UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Applying Code Property Graphs On Modern Web Languages For Security and Privacy Analysis

Tiago Luís de Oliveira Brito

Supervisor: Doctor Nuno Miguel Carvalho dos Santos
Co-Supervisor: Doctor José Faustino Fragoso Femenin dos Santos

Thesis approved in public session to obtain the PhD Degree in Computer Science and Engineering

Jury Final Classification: Pass with Distinction

2024

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Applying Code Property Graphs On Modern Web Languages For Security and Privacy Analysis

Tiago Luís de Oliveira Brito

Supervisor:	Doctor Nuno Miguel Carvalho dos Santos
Co-Supervisor:	Doctor José Faustino Fragoso Femenin dos Santos

Thesis approved in public session to obtain the PhD Degree in Computer Science and Engineering

Jury Final Classification: Pass with Distinction

Jury

Chairperson:

Doctor Luís Eduardo Teixeira Rodrigues, Instituto Superior Técnico, Universidade de Lisboa

Members of the Committee:

Doctor Marco Paulo Amorim Vieira, Faculdade de Ciências e Tecnologia, Universidade de Coimbra

Doctor Rui Filipe Lima Maranhão de Abreu, Faculdade de Engenharia, Universidade do Porto

Doctor Miguel Nuno Dias Alves Pupo Correia, Instituto Superior Técnico, Universidade de Lisboa

Doctor Nuno Miguel Carvalho dos Santos, Instituto Superior Técnico, Universidade de Lisboa

Funding Institutions

FCT - Fundação para a Ciência e a Tecnologia

2024

Acknowledgements

The work presented in this thesis took a lot of effort, concentration and time to complete, which was only made possible with the help, patience and support of many people who crossed my path during the last few years. I am very grateful to all of them.

I would like to start by thanking the two people directly responsible for guiding me through all the challenges and successes of this work: my supervisor Prof. Nuno Santos, and my co-supervisor Prof. José Santos. I am grateful for their constant encouragement, even after several rejection letters, their out-of-the-box brainstorming meetings and hours of editing that have helped me improve my writing and the way I present and pitch my ideas. They have definitely implanted the *research bug* in me, especially Prof. Nuno Santos, who has been responsible for my entire academic career ever since the development of my master's thesis. Although I am now ready to tackle other professional challenges, they have made it so that I want to keep involved with all the future research and projects that have sprung from our collaborations.

I also want to thank all the people I had the opportunity to collaborate with and learn from while working on my thesis. I thank my co-authors Mafalda Ferreira, Miguel Monteiro, Pedro Lopes, Miguel Barros, Filipe Marques, Miguel Coimbra and Javier Santa-Maria. I have learned a lot from every single one of them. I would also like to thank the teachers, researchers and students of the Distributed, Parallel and Secure Systems (DPSS) group at INESC-ID for their feedback, which helped me improve my presentations, and for all the *sysadmin* effort to keep the DPSS infrastructure working for our common goal of innovative scientific research.

I would like to thank my friends and colleagues at DPSS, especially those from room 501, who have shared this experience with me. In no particular order, I thank Diogo Barradas, Cláudio Correia, Maria Casimiro, João Gonçalves, Rafael Soares, Daniela Lopes, Nuno Duarte and João Loff. We have all worked together to make sure we achieved our goals.

Lastly, I want to thank my family for all the support, and most importantly, I want to thank my wife Cátia for her patience in being there for me and understanding that at times, although I would like to go to the beach, go walking and travelling or go to other social commitments, I had to stay behind to work on a paper, review papers, analyse experimentation results, write another rebuttal letter (and there were a lot of them) or write this thesis. For all that, I am very thankful.

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) (via the SFRH/BD/146698/2019 and 2021.06134.BD grants), project UIDB/50021/2020, project DIVINA (ref. CMU/TIC/0053/2021) and supported by national funds through IAPMEI via the SmartRetail project (ref. C6632206063-00466847).

To a world where people have the patience and persistence to keep fixing the problems and improving the solutions,

Abstract

Many technologies and programming languages have been designed to take full advantage of the potential of the World Wide Web, increasing its usability, usefulness and efficiency for use cases such as social networking, multimedia sharing, online shopping and banking, education, gaming and others. Most of these use cases are nowadays implemented using highly dynamic web applications with modern web programming languages, such as JavaScript and WebAssembly, which might introduce new vulnerabilities, some of which are language specific. Code analysis tools are essential for detecting vulnerabilities in these modern codebases, as they take away the burden of manually analyzing large applications and allow vulnerability testing to be integrated into continuous integration / continuous delivery (CI/CD) pipelines. In the last few years, the research community proposed a static analysis technique for vulnerability detection called Code Property Graph (CPG), which despite the promising results is still relatively unexplored, particularly for highly dynamic untyped languages and web binaries. Additionally, it is difficult to evaluate the available state-of-the-art tools that implement the CPG technique because there are no gold-standard datasets that allow for a rigorous evaluation.

The goals of this thesis are to research how Code Property Graphs can be applied to these highly dynamic languages and develop frameworks and datasets for evaluating these techniques in modern web codebases. To achieve these goals we made the following contributions: i) tested the viability of applying CPGs for detecting vulnerabilities in WebAssembly binaries, ii) studied state-of-the-art static analysis techniques used for vulnerability detection in the Node.js ecosystem and Node Package Manager (`npm`) repositories, iii) created an annotated dataset for vulnerabilities in `npm` packages for testing those state-of-the-art static analysis techniques, and iv) designed a custom CPG-based analysis for detecting violations of privacy policies in web applications and detecting injection vulnerabilities in our curated dataset.

Organization of the document: The dissertation is divided into two parts. The first part provides the motivation and background for my work, identifies the main contributions of the thesis, offers an overview of the results achieved, and proposes directions for future work. The second part consists of a collection of the main papers that resulted from my work. This part reflects the content of the published papers, with minor formatting adjustments to fit the layout of the dissertation.

Resumo

Muitas tecnologias e linguagens de programação têm sido desenvolvidas para aproveitar o potencial da World Wide Web, aumentado a sua usabilidade, utilidade e eficiência em diferentes casos de uso, como as redes sociais, partilha de multimédia, compras online, *homebanking*, educação, jogos e outros. A maioria destes casos são, hoje em dia, implementados usando aplicações *web* altamente dinâmicas através de modernas linguagens de programação, como o JavaScript e WebAssembly, que podem introduzir novas vulnerabilidades, muitas delas específicas para a linguagem de programação. Ferramentas de análise de código são essenciais hoje em dia para detetar vulnerabilidades no código destas modernas aplicações, pois permitem mitigar o esforço manual de analisar grandes aplicações e permitem que o teste de vulnerabilidades seja adicionado a *pipelines* de integração e entregas contínuas (CI/CD). Nos últimos anos a comunidade científica propôs uma técnica de análise estática de código para a deteção de vulnerabilidades designada *Code Property Graph* (CPG), que apesar de apresentar resultados promissores ainda está relativamente pouco explorada, em particular para linguagens altamente dinâmicas, sem tipos estáticos, como o JavaScript e binários usados na *web*. Adicionalmente, é difícil avaliar rigorosamente o estado da arte de ferramentas que implementam CPGs porque não existe um padrão de ouro que permita comparar as ferramentas entre si.

Os objetivos desta tese são investigar como é que os *Code Property Graphs* podem ser aplicados a linguagens web altamente dinâmicas e desenvolver *frameworks* e *datasets* que permitam a avaliação desta técnica em aplicações modernas. Para atingir estes objetivos fizemos as seguintes contribuições: i) testar a viabilidade de aplicar CPGs para detetar vulnerabilidades em binários WebAssembly, ii) estudar o estado da arte das técnicas e ferramentas de análise estática usadas para a deteção de vulnerabilidades no ecossistema Node.js e em repositórios do Node Package Manager (`npm`), iii) criar um *dataset* anotado de vulnerabilidades em código `npm` para avaliação das técnicas e ferramentas de análise estática estudadas e iv) projetar uma versão customizada da análise baseada em CPGs e desenvolver um protótipo para detetar violações de privacidade em aplicações web, bem como detetar vulnerabilidades de injeção no nosso *dataset* anotado.

Organização do documento: Esta dissertação encontra-se dividida em duas partes. A primeira parte apresenta a motivação e o trabalho relacionado, identifica as contribuições principais da tese, oferece uma panorâmica sobre os resultados atingidos, e propõe direcções para trabalho futuro. A segunda parte consiste numa seleção das principais publicações que resultaram do meu trabalho. Esta parte espelha o conteúdo dos artigos tal como foram publicados, salvo alterações de pormenor para acomodar a formatação usada na dissertação.

Keywords
Palavras Chave

Keywords

Code Property Graph

Static Analysis

Software Security

Vulnerability Detection

GDPR Compliance

WebAssembly

JavaScript

Node.js

Palavras Chave

Grafos de Propriedades de Código

Análise Estática

Segurança de Software

Deteção de Vulnerabilidades

Verificação do RGPD

WebAssembly

JavaScript

Node.js

Table of Contents

1 Introduction

1

Overview

Nowadays, we are witnessing a shift in the development of software. Instead of developing native applications, now mostly reserved for high-performance use cases such as gaming, video and photo editing, etc., most modern apps are either Web applications or desktop applications developed using Web-based frameworks. In the realm of both Web applications and Web-based desktop applications, the most emerging programming languages are JavaScript and WebAssembly. JavaScript is ubiquitous for Web development, being used mostly on the client side for adding dynamic behaviour to Web pages when interpreted by the Web browser and also on the server side using the Node.js runtime environment. But JavaScript wasn't originally designed to handle performance-sensitive applications, where it lags behind due to the costly interpretation by the browser. WebAssembly is a portable low-level byte code that was introduced to allow near-native performance in the browser, thus overcoming the overhead of JavaScript.

Both JavaScript and WebAssembly code can introduce specific vulnerabilities into Web applications. JavaScript used on the client side often allows malicious users to bypass (client-side) input validation, and inject code into a Web page using cross-site scripting (XSS). This attack allows the exposure of a user's session data, or cross-site request forgery (CSRF or XSRF), which forces users to execute malicious or unwanted actions on a Web application [95, 102]. These attacks are often chained together to allow malicious users to compromise a Website and install sniffers and other data-stealing attacks, such as e-skimming attacks [10, 134], that execute in the victim's browser. Server-side JavaScript, running in the Node.js runtime environment, suffers from many of the same vulnerabilities as its client-side counterpart, but with even worse consequences. In particular, injection vulnerabilities in Node.js may not only compromise the Website but could also be used to compromise the underlying operating system, because the API of Node.js includes functionality for interacting with OS-level resources, such as the file system, network sockets, databases and others [147]. Much like JavaScript, WebAssembly code may also lead to security vulnerabilities in Web applications. WebAssembly is a compilation target for multiple languages, including memory-unsafe languages. This means that vulnerabilities in memory-unsafe source languages can translate to vulnerabilities in WebAssembly binaries, such as overwriting supposedly constant data or manipulating the heap using a stack overflow, writing arbitrary memory, overwriting sensitive data, and triggering unexpected behaviour by diverting control flow or manipulating the host environment [93].

Consequently, tools employing code and binary analysis techniques are essential to identify and mitigate security vulnerabilities, and the research community has put quite some effort into developing tools that help developers automate the process of vulnerability detection using

techniques such as fuzzing and dynamic symbolic execution [138, 143], dynamic taint analysis [73, 172], static analysis [119, 120, 48] and machine learning [27]. However, software development covers many distinct programming languages, frameworks and environments. A tool developed to detect vulnerabilities for a specific use case is not effective for all other development stacks.

Recently, the research community has proposed and improved a static analysis vulnerability detection technique called Code Property Graphs, which showed potential for detecting vulnerabilities in low-level languages like C and C++ [170, 171], as well as high-level Web languages like PHP [19, 9]. But, despite the research effort, this technique is still rather unexplored for modern Web languages. For one, there are open challenges for applying this technique to highly dynamic programming languages, akin to JavaScript, such as code coverage and precision when detecting vulnerabilities. It is also unclear if it can be applied to low-level byte code, akin to WebAssembly. Additionally, there are no systematic studies to compare the CPGs for vulnerability detection in these Web programming languages with other static analysis techniques, and the available annotated datasets of JavaScript vulnerabilities are very limited and not representative of vulnerabilities found in popular development ecosystems, such as Node.js. For these reasons, the available annotated datasets are not appropriate for such a comparison.

To this end, the goals of this thesis are to research how Code Property Graphs can be applied to modern Web languages, like JavaScript and WebAssembly, and develop frameworks and datasets for evaluating these techniques. To achieve these goals we start by i) testing the viability of applying CPGs for detecting vulnerabilities in WebAssembly binaries, ii) studying state-of-the-art static analysis techniques used for vulnerability detection in the Node.js ecosystem and Node Package Manager (`npm`) repositories, iii) creating an annotated dataset for vulnerabilities in `npm` packages for testing those state-of-the-art static analysis techniques, and iv) designing a custom CPG-based analysis for detecting violations of privacy policies in Web applications and detecting vulnerabilities in our curated dataset.

The remainder of this chapter is organized as follows. Section 1.1 describes the necessary background on Code Property Graphs and the Web languages we propose to apply them (WebAssembly and JavaScript). We also discuss the challenges of detecting security vulnerabilities in these languages and their security concerns. Section 1.2 details the contributions of this thesis. Lastly, Section 1.3 presents our conclusions and discusses directions for future work.

1.1 Background

In this section, we provide the necessary background to understand and discuss the challenges of detecting vulnerabilities in Web applications. First, Section 1.1.1 describes the notion of a Code Property Graph (CPG), a structure that combines multiple distinct graph representations to statically detect vulnerabilities across different programming languages. Then, Section 1.1.2 covers the background of WebAssembly, a binary code format for Web applications that can inherit vulnerabilities from C/C++ code, and Section 1.1.3 provides a deeper look at the JavaScript programming language, one of the most popular languages for client and server-side Web applications, for which vulnerability detection approaches based on static analysis, like Code Property Graphs, are particularly challenging to apply.

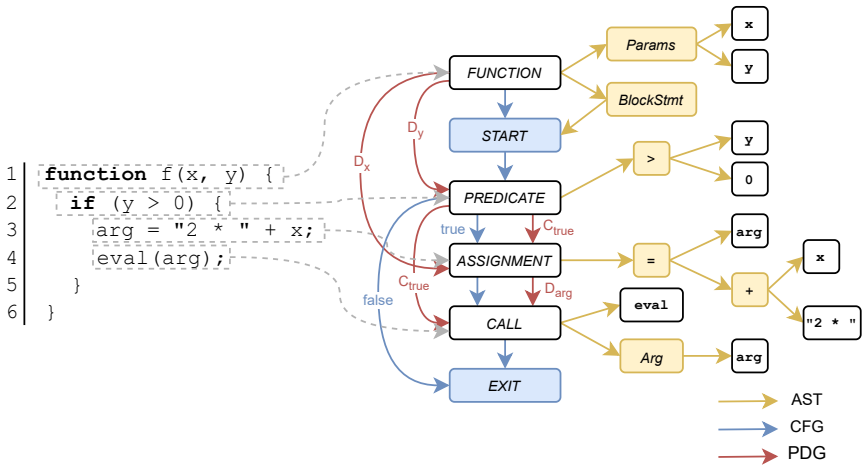

Figure 1.1: Code injection example and its Code Property Graph.

1.1.1 Static Vulnerability Detection Using Code Property Graphs

There are two distinct strategies for detecting vulnerabilities through code analysis: dynamic and static code analysis techniques. Dynamic techniques [14, 60, 77, 76, 29] instrument and inspect the code while it is executed. This strategy can improve precision, but relies heavily on manually generated/crafted inputs and tests, as well as recorded interactions between a human user and the application to establish a baseline for “normal" application behaviour. Additionally, recall tends to be lower because it’s hard to analyze all possible paths in an application. For these reasons, it’s difficult to apply dynamic code analysis techniques for automated vulnerability detection. Additionally, executing a program can be challenging to perform at scale, as each program requires a particular environmental configuration and dependencies. Dynamic analysis also tends to rely on code instrumentation and callback hooks, which add overhead to the application. On the other hand, static code analysis [54, 85, 102] allows code inspection without having to execute the application, generate custom inputs, or record user interactions. Instead, static code analysis techniques parse the application code into specific data structures which are queried for known vulnerable patterns [170, 43, 8, 19, 127, 9, 97, 96, 86]. Lastly, static analysis also tends to have better code coverage, because the analyzed code is not limited to the control-flow paths triggered by the generated test cases and inputs. The characteristics of static analysis for vulnerability detection make it a better candidate for automatic vulnerability detection. To that end, the research community has introduced static analysis techniques, such as Code Property Graphs, specifically for vulnerability detection. In this Section, we aim to describe how CPGs can be used to detect vulnerabilities.

1.1.1.1 How Code Property Graphs Work

Figure 1.1 provides the code injection example we will use to explain how traditional CPGs can be used to detect the vulnerability. This is a JavaScript code snippet that declares a function with two parameters: `x` and `y`. For this example, we will consider these parameters as application inputs, i.e., parameters that might hold attacker-controlled data. Since JavaScript is a dynamically typed language, it is not clear from this snippet alone what is the intended type for each of the

parameters. These parameters will be typecasted during the execution depending on each code statement. After the function declaration, there is an `if` statement dependent on the `y` parameter with a block of statements where the new variable `arg` is created based on the value of the `x` parameter and is then used as an argument to the `eval` function. This function is vulnerable to code injection because, whenever parameter `y` is greater than zero, user input (parameter `x`) reaches the known-dangerous function `eval` via the `arg` variable.

Yamaguchi et al. [170] proposed a general approach for finding code vulnerabilities similar to the injection example described above by introducing the concept of a Code Property Graph, which is a graph-based representation of a program used to identify security vulnerabilities in C and C++ code. In particular, these authors implemented this technique in a tool called Joern, which effectively detected vulnerabilities in the Linux kernel. The authors realized that many vulnerabilities can only be statically discovered by reasoning, simultaneously, about the structure, control flow and dependencies of code. A CPG can reason about these properties by combining the *Abstract Syntax Tree (AST)*, *Control Flow Graph (CFG)*, and *Program Dependence Graph (PDG)* in a joint data structure. This joint structure is a comprehensive view of the code that enables common vulnerabilities to be modelled using graph traversals, which work similarly to a query in a database. These graph traversals are crafted templates for how several flaws in code are represented in the graph and are independent of specific semantics of the code. Figure 1.1 provides the CPG for the previous vulnerable example to illustrate how the subgraphs are assembled into a consistent CPG data structure. Next, we present each of these subgraphs in more detail and explain how the CPG can be used to identify the vulnerability in that code example.

Abstract Syntax Tree (AST): Compilers transform our code into structures that are better suited for reasoning about the code. A compiler usually begins by performing lexical analysis to tokenize different parts of the code like variables and functions. It then parses these tokens into a tree that represents the structure of the code: the abstract syntax tree. The AST can then be safely used to translate code, e.g., converting C code to machine code. Each node of the AST denotes a construct occurring in the text and can be inserted into one of two categories: *inner nodes* and *leaf nodes*. Inner nodes represent operators such as assignments and function calls, while leaf nodes represent operands such as constants or identifiers. In Figure 1.1 the *CALL* node is an inner node as it represents a function call, whereas the `eval` and `arg` nodes are leaf nodes, as they represent the called function label and the function argument, respectively. Yamaguchi et al. [170] show that abstract syntax trees are useful to model vulnerabilities, but they lack semantic information such as the program's control or data flow. As a result, there is a need for additional structures, such as a Control Flow Graph and a Program Dependence Graph, to successfully identify a vulnerable program flow.

Control Flow Graph (CFG): All the possible execution paths of a program can be described using a control flow graph. It explicitly describes the order in which statements are executed, as well as the predicates necessary for a certain execution path to be taken. The CFG of a function contains a starting node, a node for each statement, predicates contained in the function and an end node. Nodes are connected by directed edges that indicate control flow. Edges originating from predicate nodes are labelled as either `true` or `false` denoting the boolean value the predicate must evaluate for the control to be transferred to the destination node. In Figure 1.1 the assignment in line 3 is represented by an assign statement, *ASSIGNMENT* node in the CFG, whereas the if in line 2 is represented by a predicate, *PREDICATE* node in the CFG, and follows either the edge labelled as `true` or `false` depending on the predicate value.

```
1  MATCH (source:CFG)-[data_dep_edges:PDG]->(call_node:CFG_CALL)-[ast_edge:AST]->(sink:AST)
2  WHERE sink in ["eval", "exec", "readFile",...]
```

Listing 1: Query pseudo-code to find an injection vulnerability.

Program Dependence Graph (PDG): Program dependence graphs were originally introduced by Ferrante et al. [167] to execute program slicing. PDGs represent dependencies between statements and predicates, making it possible, through static analysis, to track down the data flow of a program. This is useful for vulnerability detection because it allows us to reason about the propagation of attacker-controlled data. Just like control flow graphs, program dependence graphs' nodes include the statements and predicates of a function and its edges have one of the following types: *data dependence edges* indicate that a variable defined at the source statement is used at the destination statement and *control dependence edges* indicate that a predicate influences the execution of a statement. In Figure 1.1, the *FUNCTION* node has a data dependence edge, labelled as D_x, pointing to the *ASSIGNMENT* node because the value of `arg` depends on the value of `x`. The same logic applies to the data dependency between the *ASSIGNMENT* node and the *CALL* node. The control dependence edge between the *PREDICATE* node and the *CALL* node shows that *CALL* is only executed if the predicate value is `true`.

Vulnerability detection: To detect vulnerabilities using CPGs we need to analyze the graph in search of specific known-vulnerable patterns. Figure 1 shows the pseudocode of a query we can execute over a Code Property Graph, including the one presented in Figure 1.1, to detect an injection vulnerability. The query starts by finding a `source` (CFG node), which represents an attacker-controlled flow entering the application, e.g., a parameter of a *FUNCTION* statement. There must be one or more data dependency edges (`data_dep_edges`) linking the `source` to a `call_node` statement, also part of the CFG. The `call_node` statement needs to be connected by an AST edge (`ast_edge`) to a `sink`, part of the AST, where the called function is a dangerous sink, like `eval` or `exec`.

1.1.1.2 Tools Implementing Code Property Graphs

CPGs were originally developed to help detect buffer overflows, integer overflows, format string vulnerabilities, and memory disclosures in low-level languages, such as C or C++. But in the original work by Yamaguchi et al. [170] CPGs did not support inter-procedural analysis. The same authors later expanded on their work to support inter-procedural analysis [171] by checking for data flows between call sites and their callees, introducing edges between arguments and function parameters, as well as between return statements and the original call sites. This allowed their tool to overcome the known limitation of only considering the local scope within a function and improving taint-style vulnerability detection across multiple functions.

Although not originally developed for the Web, GPGs were soon adapted for it and extended with new dynamic code analysis techniques for PHP-based Web applications [43, 8, 127, 9, 19]. Most notably, Backes et al. [19] applied and extended CPGs for PHP code analogously to the work by Yamaguchi et al. [171], thus supporting inter-procedural analysis via call graphs. Their code representation was proven to discover vulnerabilities such as SQL injections, code injections, and command injections on the server side and cross-site scripting and session fixation on the client side. Alhuzali et al. [8, 9] was also inspired by the concept of a CPG and implemented a

```
1  function f(x, y) {
2      if (y.age > 0) {
3          arg = {};
4          arg.age = "2 * " + x.age;
5          eval(arg.age);
6      }
7  }
```

Listing 2: JavaScript code vulnerable to an injection vulnerability using object properties.

graph-based code representation that was also proven effective in detecting code injections, *XSS*, and *CSRF injections* in PHP applications. They improve the state-of-the-art by generating basic PHP vulnerability exploits for the vulnerabilities detected using the static analysis technique.

There is also work on applying CPGs to JavaScript code. In particular, Khodayari et al. [86] developed a framework that adapts CPGs exclusively to study client-side cross-site request forgery (CSRF) vulnerabilities. This work had to support language features that are not present in C/C++ or PHP code, such as asynchronous events or the execution environment, which are essential components of client-side JavaScript. The authors implement a prototype called JAW that introduces the concept of a Hybrid Property Graph (HPG) by merging the CPG with a specific call graph for JavaScript, as well as event registration, for the event loop, and a specific dependence graph that takes into account dependencies between event handlers. However, JAW does not focus on supporting the object-oriented features of JavaScript and does not support the Node.js runtime, limiting it to imperative-style client-side JavaScript.

Although these initial and extended CPG specifications can be applied to our (non-object-oriented) example from Figure 1.1, modern Web application code is more akin to that presented in Listing 2, where the user input is a JavaScript object (parameter `x` with property `age`) which reaches the dangerous `eval` sink via the `arg.age` property. If the CPG implementation does not support dependencies between variables, objects and properties, then the example in Listing 2 is not detected. Both the generation of the CPG and the query implemented for detecting injection-style vulnerabilities need to be adapted for object-oriented JavaScript.

Most recently, CPGs were adapted to object-oriented server-side JavaScript vulnerabilities [96, 97, 67]. Li et al. [96] adapted CPGs for detecting prototype pollution vulnerabilities in Node.js applications. Their prototype creates a new structure called an Object Property Graph (OPG) that represents JavaScript objects, variable names, properties and the relationships between these entities. They query for specific property assignments and lookups that typically represent prototype pollution vulnerabilities, which allowed them to detect 61 zero-day vulnerabilities with 11 CVE identifiers being assigned. Later on, Li et al. [97] improved their approach to identify both taint-style and prototype pollution vulnerabilities in Node.js applications by creating a new graph called Object Dependence Graph (ODG) which, similar to the OPG, also represents objects, variables, properties, but adds representations for scopes and connects edges between ODG object nodes and AST nodes to represent object definition, variable assignments and property lookups. They were able to detect 180 zero-day vulnerabilities with 70 CVE identifiers assigned. CodeQL [67] is an industrial tool for code mining in multiple programming languages. It represents code using a proprietary graph-based representation that allows users to specify queries capable of detecting some of the most common Node.js vulnerabilities like command injection, path traversal and prototype pollution.

C program code

```c
int factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n-1);
  }
}
```

Text Representation

```
;; magic number
(func $factorial
  (param $n i64)
  (result i64)
;; function section
;; code section start
  local.get $n
  i64.eqz
  if (result i64)
      i64.const 1
  else
      local.get $n
      local.get $n
      i64.const 1
      i64.sub
      call $factorial
      i64.mul
  end)
;; Module End, size fixups
```

Table 1.1: A simple C function is on the left and the corresponding WebAssembly's text format is on the right side.

Both ODGen [97] and CodeQL [67] were evaluated and compared with other static analysis tools in the context of this thesis (Chapter 3) and were shown to be the best state-of-the-art tools for statically detecting vulnerabilities in npm packages. Nevertheless, their results still show relatively low recall and low precision in a curated dataset of Node.js vulnerabilities, which we believe can be improved by designing a custom CPG specification that focuses on the most common Node.js vulnerabilities. Additionally, no CPG-based vulnerability detection tool exists for detecting vulnerabilities in other popular Web languages, like WebAssembly, and it's not clear if this technique can be applied to other use cases besides vulnerability detection, because there would need to exist other use cases that could be successfully modelled using graphs, but no previous work has shown this to be possible.

1.1.2 WebAssembly Security

JavaScript is the ubiquitous programming language for the client side of the Web, though it wasn't originally designed to handle performance-sensitive applications. With the evolution of the Web and the increasing complexity of Web applications, JavaScript lags behind in creating efficient implementations due to the costly interpretation by the browser. To overcome this challenge, a new language specification was proposed, which led to the creation of WebAssembly [74, 133], a portable low-level byte code and a target compilation for efficient statically typed code.

1.1.2.1 How WebAssembly Works

Unlike most programming languages, developers are not meant to program in WebAssembly. It is a target language whose goal is to provide a set of instructions that can run at near-native speed. It is usually compiled from high-level languages, such as C/C++ and Rust, via the Emscripten [51] and rustc [135] compilers, respectively. The resulting WebAssembly code (usually

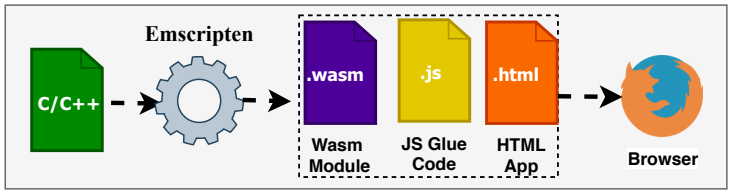

Figure 1.2: Compiling code into WebAssembly.

referred to as Wasm) is a sequence of instructions, which is then interpreted and executed in a stacked-based machine. Wasm also comes with a text format representation for readability and to simplify debugging. Table 1.1 pictures the function `factorial` written in C and the corresponding WebAssembly's text format. The original C code describes a recursive function that receives an integer as an argument (line 1), contains one `if-else` statement (lines 2 to 6), and returns the resulting computation as an integer. The WebAssembly code is a straightforward translation of C code to stack-based assembly language and contains a function declaration (line 1), a 64-bit integer (`i64`) parameter as the argument with label `$n` (line 2) and a 64-bit integer as a return (line 3). The first function statement pushes the value of `$n` to the stack (line 7), which is then compared to the value zero (line 8). If `$n` is zero (line 9), then the constant value of one is pushed to the stack (line 10) and the function ends (line 18). If the `$n` is not zero, then the value `$n-1` is pushed to the stack (lines 13 to 15) and the function is called again, using the `$n-1` value as an argument. The result of calling the function again is multiplied (line 17) with the value of `$n` that was pushed to the stack (line 12), and the function ends. Figure 1.2 demonstrates the pipeline of compiling WebAssembly using the Emscripten compiler from C code. The Emscripten compiler parses the C/C++ code, building the WebAssembly binary (.wasm) and making it callable to the JavaScript code that executes in the browser. This is done by generating JavaScript and HTML *glue code* that handles the linear memory, inputs and outputs of WebAssembly.

1.1.2.2 WebAssembly Security Concepts

WebAssembly is designed to be fast and safe to execute, easy to validate, decode and generate. To try and protect users from vulnerabilities, buggy or malicious code, the specification features four main security concepts: sandboxing, secure memory model, control flow integrity and vulnerability mitigation at compile time. Next, we explain each of these security concepts.

Sandboxing: A WebAssembly application is executed within a sandbox environment and it can only interact with the outside environment via dedicated APIs. When Wasm code is executed in the browser, the host environment for the sandbox is the JavaScript interpreter, which enforces its security policies to each Wasm module, for example, restrictions on information flow through same-origin policies [79]. The sandbox environment is also meant to restrict access to system resources, such as peripherals, network sockets and the DOM. This means that Wasm code cannot handle peripherals, make network requests or change the content of Web pages alone. All this functionality must be delegated to the host by existing APIs. This also applies to host environments outside the browser, such as Node.js. However, this mechanism might not be sufficient to ensure security, because the host environment may give full access to the existent capabilities of the system without confirming if these requests are legitimate and necessary.

