

5.2.2 Top-Down Algorithm

The Top-Down Algorithm corresponds to Algorithm 4. This Algorithm works differently depending on where the query stops. If it stops at a call node, it connects paths by running the Top-Down Taint query on the sub-graph of the called function. If it stops at a return node or sensitive sink, it saves the path for later reporting. The algorithm uses two lists during execution: *results* and *work_list*. *Results* stores identified vulnerable paths, while *work_list* holds incomplete paths. Initially, *results* is empty, and *work_list* starts

```
1 MATCH path =
2     (start)
3         -[path_edges:MDG_EDGE*1..]
4             ->(sink:TAINT_SINK|RETURN_NODE|CALL_NODE)
5 WHERE
6     start.Id = <Id>
```

Listing 5.2: Top Down Taint Propagation Query in Cypher Query Language (Neo4j)

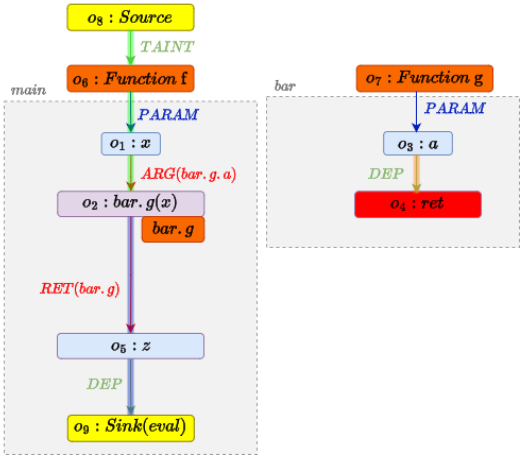

Figure 5.1: MDG to illustrate the Top-Down approach

with a single path containing only the Taint Source node. For example, to detect the vulnerability in the graph shown in Figure 5.1 we follow these steps:

1. We start by calling `Find_Taint_Paths` with `results` initialized as an empty list and `work_list` initialized with `[[`o_8`]]`, since o_8 is the Taint Source node.

2. Then, the algorithm pops the first path from `work_list`. Since the last node is `TAINT_SOURCE`, which matches the default case, it runs the Top-Down Taint query from this node. This query identifies the green path in the graph, updating `work_list` to `[[`o_8`, `o_6`, `o_1`, `o_2`]]` (lines 30-34).

3. Afterwards, it pops the first path from `work_list`. This time, the last node (o_2) is a call node. Thus, `func` becomes `bar.g` (based on the orange box below o_2) and `param` becomes o_3 (since the argument edge indicates that the parameter in question is the parameter `a`, which is represented by o_3). It then calls `Find_Taint_Paths` with `results` as an empty list and `work_list` as `[[`o_3`]]` (lines 13-16)

   3.1. It starts the recursion by popping the first path from `work_list`. The last node is o_3, representing a parameter, which matches the default case. Therefore, it executes the Top-Down Taint query from o_3, identifying the orange path in the graph. Update `work_list` to `[[`o_3`,`o_4`]]` (lines 30-34).

   3.2. The only path in `work_list` ends at a Return node (o_4). Therefore, it appends this path to the `results` list (lines 8-10). With no additional paths available, the recursion returns `[[`o_3`,`o_4`]]` (line 2)

4. Subsequently, it calls `Filter_Paths` on the results obtained from the previous step. This auxiliary function, depicted in Algorithm 5, separates paths ending in a sensitive sink from those ending in a Return node (line 17)

5. Since there is a path ending in the corresponding function's return node, the function propagates taint towards its return value. Thus, it combines the current path, the return path, and o_5, which

Algorithm 4 `Find_Taint_Paths(results,work_list)`

```
1: if work_list == [] then
2:     return results
3: else
4:     // Get current and remaining paths
5:     path,remaining_paths ← work_list.pop()
6:     node ← path.last()
7:     match node do
8:         case SINK|RETURN:
9:             results.append(path)
10:            return Find_Taint_Paths(results,remaining_paths)
11:        case CALL:
12:            // Get called function and param
13:            func ← node.func
14:            param_node ← get_param(func,node.previous())
15:            // Check taint propagation by the param
16:            param_paths ← Find_Taint_Paths([],[[param_node]])
17:            sink_paths,ret_path ← filter_paths(param_paths)
18:            sink_paths ← [path + cont | for cont in sink_paths]
19:            new_results ← sink_paths + results
20:            if ret_path ≠ NULL then
21:                // Param propagates taint, so continue the path
22:                full_path ← path + ret_path + node.ret
23:                new_worklist ← full_path + remaining_paths
24:                return Find_Taint_Paths(new_results, new_worklist)
25:            else
26:                // Param does not propagate taint, disregard the path
27:                return Find_Taint_Paths(new_results, remaining_paths)
28:            end if
29:        default:
30:            // Continue the path by running the taint query at node
31:            paths ← taint_query(node.Id)
32:            new_paths ← [path + cont | for cont in paths]
33:            new_worklist ← new_paths + remaining_paths
34:            return Find_Taint_Paths(results,new_worklist)
35: end if
```

represents the object storing the function call's return value (the variable `z`), updating `work_list` to `[[`o_8`,`o_6`,`o_1`,`o_2`,`o_3`,`o_4`,`o_2`,`o_5`]]` (lines 18-24).

6. After that, it continues the current path by executing the Top-Down Taint query from o_5, as we are in the default case again. The query finds the blue path in the graph, updating `work_list` to `[[`o_8`, `o_6`, `o_1`, `o_2`, `o_3`, `o_4`, `o_2`, `o_5`, `o_9`]]` (lines 30-34).

7. Finally, as the only path in `work_list` concludes at a `Taint Sink` node (o_9), add this path to the results (lines 8-10). With no further paths remaining, the algorithm's execution ends (line 2).

5.3 Bottom-Up Vulnerability Detection

Unlike the Top-Down approach, the Bottom-Up approach traces call chains in reverse, starting from the sinks and working back to the parameters of exported functions (i.e., functions connected to the

Algorithm 5 `Filter_Paths(paths,start)`

```
1: // Get the paths that end in a sink
2: sink_paths ← paths.filter((path) => is_sink(path.last()))
3: // Get the paths that end in a return node
4: ret_paths ← paths.filter((path) => is_ret(path.last()))
5: if ret_paths.length > 0 then
6:     ret_path ← ret_paths.get(0)
7: else
8:     ret_path ← NULL
9: end if
10: return sink_paths,ret_path
```

Taint Source node). In this section, we present two algorithms that use this approach, along with the queries that support their execution. Section 5.3.1 explains the Cypher queries required by the algorithms. Additionally, Section 5.3.2 overviews the algorithm that pre-processes the EMDG to detect vulnerabilities, while Section 5.3.3 describes the Greedy algorithm.

5.3.1 Bottom-up Queries

This section introduces the Cypher queries used in the Bottom-Up approach to vulnerability detection. Two key queries are required: the *Bottom-Up Taint Query* and the *Call Graph Query*.

Bottom-Up Taint Query The Bottom-Up Taint query, shown in Listing 5.3, identifies which function parameters lead to a sink. It starts at a node *func*, representing a function's parameter, and follows EMDG edges to a Taint Sink node *sink*, also capturing the corresponding parameter *param*. For example, in Figure 5.2, this query return the red path in the graph.