Secure linear memory model: WebAssembly's linear memory consists of a contiguous, untyped, byte-addressable array, usually a JavaScript's `ArrayBuffer`. A program can load/store values

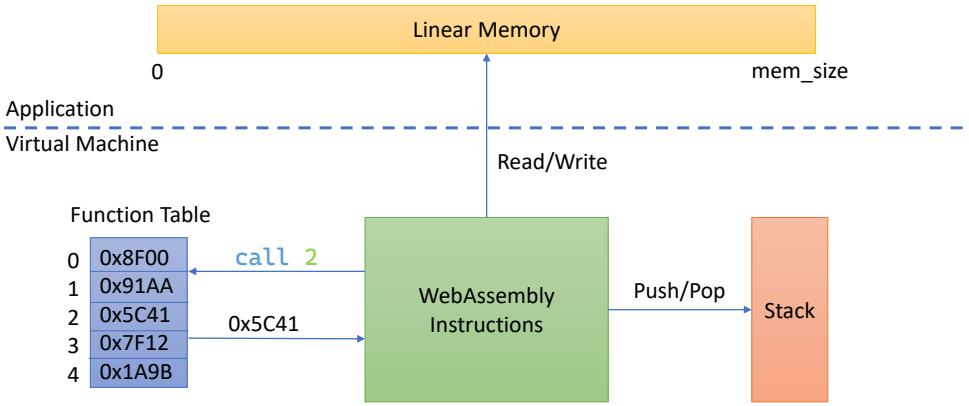

Figure 1.3: Memory Model of Wasm.

from and to linear memory at any byte address. Since WebAssembly has only four primitive types, for 32 and 64-bit integers and floats, all non-scalar types, such as strings, arrays, and other buffers, must be stored in the linear memory. Global variables, local variables and return addresses are managed in the stack, with global variables stored apart in a table named *global index space* and local variables stored within the protected call stack, which is a structure that also holds the return addresses of the function calls. Figure 1.3 shows WebAssembly's memory model and how Wasm instructions interact with the stack and linear memory. This Figure demonstrates how WebAssembly instructions push and pop values from the isolated stack inside the virtual machine, while also querying the function table for the addresses of the binary's functions. The interaction with the application is done using reads and writes to linear memory. These reads and writes are the boundary between the application and the isolated WebAssembly virtual machine.

In contrast to native binaries, such as ELF, WebAssembly's linear memory does not have custom sections with different read/write/execution policies. All memory can be read and written without any restrictions, which may lead to changes in constant data, such as constant strings or constant values, that should not be changed at any point in the execution of the program. Buffer overflows, which result from exceeding the boundaries of an object by writing to adjacent memory, do not affect local and global variables but affect data stored in linear memory since the bound check is performed only for the boundaries of the entire linear memory region.

Although one might change assumed immutable (constant) data using buffer overflows, the linear memory is, by design, non-executable, since the instructions are static and stored apart from linear memory. In the browser, as WebAssembly memory are objects in JavaScript, memory leaks do not occur due to poor memory management by the programmer, because the JavaScript garbage collector will clear the forgotten memory regions. The same does not hold for runtimes other than the browser, such as Wasmer [162] and Wasmtime [26], where memory leaks can occur if not properly handled by the runtime.

Control/flow integrity: Unlike C code, where functions live in memory and function pointers can be corrupted to point to a location holding injected malicious code, WebAssembly functions are indexed in a table. There are two types of function calls: direct and indirect, to perform a direct function call, the instruction must supply a target index that is a valid entry in the function table. For indirect calls, the target index is computed at runtime and the type signature of the function must be supplied and match the signature at the call site. All these calls happen

in the protected call stack, which is protected because it is not possible to overwrite a return pointer, making it invulnerable to buffer overflows.

Consequently, control flow instructions are designed so that calling an unexpected function is likely to fail. The expected and unexpected paths of execution are statically analyzed at compile time, which hardens the possibility of hijacking the control flow of the program but does not eliminate the possibility. It is still possible to gain program control using code reuse attacks against indirect calls. However, it is not possible to use the classic technique of return-oriented programming (ROP), which takes advantage of the execution of the few last instructions of a function, called "gadgets", because call targets must be a valid index in the function table.

Vulnerability mitigation at compile time: Compilers help protect users from vulnerable code by implementing security measures to attenuate or eliminate common vulnerabilities, such as buffer overflows, pointer subterfuge, division by zero, etc. However, some of these measures do not apply to WebAssembly. For example, the control-flow integrity mechanisms and call stack protection prevent direct code injection, which means that canaries for stack smashing protection (SSP) [37] and data execution prevention (DEP) [105] for certain sections of memory are not necessary for the protected call stack. Warnings against the use of potentially vulnerable functions (like `gets()` and `strcpy()`) are supported by state-of-the-art WebAssembly compilers, which also provide control-flow integrity checks to protect code compiled to Wasm.

1.1.2.3 Studies of WebAssembly Security

The initial publication [74] of WebAssembly states that security and safety are the main design goals. However, the research community demonstrated that vulnerabilities not expressed in the original source languages, such as C/C++, can be re-introduced and impact WebAssembly binaries, expressing many classes of vulnerabilities such as buffer overflows, integer overflows, type confusion, use-after-free and double-free. These types of vulnerabilities were first highlighted by McFadden et al. [101] and more recently by Lehmann et al. [93].

Lehmann et al. [93] described how different toolchains used for compiling WebAssembly can expose flaws in WebAssembly's memory protections and provided a set of primitive attacks for exploiting these flaws in the linear memory, thus achieving vulnerabilities such as buffer overflows and memory corruption, which allows changes to (assumed) constant data. They also provided examples of real end-to-end attacks that can lead to cross-site scripting, remote code execution in Node.js and arbitrary file writes. They also showed how attacks can impact the flow of the program and alter the execution order of real-world Web applications compiled from large C/C++ programs. In a subsequent work, Hilbig et al. [78] performed an empirical study of a large number of real-world WebAssembly binaries and confirmed that vulnerabilities can propagate from insecure source languages (memory unsafe languages, such as C/C++) and that 21% of all binaries import potentially dangerous APIs from their host environment. They also provide a large dataset of real-world binaries that can be used to evaluate vulnerability detection tools for Wasm code.

1.1.2.4 WebAssembly Analysis Tools

The community also tackled the challenge of implementing tools for WebAssembly analysis, such as disassemblers [166, 63], dynamic analysis and instrumentation engines [94], and taint

analysis tools [65, 155, 151]. The WebAssembly Binary Toolkit (WABT) [166] is an official toolkit for converting between WAT and Wasm files and decompiling Wasm files to readable C source and header files. It also supports basic static analysis by printing information about a binary's sections, removing sections or counting opcode usage. It also supports dynamic analysis by running a binary using a stack-based machine interpreter. There is also a plugin to the popular Interactive Disassembler (IDA) named *idawasm* [63], which can represent the control flow graph, code and data cross-references, and allows the rename of globals, locals and function parameters to facilitate the analysis.

Wasabi [94] is a dynamic analysis framework for WebAssembly, similar to other binary analysis tools such as Valgrind and Jalangi, which instruments the Wasm binary with callbacks and hooks. It can be used to implement several types of analysis, such as analysis for detecting possible dangerous function calls via call graph analysis or searching taint-style vulnerabilities by performing dynamic taint analysis. However, these analyses have to be manually implemented and are not included with Wasabi.

There are also three other tools specifically designed for taint analysis of Wasm binaries. TaintAssembly [65] is a monitor implemented in the V8 engine that provides basic taint-tracking functionality to follow specific data throughout the execution of the program. It is possible to specify user inputs, network packets and other data as tainted, as well as monitor taint in linear memory. But these taint-tracking features come at the cost of an overhead of up to 4x compared to the unmonitored version of the binary. Szanto et. al [155] introduces another dynamic taint-tracking tool in the form of a custom implementation of a JavaScript VM that executes WebAssembly and achieves an overhead ranging from 20% up to 50%. However, the implemented VM was only designed to support this particular analysis and is slower in comparison with high-performance runtime engines, such as V8. Finally, the community also developed a static taint analysis tool, called Wassail [151], which parses a WAT file into control flow graphs for each function and traverses the graphs to reason about dependency propagation for each instruction. Notice how this approach is similar to that of applying CPGs for WebAssembly, but Wassail only focuses on the control flow graph of the program and ignores the remaining information in the code. The evaluation shows a precision of 64% using the PolyBenchC benchmark, but it only supports the WAT file type and is not capable of analysing WebAssembly binaries.

In summary, there are few tools focused on performing security-oriented analysis for WebAssembly binaries. The dynamic taint analysis tools described above show high overhead, while the static analysis tool does not directly support Wasm binaries. Additionally, none of these tools was shown to work for detecting vulnerabilities transferred from memory-unsafe source languages, which the community has demonstrated to be a significant security issue. Consequently, there is a need to design a system that can efficiently detect vulnerabilities in WebAssembly binaries.

1.1.3 JavaScript Security

JavaScript is a programming language that became one of the core technologies of the Web. It came into the limelight for allowing content to be dynamically generated and displayed to the users on the Web browser. This versatility was a major improvement over the previous state of affairs where websites consisted exclusively of static Web pages. Since then, JavaScript's popularity has been on the rise, being predominantly used not only on the client side (i.e., Web application code running in the browser) but increasingly on the server side. This latest trend has been fueled by the introduction of Node.js and the growth of a community of third-party

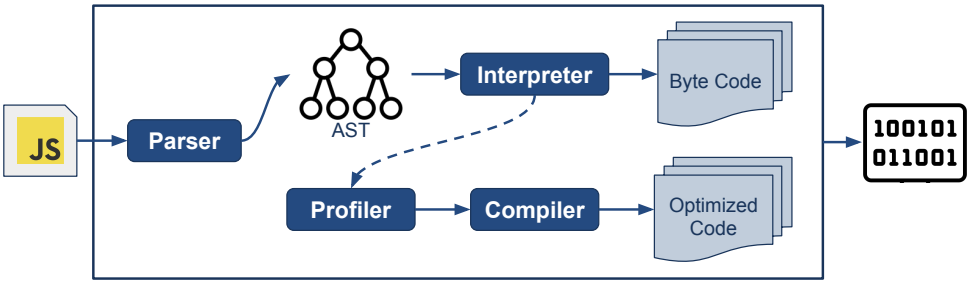

Figure 1.4: V8 Engine Compilation Model.

package developers. Next, we review the existing efforts in characterizing the security of both (client- and server-side) JavaScript ecosystems.

1.1.3.1 How JavaScript Works

JavaScript is a single-threaded interpreted programming language, which relies on the runtime/engine to translate the human-readable JavaScript code into byte code, which is then executed by the Web browser. Nowadays, Google's V8 engine[1] combines both the interpreter and a compiler, known as JIT (Just In Time) compiler to make JavaScript's execution faster. Figure 1.4 demonstrates the compilation model of JavaScript employed by the V8 engine in the Chrome browser and Node.js runtime environment. First, the JavaScript code is parsed into an Abstract Syntax Tree (AST), which is then fed into the Interpreter. The Interpreter creates the corresponding byte code, which can be executed, but it employs a Profiler that analyzes the code as it is running and allows the compiler to optimize the code in cases of code reuse. Finally, the engine can combine the interpreted byte code with the optimized code for faster execution of JavaScript.

The execution of JavaScript occurs in the context of either the Web browser or the Node.js runtime environment. Figure 1.5 shows the components necessary for executing JavaScript in the Chrome browser. The execution of JavaScript code is made possible by the interaction between the V8 engine components and the browser (host environment) components. When a Web page is loaded, user interactions trigger a series of events, which are added to the callback queue along with associated callback functions. The event loop works like an infinite while-loop that keeps fetching callbacks from the queue. Then, the callback JavaScript code is compiled and executed as explained before, at the same time that intermediate data is stored in the call stack, while data such as arrays and objects are stored in the heap. The Node.js runtime works in much the same way, but instead of the JavaScript code interacting with the Web page via the Web API and Web DOM (Document Object Model), it interacts with the Node.js API and the operating system via Node.js bindings. The V8 engine is responsible for handling the execution, garbage collection, coroutines and other features in both the browser and Node.js runtime environments.

1.1.3.2 Client-Side JavaScript

While it has enabled the development of full-fledged Web applications, the inclusion of client-side JavaScript code has led to the introduction of new classes of Web vulnerabilities, most

[1]V8 Engine Website: `https://v8.dev/`

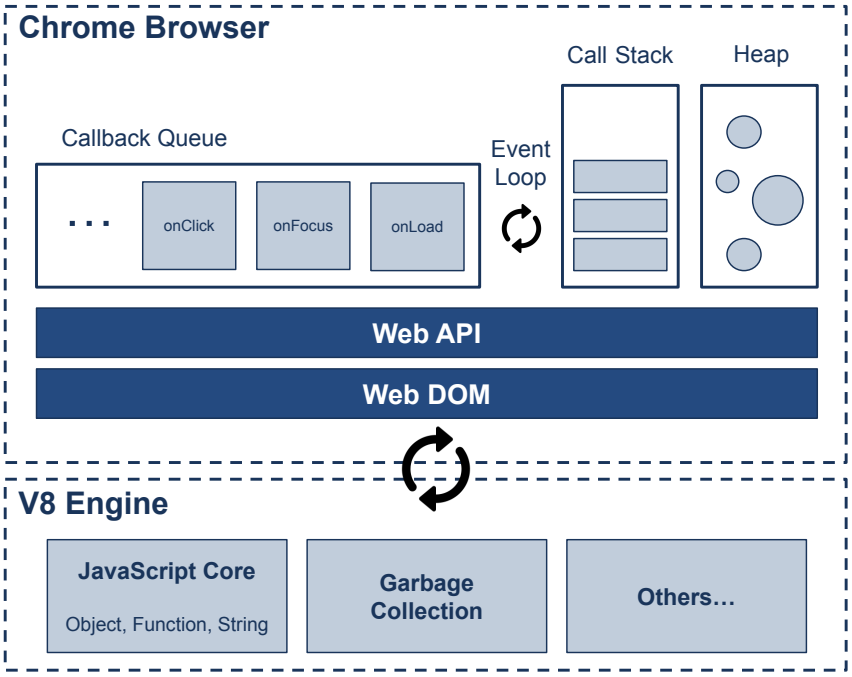

Figure 1.5: Components of a JavaScript Runtime Environment.

notably Cross-Site Scripting (XSS). Given the hazards that these vulnerabilities have brought about to Internet users, the research community has paid much attention to studying the security implications of JavaScript code in the browser.

For example, Steffens et al. [150] have specifically studied the prevalence of persistent client-side XSS in the wild, while Melicher et al. [102] evaluated 3 automated static analysis tools for detecting DOM-based XSS (Esflow [53], ScanJS [109], and Burp Suite Pro [130]). They created a dataset with 3219 confirmed vulnerabilities. Stock et al. [152] evaluated how Web security has evolved with the introduction and adoption of full-fledged client-side JavaScript Web applications. They focused particularly on enumerating which security issues surfaced over time, which countermeasures have been engineered, and how developers have adopted these countermeasures in the wild. Some common security issues discussed in this paper include XSS, insecure *postMessage* handling, and the usage of known vulnerable versions of the *JQuery* library. Lauinger et al. [91] studied how libraries with known vulnerabilities are included in popular Websites. Third-party client-side JavaScript libraries, such as the famous *JQuery*, have allowed developers to reuse code and functionalities across several projects which, on one hand, increases productivity and allows for better code management, but on the other hand, enables vulnerabilities in popular libraries to potentially affect thousands of projects deployed in the wild. These authors showed that 37% of Websites include at least one vulnerable library.

Additionally, the literature also includes some works [46, 95, 153] that introduce specific tools for detecting the classic client-side JavaScript vulnerabilities studied in the formerly described papers. Doupe et al. [46] proposed a technique for inferring the Web application's internal state machine by navigating through the Web application, observing differences in output and incrementally producing a model of the Web application's state. The scanner traverses the model to find user-input vectors that can be fuzzed to discover vulnerabilities. Lekies et al. [95] proposed a fully automated system to detect and validate DOM-based XSS vulnerabilities. They

implemented a taint-aware JavaScript engine that can reason about data flows in a Web page, including the DOM, and developed a context-sensitive exploit generation approach that verifies the potential detected vulnerabilities. The authors evaluated their prototype by analysing the Alexa top 5000 websites, identifying 6167 unique vulnerabilities and showing that 9.6% of the examined sites carry at least one DOM-based XSS vulnerability. Finally, Stock et al. [153] studied the state-of-the-art in browser-based XSS filtering and uncovered a set of conceptual shortcomings, that enable filter evasions for DOM-based XSS. They empirically test websites and show a rate of 73% successful filter bypasses. To overcome this issue, the authors propose an alternative filter design based on taint-tracking and implement a prototype based on the Chromium browser that helps protect against DOM-based XSS exploits.

As a result of the studies mentioned above, as well as the tools implemented for detecting vulnerabilities, the nature and typology of JavaScript vulnerabilities in client-side code are today very well understood by the research community. More recently, JavaScript has also been employed on the server, using the Node.js runtime. Given the most recent transition of JavaScript to the server side, there isn't as much research on the security of server-side JavaScript nor on the implementation of vulnerability detection tools for the Node.js environment.

1.1.3.3 Server-Side JavaScript

As JavaScript became increasingly popular amongst the Web developer community, this language was eventually propagated from the browser to other platforms. In particular, it has become a full-stack development language, thanks mainly to the adaptation of JavaScript engines originally used in Web browsers into fully-fledged server-side runtimes like Node.js. Being the most popular server-side JavaScript runtime[2], Node.js also allows developers to use and develop third-party libraries via the Node Package Manager (`npm`) system. At the time of writing, `npm` contains over 2.1 million packages, making it the biggest single language code repository on Earth.

However, server-side JavaScript code introduces new challenges for vulnerability detection when compared to its client-side counterpart. Similar to client-side code, server-side JavaScript can access the traditional JavaScript API and work with text, dates, regular expressions, standard data structures, etc. However, unlike client-side JavaScript, the code running in Node.js can interact with OS, networking and storage resources on the server. Consequently, the nature and typology of vulnerabilities in Node.js code are substantially different from their client-side counterpart. In fact, vulnerabilities such as directory traversals, SQL injections, and others, which are not possible in client-side JavaScript, can occur in Node.js. Additionally, the vulnerabilities that can occur in both environments show that server-side JavaScript vulnerabilities can have a much higher impact on the system's security than vulnerabilities in client-side JavaScript. As a result of this shift, the studies that had been performed about client-side JavaScript vulnerabilities had to be revisited.

Staicu et al. [147] discuss the difference between injection attacks in client-side JavaScript and on Node.js by studying a large set of `npm` packages. They found thousands of modules vulnerable to command injection attacks because attacker-controlled data can reach either the *eval* or *exec* API functions. In Node.js these injection attacks lead to malicious remote code execution on the server. Attacker code running on the privileged context of the Node.js process may provide full

[2]PYPL PopularitY of Programming Language: `https://pypl.github.io/PYPL.html`

control of the system. This demonstrates the severity of vulnerabilities in Node.js, in contrast to client-side vulnerabilities, where code execution impact is limited by the browser sandbox.

Some additional characteristics of the Node.js ecosystem contribute to the dissemination of vulnerabilities in server-side JavaScript code. For instance, `npm` allows developers to easily create and import third-party code into their applications with minimal overhead and security triage. Unfortunately, as shown by Abdalkareem et al. [1], although developers consider these packages to be correct, well tested, and increase productivity by avoiding repetitive implementation of common functions, many third-party packages are not regularly maintained. As a result, many vulnerabilities tend to creep into the code and remain unfixed for long periods of time.

Zimmermann et al. [173] found that individual packages can impact large parts of the Node.js ecosystem, meaning that vulnerabilities in individual packages can trickle down the dependency chain and affect thousands of other packages and projects. Most importantly, they show that, due to lack of maintenance, many packages depend on vulnerable code even years after a vulnerability has become public knowledge. These facts, allied with the high severity potential of server-side JavaScript vulnerabilities, motivate the need for effective vulnerability detection tools.

In the same study, Zimmermann et al. [173] further discuss the weaknesses of the `npm` security model by demonstrating that a very small number of maintainer accounts control a large number of packages. This means that, if an attacker successfully takes over these accounts, she can inject malicious code into the majority of all packages (which has happened in the past[3]) and potentially compromise a large number of applications. In fact, most recently, Duan et al. [47] studied precisely the security measures of package managers (including `npm`) and applied program analysis techniques such as metadata, static, and dynamic analysis to study registry abuse attacks, such as account hijacking, social engineering, and typosquatting; typosquatting is an attack that occurs when the attacker purposely misspells the name of a popular package and waits for users installing the popular package to typo the name, which consequently leads to the installation of the malicious package rather than the intended original one. Using this approach, they were able to identify 41 malware packages in `npm`.

Furthermore, after reviewing more than 200K *npm* applications, Staicu et al. [147] conclude that 20% of the analyzed applications either directly or indirectly make use of an injection API. Despite this security-critical situation, there is only a small number of research tools for detecting vulnerabilities in Node.js applications and their underlying infrastructure, most of which are based on dynamic code analysis. For instance, Synode [147] aims to prevent injection attacks in Node.js applications, and NodeSec [72] aims to detect vulnerabilities in Node.js applications. The authors of [146] and [44] design specific dynamic analysis for finding regular expression denial of service (ReDoS) vulnerabilities and NodeXP [118] employ dynamic analysis to automatically detect and exploit server-side JavaScript injection (SSJI) attacks. Xiao et al. [169] also apply dynamic analysis and symbolic execution to detect attacks that leverage hidden properties in client- and server-side JavaScript. There are also academic works that employ static analysis techniques for detecting vulnerabilities in Node.js, but most focus on detecting prototype pollution vulnerabilities [96, 88].

Code analysis graphs have also been recently applied to detect vulnerabilities in Node.js code. Particularly, as explained in Section 1.1.1, static analysis tools based on Code Property Graphs for vulnerability detection have been extended to analyse Node.js code. For example,

[3] `https://www.infoworld.com/article/3047177/how-one-yanked-javascript-package-wreaked-havoc.html`

CodeQL [67] uses a graph-based semantic analysis technique that allows developers to build rules for detecting specific code patterns in several distinct languages, including Node.js. DGen [97, 96] merges the AST of the CPG with a new graph named *Object Dependence Graph (ODG)* [97]. The ODG represents objects, variables, and scopes as nodes and their relations as edges. These edges include relations such as object definition, object lookup and property lookup. ODGen was used to mine vulnerabilities in *npm* packages, and it was able to detect taint-style vulnerabilities such as command injection and path traversal, as well as object-related vulnerabilities like prototype pollution. The same tool was redesigned to automatically detect prototype pollution vulnerabilities in Node.js through object lookup analysis [96]. The authors created a new structure called *Object Property Graph (OPG)* capable of representing JavaScript objects and used it to detect 61 zero-day exploitable vulnerabilities, outperforming the previous state-of-the-art [12] tool that only detected 18.

In summary, many of the recent works on detecting vulnerabilities in Node.js code focus exclusively on highly-specific vulnerability types, such as Regular Expression Denial-of-Service (ReDoS) [146], Command Injection [147] and Prototype Pollution [97, 96], which are unrepresentative of the whole Node.js ecosystem security and these tools do not generalize for other common Node.js vulnerability types. Additionally, little is known about the effectiveness of currently available static analysis JavaScript vulnerability detection tools when applied to Node.js code, with an additional challenge that arises from the nonexistence of a gold-standard benchmark to be used for evaluating these tools. Although very recently Bhuiyan et al. [23] published SecBench, a benchmark suite of vulnerabilities and executable exploits for `npm`, at the time of developing this work no such benchmark existed. Additionally, SecBench only contains vulnerable snippets for the five most common vulnerabilities in *npm*, which might miss other relevant vulnerabilities. These reasons led us to create and publish our own annotated dataset (Chapter 3).

1.2 Contributions

This thesis focuses on answering the following question: *Is it possible to successfully employ Code Property Graphs to detect injection-style vulnerabilities and privacy violations in code written using modern Web languages and technologies, such as WebAssembly and Node.js?* In particular, we concentrate on studying static analysis tools for vulnerability detection, as these offer lower overhead and can be integrated into automated testing pipelines. To evaluate these security tools, we had to first study automated vulnerability detection techniques, build a representative dataset of Web vulnerabilities, to use as a gold standard for evaluation, and evaluate the current state-of-the-art static analysis tools used for vulnerability detection. Code Property Graph-based approaches showed a better potential for successfully detecting vulnerabilities. Yet, as we will explain, the original specification of CPGs is not directly applicable to these modern Web languages, so we had to modify and design custom CPG specifications to successfully apply this technique to those use cases. Motivated by the above considerations, we address the following questions in this thesis:

- **Q1:** *Can Code Property Graphs be applied to a new Web language, such as WebAssembly, for vulnerability detection?*
- **Q2:** *Is it possible to build a representative vulnerability dataset of Web code for evaluating vulnerability detection tools?*

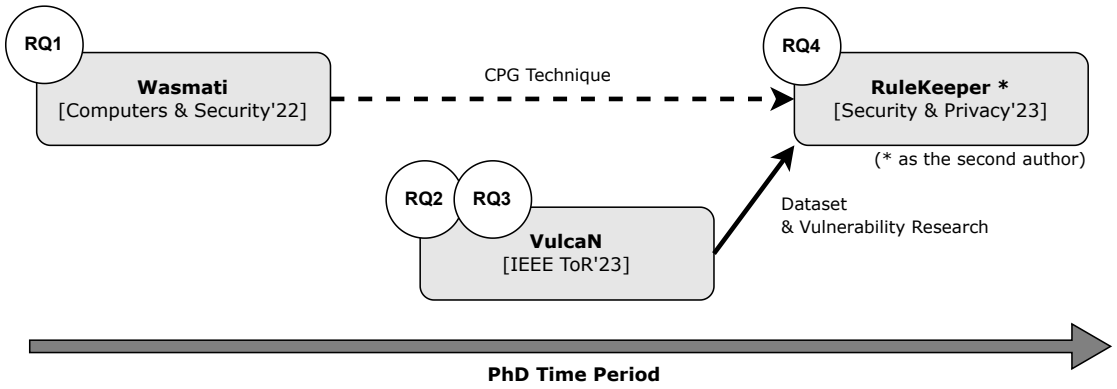

Figure 1.6: Publication Roadmap.

- **Q3:** *Is it possible to establish a baseline for the effectiveness of state-of-the-art static analysis vulnerability detection tools for Web application code?*
- **Q4:** *Can Code Property Graphs be adapted to detect violations of privacy policies in Web applications?*

Figure 1.6 illustrates the publication roadmap consisting of a set of three representative publications that tackle the specific research questions presented above. This figure also represents the chronological order for each publication and how each paper influenced the next, either indirectly, through the application of the CPG technique, or directly, by using the representative vulnerability dataset and knowledge acquired from researching and characterizing vulnerabilities. Next, we detail the main contributions of this thesis.

1.2.1 (Q1) Applying Code Property Graphs to a New Web Language

There are several approaches for detecting vulnerabilities in Web applications, such as input fuzzing, dynamic analysis, symbolic execution and manual code auditing. But, as explained in Section 1.1.1, these approaches require generating valid inputs and tests for the program to run, or require manual labour from a code auditing specialist. These requirements hinder the scalability of these approaches, as they have to be adapted for each specific program. Automated static analysis techniques do not face these challenges and, thus, scale better and are better suited for automated testing pipelines.

Of all the static analysis techniques, Code Property Graphs have shown prominent results for vulnerability detection in statically compiled programming languages, such as C/C++, as well as popular server-side Web languages, such as PHP. Yet, CPGs have not been applied to transpiled languages, such as the WebAssembly binary format, which has a completely different syntax, memory model and execution environment from its C/C++ and PHP counterparts. Consequently, one might question whether it is possible to apply CPGs to detect vulnerabilities in transpiled languages for the Web. For one, transpiled languages such as WebAssembly produce binaries as output, instead of C/C++ or PHP source code, which can be parsed for generating the CPGs. Additionally, C/C++ and PHP are imperative and functional programming languages that share many syntax features. Consequently, code patterns in one of these languages that can

be described by CPG queries should be similar to code patterns in the other language without major adaptations. This might not be true for WebAssembly, which is a binary code format that shares no syntax features with the previous adaptations of CPGs. Nevertheless, WebAssembly is a natural candidate for testing CPGs, because there are many classes of vulnerabilities such as format strings, use after free, double free, and buffer overflows that exist in languages like C and C++ and can be imported into WebAssembly code during the transpilation process [93], and be exploited in the context of the Web, in the form of cross-site scripting (XSS) and code injection attacks, for example.

We studied the hypothesis of applying CPGs to WebAssembly binaries by building the first static analysis tool for detecting vulnerabilities in WebAssembly bytecode, called Wasmati. We developed a domain-specific language named Wasmati Query Language (WQL), which allows security analysts to specify vulnerability patterns in a developer-friendly way while offering efficient query execution times, and compared it to alternative query specification languages: C++, Datalog, and Neo4j's Cypher. We implemented 10 different vulnerability queries and extensively evaluated Wasmati using four different Wasm binary datasets. Wasmati found 100 vulnerabilities out of 108 present in these datasets, representing a precision of 92.6%. We tested Wasmati on a dataset comprising real-world WebAssembly binaries and found several potential vulnerabilities. We manually analyzed some of the flagged vulnerabilities and confirmed that they can be triggered by crafted inputs provided into the affected module.

In summary, our study shows that vulnerabilities in WebAssembly can be modelled using CPGs. We also designed and implemented a prototype tool, called Wasmati, that generates CPGs that can scale for large real-world binaries. If Wasmati is applied by developers using WebAssembly in their applications, these developers can detect potential vulnerabilities before their code is executed in a production environment.

The results of our study, the design and implementation of Wasmati were published in the Computers & Security journal [25]. This publication is included as Chapter 2 of this thesis.

1.2.2 (Q2) Building a Vulnerability Dataset for Web Packages

Before tackling the improvement of CPGs for server-side JavaScript code, we needed to assess the state-of-the-art of publicly available tools and compare them using a common gold standard, so as to understand which techniques are better suited for our goal. However, there was no curated dataset of Node.js vulnerabilities at the time of developing this work. Building a dataset of Node.js vulnerabilities is in itself a challenging endeavour because of the need to identify real vulnerabilities in a large corpus of *npm* packages. Although Node.js vulnerabilities are reported by the community in security advisory reports, these reports are not represented in a format that allows for automatic processing. Moreover, some of them may contain errors and therefore cannot be used unless a thorough analysis and verification are performed. Consequently, our first step was to overcome these challenges and build our own dataset.

To do so, we used the *npm* system itself as a starting point. The *npm* system runs a vulnerability report service that results in the generation of so-called *advisory* reports. These consist of textual descriptions of real (confirmed) security vulnerabilities identified inside specific packages. As such, advisory reports provide a reliable source for building our dataset. Unfortunately, these reports are not represented in a format that allows for automatic processing, and some reports contain errors and therefore cannot be used unless a thorough analysis and verification are

performed. Consequently, we manually analyzed 1359 advisory reports covering an equal number of vulnerable *npm* package versions. These advisories represent 74% of all the vulnerabilities officially reported inside benign *npm* package versions. We have then generated a curated dataset covering 957 of these advisories, extended with annotations that specify the precise location of the reported code vulnerabilities. We found that the location of a large fraction of existing vulnerabilities can be fully expressed through *source-sink* pair annotations.

We specifically built our curated dataset to contain a large set of confirmed real-world vulnerabilities, that can be used for assessing existing (and future) vulnerability detection tools. Consequently, our publicly available dataset can help the research community to i) characterize the vulnerabilities already detected within the *npm* ecosystem, and ii) benchmark vulnerability detection tools.

The results of our study were published in the IEEE Transactions on Reliability journal [24]. This publication is included as Chapter 3 of this thesis.

1.2.3 (Q3) Evaluating the State-of-the-art of Static Vulnerability Detection

The popularity of Node.js and the increase of available *npm* packages make the development of effective server-side JavaScript vulnerability scanners a pressing matter. The research community has shown that an attacker may be able to take over an entire server and/or affect many application users through SQL injection, remote code execution, and other attacks [147, 146] by exploiting security bugs, or because of security issues caused by the intricate *npm* inter-package dependency system [173].

An effective technique to prevent security vulnerabilities from creeping into production code is to integrate security analysis tools as part of Continuous Integration/Continuous Deployment (CI/CD) pipelines, which require the use of automatic vulnerability detection tools. And such tools should have high detection quality (i.e., low false-positive rate), and high coverage (i.e., low false-negative rate). However, no study empirically tested the available state-of-the-art tools to assess their effectiveness at detecting vulnerabilities in *npm* packages. We found a large body of work on client-side JavaScript security [152, 91, 150], and some recent work in the study of vulnerabilities in *npm* packages [147, 146]. However, no prior work has focused on evaluating tools that analyze server-side JavaScript code vulnerabilities, let alone on studying their effectiveness at finding security flaws in *npm* packages.

Consequently, motivated by this need, we present the first empirical study aimed at evaluating existing JavaScript vulnerability detection tools on Node.js packages. We focus exclusively on fully automatic, static code analysis tools that can be used in CI/CD pipelines. In total, we screened 40 analysis tools for JavaScript and selected nine that can detect vulnerabilities at continuous integration time. We executed them against a curated dataset created by us containing *npm* packages with annotated vulnerabilities, mainly: path traversal, cross-site scripting, insecure transfer using HTTP, resource exhaustion/denial-of-service, prototype pollution, OS command injection, code injection, and improper input validation. Then, we checked whether these tools can correctly identify these vulnerabilities.

We tested these nine tools and found they perform rather poorly, missing many vulnerabilities and showing a high false positive rate (low precision). On average, they were able to correctly identify only 15.1% of the total number of vulnerabilities in our dataset. The combination of the three best-performing tools detects 57.6% of all vulnerabilities, albeit with only 0.11% precision.

The best-performing tools manage to detect 41.5% and 31.3% across all types of vulnerabilities and reach their peaks when it comes to identifying prototype pollution and path traversal vulnerabilities, respectively. The best-performing tools approach vulnerability detection using a Code Property Graph-based analysis, which demonstrates that CPGs are the state-of-the-art of static vulnerability detection. Of the 957 known vulnerabilities in the dataset, 324 (33.8%) were not detected by any of the selected tools. Some of the causes are tied to fundamental limitations of state-of-the-art code analysis techniques when it comes to analyzing server-side JavaScript code vulnerabilities in the *npm* ecosystem. Addressing these limitations is an interesting research direction for future work.

In summary, we found that the nine evaluated tools fail to detect many vulnerabilities and exhibit high false positive rates. Of all the tested tools, CPG-based analysis showed better results, which confirms their effectiveness at detecting vulnerabilities and motivates our remaining work in this thesis. Additionally, we show that many important vulnerabilities appearing in the OWASP Top-10 are not detected by any evaluated tool or even when using the combination of all tools.

The results of our study were published in the IEEE Transactions on Reliability journal [24]. This publication is included as Chapter 3 of this thesis.

1.2.4 (Q4) Designing a Custom CPG for Detecting GDPR Violations

The research community showed that CPGs successfully model data flows within a codebase, which means they can be used to analyse how data is handled, modified and sanitized. But, at the time, the state-of-the-art implementations of CPGs were based on specifications for imperative, non-object-oriented programming languages [170, 171] or for Node.js vulnerability detection [96, 97]. This represents a challenge for detecting sensitive flows in JavaScript Web applications because many applications use objects to represent data entities, for example,`Person`, which will include privacy-sensitive data as object properties, such as `name, date of birth, email, phone number` and others.

To overcome this challenge, we have designed a custom specification of CPGs for Node.js code, employing a points-to static analysis, that can reason about data flows using objects and object properties.Our prototype implementation consists of two modules, that build and query the CPG, respectively. The Builder module receives Node.js files as input and parses the code into our custom representation. The Query Execution module receives the custom structure from the Builder module, imports it into a graph database and executes queries to identify specific security and privacy-sensitive data flows. The prototype then outputs a file listing all the potentially vulnerable data flows detected.

We implemented a prototype based on our new custom CPG specification for detecting injection-style vulnerabilities and violations of the General Data Protection Regulation (GDPR) [58], which is an European Union (EU) law that aims to protect and regulate the processing of personal data and how that data can be transferred outside the EU. Our prototype implementation was incorporated into a GDPR-aware personal data policy compliance system for Web development frameworks, called RuleKeeper, which allows Web developers to specify a GDPR manifest from which the data protection policy of the Web application is automatically generated and transparently enforced through static code analysis and runtime access control mechanisms. Our system can model realistic GDPR data protection requirements, add modest performance overheads to the Web application and detect GDPR violation bugs. We also evaluated this early prototype using a subset of vulnerabilities from our curated dataset of vulnerable

npm packages, where we show improvements in recall when compared to some state-of-the-art CPG-based vulnerability detection tools.

The application of our CPGs for verifying GDPR policies was published at the 2023 IEEE Symposium on Security and Privacy [61]. A description of our custom CPG specification and how it was applied in this publication is included in Chapter 4 of this thesis.

1.2.5 Ramifications and Other Collaborations

In addition to the main contributions presented in this document, our research on static analysis techniques for vulnerability detection led us to collaborate on other related projects. Our early improvement on CPGs has been further developed and we have studied the soundness of this new graph design and evaluated it using both our curated database and real-world *npm* packages, where we show improvements in recall when compared to other state-of-the-art CPG-based vulnerability detection tools, as well as the detection of new, previously unknown, vulnerabilities. This work [62] was accepted for publication in PLDI'24 (Core A*) in April 2024.

Additionally, our new and improved CPG design was further improved and designed to produce valid taint summaries for a JavaScript symbolic execution engine. This is to allow for automatic confirmation of vulnerabilities based on the output of our analysis. A symbolic execution engine is capable of producing an input that triggers the vulnerability reported by the static analysis, thus confirming it without manual human intervention. False positives, for which the symbolic execution engine cannot produce a valid input, are filtered and not reported back to the user. This work is currently being developed and will be submitted in 2024.

Additionally, using a similar approach to that of our empirical study of state-of-the-art static analysis for vulnerability detection tools, we performed the first empirical study on the application of program analysis and software testing tools to student code, focusing specifically on the detection of memory errors. This collaboration follows the premise that program analysis and testing tools have become increasingly more effective over the past decade, however, these are still rarely used in undergraduate programming courses, where they could assist students in producing more resilient code. In this work, we perform an empirical study on the application of program analysis and software testing tools to student code. We evaluated 14 high-profile program analysis tools, including fuzzers, symbolic execution tools, and static analysers, on our dataset of student projects. Overall, the evaluated tools perform substantially worse than simple project-specific input generators designed by the course instructors, clearly demonstrating that there is substantial room for improvement when it comes to designing effective and precise fully automatic techniques for detecting memory errors in C projects developed by early-stage students. This work was submitted once before and is currently being improved for a future submission.

1.2.6 Summary of Contributions and Results

In summary, the primary contributions of this thesis are the following:

- The design of a static analysis vulnerability detection technique, based on Code Property Graphs, applied to the WebAssembly binary format for detecting vulnerabilities that persist after transpilation from memory unsafe programming languages (such as C/C++).

- A comprehensive dataset of real-world Node.js vulnerabilities from a large number of `npm` packages, which can be used to evaluate vulnerability detection tools for server-side JavaScript.
- An empirical study of state-of-the-art static vulnerability detection tools for Node.js code using the aforementioned dataset, and the introduction of the VulcaN framework, which allows for comparing different static vulnerability detection tools.
- The design of a novel static analysis technique, based on Code Property Graphs, for analyzing Node.js code. In particular, this technique was successfully applied for detecting privacy-sensitive data flows and GDPR policy violations in Node.js code.

Considering the above contributions, the main results of this thesis are the following:

- An implementation [25] of a framework, called Wasmati, that was able to identify several potential vulnerabilities in large real-world WebAssembly binaries.
- A publicly available dataset [24] that can be used to consistently evaluate vulnerability detection tools for Node.js code.
- A list of the most effective state-of-the-art static analysis vulnerability detection tools for Node.js, their evaluation metrics for a publicly available dataset, and a framework for comparing vulnerability detection tools called VulcaN [24].
- The application of a novel static analysis technique on the implementation of a GDPR-compliance tool, called RuleKeeper [61], which ensures Web developers follow the GDPR regulation when developing new Web applications.

1.3 Conclusion and Future Work

In this thesis, we focus on studying and applying a particular static analysis for detecting vulnerabilities and privacy violations in Web-based programming languages. In particular, we take inspiration from the concept of a Code Property Graph, a canonical representation of code that combines information from a program's syntax, control flow, and data dependencies into one single graph. Each vulnerability can be described as a specific pattern in the graph, which reduces vulnerability detection as traversals (queries). We show that it is possible to improve the recall of this technique for classic injection-style vulnerabilities and that it is possible to adapt this technique for detecting privacy violations to the GDPR in MERN Web applications. Before reaching these conclusions, we worked in multiple directions: (i) we started by detecting vulnerabilities in WebAssembly binaries using Code Property Graphs (Chapter 2), (ii) assessed the state-of-the-art of static analysis vulnerability detection tools for Node.js code, by building the first curated dataset of *npm* package vulnerabilities and evaluating a set of tools on this dataset (Chapter 3), and (iii) designed, implemented and applied our graph-based static analysis for detecting classic injection vulnerabilities and privacy-sensitive dataflows in Node.js code, successfully improving recall when compared to similar tools (Chapter 4). Next, we draw conclusions on the major outcomes of this thesis and point to future directions.

First, we show that it is possible to apply Code Property Graphs to detect vulnerabilities in a transpiled Web language, WebAssembly. As detailed in our paper [25], vulnerabilities in

WebAssembly can be modelled using a custom design of CPGs that is implemented in our prototype called Wasmati. We used our implementation to detect verified vulnerabilities in real-world Wasm binaries. We also show that Wasmati scales well for large binaries. Although this work focused primarily on the construction of Code Property Graphs and demonstrating how they can be used to detect vulnerabilities in WebAssembly binaries, it would be interesting to perform a large-scale evaluation of WebAssembly binaries in the wild to identify their vulnerabilities using Wasmati. To do so one would have to start by identifying where WebAssembly is mostly used, for example, client-side processing, browser extensions, etc., and collect all this code for analysis with Wasmati. This would be relevant to determine the prevalence of security vulnerabilities in Wasm code. Additionally, because Wasm code often interacts with its host JavaScript program via several APIs, it would also be interesting to integrate Wasm CPGs with JavaScript CPGs into one unified structure that could be analyzed using queries. The search for vulnerabilities would take into account both domains.

Second, in our empirical study [24] we built a curated dataset of reported vulnerabilities in *npm* packages to assess the effectiveness of state-of-the-art static analysis tools for vulnerability detection. We show that the tested tools fail to detect many of the reported vulnerabilities and exhibit high false positive rates, which hinders their usability for Web developers, who would have to manually verify all reported output. Additionally, we also show that many important vulnerabilities appearing in the OWASP Top-10 are not detected by any evaluated tool or combination of tools. One possible direction for future work is to focus on detecting specific vulnerabilities that have particularly low detection rates. For example, one could focus on detecting resource exhaustion vulnerabilities using Code Property Graphs by analyzing the vulnerable examples in the curated dataset to understand the most prevalent patterns of resource exhaustion vulnerabilities. Once familiar with the identified patterns, we would develop a new library of queries tailored to detecting the identified resource exhaustion patterns.

Finally, we show that it is possible to improve the recall of static analysis vulnerability detection tools for injection vulnerabilities by creating a new Code Property Graph specification for Node.js code. We evaluated this prototype using a subset of our curated dataset of vulnerable `npm` packages. We also demonstrate that it is possible to adapt this technique for detecting privacy-sensitive data flows in Node.js applications, which are then analyzed to verify GDPR policy compliance [61] in conjunction with dynamic analysis techniques. We are already working on improving our CPG specification for generating taint summaries that can be fed into a JavaScript symbolic execution engine for verifying the potential vulnerabilities, thus increasing precision, and for automated exploit generation. Another possible direction for future work is to adopt our graph specification to automatically detect potential GDPR violations by statically analyzing the source code of Web applications. This tool could be designed to incorporate CD/CI pipelines giving developers immediate feedback that enables them to detect and fix costly GDPR compliance bugs.

II Publications

List of Publications

The work presented in this thesis has contributed to two main publications in top-tier journals (Q1) and one other publication in a top-tier conference (A*), as the second author, listed below:

- Tiago Brito, Pedro Lopes, José Fragoso Santos, Nuno Santos. Wasmati: An Efficient Static Vulnerability Scanner for WebAssembly. Computers & Security 118 (2022).
- Tiago Brito, Mafalda Ferreira, Miguel Monteiro, Pedro Lopes, Miguel Barros, José Fragoso Santos, Nuno Santos. Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages. IEEE Transactions on Reliability.
- Mafalda Ferreira, Tiago Brito, José Fragoso Santos, Nuno Santos. RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks. 2023 IEEE Symposium on Security and Privacy (SP).

2

Wasmati: An Efficient Static Vulnerability Scanner for WebAssembly

Publication Data

Tiago Brito, Pedro Lopes, José Fragoso Santos, Nuno Santos. Wasmati: An efficient static vulnerability scanner for WebAssembly. Computers & Security 118 (2022): 102745.

Abstract

WebAssembly is a new binary instruction format that allows targeted compiled code written in high-level languages to be executed with near-native speed by the browser's JavaScript engine. However, given that WebAssembly binaries can be compiled from unsafe languages like C/C++, classical code vulnerabilities such as buffer overflows or format strings can be transferred over from the original programs down to the cross-compiled binaries. As a result, this possibility of incorporating vulnerabilities in WebAssembly modules has widened the attack surface of modern web applications.

This paper presents Wasmati, a static analysis tool for finding security vulnerabilities in WebAssembly binaries. It is based on the generation of a code property graph (CPG), a program representation previously adopted for detecting vulnerabilities in various languages but hitherto unapplied to WebAssembly. We formalize the definition of CPG for WebAssembly, introduce techniques to generate CPG for complex WebAssembly, and present four different query specification languages for finding vulnerabilities by traversing a program's CPG. We implemented ten queries capturing different vulnerability types and extensively tested Wasmati on four heterogeneous datasets. We show that Wasmati can scale the generation of CPGs for large real-world applications and can efficiently find vulnerabilities for all our query types. We have also tested our tool on WebAssembly binaries collected in the wild and identified several potential vulnerabilities, some of which we have manually confirmed to exist unless the enclosing application properly sanitizes the interaction with such affected binaries.

The reproduction of this publication was slightly adapted to adhere to formatting requirements. The original version of this publication can be found at: `https://www.sciencedirect.com/science/article/pii/S0167404822001407` and `https://arxiv.org/pdf/2204.12575.pdf`.

2.1 Introduction

WebAssembly [74] is an emerging binary code format designed for speeding up the Web. Currently supported by all major browsers, WebAssembly bytecode runs on a stack-based virtual machine that takes advantage of the local hardware capabilities to achieve near-native execution performance. Also known as Wasm, WebAssembly is a low-level assembly-like language as well as a compilation target for higher-level programming languages like C++. This feature has prompted swathes of pre-existing C++ libraries and applications to be ported to run in the browser [18], boosting the adoption of Wasm in the Web [111]. WebAssembly's portability and efficiency have led to its usage far beyond the browser, being employed for running sandboxed code of server-side web applications [115, 162, 30], IoT apps [75], edge computing logic [125], or smart contracts [124].

However, WebAssembly opens up new avenues for the introduction of security vulnerabilities in the Web and other Wasm-based environments [93]. Albeit the extensive safety mechanisms incorporated into WebAssembly itself, many coding errors and unsafe functions written in C or C++ can still be transposed into WebAssembly binaries. As a result, web application code that depends on WebAssembly modules may now become crippled by the introduction of classic forms of software security flaws, such as format strings, use-after-free, double-free, or buffer overflow vulnerabilities. Preliminary studies have shown that when such flaws are present in Web applications, they may be leveraged for launching several web attacks, such as cross-site scripting (XSS) or code injections [101].

Given the pace at which many software projects are being ported to WebAssembly and being adopted worldwide [78], we foresee the imminent danger of pre-existing vulnerabilities to creep into many applications and potentially cause much damage in the near future. As a preemptive measure to tackle this risk, our focus in this work is to help eliminate potential security flaws in WebAssembly binaries, regardless of the specific target application where these binaries are used, e.g., as part of a client-side web page running alongside JavaScript code, as a server-side module running on a Node.js application, or even as a component of a browser extension.

To root out many security flaws in Wasm modules, one can compile these modules from code written in safe programming languages like Go or Rust. However, porting countless software written in C/C++ to a safe language would be a daunting and impractical endeavour. Alternatively, checking and fixing vulnerabilities in C/C++ code using vulnerability scanner tools would proactively counter the transposition of security bugs to Wasm binaries. However, WebAssembly applications are normally composed of multiple libraries or modules, most of which have been precompiled independently by third parties. Access to the original sources may not even be possible (e.g., proprietary code). We then propose a third alternative based on the direct analysis of WebAssembly binaries.

This paper presents Wasmati, an efficient static analysis tool for finding vulnerabilities in Wasm binaries. Wasmati can be shipped in the form of a library that can be linked to other programs (e.g., infrastructure-level software or security analysis frameworks) or packaged as a standalone CLI program. It can then be used for checking vulnerabilities at the development stage (e.g., in the software development toolchain) or in the production stage (e.g., for analyzing client-side web applications in the wild). Currently, our tool can analyze Wasm binaries generated by the popular Emscripten [51] compiler. Wasmati is parameterized by a collection of vulnerability queries. By enriching this set of queries, one can augment the typology of vulnerabilities that Wasmati can detect.

To build Wasmati, we adopt a recently proposed static program analysis technique named *code property graph* (CPG) [170, 171]. CPGs have proved to be powerful constructs for building vulnerability scanners at the source code level for languages such as C/C++ [137, 83, 67], Java [137, 67], Python [137, 67], or PHP [19], and also at the low virtual machine code level for LLVM [137] and Java bytecode [128]. Wasmati is the first tool of this kind to analyze WebAssembly bytecode.

In the design of Wasmati, the low-level nature and specific semantics of WebAssembly brought about several non-trivial challenges. In the construction of the CPG, a major obstacle was in devising a scalable technique to analyze complex Wasm binaries. In Wasm code, the number of nodes of a CPG grows dramatically. Adding to the fact that CPG requires multiple graph traversals for adding edges relative to its distinct subgraphs, the overall construction of a CPG can become prohibitively slow even for modest-sized libraries. For query specification, a core challenge is to represent CPG search patterns while satisfying three requirements: i) expressiveness power, i.e., allow the detection of vulnerability types through the traversal and inspection of the CPG's properties, ii) conciseness and simplicity, i.e., allows security analysts to easily specify new queries, and iii) performance, i.e., the evaluation of queries must terminate in an acceptable time, viz. in a few seconds or minutes.

Wasmati features an optimized CPG data structure and generator engine that significantly speeds up the CPG generation process. Some of the optimization strategies include i) enriching the GPG graph with additional annotations, ii) caching intermediate results, and iii) employing efficient graph traversal algorithms. As for query specification, to strike a good balance between expressiveness power, conciseness, and performance, we developed a domain-specific language named Wasmati Query Language (WQL). WQL allows security analysts to specify vulnerability patterns in a developer-friendly way while offering efficient query execution times. To study how WQL performs in comparison with alternative query specification languages, Wasmati features a generic CPG query engine that enables queries to be specified and executed in three additional languages: C++, Datalog, and Neo4j's Cypher.

We implemented 10 different vulnerability queries and extensively evaluated Wasmati using four different Wasm binary datasets. Wasmati found 100 vulnerabilities out of 108 present in these datasets, representing a precision of 92.6%. This shows that vulnerabilities in WebAssembly can be modelled using CPGs. Wasmati's generation of CPGs can scale for large real-world applications. Using SPEC CPU 2017, the construction time averaged 58 seconds per binary. Graph construction is polynomial in time given the graph's size. We executed the 10 queries in generated CPGs of SPEC CPU 2017 with an average execution time of 77 seconds. We tested Wasmati on a dataset comprising real-world WebAssembly binaries and found several potential vulnerabilities. We manually analyzed some of the flagged vulnerabilities and confirmed that they can be triggered by crafted inputs provided into the affected module.

In summary, this paper makes the following contributions: (1) first formalization of CPGs for WebAssembly, (2) techniques for efficient generation of WebAssembly CPG and specification of vulnerability queries for Wasm code, (3) robust implementation of the Wasmati tool, (4) extensive evaluation using real-world Wasm binaries and queries written in four different languages.

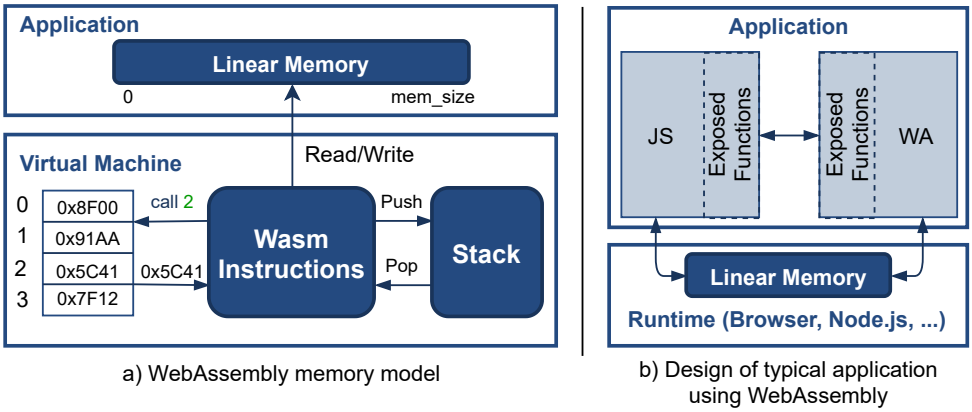

Figure 2.1: WebAssembly high-level architecture.

2.2 Background and Overview

In this section, we introduce WebAssembly, motivate our work using a real-world vulnerability that persists in WebAssembly, provide an overview of how Wasmati detects this vulnerability and, finally, clarify the design goals and scope of our new tool.

2.2.1 Background on WebAssembly

The WebAssembly binary is a sequence of instructions that are executed on a stack-based machine. Simple instructions perform operations on data; they consume their operands from the stack and produce a result that is placed on the stack. Control instructions alter the control flow of the code. A program can call functions directly or indirectly through a function table. Figure 2.1 a) shows a direct call of the index 2. Indirect calls allow the emulation of function pointers and polymorphism in OOP languages such as C++. The table index value is pushed into the stack, evaluated at execution time, and the function indexed by that value is executed. Function tables can be defined in modules. A module represents the binary format of Wasm that has been compiled.

WebAssembly has only four primitive types: `i32`, `i64`, `f32` and `f64`. The first two represent integers with 32 and 64 bits respectively, whereas the last two denote 32 and 64-bit floating-point data. Global variables, local variables, and return addresses are managed in the stack. All non-scalar types, such as strings, arrays, and other buffers, must be stored in *linear memory*, which is a contiguous, untyped, byte-addressable array, multiple of 64Kib. A program can load/store values from/to linear memory at any byte address. A trap occurs if an access is not within the bounds of the current memory size.

WebAssembly binaries are executed by a runtime engine. They are normally deployed in the form of modules that pertain to a larger JavaScript application (see Figure 2.1 b)). Applications use dedicated JavaScript code to bootstrap the Wasm module into a sandboxed environment and to interface with external resources (e.g. DOM). JavaScript code and WebAssembly modules can mutually expose function calls and communicate with each other through shared linear memory. WebAssembly-based applications can run on various platforms, most commonly on the browser, but also on web servers (e.g., powered by Node.js) or desktops.

```
void get_token(FILE *pnm_file, char *token) {
    int i = 0;
    int ret;
    // (...)
    do {
        ret = fgetc(pnm_file);
        if (ret == EOF) break;
        i++;
        token[i] = (unsigned char) ret;
    } while ((token[i] != '\n') && (token[i] != '\r') && (token[i] != ' '));
    token[i] = '\0';
    return;
}
```

Listing 3: Buffer overflow in libpng (CVE-2018-14550).

```
void main() {
    std::string img_tag = "<img src='";
    // bad input: AAAA...AA<script>alert("XSS!")</script><!--
    pnm2png("input.pnm", "output.png");  // CVE-2018-14550
    img_tag += file_to_base64("output.png") + "'>";
    emcc::global("document").call("write", img_tag);
}
```

Listing 4: Exploit for CVE-2018-14550 by Lehman et al. [93].

2.2.2 Practical Vulnerability Example

Figure 3 presents an example of a real stack buffer overflow vulnerability (CVE-2018-14550 [38]) existing inside libpng, which is the official PNG reference library and it is widely used by many applications. Affecting the function `get_token`, this vulnerability can persist when this code is compiled from C to WebAssembly and it has been previously used by Lehman et al. [93] to showcase how vulnerable code written in memory-unsafe languages can be transferred to WebAssembly modules. By developing the exploit presented in Figure 4, the same authors have shown that this flaw can be harnessed to launch a cross-site scripting (XSS) attack. We leverage the same example to motivate our work and then, in the next section, we illustrate how our tool can detect this vulnerability.

The buffer overflow vulnerability shown in Figure 3 can be triggered when converting a PNM file to a PNG file using libpng. This operation calls `get_token` by providing a 16-byte length local buffer as the `token` parameter. Inside `get_token` no check is performed to assess if this buffer is being written beyond its 16-byte length, which allows for a stack buffer overflow to occur whenever the `pnm_file` parameter exceeds 16 bytes. Normally, when libpng is compiled to native binary code, stack canaries prevent this vulnerability from being exploited. Even if stack canaries are not employed by the compiler, the buffer overflow is limited to the stack. However, in WebAssembly this vulnerability can be freely exploited without any of these mitigation strategies. Additionally, when taking into account different linear memory layouts from WebAssembly compilers and backends, this vulnerability can lead to writes not only in the stack but also in the heap and data sections of the memory.

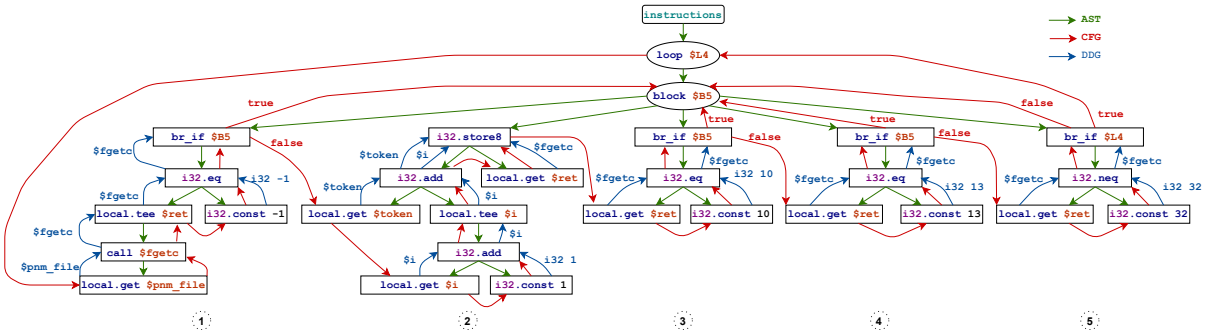

Figure 2.2: Simplified Code Property Graph generated by Wasmati for WebAssembly code fragment depicted in Figure 5.

The C++ code in Figure 4 represents a simplified version of a service that converts images using the libpng library and then displays the converted image by writing the content to the DOM using the `document.write` function. This code converts a PNM image to PNG (line 4), encodes the image content in base64, appends the image content into the `img` tag (line 5), and then adds the tag to the document by manipulating the DOM (line 6). Since the image content is embedded into the DOM as a base64-encoded string, it normally cannot lead to XSS. However, the stack-based buffer overflow in libpng allows the attacker to overwrite higher addresses, including the heap, which holds the C++ string with the `img` tag (line 2). This way, the attacker can use a crafted malicious input, such as the `script` tag string containing an alert (line 3) as the content of the image to convert and override the `img` tag with the newly crafted `script` tag, thus causing an XSS attack.

2.2.3 Finding Vulnerabilities With Wasmati

To cater for the static detection of C/C++-style vulnerabilities in Wasm code, we generate and analyze WebAssembly-specific Code Property Graphs (CPGs). GPGs are graph-based data structures which include information about the analyzed code in the form of property-value pairs (hence the name CPG). These pairs can then be queried in different ways to search for different types of vulnerabilities that can exist in the analyzed code. Using our CPG specification for WebAssembly, Wasmati can detect the buffer overflow vulnerability in the libpng presented in Figure 3 by analyzing the corresponding Wasm code representation listed in Figure 5. Next, we give an overview of Wasmati CPGs and then explain how this vulnerability can be detected by searching for a specific pattern in the CPG's sub-graphs of the corresponding WebAssembly code.

1. Generating the CPG: Figure 2.2 depicts part of the CPG generated by Wasmati for the code shown in Figure 5. The WebAssembly CPG is comprised of four different graps: Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Dependency Graph (DDG), and Call Graph (CG). The first three are portrayed in Figure 2.2 being distinguished by the edge colours. For clarity, AST edges are illustrated by green edges and node properties are not represented in the figure. A full description of node properties can be found in the appendix. The CFG explicitly describes the order by which instructions are executed and the conditions necessary for taking a particular execution path. The CFG edge, illustrated in red, may contain a label that helps identify the condition that allows that particular flow to be followed. The DDG explicitly represents dependencies between the instructions of the program. Dependencies can be function calls, local variables, global variables or constants. Each dependency is expressed as an edge

painted in blue and its label refers to the name of the dependency. The CG is essential to support inter-procedural analysis and is a simple directed graph that connects call nodes and the root of the corresponding called function. It is not visible in Figure 2.2 as we are not analyzing an inter-procedural case, but we explain its importance in Section 2.4. To better visualize the CPG, the nodes in this figure are arranged in a tree layout (corresponding to the AST) with five sub-trees which are numbered from 1 to 5. Each sub-tree can be seen as a bigger statement which is a composition of different instructions. This layout and an understanding of the CPG help us go through the steps needed to find the vulnerability reported in CVE-2018-14550.

2. Querying the CPG: The idea to find this type of vulnerability – i.e., buffer overflows – in WebAssembly code is to analyze the CPG searching for characteristic patterns. In WebAssembly, buffer overflows happen when a buffer is incorrectly used inside a loop, e.g., when handling strings. A vulnerability may occur because the index of the buffer is incremented and the buffer's bounds are not being checked as part of the loop's exit condition. The idea to detect buffer overflows then is to search for instructions in a loop where the AST descendants (loop's block and condition): 1) contain a local variable `$i` representing the index that is being incremented, 2) have a store instruction that depends on a buffer and `$i` (assignment), and 3) lack an exit `br_if` whose condition test verifies the boundaries of `$i`. Thus, to find the vulnerability in our running example, we can query the CPG looking for patterns that satisfy these three conditions. For the first, we look for instructions `i32.add` that have an incoming local DDG edge for `$i` and a constant DDG edge (which is the increment value). For the second, we simply look for `store` instruction with incoming local DDG edge for `$i`. Lastly, we search for `br_if` instructions and query its AST descendants (representing the composite condition) for the existence of comparison instruction with incoming local DDG edge for `$i`. This idea can then be generalized to look for other types of vulnerabilities by adjusting the CPG query accordingly.