Call Graph Query The Call Graph query connects function parameters used as arguments in the caller to their corresponding parameters in the callee, building the program's call graph. This query is depicted in Listing 5.4 and consists of the following sub-queries:

- ***Caller Sub-Query*** (lines 3-7): This query finds which parameters of a function reach which arguments. It traces paths from a function node (*caller*) through a parameter (*argument*), to a call node (*call_node*). The argument is specified by the edge connecting the path to the call node. For instance, in Figure 5.2, this query returns the path $o_7 \xrightarrow{\text{PARAM}} o_1 \xrightarrow{\text{ARG(bar.f.a)}} o_2$. In this path, o_7 corresponds to *caller*, o_1 to *argument* and o_2 to *call_node*.
- ***Callee Sub-Query*** (lines 9-11): This query finds a function's parameters by locating edges that connect the function node (*callee*) to its parameter nodes (*param*). The *WHERE* clause filters the results by matching the argument's name from the last edge in *path_edges* (which corresponds to an argument edge) with the *param*'s name, linking callers to callees. If a *paramName* variable is provided, the query only includes *param* nodes that match *paramName*. For example, in Figure 5.2, if *paramName* is not specified, one of the paths that the query returns is the path $o_8 \xrightarrow{\text{PARAM}} o_4$, where o_4 (*param*) represents `Bar.f`'s parameter `a`, which was also found by the previous query.

```
MATCH
    (func:FUNCTION_NODE)
        -[ref_edge:REFERENCE_EDGE]
            ->(param:EMDG_VALUE_NODE)
                -[edges:EMDG_EDGE*1..]
                    ->(sink:TAINT_SINK)
WHERE
    ref_edge.RelationType = "param"
RETURN *
```

Listing 5.3: Bottom Up Taint Propagation Query in Cypher Query Language (Neo4j)

```
MATCH
    // caller sub-query
    (caller:FUNCTION_NODE)
        -[argument_edge:REF]
            ->(argument:EMDG_VALUE_NODE)
                -[path_edges:EMDG_EDGE*1..]
                    ->(call_node:CALL_NODE),
    // callee sub-query
    (callee:FUNCTION_NODE)
        -[param_edge:REF]
            ->(param:EMDG_VALUE_NODE)
WHERE
    argument_edge.RelationType = 'param' AND
    argument_edge.ParamIndex <> 'this' AND
    param.IdentifierName = LAST(path_edges).IdentifierName AND
    ( <paramIdentifierName> IS NULL OR
    param.IdentifierName = <paramIdentifierName> )
RETURN *
```

Listing 5.4: Build Call graph in Cypher Query Language (Neo4j)

5.3.2 Algorithm with Pre-Processing

The Bottom-Up algorithm with Pre-Processing first computes the transposed call graph using the Call Graph query. Then, we use the Bottom-Up Taint query to identify potential vulnerabilities by tracing which objects reach a sink. Finally, we confirm these vulnerabilities using the transposed call graph and Algorithm 6. For instance, to detect the vulnerability present in the graph in Figure 5.2, we follow these steps:

1. We start by constructing the transpose call graph (`CGT`), similar to the one shown in Figure 5.3.

2. Then, we run the Bottom-Up Taint query to find the red path in the graph. Subsequently, we call `Confirm_Vuln(`o_5`,CGT)`, as o_5 represents the function `bar.g`'s parameter `b` and this parameter reaches and sensitive sink.

3. At this step, the `stack` list is `[`o_5`]`. The algorithm pops the first value from the stack to get o_5, representing the parameter `b` of function `bar.g`. Since `b` is not a parameter of an exported function (i.e, a function connected to the Taint Source node), it adds the parameters that reach it by referencing the transposed call graph. The only parameter that reaches the parameter `b` is `bar.f`'s parameter `a`, represented by o_3. Finally, it updates `stack` to `[`o_3`]` (lines 3-8).

4. It repeats the previous step for o_3 (the parameter `a`), which is not a parameter of an exported function. After this step, the `stack` becomes `[`o_1`]` because o_1, representing `main.f`'s parameter `x`, is the only parameter that reaches the parameter `a` (lines 3-8).

5. o_1 (the parameter `x`) is a parameter of the exported function `main.f`. It reports the vulnerability and ends its execution (lines 4-5).

Algorithm 6 `Confirm_Vuln(param,CGT)`

```
1:  stack ← [param]
2:  while stack.length != 0 do
3:      caller ← stack.pop()
4:      if caller.isExported() then
5:          return True
6:      end if
7:      // Params reaching caller
8:      stack.append(CGT[caller])
9:  end while
10: return False
```

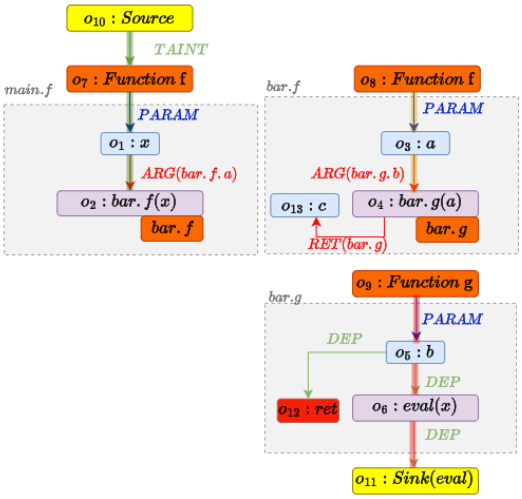

Figure 5.2: EMDG to illustrate the Bottom-Up approach

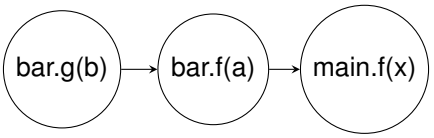

Figure 5.3: Transposed Call Graph Generated for the graph in Figure 5.2

5.3.3 Greedy Algorithm

The Bottom-Up Greedy algorithm connects only the necessary paths to confirm a vulnerability, caching them to avoid repetition. We start by identifying which parameters reach a sink using the Bottom-Up Taint query. Once we've found these parameters, we employ Algorithm 7 to confirm the vulnerability. Contrary to the Bottom-Up algorithm with pre-processing, Algorithm 7 does not build the transpose graph at the beginning but rather uses the Call graph query to build the transpose graph as needed. For instance, detecting the vulnerability present in the graph of Figure 5.2, involves the following steps:

1. We start by running the Bottom-Up Taint query to find the red path in the graph. Then, we call `Confirm_Vuln(`o_9`,`o_5`)`, as o_9 represents the function `bar.g` and o_5 represents its parameter `b` and this parameter reaches and sensitive sink.

2. Since o_9 (`bar.g`) is not exported (i.e, is not connected to the Taint Source node), the algorithm uses the Call Graph query to identify the parameters reaching o_5 (`bar.g`'s `b` parameter). This yields the yellow path in the graph. Thus, it calls `Confirm_Vulnerability(`o_8`,`o_3`)`, since o_8 represents the function `bar.f` and o_3 represents its parameter `a` (lines 4-9).

3. It repeats the previous step for o_8 (`bar.f`) and o_3 (`bar.f`'s `a` parameter), as `bar.f` is not exported. This step finds the green path in the graph. Call `Confirm_Vulnerability(`o_7`,`o_1`)`, as o_7 represents the function `main.f` and o_1 represents the parameter `x`, which is the only parameter that reaches the parameter `a` (lines 4-9).

4. Finally, it report the vulnerability since o_7 (`main.f`) is exported, and ends the algorithm (lines 1-2)

Algorithm 7 `ConfirmGreedy(func,param)`

```
1: if is_exported(func) then
2:     return True
3: else
4:     callers ← call_graph_query(param.name)
5:     for call ∈ callers do
6:         // caller is vuln ⟹ param is vuln
7:         if ConfirmGreedy(call.func,call.arg) then
8:             return True
9:         end if
10:    end for
11: end if
12: return False
```

5.4 Soundness Considerations

The algorithms discussed earlier detect vulnerabilities effectively with minimal false positives but are not sound, as they miss certain vulnerabilities. This section outlines how to make vulnerability detection as close to sound as possible.