2.2.4 Design Goals and Scope

Our main goal is then to build a static analysis tool that can detect security flaws that can be propagated from the original programs (typically written in a high-level language like C/C++) into WebAssembly binaries. We are also interested in building efficient query engines that can help us study several inherent trade-offs between query expressiveness power, conciseness, and performance.

The analysis implemented by our tool will be focused on individual Wasm modules independently of how they are used by a given application or the platform where they are deployed. To examine how a particular security flaw in a Wasm module would manifest itself in a full-blown application, it would be necessary to analyze not only individual WebAssembly binaries but also all other application components (including JavaScript code), which fall outside the scope of this paper due to a significant added complexity.

2.3 Wasmati Architecture

We present Wasmati, a new static analysis tool based on the generation of CPGs for finding security vulnerabilities in WebAssembly binaries. We implemented Wasmati using C++11 in about 12,350 lines of code. Figure 2.3 represents the internal components of our tool. Wasmati consists of two processing pipelines: the *CPG generator pipeline* and the *query engine pipeline*.

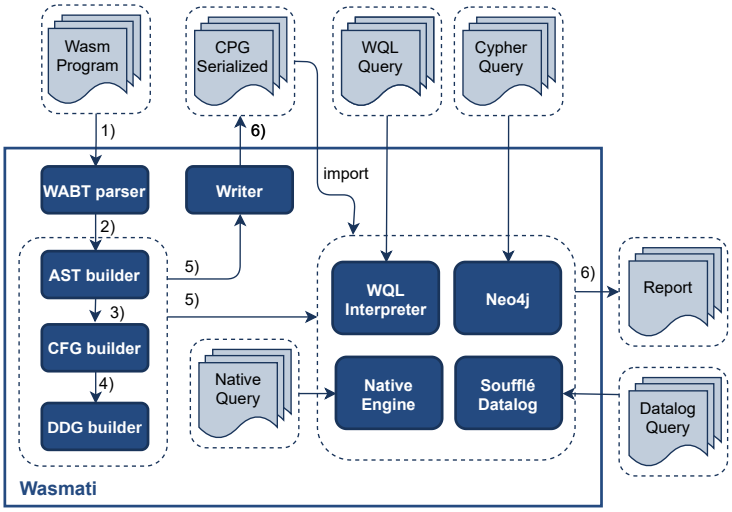

Figure 2.3: Wasmati architecture.

The former is responsible for analyzing an input Wasm program and generating the corresponding CPG data structure into a file. The query engine pipeline loads the CPG from this file and executes a series of queries by searching for specific vulnerability patterns in the CPG. Below, we briefly describe the inner workings of each processing pipeline.

CPG generator pipeline: Wasmati receives a Wasm module, in binary or text format, which will be fed into WebAssembly Binary Toolkit (WABT) [166]. We used WABT, an open-source project maintained by the official WebAssembly community, to parse the WebAssembly's binary/text formats. The parser produces a list of functions. This list is processed by a chain of components which progressively build the CPG. First, this list is provided as input to the abstract syntax tree (AST) builder which creates the AST. Then, the control flow graph (CFG) builder generates the CFG by making another iteration over the function list; the call graph (CG) is also generated in this stage. Lastly, the data dependency graph (DDG) builder creates the DDG. The resulting CPG is then written to a file in one of several possible serialization formats.

Query engine pipeline: The query pipeline supports four query languages/back-ends to execute the CPG traversals:

1. WQL: is the Wasmati Query Language, a DSL that eases the writing of CPG traversals for WebAssembly. The queries written in WQL are interpreted by the WQL interpreter. WQL strikes a good balance between expressiveness power, specification simplicity, and performance. The following back-ends explore different trade-offs in the design space which we explain more thoroughly in Section 2.5.

2. Native: consists of a query API that enables Wasmati to be extended with additional queries written in C++. Adding new queries requires the re-compilation of Wasmati.

3. Neo4j: is a graph database that can import the serialized CPG and be traversed using Cypher Query Language (CQL). Wasmati comes with a Dockerfile and scripts to import and run queries automatically. A new query can be added to a specific folder.

4. Datalog: is a declarative logic programming language that allows the Wasm CPG to be queried in a deductive manner. Wasmati uses Soufflé's [84] flavoured datalog and its engine. Alongside a

Docker configuration file, which automatically imports and executes datalog queries, Wasmati also comes with a library containing common predicates and definitions to query the CPG.

Currently implemented queries: We focused on five main classes of common vulnerabilities in C/C++ that can still persist in the compiled Wasm code: format strings, dangerous functions, use-after-free/double-free and different variations of tainted-style vulnerabilities and buffer overflows. To detect these types of vulnerabilities, we implemented a total of ten different queries in each of Wasmati's supported query languages. Regarding tainted-style vulnerabilities, we implemented three variants: tainted call indirect, tainted function-to-function, and tainted local-to-function. In the first, we query the taintability of the last argument of `call_indirect` which controls the function to be called. The second looks for the classical result of an input source reaching a sink. The third looks for a possible tainted parameter that reaches a sink. With respect to buffer overflows, we also target three variants: static buffer overflow, where we compare the size of the buffer against the size of the data being written to it; malloc buffer overflow, similar to static buffer overflow but the buffer is allocated dynamically with a constant value; and loop buffer overflow, where we search for buffer writes without boundary checks – this variant can detect a buffer overflow in libpng (CVE-2018-14550) [38].

In the following sections, we describe in detail the core challenges and operations performed by the processing pipelines of Wasmati, namely building and querying WebAssembly CPGs.

2.4 Building WebAssembly CPGs

The specific features of WebAssembly introduce non-trivial obstacles to the construction of WebAssembly CPGs which required us to: (i) formalize a tailor-made data structure for representing WebAssembly CPGs, (ii) develop a specific data flow analysis algorithm for computing WebAssembly DDGs, and (iii) incorporate a set of optimizations to speed up the process of building CPGs. In this section, we present how Wasmati builds WebAssembly CPGs emphasizing these main distinguishing features of our system.

2.4.1 Specification of WebAssembly CPGs

Formally, a CPG $G = (V, E, \mu)$ is a triple such that: (i) V is the set of nodes, (ii) E is the set of edges connecting the nodes in V, and (iii) $\mu : V \cup E \to \mathcal{K} \rightharpoonup \mathcal{V}$ is a total function linking each vertex/edge to a (partial) map connecting its properties to their values. For instance, let n be a node in V, the function $\mu(n) : \mathcal{K} \rightharpoonup \mathcal{V}$ maps the properties of n to their corresponding values. For clarity, we use $\mu(n, \kappa)$ instead of $\mu(n)(\kappa)$ to denote the value of property κ of node n. Not all nodes/edges define all the properties in $\mathcal{K}$. More concretely, each type of node/edge defines its own specific properties; for instance, store/load instruction nodes define a property *offset* for holding the offset associated with their respective instructions.

A CPG $G = (V, E, \mu)$ can be decomposed into four sub-graphs, respectively corresponding to the AST, CFG, CG, and DDG of the program to be analyzed. For instance, we write $G_{AST} = (V, E_{AST}, \mu_{AST})$ for the AST component of G. While these four graphs share the same underlying set of nodes, V, they have different edges and they store different property-value pairs. Importantly, each edge $e \in E$ is associated with a property *type*, indicating the sub-graph to which it belongs (i.e. $\mu(e, \textit{type}) = \mathrm{AST}$ means that $e \in E_{AST}$).

Abstract Syntax Tree (AST): To build the AST, we leverage the official open-source parser included in WABT[1]. However, the AST produced by the WABT parser is a flat tree, and for the purpose of CPG query traversal, this lack of structure makes it hard to analyze instructions that take multiple input arguments (e.g., `i32.add`, or function calls). To overcome this limitation, we re-arrange the array of instructions produced by the parser to make sure that direct dependencies are taken into account resulting in a hierarchical organization. For instance, in the program of Figure 2.2, the instructions `local.get $i` and `i32.const 1` become children of `i32.add` (sub-tree 2), while the original parser represents the three instructions at the same level. To perform this re-organization, Wasmati implements an "AST folding" algorithm (shown in the appendix). Our AST includes additional nodes that store meta-information concerning the WebAssembly program; for instance, we use the node `return` to pinpoint the expression that computes the return value of a function.

Control Flow Graph (CFG): In our case, building the CFG is relatively simple because the WebAssembly control flow is structured and can be statically verified. In contrast to typical native binaries (e.g., for x86 or Arm), in WebAssembly there are no relative jumps and a jump target cannot be an instruction from the middle of a block. Wrong paths are not allowed and are validated before execution by the runtime. As a result, the WebAssembly CFG is mostly linear: each instruction has a CFG edge connecting it to the next instruction in the code. The exceptions are the branching instructions `if`, `br_if` and `br_table`, which have more than one successor node. Unsurprisingly, the two outgoing edges of `if` and `br_if` are labelled with either **true** or **false** to distinguish the *then* branch from the *else* branch. Analogously, the instruction `br_table`, which works as a `switch` statement in a high-level language, has its outgoing branches annotated with the concrete values that cause the control to be transferred to each of its branches. Finally, branching labels are stored in the property *label* of the corresponding CFG edge. For instance, given a CFG edge $e \in E_{CFG}$, $\mu(e, label) = true$ means that e is the *then* branch of a `br_if` instruction.

Call Graph (CG): Call graphs are essential to support inter-procedural analysis. In a nutshell, a call graph is a simple directed graph that connects call nodes (i.e., nodes representing call instructions) to the root nodes of the corresponding functions. To implement a CG in WebAssembly, we have to consider both *direct calls* and *indirect calls*. Direct calls are processed straightforwardly: each direct call instruction is directly connected to the node representing the function being called. However, indirect calls are more difficult to analyze, as the index of the function being called is computed at runtime. An indirect call is a mechanism that allows polymorphism from OOP source languages (e.g. C++) and is dependent on its execution. As a result, it is impossible to determine statically which functions will be executed. In fact, multiple functions can be executed depending on different executions.

We solve this challenge as follows. In WebAssembly, for a dynamic call to be executed successfully: 1) the called function must be stored in the function table, and 2) the signature of the called function must coincide with the signature supplied to the call instruction. Hence, for indirect calls, we simply connect every indirect function call to all the function nodes whose signatures match the statically supplied signature. Finally, all CG edges have a single property *type* with the value CG, tagging them as part of the call graph.

Data Dependency Graph (DDG): A DDG explicitly represents dependencies between the instructions of the program to be analyzed. In the original CPG paper [170], it is comprised of

[1] `https://github.com/WebAssembly/wabt`

```
1  (module
2      (func $test
3          (param $y i32)
4          (param $z i32)
5          (result i32)
6          call $source
7          if (result i32)
8              local.get $y
9              i32.const 2
10             i32.add
11         else
12             local.get $z
13             i32.const 3
14             i32.mul
15         end
16         i32.const 1
17         i32.add))
```

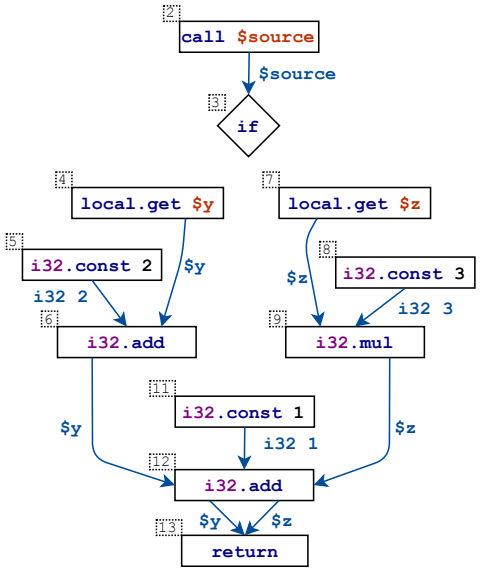

Figure 2.4: Simple WebAssembly program and its DDG

both *data dependencies* and *control dependencies*, which the authors coalesce into a Program Dependency Graph (PDG). An instruction $inst_2$ data-depends on another instruction $inst_1$, if $inst_2$ uses a variable defined by $inst_1$. In turn, $inst_2$ control-depends on $inst_1$, if the execution of $inst_2$ depends on $inst_1$ (e.g., $inst_1$is a branch instruction).

However, the original PGD definition for CPGs is not ideal for WebAssembly. For one, keeping track of both dependency types leads to inefficiencies and scalability bottlenecks when constructing and querying the graph. Compiled WebAssembly code contains many instructions organized in long chains of conditional blocks and loops which may result in an exceedingly large number of control dependencies edges (in our preliminary experiments, comprising in some cases 98% of the total edges of a program's CPG and bloating its size up to 65 times). Secondly, the original PDG definition [170] is too coarse. In WebAssembly, we require richer semantics that allow us to reason about different variable scopes (local and global), return values from function calls, and constant value propagation. For instance, we need to differentiate variables from constants given that oftentimes the compilation of an instruction in C that uses a constant value translates into multiple WebAssembly instructions accessing local variables, making it difficult to keep track of the constant in the WebAssembly binary.

Hence, for building WebAssembly data dependencies we employ two specific adaptations: i) we discard the calculation of control dependencies as it is easily queried later through the CFG, and ii) use a custom-made dependency analysis that combines reaching-definitions and constant/function propagation. This leads to our definition of a Data Dependency Graph (DDG). More concretely, we track four types of data-dependencies: (1) a *constant dependency* $\texttt{C}_{id}(v, t)$ represents a data-dependency on a constant value v of type t, generated by the instruction with identifier id; (2) a *function dependency* $\texttt{F}_{id}(f)$ represents a data-dependency on the value returned by function f, called at instruction with identifier id; (3) a *global dependency* $\texttt{GV}_{id}(x)$ represents a data-dependency on the value of a global variable x (through the instruction with identifier id); and (4) a *local dependency* $\texttt{LV}_{id}(x)$ represents a data-dependency on the value of a local variable x. In the following, we use φ to range over the set of dependencies ($\varphi \in \Phi$) and ϕ to denote an

arbitrary set of dependencies ($\phi \subseteq \Phi$). Put formally:

$$\varphi \in \Phi \quad := \quad \mathtt{C}_{id}(v,t) \mid \mathtt{F}_{id}(f) \mid \mathtt{GV}_{id}(x) \mid \mathtt{LV}_{id}(x) \tag{2.1}$$

where: v ranges over the set of WebAssembly values, t over the set of WebAssembly types, f over the set of function identifiers, and x over the set of global and local variable names.

To better understand how dependencies are modelled, let us consider the WebAssembly program and DDG given in Figure 2.4. Each node's identifier is displayed in its upper-left corner. In particular, one can see that instruction 6 depends on the value of the local variable `$y`, which is put on top of the stack by instruction 4. This dependency is modelled as: $\mathtt{LV}_4(\$\mathtt{y})$. One can further see that instruction 6 also depends on the constant value 2 put on top of the stack by instruction 5. This dependency is modelled as: $\mathtt{C}_5(2, \mathtt{i32})$.

Dependencies are computed by a data-flow analysis explained next. Having computed the dependencies of every instruction, the construction of the DDG is straightforward. Each node is simply connected to the nodes on which it depends and DDG edges are labelled with attributes of their corresponding dependencies. For instance, let e_1 be the edge connecting nodes 4 and 6 in the DDG given in Figure 2.4, we have that: $\mu(e_1, \mathit{type}) = \mathrm{DDG}$, $\mu(e_1, \mathit{ddgType}) = \textsc{Local}$, and $\mu(e_1, \mathit{vName}) = \$\mathtt{y}$. A complete description of the property-value pairs of DDGs is given in the appendix.

2.4.2 Dataflow Analysis for WebAssembly

The construction of the DDG is the most challenging of all the CPG's sub-graphs as it requires a data flow analysis that takes into account the specific features of WebAssembly. In this section, we formally explain the steps and calculations implemented to track the data dependencies necessary for the DDG use case. As discussed above, despite applying known dataflow analysis, these data dependencies had to be adapted to our model and WebAssembly itself.

We apply the monotone framework to compute WebAssembly dependencies by lifting concrete states to abstract states. While a concrete state is composed of a concrete linear memory, global store, local store, and stack. Our abstract states do not have the linear memory component as we do not track dependencies that are established through the use of linear memory operations. Instead, abstract states are simply composed of an abstract global store, local store, and stack. In a nutshell, an abstract global store $\hat{g} : \mathcal{GV} \rightharpoonup \wp(\Phi)$ is a mapping from global variables to sets of dependencies (*mutatis mutandis* for local stores: $\hat{l} : \mathcal{LV} \rightharpoonup \wp(\Phi)$). Analogously, an abstract stack, $\hat{st}$, is simply a list of sets of abstract dependencies. For instance, $\hat{g}(\$\mathtt{g0}) = \mathtt{C}_{20}(3, \mathtt{i32})$ means that `$g0` depends on the constant value 3 via the instruction with identifier 20.

To calculate the data dependencies at each execution point, we traverse the CFG of the program to be analyzed, propagating dependencies in a forward manner. More concretely, we define, for each instruction, a transfer function that describes how that instruction propagates data dependencies by specifying how the output data dependencies are computed using the input data dependencies. Put formally, we define a general transfer function $\mathcal{T} : \mathcal{I} \times \hat{\mathcal{S}} \rightharpoonup \hat{\mathcal{S}}$ that computes an output abstract state given an instruction in I and an input abstract state. The function $\mathcal{T}$ is defined as a set of rules that follow the syntax of instructions. Below, we show a few selected rules, while the complete set of rules is given in the appendix. Transfer function rules are annotated with the identifier of the executing instruction, `id`. Their interpretation is

straightforward. For instance, the transfer function for a `const` instruction extends the stack with a constant value dependency.

Transfer functions (fragment): $\mathcal{T} : \mathcal{I} \times \hat{\mathcal{S}} \rightharpoonup \hat{\mathcal{S}}$

$$\mathcal{T}(\texttt{t.const c}, (\hat{g}, \hat{l}, \hat{st})) \hookrightarrow_{id} (\hat{g}, \hat{l}, \hat{st} :: \texttt{C}_{id}(c, t))$$
$$\mathcal{T}(\texttt{local.get x}, (\hat{g}, \hat{l}, \hat{st})) \hookrightarrow_{id} (\hat{g}, \hat{l}, \hat{st} :: \hat{l}(\texttt{x}))$$
$$\mathcal{T}(\texttt{local.set x}, (\hat{g}, \hat{l}, \phi :: \hat{st})) \hookrightarrow_{id} (\hat{g}, \hat{l}[\texttt{x} \mapsto \phi], \hat{st})$$
$$\mathcal{T}(\texttt{global.get x}, (\hat{g}, \hat{l}, \hat{st})) \hookrightarrow_{id} (\hat{g}, \hat{l}, \hat{st} :: \hat{g}(\texttt{x}))$$
$$\mathcal{T}(\texttt{global.set x}, (\hat{g}, \hat{l}, \phi :: \hat{st})) \hookrightarrow_{id} (\hat{g}[\texttt{x} \mapsto \phi], \hat{l}, \hat{st})$$

Given the transfer functions, we compute the dependencies of each instruction by traversing the CFG of the program starting from the entry point of each function. Loop instructions are re-visited if their input dependencies change. The complete algorithm and the full list of transfer functions are given in the appendix.

2.4.3 Algorithmic Complexity

We analyze the complexity of our algorithm for constructing WebAssembly CPGs by focusing on each sub-graph at a time.

The construction of the AST graph is done in linear time in the number of instructions of the original program. The complexity of the construction algorithm is dominated by the cost of the AST folding algorithm. Using aggregate analysis, we conclude that the total number of iterations of both the outermost and the innermost loops is linear in the number of instructions. In particular, we note that each node has at most one parent in the AST graph, meaning that it can only be visited once by the innermost loop.

The construction of both the CFG and the CG graphs is done in linear time in the number of instructions of the original program. The construction algorithms simply traverse the instruction nodes of the program; the CFG algorithm connects each node to its immediate successors, while the CG algorithm connects direct and indirect call nodes to their corresponding functions. In both cases, the processing of each instruction is done in constant time, making both algorithms linear in the number of traversed instructions.

The construction of the DDG is done in quadratic time in the number of instructions of the original program. The cost of this algorithm is dominated by the dataflow analysis described in Section 2.4.2. To determine the cost of the dataflow analysis, we have to determine an upper-bound on the height of the state lattice, which bounds the number of iterations that the dataflow analysis can perform [112]. Recalling that each state is composed of an abstract global store $\hat{g} : \mathcal{GV} \rightharpoonup \wp(\Phi)$, an abstract local store $\hat{l} : \mathcal{LV} \rightharpoonup \wp(\Phi)$, and an abstract stack $\hat{st}$, we note that the height of the state lattice corresponds to the joint size of the domains of the three state components ($|\mathcal{GV}| + |\mathcal{LV}| + ST_{size}$) times the height of the dependency lattice ($\wp(\Phi)$); put formally:

$$H = (|\mathcal{GV}| + |\mathcal{LV}| + ST_{size}) \times |\Phi| \tag{2.2}$$

where ST_{size} denotes an upper bound on the size of the stack. Observing that both $|\mathcal{GV}| + |\mathcal{LV}| + ST_{size}$ and $|\Phi|$ are bounded by the total number of instructions, we conclude that the construction of the DDG is done in quadratic time.

2.4.4 Optimizations

The distinct structures that make up a CPG amount to a high number of nodes and edges. This is especially true for WebAssembly, which is a low-level language with many instructions which tends to require the generation and analysis of a very large graph even for relatively small binaries. The high number of nodes and edges hampers the scalability of our system with respect to memory usage and computing time for both CPG generation and graph traversals (queries). For example, regarding CPG generation, the biggest bottleneck is the construction of the DDG, mainly in the analysis of loops, because it has to traverse the graph multiple times and calculate data dependencies until no changes are made to the set of dependencies. This is very expensive as we realized that typical WebAssembly programs contain multiple chained loops.

To improve the efficiency and scalability of Wasmati, we employed several optimizations to construct the CPG in useful time without overwhelming the memory usage. The most relevant were as follows: 1) We pre-compute, for each type signature in the program, the set of functions with that signature that can be called indirectly, thus simplifying the generation of the call graph; 2) We propagate dependencies in a modular fashion, meaning that we only compute the dependencies of the successor of a loop, after computing the dependencies of the loop itself; this requires us to keep track of loop exits during data-flow analysis; 3) We cache the dependencies of inner loops to avoid re-analyzing them during new iterations of the outer loops; 4) We avoid the implementation of recursion as it had a negative impact due to indirection and a high number of call stack frames (eventually running out of stack memory). Every traversal/calculation was made in an iterative version.

2.5 Querying WebAssembly CPGs

Vulnerabilities often give rise to specific CPG patterns that can be found using graph queries. Hence, the detection of a security flaw in WebAssembly can be achieved by specifying a graph query that captures the corresponding pattern. Wasmati comes with four query engine pipelines that can process queries in specified different languages. Next, we present WQL: Wasmati's dedicated query language for specifying and executing CPG queries. Then we give a brief account of the remaining query back-ends, explaining the rationale for their development and describing how they work.

2.5.1 Wasmati Query Language (WQL)

We introduce WQL by example, explaining how it can be used to encode and detect typical C/C++-style security vulnerabilities, namely *use-after-free* vulnerabilities, *taint-style* vulnerabilities, and *buffer overflows*. Other common security vulnerabilities, such as use of *dangerous functions*, *format strings*, and *double free* can also be easily expressed using WQL. The complete list of queries used to assess Wasmati is given in the appendix.

WQL features: WQL is a simple interpreted imperative language that offers a wide range of built-in functions for traversing and inspecting the underlying CPG. WQL supports the standard primitive types: booleans, floats, integers, and strings. It also supports polymorphic lists and maps, as well as CPG nodes and edges. When it comes to control flow, WQL includes the typical control flow constructs: `if-then-else`, `while`, `foreach`, `break`, and `continue`. Furthermore, WQL

has a dedicated syntax for range-expression, which is often useful for concisely implementing queries. We write `[ n in lst : pred ]` to denote the list obtained by removing from `lst` all its elements that do not satisfy `pred`.

Use-after-free vulnerabilities occur when one uses a reference to a memory location that has already been freed. This may lead to undefined system behaviour and, in many cases, to a write-what-where condition. It can compromise the integrity and/or availability of the system by causing it to crash or lead to the corruption of valid data when that memory area has already been reallocated.

Figure 6 shows the WQL query for detecting use-after-frees. Our goal is to find three nodes `n1`, `n2`, and `n3` such that: (1) `n1` holds a call to `malloc`; (2) `n2` holds a call to `free`, which frees the memory segment allocated at `n1`; and (3) `n3` holds any instruction that executes after `n2` and uses the pointer returned by the instruction at `n1`. In order to find such three nodes, we iterate over each function in the CPG. For each function, we first obtain all possible candidates for `n1` (line 2). For each possible `n1`, we then obtain all possible candidates for `n2` (line 4). In order to do this, we make use of the predicate `reachesDDG(n1, n, "Function", "$malloc")` to identify only the nodes that depend on `n1` via the value returned by `$malloc`. Finally, we use the built-in function `descendantsCFG` to obtain all the descendants of `n2` and then filter these descendants to obtain only those that depend on `n1` (line 6); if the resulting list is not empty, a *use-after-free* vulnerability is flagged.

Taint-style vulnerabilities refer to data flows from attacker-con-trolled sources to security-sensitive sinks that do not undergo sanitization. With our CPGs, we can identify such flows simply by checking the data dependencies of all the possible sinks. Figure 7 shows the WQL query for detecting taint-style vulnerabilities. Our goal is to find the sink nodes that depend on source nodes. To this end, we inspect the dependencies of each sink, checking if any of them depend on a sensitive source. If the obtained list is not empty, a vulnerability is flagged for each individual source-sink pair.

Buffer overflows: Despite having long been identified as a potential source of security vulnerabilities, buffer overflows remain a common gateway for attackers in today's code. Many mitigation techniques are in place to prevent buffer overruns, such as the use of stack guards to protect local data. Unfortunately, no such technique has been implemented by the existing WebAssembly compilers.

While buffer overflows can happen in various ways, many actually happen when handling strings inside loops. Figure 3 exemplifies one occurring inside a loop in a PNM decoding procedure in the module `pnm2png` from the `libpng` library (CVE-2018-14550) [38]. The loop reads characters from a file and stores them in the buffer `token` until the character read is one of those given in line 7. The problem is that the loop can have an arbitrary number of iterations, while the buffer has a fixed size. As a result, we can trigger a buffer overflow by picking a file containing a large enough string.

The WQL query used to find this vulnerability is given in the appendix. In a nutshell, we have to search for loop instructions for which: (1) there is a local variable `$i` representing an index being incremented inside the loop; (2) there is a store instruction within the body of the loop that depends on `$i` (buffer write operation); and (3) there is no explicit loop exit (`br_if`) instruction that depends on the result of a comparison operation directly involving the value of `$i` (e.g. `i < BUF_SIZE`). For (1), we look for `i32.add` instructions with two incoming DDG edges: one expressing a *local dependency* on variable `$i` and another expressing a constant dependency

#	Query Description	LOC				Basic			Lehmann			STT			CWE		
		NAT	WQL	N4J	DTL	TP	FP	P	TP	FP	P	TP	FP	P	TP	FP	P
1	Format Strings	24	10	19	139	5	0	7	-	-	-	11	8	11	2	0	2
2	Dangerous Function	14	8	5	137	7	0	7	1	0	1	10	0	10	2	0	2
3	Use After Free	67	15	16	140	2	0	2	-	-	-	-	-	-	-	-	-
4	Double Free	60	16	19	141	1	0	1	-	-	-	-	-	-	-	-	-
5	Tainted CallIndirect	55	12	13	139	-	-	-	-	-	-	5	0	5	-	-	-
6	Tainted Func-to-Func	55	12	13	142	2	0	2	-	-	-	-	-	-	1	0	1
7	Tainted Local-to-Func	84	51	55	177	16	0	16	4	0	4	-	-	-	4	0	4
8	BO - Static Buffer	141	58	58	172	1	0	2	0	0	1	12	0	15	8	0	9
9	BO - Static Buffer (malloc)	86	27	19	142	1	0	1	-	-	-	-	-	-	1	0	1
10	BO - Loops	94	23	47	147	-	-	-	1	0	1	2	0	2	1	0	1
	Total:	**680**	**232**	**264**	**1476**	**35**	**0**	**38**	**6**	**0**	**7**	**40**	**8**	**43**	**19**	**0**	**20**

Table 2.1: Query vulnerability report by dataset and LOC metrics for each querying approach. Acronyms: BO (Buffer Overflow), NAT (native, C++), WQL (Wasmati Query Language), N4J (Neo4J Cypher) and DTL (Datalog).

on the value to be added. For (2), we simply look for `store` instructions that directly use the result of the `i32.add` instructions found in (1). Finally, for (3), we search for `br_if` instructions that rely on the result of a comparison operation involving the value of `$i`. If no such instructions are found, a vulnerability is flagged.

2.5.2 Other Query Back-ends

By developing multiple query engine back-ends, our primary goal was to investigate the trade-offs that different query language paradigms can offer regarding: i) expressiveness power, ii) conciseness and simplicity, and iii) query execution performance. After developing and experimentally analyzing three query back-ends that support imperative, logic, and database processing paradigms – C++, Datalog, and Neo4j, respectively – we developed our own domain-specific language (WSL) which i) offers enough expressiveness for capturing WebAssembly vulnerability patterns, ii) is relatively intuitive to program, and iii) performs well. Next, we review the remaining three query back-ends, explaining their main properties and how they can be used to query WebAssembly CPGs.

Native back-end: Wasmati comes with an API for users to implement their queries directly in C++ and integrate them within the Wasmati code base. To this end, a user simply needs to write a file containing a C++ function that describes their query and then adds the query to a configuration file with all the queries to be executed. A complete description of our query API is given in the appendix. By writing queries natively, the user can leverage the high performance and expressiveness of C++. However, users have to re-compile Wasmati every time they need to change or add a query to the system and they have to be acquainted with both C++ and the structure of the Wasmati code-base. The implementation of some search patterns for Wasm code vulnerabilities (e.g., buffer overflows) is also rather complex and prone to programming errors.

Neo4j: The CPG can be automatically loaded into a Neo4j [114] database and run graph queries specified in Cypher, an SQL-like language. Figure 8 shows the Cypher query for finding taint-style vulnerabilities. The execution of lines 1 and 2 yields an interim table of pairs, each consisting of a function node and one of its sink calls. In lines 4-6, we look for source calls that reach any of the sinks stored in the interim table via DDG function edges. Finally, in line 7, we return a table with the computed results, with each result corresponding to a taint-style vulnerability (source name, sink name, enclosing function name). Neo4j relies on sophisticated planning algorithms

for maximizing the query execution performance. From our experience, however, query planning works better for simpler queries, as we have observed significant performance degradation for complex queries involving multiple graph traversals.