Section 5.4.1 presents a vulnerability example that the current methods overlook. Section 5.4.2 explains the definition of an attacker-controlled object, which is why these algorithms are not sound. Section 5.4.3 proposes a a new definition of an attacker-controlled object to make the detection sound.

5.4.1 Motivating Example

Listing 5.5 demonstrates a scenario where our enhanced analysis fails to detect a vulnerability. The listing includes two functions, `g` and `f`, with function `g` being exported as the module's primary function. Function `g` locally instantiates an object `o` and assigns the value `33` to its `foo` property. It then calls function `f`, passing `x` and `o` as arguments. Inside function `f`, the parameter `y` assigns its value to `z.foo`, effectively modifying `o.foo` to match `x`'s value. The execution of `g` concludes with a call to `eval(o.foo)`, which attempts to evaluate the value of `o.foo`.

The vulnerability in Listing 5.5 arises from the use of `eval(o.foo)` in function `g`. Within function `g`, function `f` is called with arguments `x` and `o`. Inside function `f`, `z.foo` (which is `o.foo`) is set to the value of `y` (which has the same value as the attacker controlled variable `x`). As a result, the attacker can manipulate the content of `o.foo`, which is subsequently passed to `eval` in `g`, making it susceptible to code injection. The analysis overlooks this vulnerability because it fails to connect the assignment (`z.foo = y`) in `f` with the subsequent execution (`eval(o.foo)`) in `g`. In other words, it fails to determine that the `x` parameter propagates taint to the object `o.foo` through the call to the function `f`.

5.4.2 Unsound Attacker-Controlled Object Definition

As shown in the previous section, the earlier algorithms fail to detect the vulnerability in the example because they rely on the definition of an attacker-controlled object depicted in Algorithm 8. According to this definition, an attacker controls the parameters of the exported functions and the objects they can

```
1  function g(x) {
2      let o = {}
3      o.foo = 33;
4      f(x, o);
5      eval(o.foo);
6  }
7  function f(y,z){
8     z.foo = y;
9  }
10 module.exports = g
```

Listing 5.5: Overlooked Vulnerability Example

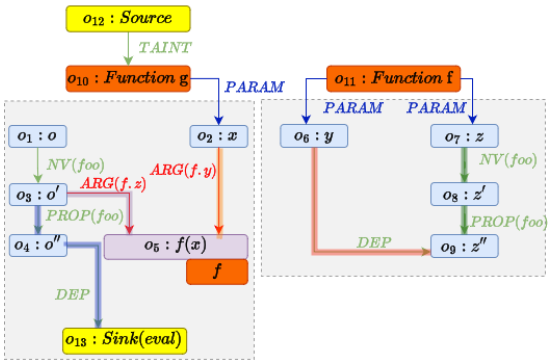

Figure 5.4: Corresponding MDG of Listing 5.5

Algorithm 8 `Controls(n)`

```
1:  // get the function that contains n
2:  f ← get_func(n)
3:  params ← get_params(f)
4:  if ∃ p ∈ params: Reaches(p,n) then
5:      if is_exported(f) then
6:          return True
7:      else
8:          callers ← get_calls(f,p)
9:          if ∃ call ∈ callers: Controls(call.arg) then
10:             return True
11:         end if
12:     end if
13: end if
14: return False
```

reach:

- ***Directly*** (line 4): Objects directly connected to parameters of the exported functions. As depicted in Algorithm 9, node `n1` reaches `n2` if there is a path connecting them via dependency, new version, and/or property edges.
- ***Indirectly*** (lines 8-9): Objects reachable from the parameters of the exported functions through function calls. In Algorithm 8, we follow the call chains using a bottom-up approach until we reach a parameter of an exported function to determine if the attacker controls the object.

However, this definition fails to capture the vulnerability shown in Figure 5.4. In the example, the node that represents the parameter `x` of the function `g` (o_2) does not have an explicit path (either direct or indirectly) to the node representing `o.foo` (o_4), even though taint is propagated through the call to `f(o,x)`.

5.4.3 New Attacker-Controlled Object Definition

To detect the vulnerability in the motivating example of Listing 5.5, we propose replacing the function `Reaches` (Algorithm 9) with `Reaches2.0` (Algorithm 10) in Algorithm 8. In Algorithm 10, `n1` reaches `n2` if they are direct or indirectly connected or if taint propagation occurs inside a function call.

Recalling the motivating example, we can now see that the node representing the parameter `x` of the function `g` (o_2) taints the node representing `o.foo` (o_4). In the call to `Reaches2.0`(o_2, o_4), although

Algorithm 9 `Reaches(n1,n2)`

1: **if** n1 $\xrightarrow{\texttt{(DEP/PROP/NV)}^+}$ n2 **then**
2: `return True`
3: **end if**
4: `return False`

Algorithm 10 `Reaches2.0(n1,n2):-`

1: `Reaches(n1,n2)` $\vee$
2: $\exists$ `n1',n2',q,p,call:`
3: `Reaches2.0(n2',n2)` $\&$ *// n2 sub-object n2'*
4: `n2'` $\xrightarrow{\texttt{ARG(q)}}$ `call` $\&$
5: `Reaches2.0(n1,n1')` $\&$ *// n1' depends on n1*
6: `n1'` $\xrightarrow{\texttt{ARG(p)}}$ `call` $\&$
7: `Reaches2.0(node(q),q')` $\&$ *// q' sub-object q*
8: `Reaches2.0(node(p),q')` $\&$ *// q' depends on p*
9: `PropsTraversed(n2',n2) == PropsTraversed(node(q),q');` *// the paths* `n2'` $\rightarrow$ `n2` *and* `q` $\rightarrow$ `q'` *have the same sequence of property edges*

line 1 of the algorithm is not satisfied (i.e, they are not directly connected), line 2 is satisfied with the following setup: o_2 as `n1` and `n1'`, o_4 as `n2`, o_3 as `n2'`, `y` as the parameter `p`, with o_6 as its node, and `z` as the parameter `q`, with o_7 as its node. Based on these assumptions, we verify the following conditions:

1. **`Reaches2.0(n2',n2)`** (line 3): We verify this condition because node o_3 (variable `o'`) connects to o_4 (`o.foo`) through a property edge (blue path in the graph).

2. **`n2'`** $\xrightarrow{\texttt{ARG(q)}}$ **`call`** (line 4): We verify this condition because node o_3 (`n2'`) connects to the call to `f` (o_5) (purple path in the graph).

3. **`Reaches2.0(n1,n1')`** (line 5): We verify this condition because o_2 corresponds to both `n1` and `n1'`

4. **`n1'`** $\xrightarrow{\texttt{ARG(p)}}$ **`call`** (line 6): We verify this condition because node o_2 (`n1'`) connects to the same call node found in 2 (o_5) (orange path in the graph).

5. **`Reaches2.0(node(q),q')`** (line 7): We verify this condition because node o_7, representing the variable `z` (`q`), connects to o_9 through the edges `NV(foo)` and `PROP(foo)` (green path in the graph).

6. **`Reaches2.0(node(p),q')`** (line 8): We verify this condition because node o_6, representing the variable `y` (`p`), connects to o_9 (`q'`) through dependency edges (red path in the graph).