Datalog: The CPG can also be automatically loaded into Soufflé [84], a state-of-the-art Datalog engine. Wasmati also provides a library containing various predicates to reason about the structure of WebAssembly CPGs in a declarative fashion. For instance, it comes with a predicate `reachesDDG(X, Y, TYPE_DEP, LAB)` to denote that node `Y` is reachable from node `X` via DDG edges with the label `LAB` of type `TYPE_DEP`. Datalog queries are expressed as predicates for which the Datalog solver will try to find a model. When defining new query predicates, the user can leverage our library of CPG predicates which can result in a rather concise query specification. Figure 9 shows the Datalog query for finding taint-style vulnerabilities: we find a call to a source and a call to a sink using the predicates `sources` and `call` and then we require that the sink be reachable from the source using DDG function edges. On the other hand, Datalog is generally the less-performing back-end.

2.6 Evaluation

This section presents our experiments to evaluate Wasmati. They mainly focus on answering the following questions:

1. Can security vulnerabilities in WebAssembly code be modelled and located using a CPG?
2. Does Wasmati find security vulnerabilities in WebAssembly code collected in the wild?
3. How well does Wasmati scale when generating CPGs from large real-world applications?
4. How do the different graph querying back-ends of Wasmati scale over CPGs generated from large applications?

Next, we present our experimental setup and main findings for each of these questions in a separate section.

2.6.1 Evaluation Using Annotated Datasets

In the first part of our evaluation of Wasmati, our goal is to assess the feasibility of modeling and finding security vulnerabilities in WebAssembly using CPGs. To achieve this, we implemented and executed the ten different queries described in §2.5 over four datasets containing programs written in C with known vulnerabilities. We compiled the datasets to WebAssembly using Emscripten 2.0.9 [51] with level 1 optimization and debug information. For our dataset selection, the vulnerabilities therein contained had to be properly annotated, even if the number of programs in the dataset was relatively small, as this allows us to access the ground truth and evaluate the detection effectiveness of our system. Next, we describe the four datasets that we used, containing a total of 110 C programs.

1. **Basic:** has a total of 37 C programs compiled and created by us during the implementation and testing of Wasmati.

2. **Lehmann:** contains 7 programs from Lehman et al. [93]. The programs are attack primitives and end-to-end exploits. It includes a program using the vulnerable version of `pnm2png` from libpng depicted in Figure 3, a remote execution code in NodeJS and arbitrary file write.
3. **STT:** is a repository of 47 C vulnerable programs from Binary Analysis School [154]. The programs are exploit exercises following the style of Capture the Flag.
4. **CWE:** is a list of "Weaknesses in Software Written in C" with a total of 19 example code snippets [41]. We removed the weaknesses where no query existed targeting it or the vulnerability in C is not ported to WebAssembly. We also removed duplicated code. One removed example is CWE-467 which flags the use of `sizeof()` on a pointer type.

Query expressiveness: We found that all our queries can be encoded using the language primitives provided by C++ (native), WQL, and Datalog. As for Neo4j, queries 7 and 8 required us to develop specific workarounds. The simplicity of Neo4j's Cypher querying language shines in querying for simple logical patterns. However, since it is a declarative language, it does not natively support recursion, which is quite useful for query 7. Cypher also lacks helper data structures like lists or maps that are convenient for query 8. To overcome these limitations, we employed different approaches. For query 7, we emulated recursion by recurrently invoking a simpler query through the Neo4j Python driver. For query 8, we developed a simple aggregation plugin function in Java that builds a map of local buffer indexes and their corresponding buffer sizes. For these reasons we consider Cypher's expressiveness to be relatively more limited for representing our queries than the other query back-ends.

Query conciseness: Table 2.1 presents the lines of code (LOC) of each query written in the four query specification languages supported by Wasmati: C++, WQL, Neo4j, and Datalog. The Datalog queries include a shared library that contains common predicates. The size of this library is 127 LOC. If we exclude this library from the total size (i.e., 1476) and count only the lines of code specific to each query, then the total query size is 206 LOC. We can then see that WQL, Neo4j, and Datalog allow for writing queries with comparable levels of conciseness, averaging respectively 23.2, 26.4, and 20.6 LOC per query. As expected, C++ is considerably more verbose displaying a LOC size 3× larger than the other languages.

Query effectiveness: To gauge how effective our queries are at finding vulnerabilities in our four datasets, Table 2.1 reports on the absolute number of *true positives* (TP), *false positives* (FP) and the number of *present* vulnerabilities in the code (P), i.e., the ground truth. We count a TP as a correctly reported vulnerability and an FP otherwise. We categorize each present vulnerability according to its nature and assign it to the query that best matches its pattern.

From a total of 108 vulnerabilities in 110 programs, 100 were correctly reported (TP) and 8 were false positives (FP). The false-positive rate, which computes the proportion of incorrect reports (FP) in relation to the total results (P), is 7.41%. The precision, which reflects the proportion of true positives in the total reports made by the queries, is 92.59%. There are also 8 undetected vulnerabilities (FN): 3 in Basic, 1 in Lehmann, 3 in STT, and 1 in CWE. These anomalies, both in missed vulnerabilities and ill-classified ones, occur only in format strings (query 1) and buffer overflows in static buffers (query 8). All other queries have detected all existing vulnerabilities in the datasets while flagging no false positives. Consequently, Wasmati's recall for these datasets is 92.59%. This means that Wasmati achieves both high recall and high precision, which shows its effectiveness in detecting vulnerabilities in WebAssembly binaries.

In the case of query 8, two unreported vulnerabilities in the basic dataset arise from calls to `printf()` using global static buffers as its first argument. In WebAssembly, both constant

	Basic		Lehmann		STT		CWE		Total	
	TP	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
Wasmati	35	0	6	0	40	8	19	0	**100**	**8**
Joern	10	17	0	6	24	38	4	4	**38**	**65**

Table 2.2: Comparison of aggregate results between Wasmati (analysis of WebAssembly binary compiled from C code) and Joern (analysis of original C code).

data strings and global static buffers are indistinguishable from each other and, as a result, the vulnerability is not reported as such. In the STT dataset, a total of 8 false positives were reported by the query. Analyzing the binary, we find that when there are multiple `printf` calls sequentially. The way that the WebAssembly compiler stores the constant format templates leads the query to wrongly perceive its logic. As for the buffer overflows in static buffers, a total of 4 (all in global static buffers) were not reported. The buffer is stored in the data section and referred to by a constant pointer. The query cannot infer the size of a global static buffer so, it omits the report.

Comparison with Joern: The four datasets used to evaluate Wasmati contain programs written in C with known vulnerabilities. The results presented above show that these vulnerabilities persist in the corresponding WebAssembly binary after compiling the C code and that Wasmati can successfully detect these vulnerabilities. To offer a comparison between Wasmati CPGs and the CPGs originally developed to detect vulnerabilities in C/C++ code [170], we decided to perform an evaluation of Joern [83], which is the open-source implementation of the original published work that is actively maintained. Note that it is also the basis of a commercial tool developed by ShiftLeft [137]. Consequently, although Joern is actively maintained, we expect it not to include the latest detection features or perform on par with its commercial version. We used the queries shipping with Joern, which are aimed at detecting the most common vulnerabilities in C code, and executed Joern directly on the C files of each dataset. Table 2.2 shows the aggregate results of Joern and Wasmati. From a total of 108 vulnerabilities in 110 programs, Wasmati correctly reported 100 vulnerabilities (TP) and reported 8 false positives (FP), while Joern correctly reported only 38 vulnerabilities (TP) and reported 65 false positives (FP). This further shows that CPGs can be successfully applied to WebAssembly for detecting vulnerabilities, with Wasmati achieving 92.59% recall and 92.59% precision, and Joern 35.19% recall and 36.89% precision when analyzing the original C files. We infer that this discrepancy in the results is directly linked to the coverage of the queries implemented by both tools and does not necessarily mean that CPGs are less effective for C code. In Wasmati we focused on implementing comprehensive queries, while the open-source Joern implementation offers only 14 queries covering dangerous functions, format strings, buffer overflows, use-after-free, and others, corresponding to 101 out of 108 total vulnerabilities in the datasets.

2.6.2 Vulnerability Detection in the Wild

In this second part of our evaluation, our goal is to assess Wasmati using WebAssembly binaries deployed in the wild. To this end, under ideal conditions, we would like to test Wasmati on a dataset that: i) contains a representative and large collection of real-world WebAssembly binaries, and ii) comes with ground truth annotations that allow us to determine whether or not the vulnerabilities detected by Wasmati are real. In the absence of such an ideal dataset, we used

Deployment Location	Vuln. Binaries	Total Collected
github	3,761 (8.5%)	44,218
web/httparchive	86 (33.0%)	261
web/own-crawler	264 (9.0%)	2,923
npm/wasm	254 (7.3%)	3,488
wapm	23 (19.0%)	122
firefox-extensions	7 (24.1%)	29
manual	17 (35.4%)	48
survey	9 (20.0%)	45
npm/top	3 (21.4%)	14
Total	4,424 (8.6%)	51,148
Total (unique binaries)	3,140 (37.1%)	8,461

Table 2.3: Deployment location of binaries for which Wasmati detected at least one vulnerability. Percentage in relation to total number of collected binaries for that deployment location. Note that, in the dataset [78], the same binary might be collected from different sources, which leads to duplicates across lines in the table.

the dataset curated by Hilbig et al. [78]. It consists of 8,461 unique binaries collected from several sources (repositories, package managers, and websites), and it is not annotated.

We started by generating the CPGs for all the WebAssembly binaries of this dataset. From the 8,461 binaries, Wasmati has successfully generated CPGs for 7,879 (93.1%) binaries. Of the remaining 582 binaries, 561 (6.6%) failed mainly due to unsupported WebAssembly features in the binaries (e.g. threading, multi-return values, bad sections, etc..) and 21 (0.3%) exceeded the maximum allocated 16 GiB of RAM. These 21 programs had a mean size of 61 MiB. After gathering the generated CPG, we ran the 10 queries listed in Table 2.1. Next, we present the main findings of our analysis.

Provenance of potentially vulnerable binaries: Table 2.3 shows the deployment location of WebAssembly binaries for which Wasmati detected at least one potential vulnerability (in the dataset [78] this information corresponds to the binary collection method). We see that the original dataset is heavily skewed, where 86.5% of all binaries of the dataset originate from Github. This helps explain why the most vulnerable binaries detected by Wasmati (3,761) were originally found in GitHub repositories. We also observed that many of these binaries were obtained from repositories owned by security researchers, who collect selected WebAssembly binaries for research purposes. Beyond Github, Wasmati detects vulnerabilities in a considerable percentage of binaries that reach the production stage, such as real websites (*web/httparchive*), at the WebAssembly Package Manager (*wapm*), in *firefox-extensions* and at the Node Package Manager (*npm/top*). This demonstrates the feasibility of applying Wasmati for the analysis of real WebAssembly binaries.

Characterization of potentially vulnerable binaries: Table 2.4 shows the number of vulnerabilities detected by Wasmati in the curated dataset, characterized by vulnerability type. In total, Wasmati flags 83,202 potential vulnerabilities in 3,140 unique binaries deployed in the wild. We discuss three main observations:

1. A large number of taint-style vulnerabilities: In our query for detecting this type of vulnerability, any data external to the WebAssembly binary is considered tainted. This query flags a potential vulnerability in two conditions: i) data dependent on a tainted input reaches a known dangerous sink inside the WebAssembly module (e.g. `memcpy`), or ii) data dependent on a tainted input

Language	FS	TV	UAF	DFree	BO	DFunc	Binaries
C	654	8941	6	5	90	64	49
Rust	-	62	-	-	343	-	73
Unknown	925	71821	-	-	199	92	3018
Total:	1579	80824	6	5	632	156	3140

Table 2.4: Number of vulnerabilities detected by Wasmati: format strings (FS), tainted variable (TV), use after free (UAF), double free (DFree), buffer overflow (BO) and dangerous function (DFunc).

is reflected to the calling environment (e.g. when the WebAssembly module calls a JavaScript function). Wasmati detects that such data flows can be frequent. A closer analysis revealed a heavily skewed distribution, with 73.6% (59486) of the total tainted-variable vulnerabilities being located in 10% of the flagged binaries. Furthermore, we found multiple binaries that are similar, as is the case of the first and second binaries with most tainted variables flagged (3028 and 2973 respectively) that are compilations from libxml2. This library has many imported functions from JavaScript that handle HTML and XML and are good candidates to consider as vulnerable sinks.

2. Usage of dangerous functions: In Table 2.4 we can see 156 uses of dangerous functions: `gets` (1 occurrence) and `strcat` (155 occurrences). These are generally considered dangerous regardless of how they are used [40]. In fact, some C/C++ compilers warn the developer when `gets` is used since it was removed from ISO/IEC 9899:2011 [81]. Yet, these functions persist in WebAssembly code.

Case studies: Given that i) no tool still exists to automatically validate the vulnerabilities reported by Wasmati, and ii) the manual inspection of such a large WebAssembly codebase would be extremely laborious, we sampled a few vulnerabilities reported by Wasmati to analyze manually. From this exercise, we identified two interesting cases where reported buffer overflows may actually result in exploitable vulnerabilities unless the provided inputs are properly sanitized by external JavaScript code. In one case, we identified 4 binaries that use vulnerable versions of the libpng library. Wasmati detected the buffer overflow vulnerability in libpng reported in CVE-2018-14550 [38] (see Figure 3). This demonstrates the transfer of real vulnerabilities to WebAssembly in the wild. In a second case, we confirmed the existence of a buffer overflow in a WebAssembly binary that pertains to a Firefox Extension for a Cardano ADA Wallet. Wasmati identifies that the function `__wbg_publickey_free` is callable from JavaScript. Wasmati treats its parameter (a pointer) as tainted and tracks it as it reaches a sensitive allocation sink (`memcpy`). This pointer does not undergo sanitization until it reaches `memcpy`; if the value gets corrupted somehow in the JavaScript code, it can result in a possible buffer overwrite and memory corruption that can chain into other problems (e.g. code execution, or signature bypass/forgery).

2.6.3 CPG Generation Performance

In this section, we assess the scalability, overall performance, and resources allocated in the generation of the CPG. To achieve these goals we generated CPGs from two well-known industrial benchmarks: PolybenchC [131] and SPEC CPU 2017 [144]. The experimental evaluation was performed in a 64-bit Ubuntu 18.04LTS with 16GB RAM and 4 Intel Core i7-4700HQ 2.40GHz CPUs with all runs using the same configuration file, which specifies sources, sinks, etc.

Source	C	C++	Average (binary)
Instruct. (k)	497.5k	1M	713.0k
Size (KiB)	1,167	2,697	1,797
Nodes	525.7k	1.1M	747.6k
Edges	1.6M	9.7M	4.9M
Memory (MiB)	107.88	419.50	236.17
Time	37s	1min 43.5s	57.79s
Exported (MiB)	8.74	42.01	22.44

Table 2.5: Average CPG generation times and information from SPEC CPU 2017 binaries. Complete data by binary in appendix.

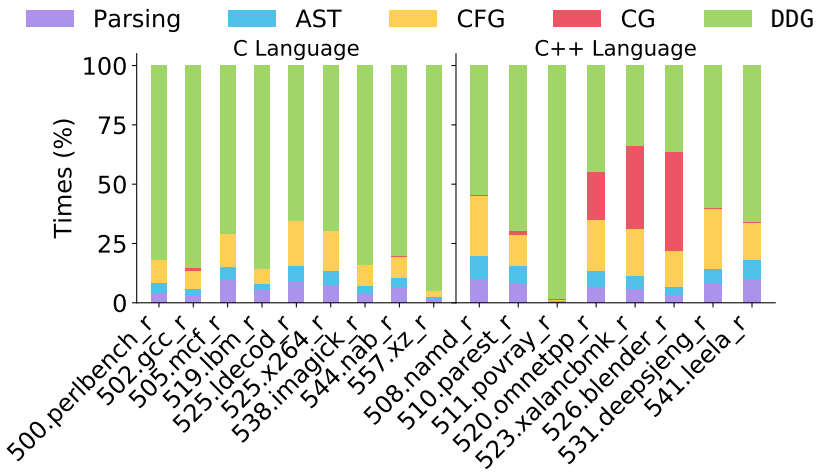

Figure 2.5: Time (%) per CPG's construction stage.

Scalability assessment: We generated CPGs for a subset of the SPEC CPU 2017 benchmark comprising a total of 17 binaries, from C and C++ source, compiled using Emscripten 2.09. The results are shown in Table 2.5. On average, the binaries are composed of 713k instructions with 502.gcc and 526.blender peeking in 2.9M and 3.2M instructions respectively. The constructions of the CPG took an average of 58 seconds per binary with a total of about 16min22s. We can see that even the largest binaries generated their CPG in less than 4min30s using up to 1.73 GB of RAM.

Figure 2.5 presents a more detailed analysis of the time taken to construct the CPG per benchmark, showing how much time is employed by each stage of the construction of the CPG, namely: parsing, construction of the AST, construction of the CFG, construction of the CG, and construction of the DDG. As expected, the DDG construction captures the most time of the construction time. Some exceptions can be seen in 2 C++ programs for which most of the time of the CPG construction is spent on building the call graph (most likely due to the presence of many call indirect instructions).

Figure 2.6 represents the CPG construction time in terms of the graph size, counting the total number of nodes and edges. The plot includes the sizes of all 158 binaries from our four datasets and two benchmarks. The regression follows a power series with $R^2 = 0.929$. This result shows that the CPG construction is polynomial in function of the graph's size and can scale to larger programs.

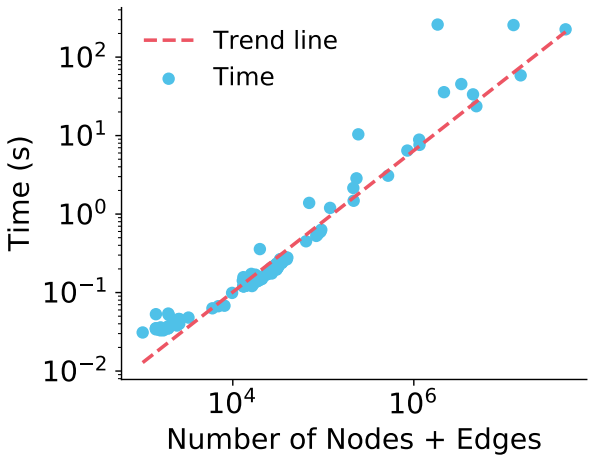

Figure 2.6: CPG construction time by size.

Comparison against related systems: To put Wasmati's results in perspective, we compared it against the closest-related system in the literature named Wassail [151]. Wassail is a static taint analysis tool for WebAssembly programs focused exclusively on information flow analysis. In contrast to Wasmati, which generates full-blown CPGs, Wassail generates a much simpler graph data structure consisting only of the control flow graph (CFG) of the program followed by a data flow analysis propagation over the CFG. Wassail was written in OCaml and evaluated using the PolybenchC suite, which is composed of 30 C programs. To compare the performance between Wasmati and Wassail, we tested both systems against PolybenchC. The data structure computed by Wassail is comparable to the DDG generated by Wasmati. Figure 2.7 displays the execution time of each tool for each program of the benchmark. On average, Wasmati is 9.1x faster than Wassail, demonstrating that our framework outperforms a simpler static analysis tool for WebAssembly.

Note that Wassail is not a vulnerability scanning tool. Wassail was designed only to generate summaries that describe where the information can flow within functions. It offers no functionality that allows us to search for vulnerabilities in WebAssembly binaries. For this reason, we cannot use it as a baseline for evaluating Wasmati's effectiveness in detecting the vulnerabilities analyzed in Table 2.1.

2.6.4 Query Execution Performance

Lastly, we measure the query execution time over the SPEC CPU 2017 benchmark to assess its scalability. Table 2.6 shows the execution times (in seconds) of the WQL queries described in Table 2.1 following the same numbering system. The first six queries are all similar in complexity, which translates into relatively similar execution times for the same binary. On average, a binary took less than 77 seconds to run all ten queries, totalling an execution time of around 22 minutes for all binaries. The larger binary, with 3.4M nodes and 44.1M edges, took about 8 minutes to complete all queries.

Figure 2.8 compares the execution times between the query back-ends. Neo4j and Datalog were not capable of executing all queries for the SPEC dataset before an established timeout

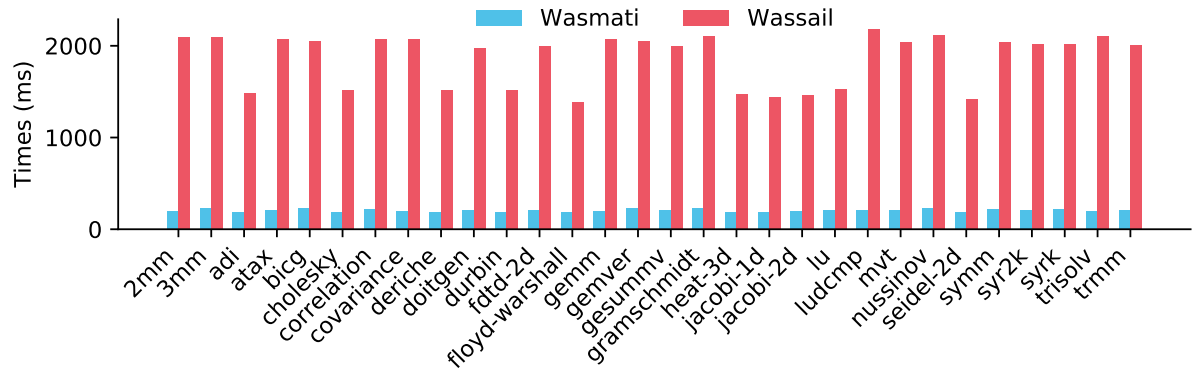

Figure 2.7: Wasmati's generation time comparing to Wassail.

Query	1	2	3	4	5	6	7	8	9	10	Total
Average	4.2	10.2	174.2	1.3	1.4	1.0	0.1	5.5	1.2	2.0	201.4
Total	90.8	105.7	93.4	93.8	102.1	101.4	2.6	490.8	94.2	129.9	1304.5

Table 2.6: Overview of WQL execution time for SPEC binaries (in seconds). The complete results can be found in the appendix.

of 10 minutes. In these cases, we capped the query execution time to the 10-minute mark, and proceeded to calculate the described averages.

As it is clear from Figure 2.8 a), the native back-end outperforms all other back-ends, but at the cost of a harder query specification process. Datalog shows consistently high execution times, with the worst result being a 100× increase in query 7 when compared to the native back-end. Neo4j shows reasonable performance for simpler queries, such as queries 1 to 6, but quickly explodes in more complex queries, some of which did not finish before a timeout occurred. Finally, WQL shows significantly better performance than Neo4j and Datalog for most queries. Compared to native execution times, WQL shows, in the worst case, a 3× overhead, but the queries are much simpler to express and the execution times are still practical.

In Figure 2.8 b), six binaries show visible spikes in the execution times. These cases also include timeouts for Datalog or Neo4j. These are also the largest binaries of the SPEC dataset, which means that CPG analysis performance is directly related to the size of the graph, and Neo4j and Datalog are particularly less efficient at analyzing larger CPGs than the native and WQL back-ends.

2.7 Limitations

The precision of Wasmati's results is bounded by the queries executed. As such, the ten queries discussed in Section 2.5 may not encode all the patterns that match a specific vulnerability type. For example, a taint vulnerability seizes to exist if proper sanitization is employed, but our taint queries do not check if common sanitization functions are called on the tainted data.

Wasmati can only analyze individual Wasm binaries. To uncover actual exploitable vulnerabilities in real applications, the analysis must also consider the application components that interact with the WebAssembly binary. An interesting direction for future work is to extend

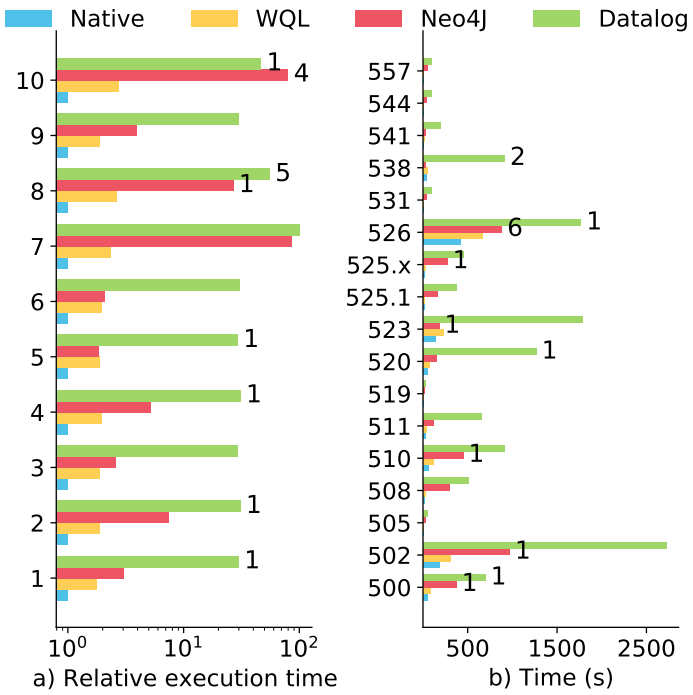

Figure 2.8: Execution time comparison across back-ends; a) shows the relative average time for each query over the SPEC dataset. The x-axis is in log scale; native is 1, and all other numbers are relative to native (lower is better); b) shows the total execution times, in seconds, of running all 10 queries in the different SPEC binaries (y-axis). Beside each bar is the number of timeouts that occurred for a query (a), or during the execution of all queries for a binary (b).

Wasmati to model JavaScript calls to exposed functions of a WebAssembly binary, as well as model accesses to the JavaScript array buffer used as the binary's linear memory.

2.8 Related Work

Empirical studies on WebAssembly: Some relevant studies characterize the performance of WebAssembly engines [132] as well as the prevalence [111] and security [93, 78] of WebAssembly code in the wild. The latter studies on security are of special interest to us, giving a comprehensive account of WebAssembly security vulnerabilities and how they can be exploited [93] and providing a dataset of real-world binaries that we use in our evaluation [78].

Program analyzes for WebAssembly: Since the proposal of the WebAssembly standard [133], several program analyses have been designed to tackle the specificities of the language. Haas et al.[74] proposed a small-step operational semantics for WebAssembly together with a type system for checking the safety of stack-based operations. Later, Watt [163] mechanized both the semantics and the type system introduced in [74]. The proof infrastructure of [163] was then re-used to formalize and prove the soundness of CT-Wasm [165], a type-driven extension of WebAssembly for provably secure implementation of cryptographic algorithms. More recently, the authors of [164] introduced Wasm Logic, a program logic for modular reasoning about heap-manipulating WebAssembly programs. So far, most practical tools for WebAssembly analysis are based on dynamic analysis. Wasabi [94] is a general framework for instrumenting Wasm binaries and can

be used to implement different types of dynamic analyses. To the best of our knowledge, there are only three taint analysis tools for WebAssembly: TaintAssembly [65], the tool presented in [155], and Wassail [151], with the former two being dynamic and the latter static. Wassail implements a data flow analysis algorithm that has not been tailored for vulnerability detection. In particular, unlike Wassail, Wasmati specifically tracks constant and function dependencies, which are fundamental to reason about a large number of security vulnerabilities. Importantly, the authors of [151] do not demonstrate how Wassail can be used to enable a vulnerability detection tool. Wasmati is the first CPG-based framework for detecting vulnerabilities in WebAssembly code.

Code Property Graphs: CPGs have been applied to find SQL injection, XSS, and CSRF vulnerabilities in PHP applications [19, 127] and, most recently, for detecting CSRF vulnerabilities in client-side JavaScript [86], all on top of Neo4J's backend. CPGs are at the core of CodeQL [67], a commercial tool for detecting security vulnerabilities in multiple programming languages, but not WebAssembly. Joern [83] leverages CPGs to help developers detect vulnerabilities in C/C++ code. Although there is an open-source version of this tool, we did not use it for building Wasmati for three reasons: i) it is a very complex project, ii) Joern's CPGs are generated from an intermediate code representation which hampers their portability and testing on various query back-ends, and iii) developing a custom DSL in C++ helps reduce the overheads of running the queries inside Joern, which runs on top of Scala, a JVM target language.

2.9 Conclusion

In this paper, we presented Wasmati, a static analysis tool for finding vulnerabilities in WebAssembly. It employs optimized techniques for generating CPGs for complex WebAssembly code, as well as four distinct back-ends for query execution. We implemented ten queries for each of the four execution back-ends, capturing different vulnerability types, and extensively tested Wasmati with four heterogeneous datasets. Wasmati can scale to large real-world applications and efficiently find vulnerabilities for all tested query types. **Availability:** Wasmati is publicly available at `https://github.com/wasmati/wasmati`.

Acknowledgements

This work was supported by Fundação para a Ciência e Tecnologia (FCT) via the SFRH/BD/146698/2019 grant, UIDB/50021/2020 (INESC-ID), project INFOCOS (PTDC/CCI-COM/32378/2017), and by Instituto Superior Técnico, Universidade de Lisboa.

```
(module
    (func $get_token (param $pnm_file i32)(param $token i32)
    (local $i i32)
    (local $ret i32)
    ;; (...)
    loop $L4
    block $B5
        local.get $pnm_file
        call $fgetc
        local.tee $ret
        i32.const -1
        i32.eq ;; ret == EOF
        br_if $B5
        local.get $token
        local.get $i
        i32.const 1
        i32.add ;; i++
        local.tee $i
        i32.add ;; token + i
        local.get $ret
        i32.store8 ;; token[i] = ret
        local.get $ret
        i32.const 10
        i32.eq ;; token[i] == '\n'
        br_if $B5
        local.get $ret
        i32.const 13
        i32.eq ;; token[i] == '\r'
        br_if $B5
        local.get $ret
        i32.const 32
        i32.ne ;; token[i] == '\r'
        br_if $L4
    end
    end
    ;; (...)
    i32.const 0))
```

Listing 5: WebAssembly textual representation of the buffer overflow vulnerability in libpng (CWE-2018-14550)

```
foreach func in functions():
    nodes := [n in instructions(func) : (n.instType = "Call") && (n.label = "$malloc")];
    foreach n1 in nodes:
        descendants := [
            n in descendantsCFG(n1) :
            (n.instType = "Call") && (n.label = "$free") &&  reachesDDG(n1, n, "Function", "$malloc")
        ];
        foreach n2 in descendants:
            uafs := [n in descendantsCFG(n2) : reachesDDG(n1, n, "Function", "$malloc")];
            if (!uafs.empty())
                vulnerability("Use after free", func.name, "$free");
```

Listing 6: Use-after-free WQL Query.

```
foreach func in functions():
    sinkCalls := [n in instructions(func) : n.instType = "Call" && n.label in sinks &&
                    !([e in n.inEdges : e.type = "DDG" && e.ddgType = "Function" &&
    foreach sink in sinkCalls:
        vulnerability("Tainted", func.name, sink.label);
```

Listing 7: Taint-flow WQL Query.

```
MATCH (f:Function)-[:AST*1..]->(sink:Instruction)
WHERE sink.instType="Call" AND sink.label IN sinks

WITH * MATCH (src:Instruction)-[:DDG*1..]->(sink)
WHERE src.instType="Call" AND src.label IN sources
    AND (source_call)-[:DDG*1.. {ddgType:"Function", label:src.label}]->(sink)
RETURN src.label as source, sink.label as sink,
    f.name as function;
```

Listing 8: Tainted source-to-sink in Neo4j CQL.

```
taintedFuncToFunc(FUNC_NAME, Y, SINK) :-
    sources(SOURCE), call(X, SOURCE, _, _),
    reachesFunc(FUNC_NAME, X),
    sinks(SINK), call(Y, SINK, _, _),
    reachesDDG(X, Y, "Function", SOURCE).
```

Listing 9: Tainted source-to-sink in Datalog.

3 Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages

Publication Data

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

Abstract

With the emergence of the Node.js ecosystem, JavaScript has become a widely used programming language for implementing server-side web applications. In this paper, we present the first empirical study of static code analysis tools for detecting vulnerabilities in Node.js code. To conduct a comprehensive tool evaluation, we created the largest known curated dataset of Node.js code vulnerabilities. We characterized and annotated a set of 957 vulnerabilities by analyzing information contained in *npm* advisory reports. We tested nine different tools and found that many important vulnerabilities appearing in the OWASP Top-10 are not detected by any tool. The three best-performing tools combined only detect up to 57.6% of all vulnerabilities in the dataset, but at a very low precision of 0.11%. Our curated dataset offers a new benchmark to help characterize existing Node.js code vulnerabilities and foster the development of better vulnerability detection tools for Node.js code.

3.1 Introduction

JavaScript has become one of the most popular programming languages for implementing server-side web applications. A driving factor in this trend has been the emergence of Node.js [115]. Node.js is a cross-platform, back-end runtime environment that executes JavaScript code. Essentially, it can be used as a web container, housing JavaScript code that handles HTTP requests. Pivoted around Node.js, there is also an ecosystem of third-party packages managed by the Node

The reproduction of this publication was slightly adapted to adhere to formatting requirements. The original version of this publication can be found at: `https://ieeexplore.ieee.org/document/10168679` and `https://arxiv.org/pdf/2301.05097.pdf`.

Package Manager (*npm*). Currently, *npm* stores thousands of packages that web developers can readily import into their code, either for writing web applications or other packages.

The widespread adoption of Node.js makes the development of effective JavaScript vulnerability scanners a pressing matter. For one, the JavaScript [50] language features various constructs that display subtle behaviours. When employed by inexperienced code developers, these constructs may all too easily lead to the introduction of vulnerabilities. In addition, the manual detection of code vulnerabilities is complicated by the intricate *npm* inter-package dependency system. In some cases, correct packages may become the source of security bugs as a result of ill-use by other packages. In others, buggy packages may end up propagating vulnerabilities up in the dependency chain to correct packages [173]. This combination of factors opens up the path for serious security breaches in web applications. By exploiting security bugs, an attacker may be able to take over the entire server and/or affect many users through SQL injection, remote code execution, and other attacks [147, 146].

An effective technique to prevent security vulnerabilities from creeping into production code is to integrate security analysis tools as part of Continuous Integration/Continuous Deployment (CI/CD) pipelines. Using automatic vulnerability detection tools, developers can seamlessly receive prompt feedback about potentially existing security flaws in their code. This enables them to apply the necessary fixes at an early code development stage, thus helping them to improve the reliability of their software. In the same vein, JavaScript developers can benefit from code analysis tools that allow them to detect and fix security flaws inside *npm* packages. Ideally, such tools should have high detection quality (i.e., low false-positive rate), and high coverage (i.e., low false-negative rate).

Motivated by this need, we set out to evaluate the effectiveness of existing JavaScript vulnerability detection tools at analyzing Node.js packages. We found a large body of work on client-side JavaScript security [152, 91, 150], and some recent work in the study of vulnerabilities in *npm* packages [147, 146]. However, no prior work has focused on evaluating tools that analyze server-side JavaScript code vulnerabilities, let alone on studying their effectiveness at finding security flaws in *npm* packages. As it turns out, performing this task is rather involved, given the absence of a gold standard for classifying such tools, and the lack of a comprehensive vulnerability dataset that can be used for benchmarking purposes.

In this paper, we present the first empirical study aimed at evaluating existing JavaScript vulnerability detection tools on Node.js packages. We focus exclusively on fully automatic, static code analysis tools that can be used in CI/CD pipelines. This excludes tools [147, 72, 66] that expect additional customization, such as some test suites that require specific application-dependent inputs, or tools that perform simple checks on known vulnerable dependencies [117, 140]. In total, we screened 40 analysis tools for JavaScript and selected nine that can detect vulnerabilities at continuous integration time: NodeJsScan [2], CodeQL [67], ODGen [97], Graudit [98], InsiderSec [80], ESLint SSC [56], Microsoft's DevSkim [104], Mosca [35] and Drek [90]. We executed them against a curated dataset created by us containing *npm* packages with annotated vulnerabilities, mainly: path traversal, cross-site scripting, insecure transfer using HTTP, resource exhaustion/denial-of-service, prototype pollution, OS command injection, code injection, and improper input validation. Then, we checked whether these tools can correctly identify these vulnerabilities.

Given that there is no curated dataset of Node.js vulnerabilities, our first step was to develop our own. Building this dataset was in itself a challenging endeavour because we needed to identify real vulnerabilities in a large dataset of *npm* packages. Our starting point was the *npm*

system itself. The *npm* system runs a vulnerability report service that results in the generation of so-called *advisory* reports. These consist of textual descriptions of security vulnerabilities identified inside specific packages. These are real security vulnerabilities collectively identified by the Node.js developer community. Reports may also include advice to upgrade the package to a fixed version. As such, advisory reports provide a reliable source for building our dataset. Unfortunately, these reports are not represented in a format that allows for automatic processing. Moreover, some of them may contain errors and therefore cannot be used unless a thorough analysis and verification are performed.

To overcome these difficulties, we manually analyzed 1359 advisory reports covering an equal number of vulnerable *npm* package versions. These advisories represent 74% of all the vulnerabilities officially reported inside benign *npm* package versions until June 2021. In this process, we identified several anomalies in the advisory reports. We have then generated a curated dataset covering 957 of these advisories extended with annotations that specify the precise location of the reported code vulnerabilities. We found that the location of a large fraction of existing vulnerabilities can be fully expressed through *source-sink* pair annotations. Our dataset can help the research community to i) characterize the vulnerabilities already detected within the *npm* ecosystem, and ii) benchmark vulnerability detection tools. Our dataset is publicly available[1].

We tested the pre-selected tools against our dataset and found they perform rather poorly, missing many vulnerabilities (low true positive rate/recall) and showing a high false positive rate (low precision). On average, they were able to correctly identify only 15.1% of the total number of vulnerabilities in our dataset. The combination of the three best-performing tools detects 57.6% of all vulnerabilities, albeit with only 0.11% precision. The best-performing tools, ESLint SSC and CodeQL, manage to detect 41.5% and 31.3% across all types of vulnerabilities and reach their peaks when it comes to identifying prototype pollution (79.2% for CWE-471 and 86.1% for CWE-1321) and path traversal (71.2%) vulnerabilities, respectively. Of the 957 known vulnerabilities in the dataset, 324 (33.8%) were not detected by any of the selected tools. Some of the causes are tied to fundamental limitations of state-of-the-art code analysis techniques when it comes to analyzing server-side, JavaScript code vulnerabilities in the *npm* ecosystem. Addressing these limitations is an interesting research direction for future work.

In summary, our paper makes four contributions: (i) a curated dataset with 957 real-world vulnerabilities in *npm* package versions, which will be fundamental to evaluate future advancements in static analysis tools for the detection of vulnerabilities in Node.js applications, (ii) a survey of existing vulnerability detection tools for JavaScript / Node.js code, (iii) a quantitative assessment of the vulnerability detection toolset against our curated dataset, and (iv) a study of the main causes of missing important vulnerabilities in *npm* packages, which opens up several research avenues in this field.

3.2 Study Design

3.2.1 Background

Node.js features a package manager system named Node Package Manager (*npm*), which currently stores thousands of third-party packages. Presently, it is difficult to guarantee the

[1] `https://github.com/VulcaN-Study/Supplementary-Material`

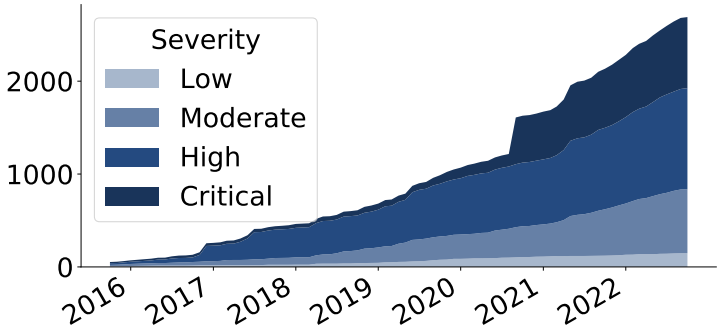

Figure 3.1: Evolution of published advisories over time.

absence of security vulnerabilities in packages uploaded to *npm* because there is no systematic code triage in place. Consequently, the *npm* community mainly relies on third-party vulnerability reports to identify the potentially vulnerable packages that have already been included in the ecosystem.

The *npm* system alerts JavaScript developers whenever they use a package version reported as vulnerable. These vulnerability alerts are called *npm advisories*, as their purpose is to advise developers to update the vulnerable dependencies to either a fixed package version or to select another package entirely. When someone identifies a potential vulnerability in an *npm* package, they produce a vulnerability report and submit it to the *npm* security team, which checks the report, notifies package maintainers, and publishes the advisory, either when the package maintainers release a fix or if they remain unresponsive for longer than 45 days [6]. Typically, an advisory report includes the package name, affected versions, description of the vulnerability, effects and references, commits, and/or code examples that help trigger the vulnerability. This information is then made available to developers on the advisory page (see example page for advisory 315 / GHSA-cwcp-6c48-fm7m in [5]). Figure 3.1 displays the evolution of the number of published advisories since *npm*'s inception until October 11$^{\text{th}}$ 2022, according to their severity level. This number has steadily grown to nearly 40 advisories per month; about 70% cover vulnerabilities with a risk factor of high/critical severity.

3.2.2 Research Questions and Scope

In this work, we investigate four main research questions:

RQ1. How to obtain an annotated dataset of vulnerabilities in *npm* packages? To evaluate JavaScript vulnerability detection tools, we require an annotated vulnerability dataset to compare the output of a given tool against ground truth data. The *npm* repository provides an excellent source for retrieving both (i) an extensive collection of vulnerable Node.js package versions, and (ii) information about real-world vulnerabilities (documented by the advisory database). Unfortunately, this information cannot be used as-is from existing advisories. First, advisories often lack relevant information about the reported vulnerability (e.g., the exact code location of the vulnerability within the package). Second, in many cases, most of the explanations regarding the reported vulnerability are given in external references, where information tends to be inconsistent and unstructured. Third, some advisories may be incorrect in places (e.g., the classification of the vulnerability type), which may lead to the mischaracterization of existing JavaScript vulnerabilities. These obstacles preclude an automated advisory analysis approach.

RQ2. What is the state-of-the-art of existing security-oriented static analysis tools for Node.js code? We are interested in assessing which static vulnerability detection tools are available. We need to distinguish between a broader set of code analysis tools which can serve many different purposes (e.g., detecting programming malpractices) from those that are specifically oriented toward the detection of vulnerabilities. Furthermore, we aim to analyze which code analysis techniques are employed by these tools. This will help us to better assess the strengths and weaknesses of each technique when evaluating each tool.

RQ3. How effective are available detection tools in uncovering vulnerabilities in JavaScript code? We are interested in empirically determining and characterizing how precise the publicly available vulnerability detection tools are at identifying vulnerabilities in known vulnerable JavaScript code.

RQ4. What are the main reasons for missing the detection of vulnerabilities? We aim to understand the key limitations of existing static vulnerability detection tools that explain their failure to detect known vulnerabilities in JavaScript code.

The research questions above have a twofold goal: (i) characterize vulnerabilities in *npm* packages in the wild, and (ii) evaluate the effectiveness of existing static JavaScript vulnerability detection tools. To better set the reader's expectations about our study, we further clarify these subgoals and discuss other relevant directions we left outside the scope of this work.

Firstly, our study is narrowed toward the characterization of vulnerabilities reported inside *npm* packages. As such, our results cannot be extrapolated to characterize the typical programming flaws introduced by JavaScript developers during the code development stage; nor is this our goal. Developers may use vulnerability detection tools in their continuous integration frameworks that may capture some vulnerabilities before the code is deployed to *npm*. This means that some security flaws may have been fixed prior to deploying the code into production. Also note that, given that we only analyze formerly identified vulnerabilities, our curated dataset may not be representative of all existing JavaScript vulnerabilities lingering inside *npm* packages; this is also not our purpose. In contrast, we intend our curated dataset contain a large set of confirmed real-world vulnerabilities that can be used for assessing existing (and future) vulnerability detection tools.

Secondly, we focus exclusively on *fully automatic vulnerability analysis tools* that can be easily integrated into existing code review pipelines. This excludes the only three existing dynamic analysis tools for detecting vulnerabilities in Node.js code: SyNode [147], NodeSec[72] and Affogato [66]. These tools require a set of unit tests covering all security-sensitive behaviours of the package to be analysed. Such test suites must be manually written as the automatic generation of high-coverage test suits for Node.js applications is still an open problem. Nevertheless, our curated dataset can still be used as a benchmark for evaluating the effectiveness of these excluded tools.

3.2.3 Study Methodology

Our approach to answering the questions above is to perform an empirical study consisting of the following three tasks:

Task 1. Manual analysis of advisory reports and building the annotated dataset: We manually analyzed all advisories, filling in the missing information, namely by identifying the

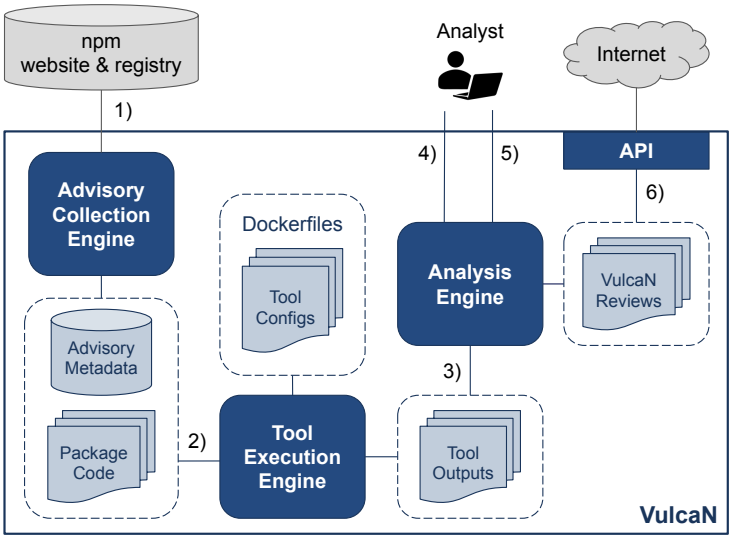

Figure 3.2: VulcaN architecture.

code location that triggers the vulnerability. This information is part of our curated dataset used in the following tasks.

Task 2. Selection and execution of code analysis tools for vulnerability detection in JavaScript code: We searched both the literature and the web for code analysis tools that can be used for vulnerability detection. We executed all of the selected tools on all the vulnerable packages in our curated dataset.

Task 3. Evaluation of tool output according to the ground truth extracted from the annotated dataset: We automatically compare a tool's result with the corresponding dataset annotations, considering the verbosity levels of each tool.

To support our methodology, we developed VulcaN, a testbed for analyzing vulnerability detection tools for Node.js packages. VulcaN is an execution and analysis framework to collect vulnerable versions of *npm* packages, run vulnerability detection tools over all collected packages, and help security analysts perform the vulnerability analysis. Currently, VulcaN supports nine tools (see Section 3.4) and has these main features:

- an interface to download the latest published advisories;
- a Docker-based extension for adding more analysis tools;
- an interface for accessing both the metadata and the output of the analysis tools for each advisory;
- an interface for annotating each advisory with a review file, which contains the code location of the vulnerability and the classification of the results of the analysis tools;
- an interface for automatically comparing the output of the tools with the corresponding review in our curated dataset, this includes a parser for the output of each tool;
- a web API exposing the curated dataset and the classification of the analysis tools.

Figure 3.2 shows the architecture of VulcaN and how each component is used in the context of our empirical study. First, the *Advisory Collection Engine* crawls the *npm* advisory website [116] and collects the available metadata about all published advisories saving it in a *MongoDB* database (1). The latest vulnerable version for each advisory, referenced in the collected metadata, is then downloaded via the *npm* registry. Second, the *Tool Execution Engine* builds a docker image for each tool, according to the specified *Dockerfile*, and runs the respective container for all downloaded packages, storing the output of each tool locally (2). Third, the advisory metadata, code, and tool outputs are fed to the *Analysis Engine*, which generates an environment for advisory report analysis (3). The analyst accesses the environment (4) and produces a review file comprising an accurate description of the vulnerable code location, which is then submitted to the framework (5). Finally, the *Analysis Engine* processes the submitted information, automatically compares the tools' outputs with the analyst reviews (dataset) using a dedicated parser for each tool, and exposes it through a web API (6).

We developed the VulcaN pipeline using Python3 and Bash scripts totalling 721 and 1095 lines of code, respectively. The Analysis Engine for advisory report analysis provides a web-based interface built using Flask@1.1.2 and a Mongo database. VulcaN executes every component inside individual Docker containers using Docker@20.10.23 and the Ubuntu and Python3 base images. The tools generated textual, JSON and SARIF files containing over 9 million lines and the VulcaN review files contain over 63 thousand lines of JSON.

3.3 Dataset of Vulnerabilities (RQ1)

In this section, we address RQ1, explaining how we created our curated dataset of *npm* package vulnerabilities.

3.3.1 Selection and Validation of Reports

To create our dataset, we collected a snapshot of the existing *npm* advisories until the end of June 2021. Then, through manual analysis, we excluded some advisories and fixed inconsistencies in the remaining ones (see Table 3.1). Out of the 1828 advisories from the original snapshot, we excluded 469, keeping 1359 for further analysis. Next, we present our exclusion criteria and discuss the detected inconsistencies.

Excluded advisories. As of June 30$^{\text{th}}$ 2021, there were 1828 advisories published in *npm*. Of these 1828 advisories, 416 are categorized by *npm* as Embedded Malicious Code (CWE-506): these are packages designed with malicious intent, named very similarly to real legitimate packages so as to deceive developers into installing them. These packages are not relevant to our study, which focuses only on unintentional vulnerabilities. From the remaining 1412 advisories, we excluded 31 for lacking available code. Lastly, out of the resulting 1381 vulnerable package versions, 22 were excluded for not including JavaScript code; instead, they had pre-transpiled variants such as CoffeeScript [13] and TypeScript [106], which prevented us from analyzing their source code directly. Consequently, in the end, VulcaN successfully collected 1359 advisories and their corresponding package versions for further manual analysis.

Detected inconsistencies. During the manual analysis, we noticed several inconsistencies in the collected advisories. Most notably, only a small minority of advisories come with the exact code

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

Table 3.1: Number of advisories excluded or inconsistent.

locations that trigger their corresponding vulnerability. Furthermore, some advisories provide an *incorrect vulnerable package version*, i.e., the advisory metadata points to a package version that does not contain the described vulnerability. When the advisory does not come with additional external references, which is the case for 21% of the analyzed advisories, correcting the incorrect vulnerable package version anomaly can be quite challenging, as the advisory metadata alone is generally insufficient for pinpointing the correct package version. Another detected anomaly is the *imprecise classification of vulnerability type/category*. Most of the time this imprecision is subjective, as a Common Weakness Enumeration (CWE) [107] class can be a subcategory (child) of another more general CWE. This is particularly common for Path Traversals (CWE-22) and Code Injections (CWE-94), to which more precise classes can be attributed; in particular, CWE-23 (Relative Path Traversal) and CWE-24 (another specific Path Traversal variant) to CWE-22 and CWE-95 (Eval Injection) to CWE-94. Some vulnerabilities are simply miscategorised; for instance, sometimes Code Injection (CWE-94) vulnerabilities are categorized as Cross-Site Scripting (CWE-79). During the analysis, we detected 63 cases of vulnerability miscategorization (different CWE) and 21 cases of incorrect vulnerable package versions referenced in the advisory. The remaining cases lack the CWE categorization.

3.3.2 Analysis of Reported Vulnerabilities

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

Figure 3.3 displays the cumulative distribution function (CDF) of the number of vulnerabilities of our dataset ranked by their CWE category. This distribution is heavily skewed toward a relatively small number of CWE categories, i.e., a large fraction of vulnerabilities pertain to a restricted set of categories. In particular, the top-10 CWEs cover 665 advisories, i.e., 69% of the total number of verified vulnerabilities.

To estimate the potential security risks of such vulnerabilities, we mapped each of the top 10 CWE categories to the latest OWASP ranking (from 2021). OWASP is an organization that works to raise awareness about web security and ultimately improve it. The OWASP Top 10 [123] list is a popular document representing a broad consensus about the most critical security risks to web applications. Updated every few years, this document describes existing risks, such as injection

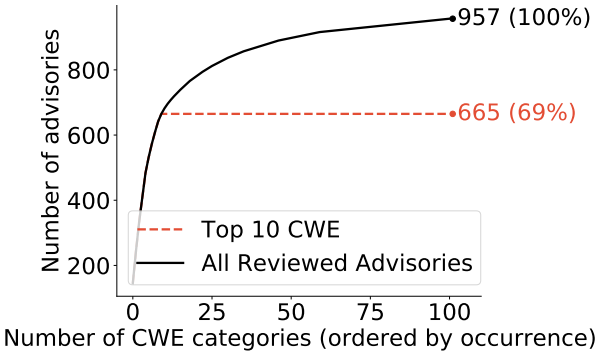

Figure 3.3: CDF of number of reviewed advisories ranked by CWE.

#	CWE	Security Risk	# Occurrences
1	CWE-22	1. Broken Access Control	146
2	CWE-79	3. Injection	99
3	CWE-400	-	89
4	CWE-78	3. Injection	75
5	CWE-818	2. Cryptographic Failures	75
6	CWE-471	3. Injection	48
7	CWE-20	3. Injection	41
8	CWE-1321	3. Injection	36
9	CWE-94	3. Injection	33
10	CWE-77	3. Injection	23

Table 3.2: Possible mapping of most occurring vulnerabilities in dataset with OWASP Top 10 Web Security Risks (2021) [108]: Path Traversal (CWE-22), Cross-site Scripting (CWE-79), Resource Exhaustion (CWE-400), Insufficient Transport Layer Protection (CWE-818), OS Command Injection (CWE-78), Modification of Assumed-Immutable Data (CWE-471), Improper Input Validation (CWE-20), Improperly Controlled Modification of Object Prototype Attributes (CWE-1321), Code Injection (CWE-94), and Improper Neutralization of Special Elements used in a Command (CWE-77).

attacks, broken authentication, and known vulnerable dependencies. As shown in Table 3.2, most vulnerability types can be mapped to a top-10 web security risk. The only exception is annotated with the symbol "–" and refers to CWE-400 (Resource Exhaustion), which represents denial of service vulnerabilities and is not considered a top web security risk by OWASP. This table shows that vulnerable *npm* packages pose well-known risks. Most notably, 9 out of 10 CWE categories (i.e., 576 advisories) appear in the top 3 OWASP list. This translates to approximately 60% of the total number of analyzed vulnerabilities in our dataset (957). In other words, many reported vulnerabilities can introduce serious security flaws in web applications.

3.3.3 Our Curated Dataset

Based on the selected advisories, we created a curated dataset aimed at providing a baseline for assessing the effectiveness of vulnerability detection tools. Table 3.3 tells the number and a few examples of advisories and packages in our dataset grouped into five popularity ranges. It shows that vulnerabilities exist across a wide range of packages, both popular and less popular ones. The dataset itself comprises (i) the code for the vulnerable version of the *npm* package indicated in the advisory, and (ii) a corresponding *review* file. This file contains ground truth

Downloads	# Advs.	Examples	# Pkgs.	Examples
[30M, 84M[	89	813, 1013	67	debug, semver
[3M, 30M[	115	736, 1435	85	validator, tree-kill
[300k, 3M[	106	762, 1216	96	grunt, jspdf
[30k, 300k[	107	335, 1526	90	string, hapi
[0, 30k[	539	249, 1498	506	pivottable, sql

Table 3.3: Characterization of popularity for packages with advisories in the dataset. Popularity extrapolated from total downloads for a 30-day period between 19th December 2022 and 19th January 2023. It includes examples of advisory ids and packages for each bin.


```
{
  "advisory": {
    "id": "GHSA-cwcp-6c48-fm7m", "npm_id": 315,
    "cve": "CVE-2017-16020", "cwe": "CWE-94",
    "github_link": "github.com/advisories/GHSA-cwcp-6c48-fm7m"
  },
  "package_link": "registry.npmjs.org/summit/-/summit-0.1.22.tgz",
  "vulnerability": [
    {
      "source": {
        "file": "lib/drivers/search/pouch.js", "lineno": 4,
        "code": "return function search (opts) {"
      },
      "sink": {
        "file": "lib/drivers/search/pouch.js", "lineno": 20,
        "code": "eval(opts.filter);"
      }
    }
  ]
}
```


Listing 10: Example of VulcaN review file for advisory 315.

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

Review example: Listing 10 shows the created review file for advisory 315 [5]. This advisory reports the presence of a code injection vulnerability in package *summit-0.1.22*. A review is a JSON object that contains several fields that describe: i) the advisory identifier (`id` from Github - *GHSA* - and *npm*), ii) the CVE, when it exists, iii) the vulnerability type as per the CWE taxonomy (`cwe`), iv) the link to the Github advisory page (`github_link`), v) the affected package version (`package_link`), and vi) the vulnerability location expressed as a `source`/`sink` pair. A source/sink is specified as a JSON object with fields denoting: the file name (`file`); the line number (`lineno`); and the corresponding line of code (`code`).

Vulnerability location: The location fields are used to determine if the output of a given vulnerability detection tool is correct. Besides using (one or multiple) `source`/`sink` pairs to locate a vulnerability (e.g.: Listing 10), we can also employ (one or multiple) `block` patterns, which indicate contiguous code regions in which the flaw exists; we give such an example for

Technique	Included Tools (& Version)	Only Package Source Code (C0)	Not Available or Proprietary (C1)	Not Scriptable Interface (C2)	Not Security Oriented (C3)	Other Exclusionary Reasons
Graph-based Analysis (GBA)	CodeQL (2.2.6) [67] ✸ ODGen (-) [97] ✸	SyNode [147] NodeSec [72] Affogato [66]	Beyond Security [22] Checkmarx [28] Fortify [64] Veracode [160] Kiuwan [89] CodeSonar [33] Thunderscan [156] WhiteHat [168]	JsPrime [126] ✸ Codeburner [32] ✸ SonarQube [142] ✸ CodeWarrior [34] ✸	WALA [161] PMD [129] Aether [7] Coala [31] JsHint [85] ✸ EsComplex [52] Coverty Scan [36] DeepScan [45] AppInspector [11] ✸ TAJS [82] SAFE [92]	ESLint SP [55] ✸ Mozilla ScanJs [109] ✸ SemGrep [136] JAW [86] ✸ Joern [170, 83] ✸
Syntax-based Analysis (SBA)	NodeJsScan (0.2.8) [2] ✸ ESLint SSC (**) [56] ✸					
Keyword-based Analysis (KBA)	Graudit (2.8) [98] ✸ InsiderSec (2.0.5) [80] ✸ MS DevSkim (0.4.109) [104] ✸ Mosca (0.8) [35] ✸ Drek (1.0.3) [90] ✸					

Table 3.4: Tools included/excluded. Excluded for reasons beyond C0, C1, C2, & C3: *ESLint Security Plugin* and *Mozilla ScanJs* use *ESLint* rules subsumed by *ESLint SSC*'s; *SemGrep* requires special rules for security purposes and is the backbone of NodeJsScan; JAW only implements rules for detecting client-side CSRF; Joern supports JavaScript inspection, but does not support default rules for detection. Some tools excluded due to C1 were tested using free trials but failed to comply with additional criteria. **ESLint SSC was used with eslint@7.32.0.

advisory GHSA-779f-wgxg-qr8f in the supplementary material[2]. Block patterns are needed when a vulnerability cannot be expressed with source/sink pairs, e.g., usage of HTTP instead of HTTPS which allows for MITM attacks (CWE-818). Interestingly, we found that, in most cases (78%), the exact location of a vulnerability is very clear, e.g., a call to `eval` in a code injection. These cases can be represented by source/sink pairs, while the remaining cases (22%) can be represented using code blocks that cover all vulnerability-relevant code. Although the block size depends on the vulnerability, in our dataset the average block size is six lines of code. Moreover, since the review files can be processed automatically, we believe that our curated dataset will be useful for benchmarking purposes beyond the scope of our work.

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

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

3.4 Vulnerability Detection Tools (RQ2)

We now focus on RQ2, explaining how we selected the tools considered in our study. We specify our eligibility criteria, survey the existing tools that satisfy them, and classify these tools according to the detection technique that they employ.

[2] `https://github.com/VulcaN-Study/Supplementary-Material/blob/master/dataset/reviews/GHSA-779f-wgxg-qr8f.json`

3.4.1 Tool Selection Criteria and Selection Process

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

C0. Depends only on the package source code: The tool requires only the source code of the package to analyze. This excludes tools that require a test suite to guide the analysis.

C1. Be available and transparent: The tool is publicly available and implements a technique that is non-proprietary. Its source code does not need to be open as long as the tool's code analysis techniques can be clearly characterized, e.g., through available documentation, rule sets, and usage examples.

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

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

Based on the above criteria, we ended up selecting nine tools for our testing purposes. We started by examining the academic literature [72, 147, 66, 86, 170, 97] and searching the Internet, including OWASP lists [122, 121], repository collections [15, 16, 17] and other websites [157, 158, 159], for suitable tools for vulnerability detection in *npm* package code. Most tools we screened were developed by the industry and the open-source community. In total, we first collected 40 JavaScript analysis tools. This full list is presented in Table 3.4.

After inspecting all 40 analysis tools, we found that we needed to manually test 19 tools, which are annotated with the symbol ✷ in Table 3.4. These 19 tools were tested against the Damn Vulnerable Node Application (DVNA) [49], a web application written in JavaScript that was purposely built with a range of vulnerabilities matching the OWASP Top 10 Web Security Risks. After executing each of the 19 tools against the vulnerable application, we excluded those that do not allow the analysis to be automated via a script and also those that fail to show security-oriented results.

Out of the remaining 19 tools, we selected 14 tools that are available, transparent, can be automated, and show security-oriented results. Out of these 14 tools, we also excluded *ESLint Security Plugin*, *Mozilla ScanJs*, *SemGrep*, *JAW*, and *Joern* as explained in the caption of Table 3.4. Consequently, we ended up with 9 distinct candidates that represent proper fully automatic vulnerability detection tools for Node.js applications.