7. **`PropsTraversed(n2',n2) == PropsTraversed(node(q),q')`** (line 9): We verify this condition because the paths from `n2'` to `n2` (blue path) and from `node(q)` to `q'` (green path) follow the same sequence of property edges.

Consequently, this analysis flags o_4 as tainted by o_2 (the parameter `x`). Since `x` is a parameter of the exported function `g`, we report the vulnerability.

Summary

In this chapter, we explored vulnerability detection in extended EMDGs. We discussed the Top-Down detection approach, outlining its algorithm and necessary queries. Then, we introduced the Bottom-Up approach, showcasing two algorithms and their corresponding queries for vulnerability detection. Finally, we discussed how to make our detection as sound as possible. In the next chapter, we will assess the effectiveness of our solution.

Chapter 6

Evaluation

In this chapter, we evaluate the effectiveness of our solution. First, in Section 6.1, we describe our experimental setup. Then, we address the following research questions:

- ***RQ1: Which of our three proposed algorithms is most effective for vulnerability detection?***
- ***RQ2: How much does our best algorithm improve detection over state-of-the-art tools?***
- ***RQ3: What is the impact of our new attacker-controlled object definition on the detection?***

6.1 Experimental Setup

In this section, we detail our experimental setup. First, we outline the benchmarks used for the evaluation (Section 6.1.2). Then, we discuss the tools employed for comparison with our solution (Section 6.1.3). Finally, we conclude this section by presenting our experimental environment (Section 6.1.1).

6.1.1 Experimental Environment

Our testbed consisted of a single 64-bit Ubuntu 22.04.3 server with 64GB of RAM and 2x Intel(R) Xeon(R) Gold 5320 2.2GHz CPUs. We set the total analysis timeout to five minutes.

6.1.2 Datasets

To address our research questions, we require two datasets: the *Combined* dataset and the *Collected* dataset. A summary of the vulnerabilities considered in these datasets is depicted in Table 6.1.

Collected Dataset: The Collected dataset consists of 32,137 popular real-world NPM packages retrieved from the NPM repository in September 2023. According to Snyk's guidelines, a package is considered popular if it had over 2,000 weekly downloads at the time of collection.

Dataset	Total	Excluded				Manually Added	Considered
		Unavailable	Incorrect Annotations	Duplicated	Out-of-Scope		
VulcaN	236	10	7	0	0	59	278
SecBench	601	10	71	38	98	150	534
Total	837	20	78	38	98	209	812

Table 6.1: Summary of the vulnerabilities considered in each dataset

Combined dataset (VulcaN and SecBench): The Combined dataset consists in the combination of our two ground truth datasets: the *VulcaN* dataset and the *SecBench* dataset.

The VulcaN dataset, introduced in Section 3.1.2, comprises 957 npm package versions containing real-world Node.js vulnerabilities. Similarly to the initial Graph.js evaluation [3], of the 957 packages, we selected all 174 that contain vulnerabilities that Graph.js targets. In other words, we selected packages that contain the following vulnerabilities: code injection, command injection, path traversal, and prototype pollution. These selected packages contain a total of 236 vulnerabilities. Out of the 236 vulnerabilities, we excluded 17 that either have incorrect annotations (e.g., the annotated vulnerability type is different from the correct type and the correct type is outside of our scope) or are located in an external imported package, whose source is unavailable for analysis.

The SecBench dataset, introduced in Section 3.1.2, contains 600 vulnerable packages (some of which may also be present in the VulcaN dataset). The SecBench dataset includes the same vulnerability types as VulcaN, along with Regular Expression Denial of Service (CWE-1333 [20]). Out of the 601 vulnerabilities, we excluded 217 vulnerabilities in total: 98 refer to out-of-scope ReDoS vulnerabilities, 71 are incorrectly annotated (e.g., non-existent or wrong sink line, or missing files), and 38 were already included in VulcaN. The rest of the excluded cases are either unavailable for download or their file type was TypeScript. While our methodology applies to TypeScript, Graph.js uses a JavaScript parser that cannot handle TypeScript. In the end, we were left with 384 vulnerabilities.

Combining these two datasets results in a dataset with 601 vulnerabilities. However, the datasets are incomplete. For instance, the SecBench dataset annotates only one vulnerability per package, even though more may exist. To address this, we identified additional vulnerabilities and their corresponding exploits, increasing the total count of the Combined dataset to 812 vulnerabilities.

Evaluation of the Combined Dataset for Precision: As previously noted, the Combined dataset contains only vulnerable packages with very few false positives, indicating a bias towards recall over precision (marking all packages as vulnerable achieves 100% recall, at a marginal cost in precision). Given this bias, it is essential to determine whether the dataset is appropriate for evaluating precision. To investigate this, we used a simple detection tool that applies regular expressions to search for sensitive sinks in every source file within the dataset. This tool likely achieves the highest possible recall, as it marks all sinks as vulnerable, regardless of their actual status. By adopting this approach, we can evaluate if prioritizing recall over precision leads to better performance in these datasets. To produce the results shown in Table 6.2, we collected the following metrics:

- ***FP (False Positives)***: Instances where the tool identified a vulnerability, but it wasn't exploitable

CWE	Total	Regex				
		TP	FP	Recall	Precision	F1
CWE-22	244	239	136	0.98	0.64	0.77
CWE-78	269	252	446	0.94	0.36	0.52
CWE-94	71	71	1694	1.00	0.04	0.08
CWE-1321	228	217	3016	0.95	0.07	0.13
Total	812	781	5282	0.96	0.13	0.23

Table 6.2: Detection results using Regular Expressions in the Combined dataset

or lacked a successful exploit. Since the datasets are incomplete and there might be more vulnerabilities present in the dataset then the ones that are annotated.

- ***TP (True Positives)***: Instances where the vulnerability type and sink line number reported by the tool match the dataset annotations.

- ***Precision*** = $\frac{\text{TP}}{\text{TP + FP}}$

- ***Recall*** = $\frac{\text{TP}}{\text{TP + FN}}$

- ***F1 Score*** = $\frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$

From these results, we conclude that the tool achieved a near 100% recall across all CWEs, with a global recall of 96%. The false negatives in the regex tool's results contributed to this recall of only 96%, despite the expectation of 100%. This occurred because some vulnerabilities did not match the defined regex patterns due to variations in format or unexpected input. However, this high recall still underscores the tool's effectiveness in identifying vulnerabilities within the dataset. Regular expressions excel at detecting calls to sensitive sinks due to their ability to efficiently search for specific patterns or structures within code, making them particularly well-suited for identifying potential security vulnerabilities.

However, the results also reveal low precision across most CWEs, except for CWE-22 (Path Traversal) which presented a 64% precision. Globally, the tool achieved a precision of 13% with over 5k false positives. This low precision can be attributed to the inherent limitations of the regular expression approach in accurately distinguishing between vulnerable and non-vulnerable code sinks.

Based on these findings, we conclude that the dataset is suitable for evaluating precision across all CWEs except for CWE-22 (path-traversal). In the case of CWE-22, a tool that marks every sink as vulnerable has a 64% chance of correctly classifying that sensitive sink. However, globally speaking, the dataset is adequate for evaluating precision because a tool that likely achieves the highest possible recall by blindly marking all sinks as vulnerable, performs poorly in terms of precision on these datasets.

Therefore, tools that achieve good recall and have a precision value above 13% demonstrate good performance within these datasets. This threshold serves as a benchmark for the evaluation conducted to address the research questions.

CWE	Top-Down			Bottom-Up Pre-Processing			Bottom-Up Greedy		
	Recall	Precision	F1	Recall	Precision	F1	Recall	Precision	F1
CWE-22	0.95	0.84	0.89	0.97	0.87	0.92	0.95	0.86	0.90
CWE-78	0.94	0.95	0.94	0.93	0.97	0.95	0.94	0.95	0.94
CWE-94	0.77	0.80	0.78	0.87	0.84	0.85	0.77	0.82	0.79
CWE-1321	0.46	0.65	0.54	0.54	0.70	0.61	0.56	0.70	0.62
Total	0.80	0.84	0.82	0.82	0.84	0.83	0.82	0.85	0.84

Table 6.3: Detection results in the Combined dataset

6.1.3 Baseline Tools

The evaluation also includes a comparison between its current version, Graph.js, and another state-of-the-art tool, ODGen. We compare with the current version of Graph.js to assess improvements and verify whether the updates have enhanced its performance. Additionally, we chose ODGen because it employs a similar detection approach and Brito *et al.* [4] elected it as the tool that offers the most favorable trade-off between effectiveness and precision.

6.2 RQ1: Which of our three proposed algorithms is most effective for vulnerability detection?

In this section, we evaluate our three algorithms (Top-Down, Bottom-Up with Pre-Processing and Bottom-Up Greedy) using the Combined dataset, as it is our ground truth dataset. We start with our evaluation methodology (Section 6.2.1), followed by a comparison of the results (Section 6.2.2).

6.2.1 Evaluation Methodology

To address this research question, we evaluated all algorithms against every package included in the Combined dataset. Specifically, for each package, we ran the tools on the files containing the reported vulnerabilities. Each run consisted of two phases: *(i)* graph construction, which converts the analyzed file into its corresponding EMDG representation, and *(ii)* detection, which executes the detection algorithm over the resulting EMDG to identify vulnerabilities. We then measured the number of *True Positives* (TP) and *False Positives* (FP) for all the tools in the datasets, using the same metrics explained above for the regular expression-based tool. Using these metrics, we calculated each tool's *Precision*, *Recall*, and *F1-score*.

6.2.2 Results

Table 6.3 depicts the results of each algorithm across the dataset. Overall, all algorithm exhibit good performance, surpassing the precision obtained by the tool employing regular expressions (13%) and

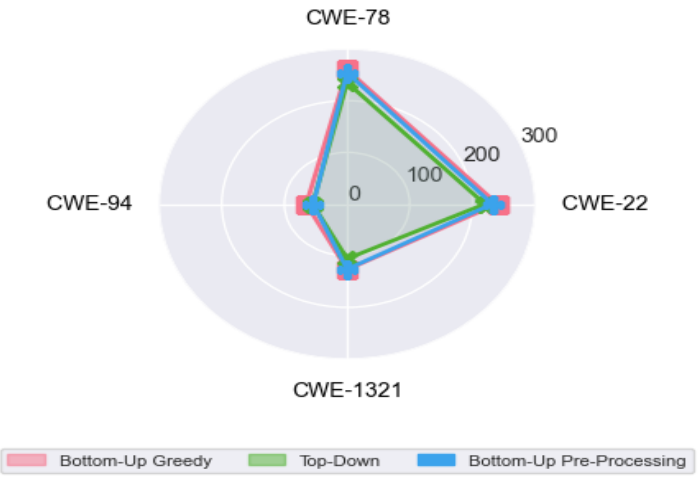

Figure 6.1: Comparison of the vulnerabilities detected in the Combined dataset

```
1  git.status = function(repoPath, file) {
2    var gitTask = git('status -s -b -u ' +
3    (file ? '"' + file + '"' : ''), repoPath)
4      .parser(gitParser.parseGitStatus)
5      .fail(task.setResult)
6      .done(function(status) {
7        status.inRebase = fs.existsSync(
8        path.join(repoPath, '.git', 'rebase-merge'))
9        || fs.existsSync(path.join(repoPath, '.git',
           ↪  'rebase-apply'));
10       status.inMerge = fs.existsSync(path.join(
           ↪ repoPath, '.git', 'MERGE_HEAD'));
11       if (status.inMerge) {
12         status.commitMessage = fs.readFileSync(
13         path.join(repoPath, '.git', 'MERGE_MSG'),
             ↪ { encoding: 'utf8' });
14       }
15     });
16 }
```

Listing 6.1: Code snippet from the package *ungit-0.8.4* to illustrate a False positive

```
1  var isExtendable = require('is-extendable
       ↪ ');
2  var forIn = require('for-in');
3
4  function mixinDeep(target, objects) {
5    var len = arguments.length, i = 0;
6    while (++i < len) {
7      var obj = arguments[i];
8      if (isObject(obj)) {
9        forIn(obj, copy, target);
10     }
11   }
12   return target;
13 }
```

Listing 6.2: Code snippet from the package *mixin-deep-1.3.0* to illustrate a False Negative

demonstrating good recall. Figure 6.1 shows that the number of vulnerabilities detected by each algorithm is similar.

Among the algorithms, the Greedy Bottom-Up approach stands out with the best balance between recall and precision, achieving scores of 82% and 85%, respectively. Following closely is the Bottom-Up with pre-processing, which achieves 82% recall and 84% precision, making it the second best approach for detecting vulnerabilities. The Top-Down approach achieves a recall of 80% and a precision of 84%, demonstrating its effectiveness in identifying vulnerabilities despite its slightly lower recall compared to the Bottom-Up algorithms.

False Positive Analysis: Despite having good precision and recall, all algorithms still suffer from false positives. Among the possible reasons, we have the following:

- ***The Require Function***: Our analysis mistakenly labels the `require` function as a sink, suggesting it always leads to code injection. This is not always accurate. Code injection through the `require`

function only happens if an attacker can both import a malicious package and execute an exported function from that package with arguments they control. Without this specific condition, marking the `require` function as a vulnerability is a false positive.

- ***Lack of Comprehensive Information on Function Calls***: While we have accurately modeled function calls within the application's own modules, we lack detailed insights into built-in Node.js modules. This limitation is evident in path traversal vulnerabilities, as shown in Listing 6.1. In line 11 of this listing, the attacker controls only the directory being read because they control the `repoPath` variable in the call to `path.join(repoPath, '.git', 'MERGE_MSG')`. Consequently, the analysis flags this line as vulnerable. However, since the attacker does not control the filename, there is no risk of a path traversal vulnerability.

- ***Reporting of Recursive Object Assignments***: While these assignments are related to the detection of prototype pollution vulnerabilities, they do not directly pollute `Object.prototype`. Instead, they may create conditions that could enable prototype pollution. Therefore, identifying these assignments as direct sinks is incorrect, leading to an overestimation of the vulnerability.

False Negative Analysis: Besides false positives, the algorithms also exhibited false negatives in the datasets due to the following factors:

- ***Incomplete Support for Javascript Features***: Our tool does not support all JavaScript features, particularly those involving the `arguments` and `this` keywords. This limited support leads to gaps in the dependency graph, causing the analysis to overlook vulnerabilities. For instance, in scenarios like the call to `this.foo(x,y)`, the tool fails to establish connections between the call's arguments and the call node because it cannot accurately determine the called function `foo`. This results in broken taint paths, which in turn causes the analysis to miss detecting vulnerabilities.

- ***Prototype Pollution Patterns***: Prototype pollution patterns often involve third-party NPM packages like *for-own* [27] and *for-in* [28], as illustrated by Listing 6.2. Since the code of these external packages is not represented in the graph, the tool fails to recognize associated vulnerability patterns. Additionally, prototype pollution sources frequently employ the `arguments` keyword, further complicating detection due to the lack of full support.