3.4.2 Detection Techniques

We characterized the nine selected tools as per the employed vulnerability detection technique. To this end, we tested each tool against the DVNA application, as explained above, read the available documentation, and manually analyzed the source code (when available). We included tools employing similar techniques in the same category; analyzing the differences between techniques helps us to understand the behaviour of each tool. We grouped these techniques into three classes: *graph-based analysis*, *syntax-based analysis* and *keyword-based analysis*.

```
1  function search (opts) {
2    if (!opts.filter && opts.collection) {
3      if (typeof opts.collection === 'string') {
4        opts.filter = "function filter (doc) { return doc.type === '" + opts.collection + "'}";
5      } else { ... }
6      eval(opts.filter);
7      opts.filter = filter;
8    }
9  }
```

Listing 11: Code injection vulnerability (*npm* advisory 315).

Graph-based analysis tools work by first constructing a graph-based model of the program to be analyzed. They usually coalesce various types of statically computed program artefacts, such as the abstract syntax tree, control-flow graph, and dependency graph, into a single graph-like data structure. The resulting graph can be inspected using queries written in domain-specific languages (DSL) that specify vulnerable code patterns. For instance, typical queries aim to identify code-flow paths between user-controllable inputs and dangerous sinks. CodeQL is one such tool that models source code as database records. These records can be queried using SQL-like statements that are specified in the form of rules/queries.

Syntax-based analysis tools employ a technique that searches the code to be analyzed for insecure syntax-aware patterns. Patterns can express simple control-flow conditions, e.g., calling a function with a particular variable. In contrast to graph-based analysis, this technique operates directly on source code and does not typically cater for more intricate dependency analysis or for matching patterns across multiple files. NodeJsScan is an example of such a tool. It is used in GitLab's CI/CD [71].

Keyword-based analysis tools employ a code analysis technique that searches the code to be analyzed for strings associated with potentially insecure code. This search is typically performed through the use of regular expressions. Note that keyword-based analysis does not model the AST of the program to analyze. Consequently, it is considerably less expressive than graph- and syntax-based analysis, as it cannot reason about fine-grained control-flow interactions, often operating on a single line of code at a time.

3.4.3 How Different Detection Techniques Work

To better understand how the vulnerability detection techniques work, we examine, as an example, the advisory 315 of our dataset. As per the advisory page [5], this example consists of a code injection vulnerability (CWE-94), which allows a malicious user to run arbitrary code on the targeted execution platform. In this type of attack, the adversary is only limited by the expressiveness of the injected language. JavaScript code injections at the server can have a more significant impact than those at the client side, given that Node.js has fewer security barriers (e.g., no sandbox), and a larger and privileged API facilitating access to critical system resources (e.g., file system).

The vulnerable code snippet of advisory 315 is given in Listing 11. In this example, the variable *opts* is bound to a user-controlled object with the properties *filter* and *collection*, which can be trivially tainted with a maliciously crafted input to produce valid JavaScript code that

```
opts.collection = `'};
const exec = require("child_process").exec;
exec("cat /etc/passwd", (err, stdout, stderr) => {
console.log(stdout); }); var a={ hello: 'world`;
search(opts)
```

Listing 12: Exploit for code injection (*npm* advisory 315).

```
1  class EvalTaint extends TaintTracking::Configuration {
2    EvalTaint() { this = "EvalTaint" }
3    override predicate isSource(Node node) {
4      node instanceof RemoteFlowSource
5    }
6    override predicate isSink(Node node) {
7      node = globalVarRef("eval").getACall().getArgument(0)
8    }
9  }
10
11 from EvalTaint cfg, Node source, Node sink
12 where cfg.hasFlow(source, sink)
13 select sink, "Eval with user input from \$@.", source
```

Listing 13: CodeQL rule for *eval* taint-tracking [68].

reaches the *eval* function. One can leverage this vulnerability to execute arbitrary JavaScript code, including OS-level commands by using a payload like the one given in Listing 12.

Using graph-based analysis: Both CodeQL and ODGen adopt graph-based analysis. Listing 13 shows an example of a CodeQL rule designed to detect calls to the *eval* function using user-controlled inputs. In order to create this CodeQL rule, one starts by specifying the appropriate configuration, that is, a code description of the targeted sources and sinks. In this case, we are interested in code flows from *remote flow sources*, described by the predicate *isSource* (lines 3 to 5), to the *eval* sink, described by the predicate *isSink* (lines 6 to 8). Then the main query (lines 11 to 13) states that, using the specified configuration (*EvalTaint cfg*), CodeQL should find code paths from the specified source to the specified sink. The output of this query is a string with a description of the source-sink pairs that match the query. This particular rule is a simplified version of one of the CodeQL rules [69] executed inside VulcaN.

In general, graph-based analysis works well for taint-tracking, but it requires every source and sink to be explicitly encoded into rules. These sources and sinks change over time as languages evolve and new popular third-party packages are created. This is why the community has started to work on automatically generating such taint-tracking specifications [149].

Using syntax-based analysis: NodeJsScan helps us showcase syntax-based analysis. Listing 14 lists an excerpt of the NodeJsScan rule that detects potentially vulnerable uses of *eval*. To be applied, this rule must match two related patterns. First, *eval* must occur inside a function receiving two or more arguments (line 2). Then, *eval* must be called with a parameter computed using one of the given arguments (line 4). NodeJsScan includes analogous rules for other vulnerability types [4].

```
patterns:
  - pattern-inside: function $FUNC($REQ, $RES, ...) {...}
  - pattern-either:
      - pattern: eval(..., <... $REQ.$QUERY ...>, ...)
```

Listing 14: NodeJsScan rule for *eval* detection [3].

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

Listing 15: Snippet of Mosca output classified with *Score A*.

Similarly to the previous technique (graph-based), syntax-based analysis also suffers from the source-sink specification limitation. Additionally, there are some other limitations specific to syntax-based analysis. For example, it may lead to a high number of false positives, as it is not expressive enough to capture the dependencies of the variables occurring in the patterns; e.g., it will detect all calls to *eval* regardless of whether or not their given input can be controlled by the user. Furthermore, it commonly leads to *rule overfitting*, resulting in over-specific rules that match known examples of vulnerabilities, but are not general enough to capture other instances of the same vulnerability. In Listing 14, the *eval* call is only detected when it occurs inside the body of a specific type of function declaration. Besides ignoring *eval* calls at the top level, this pattern also ignores calls to *eval* which occur inside the body of JavaScript functions declared using alternative syntactic constructs (e.g. function constructor and lambdas).

Using keyword-based analysis: The tool we use to illustrate keyword-based analysis is Graudit. The following is an excerpt of a Graudit rule that detects the use of *eval*:

```
eval[[:space:]]*\(
```

Here, we see a regular expression pattern that simply detects any call to the *eval* function. This technique suffers from several limitations such as the source-sink specification problem and a high number of false positives. Graudit includes many other rules for dangerous sinks in Node.js applications [99].

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

Listing 16: Snippet of Graudit output classified with *Score B*.

3.5 Effectiveness of the Tools (RQ3)

In this section, we focus on our third research question (RQ3). To perform a quantitative and qualitative assessment of the selected tools, we begin by specifying the evaluation metrics and methodology we used to rank the tools. Then we present our findings, relying on the result of running the selected tools across all 957 advisories of our curated dataset.

3.5.1 Evaluation Methodology

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

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

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

We split the TP results according to two distinct classes: A is an *explicit vulnerability notification*, and B is a *security warning notification*. The A-ranked tools provide a richer output to users and, thus, more information about the detected vulnerability. E.g., consider the output of Mosca and Graudit with regards to advisory 315 shown in Listings 15 and 16, respectively. Although both outputs flag the vulnerable *eval* call reported by the review file of Listing 10, Mosca's output clearly identifies a possible code injection, and provides a description, a severity level, and the line of code containing the vulnerability. On the other hand, Graudit only shows

Tech.	Tool	Min	Max	Mean	St. Dev.	Q-90	Total
GBA	ODGen	1.236	3653.043	148.757	385.629	370.969	110823.8
	CodeQL	1.712	736.546	119.570	98.755	177.550	28696.8
SBA	NodeJsScan	20.023	984.453	99.562	121.246	230.216	23795.4
	ESLint SSC	0.592	3556.665	29.871	237.799	25.465	7139.1
KBA	Graudit	0.042	14.632	0.550	1.624	0.704	131.3
	InsiderSec	0.000	243.000	5.749	25.186	7.000	1374.0
	MS DevSkim	0.276	186.338	6.393	23.262	8.044	1527.9
	Drek	0.300	6.649	1.022	0.865	1.949	244.3
	Mosca	0.005	245.498	7.408	25.166	12.216	1770.5

Table 3.5: Summary statistics of the analysis times (in seconds) taken by the tested tools across all 957 reviewed advisories.

the vulnerable line of code without explaining how or why it flags that particular snippet. For this reason, the output of Mosca is classified with *Score A* while the output of Graudit is classified with *Score B*.

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

3.5.2 Analysis Performance

We gauge analysis performance by measuring tools' execution time for all 957 advisories on a machine with an Intel Xeon E3-1220 v3 @ 3.10GHz processor and 32GB of memory.

Table 3.5 analyzes the execution times of each tool to scan our dataset. When compared to all other tools, ODGen, CodeQL, NodeJsScan and ESLint SSC require considerably more time. To analyze all 957 packages, ODGen took over 30 hours (110k seconds), CodeQL took nearly 8 hours (27k seconds), NodeJsScan took nearly 7 hours (24k seconds), while ESLint SSC took nearly 2 hours (7k seconds). All other tools are more efficient, taking at most 30 minutes to analyze all packages.

The mean execution time of ODGen, CodeQL and NodeJsScan is 148.8, 119.6 and 99.6 seconds, respectively. ODGen, CodeQL and NodeJsScan tools are slower because their detection techniques involve modelling statically computed structures. These operations are more complex than performing keyword-based matching searches (see Section 3.4.2). Depending on the size of the package and on the CI/CD pipeline restrictions, these tools may end up being exceedingly slow.

3.5.3 Results Across the Entire Dataset

Figure 3.4 displays the score distribution for each tool across our entire dataset and Table 3.6 shows the evaluation metrics for each tool. Globally, the tested tools perform rather poorly. We can draw the following main observations:

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

Table 3.6: TP, TPR (%), FP and Precision (P%) for each tool by CWE. TPR highlights: green (TPR $\geq$ 50%) or yellow (50% $>$ TPR $\geq$ 15%). When TPR is highlighted we also highlight the FP and P columns: yellow (50% $>$ P $\geq$ 15%), light red (15% $>$ P $\geq$ 2.5%) and dark red (P $<$ 2.5%). The CWEs are: Path Traversal (CWE-22), Cross-site Scripting (CWE-79), Resource Exhaustion (CWE-400), Insufficient Transport Layer Protection (CWE-818), OS Command Injection (CWE-78), Modification of Assumed-Immutable Data (CWE-471), Improper Input Validation (CWE-20), Code Injection (CWE-94), Improper Neutralization of Special Elements used in a Command (CWE-77), and Improperly Controlled Modification of Object Prototype Attributes (CWE-1321).

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

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

3. Graph-based analysis has the best detection capability: Figure 3.5 shows the scores according to a particular detection technique. These results show that graph-based analysis reports a significantly larger number of results with score A (explicit vulnerability notifications). Syntax and keyword-based analysis look fairly similar, with reasonable detection rates, but also a high number of reports containing only false positive results.

When considering the results of both tools in this category, i.e., ODGen and CodeQL, they strike a better balance between true positives and precision. CodeQL detects 300 vulnerabilities (31.3%) and has a significantly higher precision (7.8%), when compared to most other tools, while ODGen detects 154 vulnerabilities (16.1%). This number is significantly lower than CodeQL's,

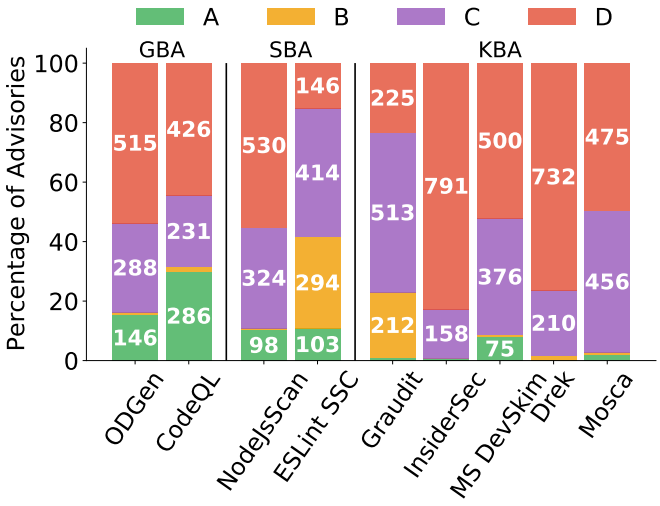

Figure 3.4: Score distribution for each tool.

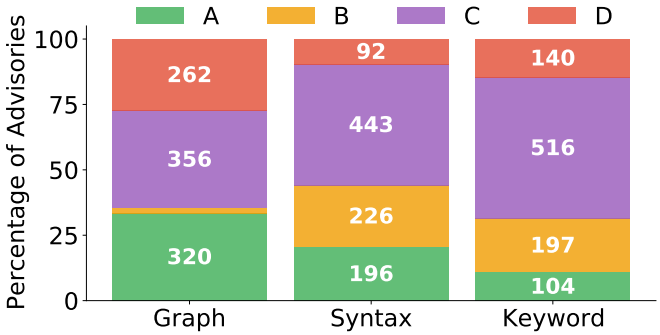

Figure 3.5: Score distribution for each detection technique.

but it represents a much higher precision (23.8%) than any other tool tested. Although both of these tools do not have the highest TPR, most of their detected vulnerabilities were classified with the A score, meaning that the reported information is richer and more meaningful to the user. Consequently, CodeQL and ODGen are the most balanced tools, achieving a reasonable detection rate (TPR) and fewer FPs, when compared to other tools with similar TPR. We also note that both these tools have the potential to be further improved by extending them with additional rules.

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

3.5.4 Results Across Specific Vulnerability Types

We now assess the performance of the tools when focusing on particular types of vulnerabilities. We concentrate on two main aspects: i) studying the types of vulnerabilities that the tools detect more frequently, and ii) analyzing which types of vulnerabilities can be detected simultaneously by several tools.

1. Most frequently detected vulnerability types: From the analysis of Table 3.6, we highlight seven CWEs that are detected most often regardless of the used tool: CWE-22, CWE-471, CWE-78, CWE-79, CWE-94, CWE-77, and CWE-1321. These are coloured in yellow and green in Table 3.6.

CWE-22 (path traversal) is the only type clearly detected by all four best-performing tools (ODGen, ESLint SSC, Graudit, and CodeQL). This is because path traversal can be found statically by searching for well-known dangerous sinks in the Node.js API, e.g., the functions *readFile*, *writeFile* and *createReadStream*. The difference in precision between these four tools lies in that ESLint SSC and Graudit simply match these function calls, while ODGen and CodeQL report only cases where the path is tainted by user input.

CWE-78 and CWE-77 (OS command injection), CWE-79 (cross-site scripting) and CWE-94 (code injection) correspond to classic injection vulnerabilities. Detecting these vulnerabilities depends on the sets of sinks considered by each tool. CodeQL detects more OS command injections, while ESLint SSC detects more code injections because each has more extensive rulesets for those particular vulnerabilities. Both tools detect about the same number of XSS vulnerabilities.

Both CWE-471 (Modification of Assumed-Immutable Data) and CWE-1321 (Improperly Controlled Modification of Object Prototype Attributes) are umbrella CWEs for several prototype tampering and prototype pollution vulnerabilities, for which both CodeQL and ESLint SSC have various rules. We expected ODGen to perform better at detecting prototype pollution vulnerabilities (CWE-471), as this is one of its central goals [97]. Although ODGen's results for this vulnerability (22.9%) fall short of those by CodeQL (27.1%) and ESLint SSC (79.2%), ODGen does achieve a much higher precision (36.7%) than CodeQL (19.4%) and, especially, ESLint SSC (0.5%).

2. Vulnerability types detected by the three best-performing tools: Figure 3.6 shows the intersections of TPs for the top-10 CWEs. We can see a substantial intersection for CWE-22, where all three tools detect the same 85 vulnerabilities. This happens because path traversals are easy to find statically using a limited set of known dangerous sinks from the Node.js API, which all tools share. For CWE-79, both CodeQL and ESLint SSC detect about the same number of vulnerabilities, but only about half intersect with each other. This is due to differences in the rules regarding XSS sources and sinks. CWE-471 shows a significant intersection, but ESLint SSC detects several vulnerabilities that CodeQL misses. This is because ESLint SSC's rules have a wider range of sinks. Other CWEs have fewer intersections because their rulesets differ. For example, CodeQL is the only tool with specific rules to detect resource downloads over HTTP, hence the results for CWE-818.

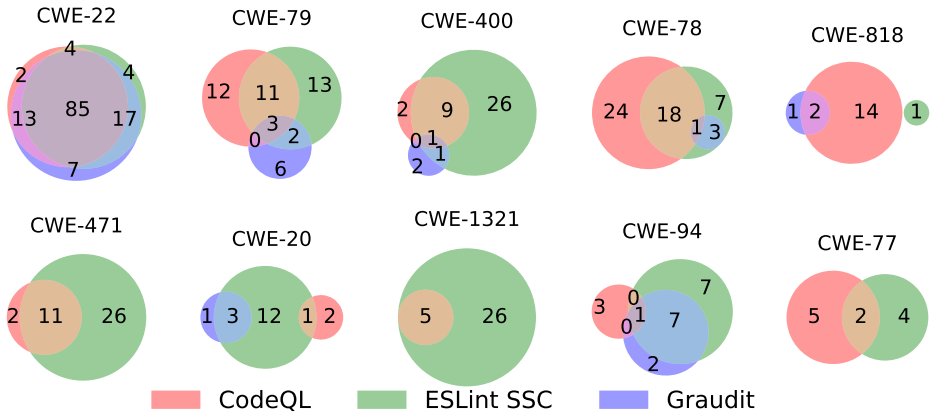

Figure 3.6: Intersections of TPs of the 3 best tools for top-10 CWEs.

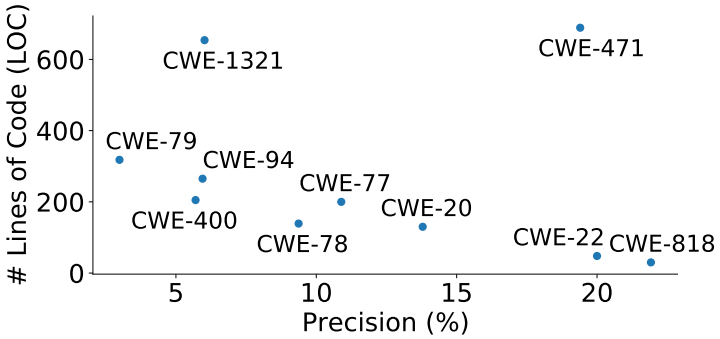

Figure 3.7: Correlation between Query LOC and Precision.

3.5.5 Results as a Function of Queries and Ruleset

To study the relationship between ruleset/queries and vulnerability detection results, we take CodeQL as an example and show, in Figure 3.7, how the precision of this tool compares with the number of lines of code of the specific queries CodeQL uses to detect the vulnerabilities in the Top-10 CWE categories in the dataset. Although this cannot be taken as a general rule, we can see that, in most cases, the precision is higher for smaller queries. Taking into account each CWE, simpler vulnerabilities, like CWE-22, can be detected using smaller queries with higher precision, while more complex vulnerabilities to detect, like CWE-1312, are harder to detect even when using larger (more complex) queries.

It is then clear that vulnerability detection results can be influenced by the ruleset/queries executed by the tool. Using a small (specific) ruleset may improve the precision in detecting a specific vulnerability. However, in the context of a CI/CD pipeline, application developers do not know beforehand which specific ruleset to select to detect the (unknown) vulnerability. Consequently, it is reasonable to apply the most comprehensive ruleset available for the tool. This is the approach we used in this paper. For every tool, we selected the most comprehensive and complete ruleset, by either combining rules into a single tool execution or executing the tool multiple times using a different rule for every execution and combining the results. We also used the rules available off-the-shelf instead of developing or customizing rules. This allows us

CWE	CWE Description	OWASP	Undetected	%
CWE-79	Cross-site Scripting	✓	50 / 99	50.5%
CWE-400	Resource Exhaustion	-	44 / 89	49.4%
CWE-78	OS Command Injection	✓	20 / 75	26.7%
CWE-20	Improper Input Validation	✓	16 / 41	39.0%
CWE-22	Path Traversal	✓	12 / 146	8.2%
CWE-94	Code Injection	✓	10 / 33	30.3%
CWE-818	Insecure Transport Layer	✓	10 / 75	13.3%
CWE-287	Improper Authentication	✓	9 / 9	100.0%
CWE-471	MAID	✓	8 / 48	16.7%
CWE-200	Information Exposure	✓	8 / 14	57.1%
Others	-	-	137 / 286	47.9%
Total	-	-	324 / 957	33.9%

Table 3.7: Number of vulnerabilities undetected by any tool.

Lx	Oc (%)	Adv. Example	Improvement
L1	36%	63 (CVE-2015-9241) 567 (CVE-2017-11429)	Cross-package CPG Auto. Taint-Specifications
L2	6%	165 (CVE-2016-10583) 305 (CVE-2016-1000249)	Resource-aware CPG
L3	18%	26 (CVE-2014-10067) 92 (CVE-2016-10533)	Test Annotations ML-augmented Analysis
L4	21%	113 (CVE-2016-10554) 43 (CVE-2014-9772)	Smart Fuzzing Tests Symbolic Execution
L5	9%	1469 (CVE-2017-1000048) 313 (CVE-2017-5954)	Hybrid Analysis

Table 3.8: Occurrence (Oc) of tool limitations (Lx) that caused undetected vulnerabilities and suggested improvement. The remaining 10% of vulnerabilities do not match any of the described limitations.

to reflect on how developers will use these tools as most of them are not technically versed in improving the ruleset specified by the tool developers.

3.6 Reasons for Missed Detection (RQ4)

In RQ4, we study why existing tools fail to detect certain vulnerabilities. Table 3.7 shows the number of undetected vulnerabilities grouped by CWE and their mapping to OWASP Top 10 Web Security Risks (2021) [108]. Of the 957 known vulnerabilities in the dataset, 324 vulnerabilities (33.9%) were not detected by any of the selected tools. To understand the underlying reasons, three authors have manually analyzed the vulnerable code and respective tools' output of 33 vulnerabilities (i.e., a 10% sample) picked randomly from the 324 undetected vulnerabilities. Using this methodology, we identified five main tool limitations whose frequency and proposed improvements we summarize in Table 3.8 and present next. Our supplementary material contains specific examples of undetected vulnerabilities indicated in this table.

L1. Cross-package vulnerabilities and incomplete rule set: Some undetected vulnerabilities exist in function calls to third-party packages whose functions execute known dangerous code, e.g., wrappers to OS-level commands. For instance, Listing 17 shows a code snippet of a command injection vulnerability for advisory 1440. In this case, user-controlled data reaches an *exec* sink

```
// Snippet of ./gnuplot.js:
var run = require('comandante');
module.exports = function () {
    var plot = run('gnuplot', []);
    plot.print = function (data, options) {
        plot.write(data); // (...)
    }; // (...)
}
```

Listing 17: Command Injection (advisory 1440).

```
{
    "scripts": {
        "preinstall":
        """wget http://s.qdcdn.com/17mon/17monipdb.zip &&
        unzip -p 17monipdb.zip 17monipdb.dat > 17monipdb.dat"""
    }
}
```

Listing 18: Insecure Transport Layer in *package.json* of *ipip-coffee* package (advisory 279) - CVE-2016-10673.

inside the third-party package *comandante*, a package meant to ease the execution of OS-level commands. The selected tools come with pre-defined sets of manually written rules, typically focusing solely on popular APIs. As a result, the tools cannot recognize this vulnerability because the command injection sinks are located inside a third-party dependency unbeknownst to the pre-defined rule sets, i.e., the rules miss *comandante*'s *write* function as a dangerous sink.

This limitation, however, is not fundamental to static analysis per se. In fact, these vulnerabilities could have been found by testing all package dependencies (can be thousands of other packages [173]) or using a more complete set of rules and queries (covering additional sources and sinks). However, the manual maintenance of such lists of sources and sinks is impractical as the Node.js ecosystem expands. To overcome this problem, we propose enhancing static analysis with two complementary techniques: *cross-package Code Property Graph (CPG)*, and *automated taint-specification*. The first consists of extending the CPG of the analyzed code base with the CPG of its dependencies. Extended CPGs would allow for an exhaustive exploration of all potentially dangerous sinks. The second is to automatically extract taint specifications (sources and sinks) from JavaScript libraries and avoid any blind spots due to incomplete rule sets. In this direction, Staicu et al. [149] propose an interesting first approach, although at the cost of an expensive, constant dynamic testing of every new *npm* package. Improving on their work is an interesting research direction for the future.

L2. Limited analysis scope: In addition to JavaScript code files, Node.js projects depend on several other artefacts, such as configuration files, front-end template code, testing frameworks, etc. However, by analyzing only the JavaScript code in isolation, certain vulnerabilities can be missed. As an example, *npm* packages contain a *package.json* file which may include bootstrap scripts. In several analyzed packages, these scripts are used to download resources over HTTP. As it turns out, using HTTP allows for man-in-the-middle attacks, where resources are replaced by malicious payloads. Listing 18 shows a snippet of the *package.json* file for the *ipip-coffee* package,

```
// Snippet of ./lib/odbc.js:
if(exports.debug) {
    console.log("""%s odbc.js : pool[%s] :
        pool.close() - processing pools %s - connections: %s""",
        getElapsedTime(), self.index, key, connections.length);
}
```

Listing 19: Credential Exposure (advisory 1185) - SNYK-JS-IBMDB-459762 [141].

in which an external resource is downloaded over HTTP. This allows for man-in-the-middle attacks that might compromise the server.

While some tools can detect insecure downloads or requests performed from the JavaScript code (e.g., by searching for HTTP URLs), they cannot detect downloads issued from *package.json*. Nevertheless, as in L1, static analysis can also be augmented to detect this kind of vulnerability. For instance, in this particular example, the vulnerability can be detected if the *package.json* file is also considered a potential taint analysis source. To this end, we propose to explore a new potential technique involving the construction of a *resource-aware CPG*. The idea is to extend further the CPG of the program under analysis with nodes and edges modelling sources, sinks, control flows, data flows, and dependency relations that involve resources specified externally to the code, such as in configuration files, front-end templates, testing frameworks, and other auxiliary artefacts. Given this diversity of artefacts, it is necessary to study their semantics and reflect the essential properties of each resource into a resource-aware CPG so as to allow the static analysis to reason about data flows that depend on sensitive external sources, e.g., network or file system.

L3. Lack of contextual knowledge: Packages may expose sensitive information, e.g., by logging plaintext passwords to a file. Listing 19 shows an example of a Credential Exposure vulnerability, in which plaintext passwords are logged to the console. The code snippet itself seems benign until one realizes that the *key* variable holds security-critical information. These vulnerabilities are application-specific and require contextual knowledge of which data is sensitive. Unfortunately, the analyzed tools are not designed to obtain this information and thus miss vulnerabilities that depend upon it, e.g., application-specific leaks. Indeed, extracting and reasoning about contextual knowledge of a program is quite challenging using automated static analysis and a relatively uncharted research space.

As a potential exploratory path to help detect vulnerabilities that depend on contextual knowledge, we propose two possible ideas: *test annotations* and *ML-augmented analysis*. The former is to let the developer/tester annotate application inputs, objects, or data flows with sensitivity levels and check which system resources handle the annotated features during the execution. The latter is to use machine learning (ML) to extract contextual information from the application and use it to enrich the static analysis. While letting a human specify the sensitive data types manually may help reduce mislabeling errors due to a better understanding of a program's semantics, ML-augmented analysis has the benefit of automation and lower user effort.

L4. Inability to reason about sanitization: Application developers often use regular expressions to detect malicious inputs. However, regular expressions are complex, and developers usually do not test them thoroughly, allowing sanitization bypasses to occur. For instance, Listing 20 contains two regular expressions aimed at preventing XSS vulnerabilities. However,

```
1  // Snippet of ./protect/lib/rules/xss.js
2  const xssSimple = new RegExp('((%3C)|<)((%2F)|/)*[a-z0-9%]+((%3E)|>)', 'i')
3  const xssImgSrc = new
   ↪  RegExp('((%3C)|<)((%69)|i|(%49))((%6D)|m|(%4D))((%67)|g|(%47))[^\n]+((%3E)|>)','i')
4
5  function isXss(value) {
6      return xssSimple.test(value) || xssImgSrc.test(value)
7  }
8  // Example payload: <input type="image" onerror="alert('XSS')">
```

Listing 20: XSS (advisory 1116) - CVE-2018-1000160.

these regular expressions are not entirely correct as there still exist specially crafted inputs, such as the one shown in the comment of Listing 20, that can bypass this regular expression validation and launch an XSS attack.

Sanitization errors are often hard to detect statically, as they require dynamically testing each regular expression ensuring that they generate semantically valid inputs that can both bypass the validation and effectively trigger the vulnerability. Consequently, static analysis tools are not ideal for detecting incorrect sanitization vulnerabilities, especially when using regular expressions. Alternatively, we propose two approaches to complement static analysis: *smart fuzzing tests* and *symbolic execution*. The first consists of extending current dynamic fuzzers with techniques for thorough testing of regular expressions. The second consists of employing symbolic execution to test sanitization functions and conditionals.

L5. Inability to cope with JavaScript dynamicity: Specific features of JavaScript can lead to vulnerabilities that are hard to detect by static analysis tools. For instance, object-based inheritance, extensible objects, and dynamic typing are key features of JavaScript that can lead to prototype pollution, authentication bypass, and business logic vulnerabilities. Listing 21 shows a Prototype Pollution vulnerability in the *qs* package, which is a *querystring* parsing library that allows developers to create objects within query strings. E.g., the string `'foo[bar]=baz'` is converted to the object `{foo:{bar:'baz'}}`. Usually, this package protects against attacks that try to overwrite the existing prototype properties of an object. However, in this vulnerable version, the protection can be circumvented by prefixing the name of the parameter with character `[` or `]`, as shown in the proof-of-concept exploit code shown in Listing 21. Consequently, calling `toString()` on the object will throw an exception. This can subvert the application logic, potentially allowing attackers to work around security controls, modify data, and make the application unstable.

The selected tools miss this example because they fail to model how objects change depending on the instructions applied to them, especially the object prototype. Static analysis tools cannot detect with sufficient precision how changes in objects occur when the code is executed. To overcome this limitation in future work, we propose using a *hybrid analysis* combining both static and dynamic analysis. Using this approach tools may start by applying an eager and imprecise static analysis (e.g. using CPG), which may lead to a high false positive rate, and then apply dynamic analysis (e.g. concolic execution) to confirm the vulnerabilities indicated by the static analysis, thus improving overall precision.