6.3 RQ2: How much does our best algorithm improve detection over state-of-the-art tools?

In this section, we assess the impact of our new inter-procedural/multi-file queries on the detection. We compare our Bottom-Up Greedy approach with the current version of Graph.js and ODGen using the Combined dataset. We also compare the number of vulnerabilities reported by the Bottom-Up Greedy approach and the current version of Graph.js in our Collected dataset, because we expect the Bottom-Up

CWE	Total	ODGen					Bottom-Up Greedy				
		TP	FP	Recall	Precision	F1	TP	FP	Recall	Precision	F1
CWE-22	244	131	8	0.54	0.94	0.69	231	38	0.95	0.86	0.90
CWE-78	269	151	29	0.56	0.84	0.67	254	13	0.94	0.95	0.94
CWE-94	71	24	113	0.34	0.18	0.24	55	12	0.77	0.82	0.79
CWE-1321	228	37	21	0.16	0.64	0.26	127	57	0.56	0.70	0.62
Total	812	343	171	0.42	0.67	0.52	667	120	0.82	0.85	0.84

Table 6.4: Comparison of ODGen and Bottom-Up greedy approach on the Combined dataset

CWE	Total	Graph.js					Bottom-Up Greedy				
		TP	FP	Recall	Precision	F1	TP	FP	Recall	Precision	F1
CWE-22	244	235	47	0.96	0.83	0.89	231	38	0.95	0.86	0.90
CWE-78	269	255	13	0.95	0.95	0.95	254	13	0.94	0.95	0.94
CWE-94	71	61	21	0.86	0.74	0.80	55	12	0.77	0.82	0.79
CWE-1321	228	132	60	0.58	0.63	0.63	127	57	0.56	0.70	0.62
Total	812	683	141	0.83	0.83	0.83	667	120	0.82	0.85	0.84

Table 6.5: Comparison of Graph.js and Bottom-Up greedy approach Combined dataset

Greedy approach to report fewer vulnerabilities than the current version of Graph.js. We start with our evaluation methodology (Section 6.3.1), followed by a comparison of the results (Section 6.3.2).

6.3.1 Evaluation Methodology

To address this research question, we conducted the following experiments:

- ***Evaluation on the Combined dataset***: We ran the current version of Graph.js and ODGen on these datasets, using the same methodology as for the first research question. We collected the same metrics for consistency.
- ***Evaluation on the Collected dataset***: We ran our best approach, Bottom-Up Greedy, alongside the current version of Graph.js on a subset of 5003 packages identified as vulnerable by Graph.js. This subset was sufficient because our new algorithms only detected vulnerabilities within those already found by Graph.js. We compared the number of reported vulnerabilities and the average package analysis time. Graph.js analyzed each package file-by-file, as it does not support multi-file analysis. The Bottom-Up Greedy approach analyzed each package both file-by-file and by entry points, identified through the *main* attribute in the package's *package.json* file or defaulted to *index.js*. We did not run ODGen on this dataset due to its long runtime.

6.3.2 Results

Comparison with ODGen: Table 6.4 shows the results of ODGen and the Bottom-Up Greedy approach on the Combined dataset. When compared to ODGen, the Bottom-Up Greedy approach achieved a 40% increase in recall and a 18% increase in precision, with approximately 30% fewer false positives.

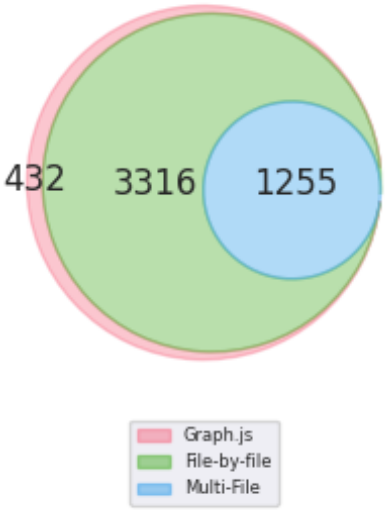

Figure 6.2: Comparison of the vulnerable packages reported in the Collected dataset

Tool	Vulnerabilities	Vulnerable Packages	Avg Analysis Time
Graph.js	14186	5003	15.110s
Bottom-Up Greedy (file-by-file)	13894	4571	15.799s
Bottom-Up Greedy (multi-file)	2327	1255	12.323s

Table 6.6: Results of the evaluation on the Collected dataset

Comparison with Graph.js (Combined dataset): Table 6.5 shows the results of Graph.js and the Bottom-Up Greedy approach on the Combined dataset. The Bottom-Up Greedy approach, despite a 1% decrease in recall, outperformed the current version of Graph.js with a 2% increase in precision and a 15% reduction in false positives. This modest increase in precision was anticipated due to the dataset's bias towards recall, as discussed in Section 6.1.2. The dataset lacks many non-vulnerable sinks (which tools might mistakenly identify as vulnerable), thus limiting the potential reduction in false positives by this approach. Nevertheless, these results already demonstrate an improvement over the current Graph.js version.

Comparison with Graph.js (Collected dataset): Table 6.6 displays the evaluation results of the collected dataset. The Bottom-Up Greedy algorithm, when applied in a file-by-file analysis, reduced reported vulnerabilities by approximately 300 compared to the current Graph.js version, with minimal impact on average package analysis time. In multi-file analysis, the algorithm decreased reported vulnerabilities by about 83% and was approximately 3 seconds faster on average than both its file-by-file counterpart and the current Graph.js version. However, we do not have the ground truth for this dataset, so we must manually analyze a sample of the reported vulnerabilities to ensure accuracy.

Analysis of Collected Dataset Vulnerabilities: To ensure that the reduction in reported vulnerabilities was not due to missed ones, we randomly sampled and manually reviewed 40 vulnerabilities detected

CWE	Graph.js		File-by-File		Multi-file		Total TP
	TP	FP	TP	FP	TP	FP	
CWE-22	1	9	0	10	9	1	10
CWE-78	1	9	1	9	9	1	11
CWE-94	0	10	1	9	8	2	9
CWE-1321	1	9	3	7	7	3	11
Total	3	37	5	35	33	7	41

Table 6.7: Sampling results in the collected dataset

CWE	Graph.js			File-by-File			Multi-File		
	Recall	Precision	F1	Recall	Precision	F1	Recall	Precision	F1
CWE-22	1	0.33	0.50	0.90	0.45	0.6	0.90	0.90	0.90
CWE-78	1	0.37	0.53	0.91	0.50	0.65	0.82	0.90	0.86
CWE-94	1	0.30	0.46	1	0.45	0.62	0.89	0.80	0.84
CWE-1321	1	0.37	0.54	0.91	0.50	0.65	0.64	0.70	0.67
Total	1	0.34	0.51	0.93	0.48	0.63	0.81	0.83	0.82

Table 6.8: Sample detection metrics in the collected dataset

only by Graph.js (red circle in Figure 6.2), 40 detected only by the file-by-file approach (green circle, excluding the blue circle), and 40 detected only by the multi-file approach (blue circle). We ensured that no vulnerabilities overlapped between these groups. Table 6.7 shows the results of this manual evaluation.

As shown in Figure 6.2, the vulnerabilities found by the multi-file approach are a subset of those detected by the file-by-file method, which are also a subset of those detected by Graph.js. This means that any true positives found by the multi-file approach are also detected by the file-by-file method and Graph.js and the true positives detected by the file-by-file approach are also detected by Graph.js. On the other hand, true positives detected only by Graph.js are false negatives for the other two methods, and those detected only by the file-by-file method are false negatives for the multi-file approach. Based on this, we estimated the Recall, Precision, and F1-score for each method, as shown in Table 6.8. Since we are treating Graph.js as our ground truth, we assumed its recall to be 1 (the highest possible recall) in order compute its F1-score.

The multi-file approach significantly improved precision, increasing it from 34% with Graph.js to 83%. It also improved precision compared to the file-by-file method, increasing from 48% to 83%. It also has a good recall of 81%, demonstrating that the reduction in false positives was not mostly due to missed vulnerabilities.

Additionally, we concluded that the reduction of the reported vulnerabilities is mostly due to the following reasons:

- ***Safe sinks in Imported Files***: Listings 6.4 and 6.3 illustrate this issue in the package *cpuprofile-webpack-plugin-1.10.3*. Both Graph.js and the Bottom-Up Greedy in a file-by-file analysis report line 5 of Listing 6.4 as as vulnerable to a path-traversal vulnerability. However, this is not an actual vulnerability. By inspecting Listing 6.3, we see that the call to the exported functions has no arguments, so both parameters hold the value `undefined`. Consequently, all subsequent calls have

```
1 // index.js
2 require("./profiler.js")();
3 ...
```

Listing 6.3: *Index.js*

```
1  // profiler.js
2  const cpuProfiler = require("sync-cpuprofiler");
3  function writeProfileFiles(profilePath, options) {
4    function onProfileDone(profilePath) {
5      const profile = readFileSync(profilePath, "utf-8");
6      ...
7    }
8    cpuProfiler(profilePath, Object.assign({}, options, {
       ↪ onProfileDone }));
9  }
10 module.exports = (profilePath, options) => {
11   try {
12     return writeProfileFiles(profilePath, options);
13   } catch (e) {
14     console.log("Writting profile failed", e);
15   }
16 };
```

Listing 6.4: *Profile.js*

Figure 6.3: Code snippet from the the *cpuprofile-webpack-plugin-1.10.3* package

```
1 var PNG = require("png-js");
2 var charm = require("charm");
3 var x256 = require("x256");
4 var buffers = require("buffers");
5 var es = require("event-stream");
6 var Stream = require("stream").Stream;
7 ...
```

Listing 6.5: *Index.js*

```
1  ...
2  var tube = require("../")(argv);
3  var fs = require("fs");
4  var file = argv._[0];
5  if (file === "-") {
6    process.stdin.pipe(tube);
7  } else {
8    fs.createReadStream(file).pipe(tube);
9  }
10 tube.pipe(process.stdout);
11 ...
```

Listing 6.6: *Tube.js*

Figure 6.4: Code snippet from the *picture-tuber-2.0.0* package

their parameters holding the value `undefined`, preventing the attacker from controlling the variable `profilePath` in the callback `onProfileDone`. The Bottom-Up Greedy approach doesn't report this package as vulnerable when using a multi-file approach

- ***Incorrect Inter-procedural analysis***: Listing 6.7 illustrates this issue in the package *node-watch-0.7.4*. Graph.js reports line 15 as vulnerable to path traversal. However, the only call to that method is on line 26, with the list containing only the variable `file`. The `file` variable is a string constructed using static variables (like `TEMP_DIR`) and random numbers, so the attacker cannot influence this variable. Therefore, line 15 is not vulnerable to path-traversal, as indicated by the Bottom-Up Greedy approach with both the file-by-file and multi-file analysis.

- ***Unused Files***: Listings 6.5 and 6.6 illustrate this issue in the package *picture-tuber-2.0.0*. Listing 6.6 depicts a path-traversal vulnerability in the *tube.js* file. However, as Listing 6.5 suggests, this file is never imported, therefore is not vulnerable. For this reason, the Bottom-Up approach doesn't report this package as vulnerable when using a multi-file approach, contrary to Graph.js and the Bottom-Up Greedy approach in a file-by-file analysis.

These findings highlight the effectiveness of combining inter-procedural queries with multi-file analysis,

```
var TEMP_DIR = os.tmpdir && os.tmpdir()
|| process.env.TMPDIR
|| process.env.TEMP
|| process.cwd();
TempStack.prototype = {
  create: function(type, base) {
    var name = path.join(base,
      'node-watch-' + Math.random().toString(16).substr(2)
    );
    this.stack.push({ name: name, type: type });
    return name;
  },
  write: function(files) {
    for (var i = 0; i < files.length; ++i) {
      fs.writeFileSync(files[i], ' ');
    }
  },
  ...
};
module.exports = function hasNativeRecursive(fn) {
var stack = new TempStack();
var parent = stack.create('dir', TEMP_DIR);
var child = stack.create('dir', parent);
var file = stack.create('file', child);
watcher.on('change', function(evt, name) {
    stack.write([file]);
});
...
}
```

Listing 6.7: Code snippet package *node-watch-0.7.4*

significantly reducing the reported vulnerabilities. By analyzing packages from their entry points and employing multi-file analysis, the tool avoids marking sensitive sinks in inner files that are never reached from entry points as vulnerable, thereby greatly minimizing incorrect vulnerability reports.

6.4 RQ3: What is the impact of our new attacker-controlled object definition on the detection?

In this section, we assess the impact of our new attacker-controlled object definition on vulnerability detection. We start with our evaluation methodology (Section 6.4.1), followed by a comparison of the results (Section 6.4.2).

6.4.1 Evaluation Methodology

To address this research question, we applied the Bottom-Up Greedy approach with the our new attacker-controlled object definition to all packages in the collected dataset using a multi-file approach. The entry points were identified in the same manner as for the previous research question. We then compared the number of reported vulnerabilities and the average package analysis time with the results obtained without using this new definition.

Tool	Vulnerabilities	Avg Analysis Time
Bottom-Up Greedy (unsound defintion)	2327	12.323s
Bottom-Up Greedy (new defintion)	3017	14.121s

Table 6.9: Results of the evaluation on the new definition on the Collected dataset.

Set	TP	FP	Precision
RQ2	1931	396	0.83
Newly reported vulnerabilities	575	115	0.83
Estimate	2506	511	0.83

Table 6.10: Precision with the new attacker-controlled object definition in the Collected dataset

6.4.2 Results

Table 6.9 shows the evaluation results. The number of reported vulnerabilities increased by about 30%, and the average package analysis time increased by approximately 1.8 seconds due to running additional queries.

Analysis of the Reported Vulnerabilities: To confirm that the increase in reported vulnerabilities wasn't just due to false positives, we randomly selected and manually reviewed 120 newly reported vulnerabilities, the results are summarised in Table 6.10 Of these, 100 (83%) were true positives, and 20 (17%) were false positives. Based on this, we estimate that out of the 690 newly reported vulnerabilities, 575 are true positives and 115 are false positives. Using the previously estimated precision of 83%, the prior analysis had 1931 true positives and 396 false positives, which are a subset of the vulnerabilities detected using the new definition. This brings the total to 2506 true positives and 511 false positives, maintaining the same precision of 83%. Thus, we detected more vulnerabilities without sacrificing precision.

Summary

In this chapter, we assessed the effectiveness of our proposed solutions. Initially, we examined the datasets to ensure their suitability for evaluating the precision of the tools and set a precision threshold of 13%. Subsequently, we measured the precision, recall, and f1-score of the original Graph.js, all three new algorithms, and ODGen. Our evaluation led us to conclude that Graph.js with the Bottom-Up greedy approach offers the best trade-off between precision and recall, surpassing the 13% precision threshold and the current version of Graph.js. On the collected dataset, the Bottom-Up Greedy approach was able to reduce both the average package analysis time (by approximately 3 seconds) and the number of reported vulnerabilities (by 83%), without sacrificing recall. Finally, the new attacker-controlled object definition increases the number of reported vulnerabilities by 30%, while being able to detect the vulnerabilities that were not detected previously without sacrificing precision. Amongst the vulnerabilities detected by our new algorithms, two of them were assigned CVEs at the time of writing this thesis. The subsequent

chapter will provide a summary of the achievements outlined in this thesis and suggest potential avenues for future research.

Chapter 7

Conclusions and Future Work

This chapter brings the document to a close by summarizing the main findings and conclusions drawn from this thesis (Section 7.1). Then, we identify potential areas for future research and development, exploring options for extending the current work (Section 7.2).

7.1 Conclusions

In this work, we addressed the challenges and vulnerabilities inherent to detect vulnerabilities in Node.js applications. 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 inter-procedural analysis and multi-file support. These limitations lead to an increased number of false positives. To addressed this issue, this work introduced modifications that enhance Graph.js' accuracy and reduce the occurrence of false positives. More concretely, the contributions of this work are the following:

1. ***Extended MDGs***: To accomplish this, we expanded the Graph Constructor Module to generate an Extended Multi-version Dependency Graph (EMDG). We improved inter-procedural analysis by introducing additional nodes, such as call and return nodes, and by adding new edges, like argument and return edges. To handle the multi-file challenge, the EMDGs combine graphs from various modules into one unified graph, which consists of interconnected sub-graphs.

2. ***New Detection Algorithms***: We developed three new algorithms: the Top-Down algorithm, the Bottom-Up with Pre-processing algorithm and the Bottom-Up Greedy algorithm. Each algorithm

finds paths from the program's sources to sensitive sinks, differing only in how they navigate the graph to identify these paths.

3. ***New Attacker-Controlled Object Definition***: While our primary goal is to reduce the false positives reported by Graph.js, we also propose a new attacker-controlled object definition. This definition is designed to enable the tool to identify additional vulnerabilities that it previously missed, by accounting for various methods of taint introduction that were overlooked before.

4. ***Evaluation on the Combined dataset***: We evaluated on the combination of two ground truth datasets: VulcaN [4] and SecBench [5]. The Bottom-Up approaches outperformed the Top-Down approach, with the Bottom-Up Greedy approach being the best, achieving an 82% recall and 85% precision. Compared to the current version of Graph.js, the Bottom-Up Greedy approach improved upon it by improving precision by 2%, worsening recall by 1%, and reports 15% fewer false positives. Against ODGen, it showed a 40% improve in recall, a 18% improve in precision, and about 30% fewer false positives, which is an improvement over ODGen.

5. ***Evaluation on the Collected dataset***: Additionally, we evaluated the tool using a dataset of real-world NPM packages. In this dataset, the Bottom-Up Greedy approach reported 83% fewer vulnerabilities compared to the current version of Graph.js, while also being 3 seconds faster on average in package analysis time. We estimate that the precision and recall for this dataset are 83% and 81%, respectively. Notably, with the new attacker-controlled object definition, the number of vulnerabilities reported increases by 30%, yet the precision remains the same.

In summary, this work offered an enhanced version of Graph.js with inter-procedural and multi-file reasoning, addressing its limitations and transforming it into a more robust tool for identifying vulnerabilities in Node.js applications.

7.2 Future Work

Despite the contributions of this work, there is still room for further improvement in Graph.js. Moving forward, several options for future work can be explored:

- ***Expansion to Other Vulnerability Types***: While this work focused on reducing false positives in Injection and Prototype Pollution Vulnerabilities, there are numerous other types of vulnerabilities that Graph.js could potentially detect. Future research could involve extending the tool's capabilities to encompass a broader range of vulnerability types, such as Regular Expression Denial of Service (ReDos) and Cross-Site Scripting (XSS), thereby enhancing its utility and relevance in real-world scenarios.

- ***Integration with Additional Datasets***: Expanding the testing and validation of Graph.js by integrating it with additional datasets beyond SecBench, VulcaN and the Collected datasets could provide further insights into its performance and effectiveness across diverse application scenarios. This

could involve testing on larger datasets, datasets from different domains, or datasets specifically tailored to capture emerging threats and vulnerabilities.

- ***Refinement of Analysis Techniques***: Continuously refining and improving the analysis techniques used by Graph.js can help further reduce false positives. This could involve fine-tuning algorithms, enhancing contextual analysis, or incorporating more sophisticated heuristics to differentiate between genuine vulnerabilities and false positives more accurately.

By pursuing these possibilities for future work, Graph.js can continue to evolve as a cutting-edge tool for vulnerability detection in Node.js applications, driving advancements in software security and enhancing the resilience of modern software systems.

Bibliography

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

[2] Node package manager. https://www.npmjs.com, 2023. Accessed: 2023-10-03.

[3] M. Ferreira, M. Monteiro, T. Brito, M. E. Coimbra, N. Santos, L. Jia, and J. F. Santos. Efficient static vulnerability analysis for javascript with multiversion dependency graphs. *Proc. ACM Program. Lang.*, 8(PLDI):25, June 2024. doi: 10.1145/3656394. URL https://doi.org/10.1145/3656394.

[4] 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*, pages 1–16, 2023. doi: 10.1109/TR.2023.3286301.

[5] 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.

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

[8] 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/, 2023. Accessed: 2023-10-03.

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

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

[11] CWE-78: Improper Neutralization of Special Elements used in an OSCommand ('OS Command Injection'). https://cwe.mitre.org/data/definitions/78.html, 2023.

[12] CWE-94: Improper Control of Generation of Code ('Code Injection'). https://cwe.mitre.org/data/definitions/94.html, 2024.

[13] CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('PathTraversal'). https://cwe.mitre.org/data/definitions/22.html, 2023.

[14] CWE-1321: Improperly Controlled Modification of Object Prototype Attributes ('Prototype Pollution'). https://cwe.mitre.org/data/definitions/1321.html, 2023.

[15] 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.

[16] 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. URL https://api.semanticscholar.org/CorpusID:51951699.

[17] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner. Containing malicious package updates in npm with a lightweight permission system. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1334–1346, 2021. doi: 10.1109/ICSE43902.2021.00121.

[18] Typescript. https://www.typescriptlang.org, 2023. Accessed: 2024-1-9.

[19] Codeql. https://github.com/github/codeql, 2023. Accessed: 2023-10-29.

[20] CWE-1333: Inefficient Regular Expression Complexity. https://cwe.mitre.org/data/definitions/1333.html, 2023.

[21] Jsjoern. https://github.com/malteskoruppa/phpjoern, 2024. Accessed: 2023-10-29.

[22] 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*, ACSAC '19, page 257–269, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450376280. doi: 10.1145/3359789.3359813. URL https://doi.org/10.1145/3359789.3359813.

[23] 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, 2023. doi: 10.1109/SP46215.2023.10179352.

[24] 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.

[25] M. Ohm, T. Pohl, and F. Boes. You can run but you can't hide: Runtime protection against malicious package updates for node.js, 2023.

[26] Dependency tree. https://www.npmjs.com/package/dependency-tree, 2024. Accessed: 2024-06-06.

[27] Npm package for-own. https://www.npmjs.com/package/for-own, 2023.

[28] Npm package for-in. https://www.npmjs.com/package/for-in/, 2023.