In summary, from these limitations, we can extract actionable insights to improve static code analysis tools for vulnerability detection in Node.js code. On one hand, these tools can potentially overcome limitations L1 and L2 by both employing improved strategies for maintaining taint

```
// Snippet of ./lib/parse.js:
module.exports = function (str, opts) {
    var options = opts || {};
    var tempObj = typeof str === 'string' ? parseValues(str, options) : str;
    var obj = options.plainObjects ? Object.create(null) : {};
    var keys = Object.keys(tempObj);
    for (var i = 0; i < keys.length; ++i) {
        var key = keys[i];
        var newObj = parseKeys(key, tempObj[key], options);
        obj = Utils.merge(obj, newObj, options);
    }
    return Utils.compact(obj);
};
// Proof-of-Concept exploit code:
qs.parse("]=toString", { allowPrototypes: false })
// {toString = true} <== prototype overwritten
```

Listing 21: Prototype Override (advisory 1469) - CVE-2017-1000048.

specifications and by considering all the appropriate analysis scopes for Node.js code. On the other hand, every static analysis tool will struggle to overcome limitations L3, L4, and L5, because they fail to capture behavioural and contextual information that is only available at runtime when the package is executed with appropriate, and application-specific, test inputs. To this end, it seems that the approaches employed by current static vulnerability detection tools can mainly be used successfully to detect classic injection-style vulnerabilities even if all the tools tested in this paper cannot do so with reasonable precision.

3.7 Threats to Validity

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

We may have missed some relevant tools, failed to evaluate an analyzer that excels above all tested tools in our study or overlooked third-party detection rules that produce better results. Furthermore, later versions of the selected tools may improve since the writing of the paper. To reduce this risk, we will promote the reproducibility of our evaluation by providing both the source code of VulcaN and our curated dataset.

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

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

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

Although our study gives a comprehensive picture of vulnerabilities in *npm* packages, our dataset may not be representative of vulnerabilities in Node.js applications. Node.js applications not only depend on multiple *npm* packages but contain application-specific code stitching together function calls from imported packages. This code is not included in our curated dataset. As a result, we cannot extrapolate that all package vulnerabilities have an impact on a Node.js application. Nevertheless, our study gives important insights to improve the detection of vulnerabilities in *npm* packages, which has been the focus of several research works [1, 173, 147, 72, 146, 44, 118, 97].

3.8 Related Work

The literature covers many tools for detecting vulnerabilities in Web applications, including static [43, 19, 8], dynamic [87, 100], and hybrid analysis tools [20, 60, 127, 9], often combining different types of program analysis techniques, such as fuzzing (e.g. [87, 100]), control-flow and data-flow analysis (e.g. [19, 20, 127, 170]), and symbolic execution (e.g. [9, 8, 60]). The great majority of these tools are, however, aimed at PHP-based Web applications. Most of the existing tools for JavaScript are aimed at client-side JavaScript code and its specific vulnerabilities: for instance, DOM-based XSS [95, 102], unrestricted inclusion of third-party cross-origin scripts [110], and potentially malicious flows via client-side persistent storage [150].

Graph-based vulnerability scanners: State-of-the-art static vulnerability analysis techniques often work by first computing a static model describing the dynamic behaviour of the application to be analyzed. Most notably, the code property graph (CPG) [170] was proposed as a compact representation of an application's behaviour. With CPGs, one can encode specific vulnerability types as simple graph traversals, which can, in turn, be expressed using graph query languages and then executed on top of off-the-shelf graph databases (e.g. Neo4J [114]). Code property graphs have successfully been applied to find SQL injection, XSS, and CSRF vulnerabilities in PHP applications [19, 127]. Furthermore, they are at the core of CodeQL [67]. For JavaScript, code property graphs were employed by JAW [86] and ODGen [97], for client-side and server-side JavaScript respectively. In our work, we have extensively evaluated CodeQL and ODGen as representative state-of-the-art, graph-based vulnerability scanners.

Vulnerability studies & analyzers for Node.js applications: Unlike client-side JavaScript applications, which run in the browser, Node.js application code is not sandboxed. Recent

empirical studies [1, 173] have shown that contrary to popular belief, *npm* applications are often poorly maintained and tested, with a significant percentage (up to 40%) of all packages depending on code with at least one publicly known vulnerability. Furthermore, after reviewing more than 200K *npm* applications, Staicu et al. [147] conclude that 20% of the analyzed applications either directly or indirectly make use of an injection API. Despite this security-critical situation, there is only a small number of research tools for detecting vulnerabilities in Node.js applications and their underlying infrastructure, most of which are based on dynamic code analysis. For instance, Synode [147] aims to prevent injection attacks in Node.js applications, and NodeSec [72] aims to detect vulnerabilities in Node.js applications. The authors of [146] and [44] design specific dynamic analysis for finding regular expression denial of service (ReDoS) vulnerabilities and NodeXP [118] employs dynamic analysis to automatically detect and exploit server-side JavaScript injection (SSJI) attacks. The authors of [169] also apply dynamic analysis and symbolic execution to detect attacks that leverage hidden properties in client- and server-side JavaScript. There are also academic works that employ static analysis techniques for detecting vulnerabilities in Node.js, but most focus on detecting prototype pollution vulnerabilities [96, 88]. ODGen [97] is the only purely static code analysis tool developed by academia that aims to detect several types of vulnerabilities in Node.js.

Empirical studies of vulnerability analyzers: Several empirical studies aim at characterizing the efficacy of existing white-box vulnerability detection tools (e.g. [48, 102, 119]). Durieux et al. [48] evaluated 9 automated analysis tools for Ethereum Smart Contracts. The authors created a curated dataset consisting of 69 annotated vulnerable smart contracts, as well as a *raw* dataset consisting of 47,518 smart contracts. They report that only 42% of the vulnerabilities on the annotated dataset were detected, with the highest-ranking tool having an accuracy of 21%. Melicher et al. [102] evaluated 3 automated static analysis tools for detecting DOM-based XSS in client-side JavaScript code (Esflow [53], ScanJS [109], and Burp Suite Pro [130]). They created a dataset with 3219 confirmed vulnerabilities. However, many security flaws in server-side code for Node.js do not exist on the client side (e.g., SQL injections), and vice-versa. As such, the dataset from [102] is not representative enough of server-side vulnerabilities. Finally, Nunes et al. [119] evaluate five free static analysis tools for detecting SQL injection and XSS vulnerabilities in PHP web applications. Our paper presents the first empirical study targeting fully automated vulnerability detection tools for *npm* packages. Our study comes with a comprehensive manually-annotated dataset based on confirmed real-world vulnerabilities.

3.9 Conclusions and Future Work

This paper presented an empirical study of static analysis tools for detecting vulnerabilities in Node.js packages. To conduct this study, we built VulcaN, an automated analysis framework, using which we created the largest known curated dataset of Node.js packages with well-characterized security vulnerabilities. Currently, our curated dataset includes 957 reviews that accurately identify the exact location of known vulnerabilities inside affected *npm* packages. We found that the nine evaluated tools fail to detect many vulnerabilities and exhibit high false positive rates. Additionally, we show that many important vulnerabilities appearing in the OWASP Top-10 are not detected by any evaluated tool or even when using the combination of all tools. To enable future research on automatic vulnerability detection tools for Node.js code, this dataset is publicly available. This work motivates future research in hybrid analysis, smart fuzzing tests, and resource-aware CPG. We also intend to work on improving the curated dataset with

new vulnerabilities and offer VulcaN as a service (akin to tool testing competitions such as TestComp[21]) so that users can check the results of new tools across this dataset.

Acknowledgements: We thank the anonymous reviewers for their comments. This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) via the 2021.06134.BD and SFRH/BD/146698/2019 grants, projects UIDB/50021/2020 and DIVINA (ref. CMU/TIC/0053/2021) and the SmartRetail project (ref. C6632206063-00466847) financed by IAPMEI.

4

Our CPG Prototype and RuleKeeper

Our previous empirical study concluded that state-of-the-art CPG-based vulnerability detection tools exhibit high false positive rates (low precision) and low true positive rates (low recall) when searching for vulnerabilities in Node.js code. All these tools aim to help developers detect vulnerabilities during the development process. Yet, because of low precision and low recall, all tools output a large number of false positive results, which have to be manually verified by a human operator. This condition hinders the usability of these tools because developers cannot waste large amounts of resources to verify many false positive results.

Additionally, most of the state-of-the-art implementations of CPGs were based on specifications for imperative, non-object-oriented programming languages [170, 171], which means they focus on modelling flows for variable declarations and assignments, instead of considering property lookup and write statements. This represents a challenge for detecting vulnerabilities in modern web applications because many applications use objects to represent data entities and their corresponding properties. Consequently, there exists a need to improve the recall and precision of static-analysis vulnerability detection tools for Node.js code.

In this chapter, we continue by discussing a custom specification of CPGs for Node.js code and how it can be used to detect injection-style vulnerabilities. We also perform a preliminary evaluation against a subset of our curated dataset described in Chapter 3, as well as against a small dataset of basic object-oriented snippets containing injection vulnerabilities. Finally, we describe how our prototype was applied for detecting flows of privacy-sensitive data to sinks that violate GDPR compliance in a published paper called RuleKeeper.

4.1 Designing Custom Code Property Graphs

To fill in the gap left by the non-existence of effective and precise static-analysis vulnerability detection tools for Node.js code, we decided to adapt Code Property Graphs for detecting injection vulnerabilities. More specifically, we wished to improve the detection (recall) of injection vulnerabilities where data flows from source to sink using JavaScript object properties. To do so, we designed a new code analysis graph specifically tailored for Node.js, using a points-to-analysis approach, and built an early prototype that generates this graph for *npm* packages.

In Figure 4.1, we present the architecture of our Code Property Graph implementation, which we call a Code Representation Graph (CRG). Our implementation consists of two modules:

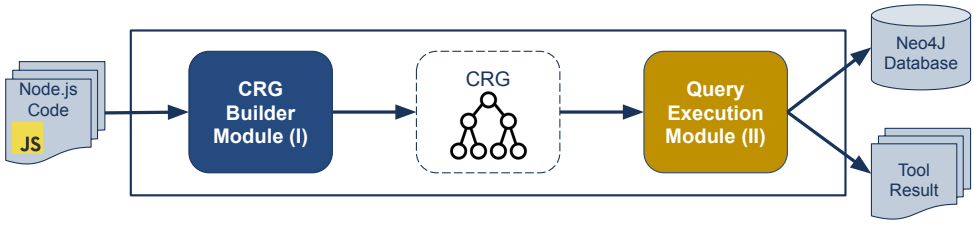

Figure 4.1: Architecture of our CRG prototype.

```
1  module.exports = function f(person) {
2      if (person.age > 0) {
3          person.status = "born";
4          return eval(person.status +
           ↪  person.name);
5      }
6  };
```

Listing 22: Node.js code vulnerable to an injection vulnerability using object properties (based on Example 3 from our Example Dataset.

the CRG Builder module and the Query Execution module. The CRG Builder module receives Node.js files as input and parses the code into our CRG representation. The Query Execution module receives the CRG structure from the CRG Builder module, imports it into a Neo4j [114] database and executes Cypher [42] queries on it to identify attacker-controlled data flows that may result in injection vulnerabilities and reconstruct the structure of the objects and variables under the attacker's control. The prototype then outputs a JSON file, which we describe as a taint summary, listing all the potentially vulnerable data flows detected, including the source (entry point) and the sink (vulnerability triggering function).

4.1.1 The Code Representation Graph (CRG)

Listing 22 shows a basic example of Node.js code vulnerable to an injection vulnerability. Suppose the `person` variable is an attacker-controlled input object. If the `age` property is greater than zero, then the `eval` function depends on the value of both the `status` and `name` properties. The `name` property's value is unknown and might hold a dangerous string which could lead to code execution when this property is evaluated at line 4. This example is based on Example 3 in our Example Dataset used during the preliminary evaluation.

Similarly to the original CPG specification [170], our CRG is generated by merging the abstract syntax tree (AST), control flow graph (CFG) and program dependence graph (PDG). Additionally, to reason about data flows of statements using object properties, we also incorporate an object dependence graph (ODG), which represents the structure and evolution of objects during the execution of the code. Next, we explain our custom object dependence graph, shown in Figure 4.2, which represents the running example illustrated in Listing 22. Our object-dependence graphs contain the following three types of nodes:

Tainted source nodes: Tainted source nodes represent untrusted data entering the application context, e.g., user input or input from other modules. For our implementation, all parameters of

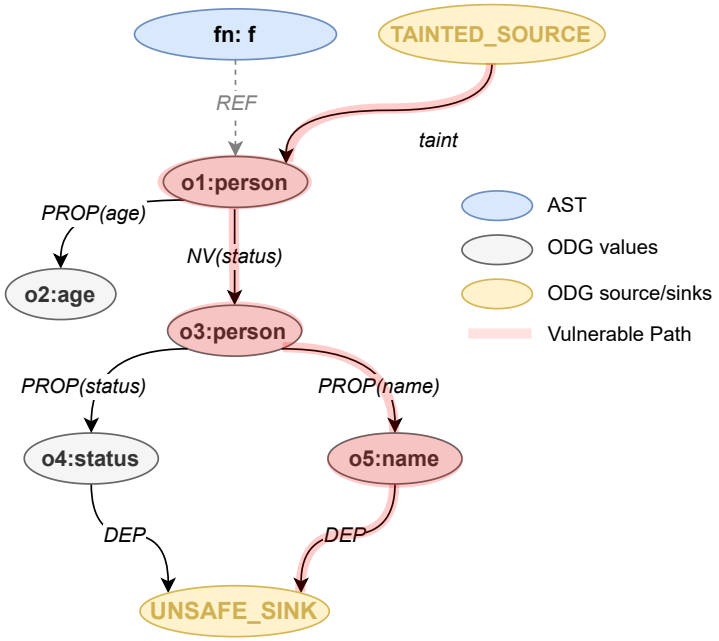

Figure 4.2: Object Dependence Graph for the CRG of the code in Listing 22.

exported functions are tainted, because these parameters carry input from outside the codebase. In Figure 4.2, the parameter `person` of the function `f` (line 1 of Listing 22) is connected by a special taint parameter edge to a global `TAINTED_SOURCE` node used to represent attacker-controlled data.

Unsafe sink nodes: Unsafe sink nodes represent calls to dangerous APIs, such as the `eval` function. In Figure 4.2, the call to the `eval` API (line 4 of Listing 22) is represented by an individual `UNSAFE_SINK` node.

Value nodes: Value nodes represent objects used during the execution of the program. In Figure 4.2, the `o1:person` is a value node that represents the object `person` with properties `age` and `status`, each of them associated with their respective value node. In our implementation, each value node contains only the properties actively used in the code and not all the properties the object might effectively hold during execution. This is enough to statically model data flows. The information in the object dependence graph is represented by the relationships between its nodes, which consist of three main types of edges:

Property edges: Property edges represent the structure of objects. In Figure 4.2, the edge between `o1:person` and `o2:age` means that the `person` object contains a property named `age`. There is also a property edge represented by an asterisk: `"*"`, for when the property is unknown. For example, in the code, `variable = person[propName]` the `propName` variable holds an unknown string, so it is not possible to statically specify the property being accessed.

Dependency edges: Dependency edges represent data dependencies between objects, sources, and sinks. In Figure 4.2, the edge between `o5:name` and `UNSAFE_SINK` states that the statement containing the unsafe sink call depends on the value of the `name` property. Our implementation does not express how that dependency occurs. For example, in line 4 of Listing 22 the `name` property is concatenated (`+` operator) with the `status` property, but that is not directly reflected in the object dependence graph because it is not necessary for reasoning about the data flow, but this information is still available in the CRG by accessing the AST for that statement.

```
{
    "sinks": {
        "code-injection": [
            {
                "sink": "eval",
                "type": "function",
                "args": [ 1 ]
            },
        ],
        "command-injection": "...",
        "path-traversal": "..."
    },
    "sources": [
        {
            "source": "parameter",
            "type": "module.exports"
        }
    ]
}
```

Listing 23: Configuration file example for listing unsafe sinks.

New Version edges: Our CRG implementation aims to keep track of the entire lifetime of objects during the execution of code. To achieve this, whenever an object is modified we create a new object with the updated property. This is represented in the graph by connecting the node representing the previous version of the object to the node representing its current version, via a new version edge. In Figure 4.2, the edge between `o1:person` and `o3:person` means that a new version of the `person` object is created by modifying the property `person.status`, which we can confirm in line 3 of Listing 22. Modifying an arbitrary property is represented by an asterisk: "*", similar to what also happens for property edges.

4.1.2 Querying for Injection Vulnerabilities

The Code Representation Graph is used as input to our Query Execution module, which loads it into a graph database and executes queries for detecting several vulnerability types. Although our evaluated prototype supports queries for detecting both injection and prototype pollution vulnerabilities, in this section we will focus on describing the querying process for detecting injection vulnerabilities, as this vulnerability type is more prevalent in the datasets used for the preliminary evaluation (Section 4.1.4).

An injection vulnerability might occur when data controlled by an attacker can reach an unsafe sink without being previously sanitised. In Listing 22 the `eval` function is called with unsanitized parameters dependent on inputs for the application, such as the `person` object, which is the argument of an exported function. This function is vulnerable to code injection and our prototype can detect such vulnerabilities by searching for data flows that create a path between the tainted source and the unsafe sink in the CRG.

To query for data flows we supply a configuration file to our query engine that contains a list of unsafe sinks and taint sources to consider for each query type. Listing 23 shows a snippet of our configuration file that establishes `eval` as an unsafe sink for *code-injection*, indicating its type

```
1  MATCH
2      -- tainted parameter subquery
3      (sourceNode:TAINTED_SOURCE)-[paramEdge:ODG_EDGE]->(paramNode:ODG_OBJECT),
4      -- source to sink subquery
5      (paramNode)-[:ODG_EDGE*1..]->(sinkNode:UNSAFE_SINK),
6      -- vulnerable function subquery
7      (functionNode)-[:REF]->(paramNode)
8  WHERE
9      paramEdge.RelationType = "taint"
10 RETURN *
```

Listing 24: Cypher query for detecting injection vulnerabilities from tainted function parameters in our prototype.

as a *function* and its first argument as the argument vulnerable to injection. We also indicate our taint sources as the parameters of exported functions.

Listing 24 shows a simplified version of our Cypher injection query for detecting injection vulnerabilities from a tainted parameter to a dangerous sink. We can describe this query using three subqueries:

Tainted parameter subquery (lines 3 and 9): This subquery matches all instances of `TAINTED_SOURCE` nodes connected to an `ODG_OBJECT` via a single `ODG_EDGE` (line 3) annotated with the *taint* marker (line 9). The goal of this subquery is to search for the parameters that contain attacker-controlled data. In Figure 4.2 this subquery would match the `TAINTED_SOURCE` and the `o1:person` objects.

Source to sink subquery (line 5): Now that we have the tainted parameter objects, we can start looking for flows between these tainted objects and sensitive sinks (like `eval`). This subquery looks for one or more ODG edges (`ODG_EDGE*1..`) between our tainted parameters and an `UNSAFE_SINK` node. Note that `ODG_EDGE` can be annotated with several labels, such as `NV`, `PROP` and `DEP` edges. This is relevant because after executing this query we will have potentially vulnerable data flows (vulnerable path) between tainted parameters and sensitive sinks, but we will have to perform checks on the edges of this path. For example, in Figure 4.2, this subquery will match two distinct flows: the flow `TAINTED_SOURCE->o1->o3->o4->UNSAFE_SINK` and the flow `TAINTED_SOURCE->o1->o3->o5->UNSAFE_SINK`. But the first flow is not vulnerable because the `person.status` property is written using a static string (line 3 of Listing 22). This operation is represented in the graph by the `NV(status)` label of the `ODG_EDGE`. This is the sort of check that needs to be performed by the Query Execution Module after executing this query inside the graph database. We also have to check this path for calls to sanitization functions, which could be described in the configuration file.

Vulnerable function subquery (line 7): This last subquery simply identifies the `FUNCTION` node that contains the tainted parameter and, consequently, the injection vulnerability. This is not relevant to detecting the injection itself but is relevant for helping the developers pinpoint the location of the vulnerability in the tested codebase.

The information extracted by this injection query is processed and presented to the user in the form of a taint summary, which is a JSON list containing an object for each of the detected vulnerable flows. Listing 25 presents a simplified taint summary for the code injection vulnerability presented in Listing 22. It identifies the file where the vulnerability was detected (`file`), the

```
[
    {
        "vuln_type": "code-injection",
        "file": "app.js",
        "vuln_function": "f",
        "vuln_function_lineno": 1,
        "sink": "eval",
        "sink_lineno": 4,
        "tainted_params": [
            "person"
        ],
        "params_types": {
            "person": {
                "age": "number",
                "status": "string",
                "name": "string"
            }
        }
    }
]
```

Listing 25: Taint summary result for running the injection query for our example in Listing 22.

vulnerability type (`vuln_type`), the vulnerable function (`vuln_function`), the vulnerable function's line number (`vuln_function_lineno`), the sink and sink's line number (`sink` and `sink_lineno`), as well as a list of the tainted parameters for that function (`tainted_params`). We also include a list of the properties handled by the function for each tainted parameter and their respective types (`params_types`).

This injection query applies to several classic injection-style vulnerabilities, such as *code injection*, *command injection*, *SQL injection*, *XSS*, *path traversal* and others. By including the appropriate known dangerous sinks for each of these vulnerability types in the configuration file, the Query Execution module can run this query for detecting specific vulnerability types and append each flow to the taint summary accordingly.

4.1.3 Prototype Implementation

Similarly to other CPG-based solutions for vulnerability detection, our prototype leverages graph traversals to search for patterns in the graph that indicate potential security vulnerabilities. As so, our prototype is composed of two processing pipelines: the *CRG builder module* and the *query execution module*. The *CRG builder module* was implemented with 5543 lines of TypeScript code and is responsible for parsing a JavaScript program and outputting the corresponding CRG. It receives as input a JavaScript program, which is parsed using Esprima v4.0.1 [57] and generates the program's AST and CFG, following the same definition as the original CPGs introduced by Yamaguchi et al. [170]. Then, the CRG builder resorts to these structures to create the CRG. The *query execution module* was implemented with 310 lines of Python code and is responsible for importing the CRG into a graph database and executing a set of queries to search for specific patterns in the CRG. The Python code that implements our query engine is used to handle the input and output of the Cypher [42] queries that execute inside the Neo4j v4.2.1 [114] graph

CWE	VulcaN Injection DS					Example DS				
	Total	*CRG Prototype*		*ODGen*		Total	*CRG Prototype*		*ODGen*	
		TP	TPR	TP	TPR		TP	TPR	TP	TPR
CWE-22	5	5	1.00	0	0.00	-	-	-	-	-
CWE-78	69	61	0.88	46	0.67	5	5	1.00	4	0.80
CWE-94	30	19	0.63	9	0.30	27	22	0.81	19	0.70
CWE-471	70	37	0.53	24	0.34	12	11	0.92	6	0.50
Total	174	122	0.70	79	0.45	44	38	0.86	29	0.66

Table 4.1: Number of vulnerabilities detected in the VulcaN Injection and Example datasets.

database. Our initial prototype supports queries for detecting both injection and prototype pollution vulnerabilities. The query engine is executed inside a docker container.

4.1.4 Preliminary Evaluation

To assess the effectiveness of our prototype, we decided to evaluate it against two datasets and compare it with other state-of-the-art CPG-based vulnerability detection tools. Our first dataset is a subset of our curated dataset of `npm` package vulnerabilities [24], consisting of 174 real-world classic injection-style vulnerabilities, such as five *Path Traversals* (CWE-22), 69 *Command Injections* (CWE-78), 30 *Code Injections* (CWE-94) and 70 Prototype Pollution/Internal Prototype Tempering vulnerabilities (CWE-471). Henceforth, we will refer to this first dataset as the VulcaN injection dataset. Our second dataset consists of 44 custom toy examples of classic injection-style vulnerabilities, such as 5 *Command Injections*, 27 *Code Injections* and 12 Prototype Pollution/Internal Prototype Tempering vulnerabilities. These toy examples, unlike the vulnerabilities in the VulcaN injection dataset, were designed to reflect all possible object-oriented data flows in Node.js code that might trick current static analysis techniques. Henceforth, we will refer to this second dataset as the Example dataset. We also decided to compare our results with ODGen [97], a system published during the development of our prototype that also designs a CPG-based graph structure for detecting vulnerabilities in object-oriented Node.js code.

Table 4.1 shows the true positive (TP) and true positive rate (TPR) of both our prototype and ODGen after executing them against the two evaluation datasets. Our prototype achieves high recall (TPR) for all tested vulnerability types in both datasets, with an average recall of 70% and 86% for the VulcaN injection and Example datasets, respectively. Our prototype also outperforms ODGen by 1.6 and 1.3 times for the VulcaN injection and Example datasets, respectively. ODGen fails to match our prototype's recall for all vulnerability types in both datasets. Note that this is particularly relevant in the Example dataset, as it was designed to focus on object-oriented flows. This means that our graph specification is more effective at modelling flows in object-oriented Node.js code when compared to ODGen.

In summary, we show that our prototype can effectively detect injection-style vulnerabilities in object-oriented Node.js code and also show improvements in recall when compared to ODGen, a state-of-the-art CPG-based vulnerability detection tool, which is shown in our previous work [24] to be among the best static analysis vulnerability detection tools for JavaScript code.

4.1.5 Conclusion and Future Work

We approached this project with the idea of not only improving recall when compared to the state-of-the-art, which we achieved as demonstrated above but also improving the precision of CPG-based vulnerability detection techniques by combining our prototype with a symbolic execution engine for filtering false positives and automatically generating exploits that trigger the detected vulnerabilities. By filtering false positives, and consequently improving precision, we would tackle one of the main drawbacks discussed in our previous work [24], where a high false positive rate makes developers waste a lot of resources and effort to manually analyze the output of vulnerability detection tools. By automatically generating exploits that trigger the detected vulnerabilities we give developers a way to test their code. This enables them to also test patches and subsequent versions of their code bases to ensure vulnerability mitigation.

We did not achieve all this in the context of this thesis but our prototype does output a taint summary that could be used as input for a symbolic execution engine. In fact, our prototype has already been improved to increase recall in a spin-off project of this thesis. In the context of that spin-off project, we have evaluated the improved system against the VulcaN injection dataset, the SecBench.js [23] dataset and a real-world (in the wild) dataset containing 430 Node.js packages. Our prototype successfully detected a total of 31 zero-day vulnerabilities across 54 `npm` packages, while signalling 47 false positives. Among these 31 vulnerabilities, we have been assigned one CVE [39] for a command injection vulnerability found in the *find-exec v1.0.3* [139] `npm` package. We expect additional CVEs to be assigned to other of our detected vulnerabilities.

In conclusion, our CPG-based graph specification was used to effectively detect injection-style vulnerabilities in object-oriented Node.js code and also showed improvements in recall when compared to ODGen, a state-of-the-art CPG-based vulnerability detection tool. We also adapted this prototype for detecting flows of privacy-sensitive data to sinks that violate GDPR compliance in a published paper called RuleKeeper, which we describe in detail in the next section. Lastly, we set the foundations for a spin-off project that was able to detect 31 zero-day vulnerabilities, with one CVE assigned. Additionally, this spin-off project is also working towards combining an improved version of our graph specification with a symbolic execution engine for improving precision and automatically generating exploits for the detected vulnerabilities.

4.2 RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks

Mafalda Ferreira, Tiago Brito, José Fragoso Santos, Nuno Santos. RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks. 2023 IEEE Symposium on Security and Privacy.

In this Section, we summarize the contributions, system design and discussion of RuleKeeper, a framework for detecting violations of GDPR policies in Web applications and help developers write compliant code. My contributions to RuleKeeper was designing and implementing the CPG-based static analysis subcomponent, which allows for detecting privacy-sensitive data flows within the tested Web applications.

The reproduction of this publication was slightly adapted to adhere to formatting requirements. The original version of this publication can be found at: `https://syssec.dpss.inesc-id.pt/papers/ferreira_sp23.pdf`.

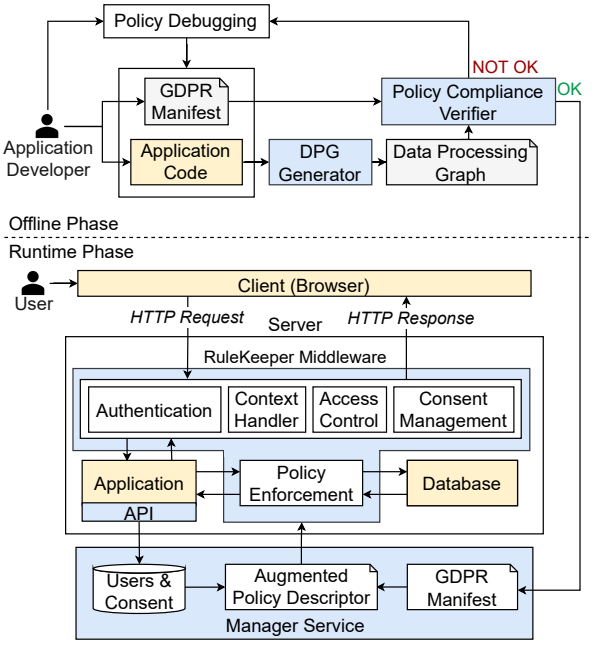

Figure 4.3: Architecture of a web application using RuleKeeper.

4.2.1 Introduction to RuleKeeper

The rise in concerns over personal data protection, especially with the enforcement of the EU General Data Protection Regulation (GDPR) [58], has highlighted challenges for online organizations. Full-stack web application development, such as MERN, lacks native support for GDPR compliance, leaving room for violations and privacy breaches.

RuleKeeper is a GDPR-aware policy compliance system for MERN [103] Web frameworks that minimizes code changes, generating a machine-readable GDPR manifest to define data processing policies. To bridge the semantic gap between GDPR concepts and application code, RuleKeeper introduces a domain-specific language (DSL).

RuleKeeper's static analysis tool component, based on this thesis' CPG-based static analysis for vulnerability detection, prevents GDPR violations by detecting privacy-sensitive data flows and ensuring that JavaScript code aligns with the specified purposes in the GDPR manifest. This tool generates a graph-based model of the JavaScript code and uses it along with the GDPR manifest to look for violations of GDPR's purpose limitation and data minimization guidelines.

In this thesis, we focus on describing our contribution to RuleKeeper, which consists of the design, implementation and evaluation of the static analysis component of RuleKeeper, based on our custom CPG specification for Node.js code.

4.2.2 System Design

Figure 4.3 depicts a RuleKeeper deployment for a 3-tier web application. The yellow boxes represent the application-specific components that are present in the typical behaviour of a 3-tier architecture system. The blue boxes represent RuleKeeper's specific software components. The system operates in two phases: *offline phase* and *runtime phase*.

The offline phase takes place at development time before deploying the web application to production. In this phase, the web developer specifies a GDPR *manifest* from which the data protection policy will be generated. RuleKeeper implements a static code analysis pipeline that allows the web developer to verify if the policy reflects the way that the application behaves. This verification is performed in two steps. First, a code analysis tool generates a model of the application code which we designate as a Data Processing Graph (DPG). Then, a compliance verification tool will look for inconsistencies between the DPG and the GDPR manifest. If this is the case, the developer needs to debug either the application code or the GDPR manifest itself to resolve these inconsistencies. When the validation step passes, the GDPR manifest is loaded to RuleKeeper's runtime components.

In the runtime phase, RuleKeeper implements dynamic policy enforcement and consent management functions using two components: *middleware* and *manager service*. The middleware consists of application-linked libraries and plays two roles: i) keeps the manager server updated with user-related information, and ii) enforces dynamic access control and consent management validations. The manager service runs in a centralized server and coordinates the middleware. Importantly, it generates a data structure named Augmented Policy Descriptor (APD) which contains the information required by the middleware to dynamically enforce the policy, such as the GDPR manifest and user consent preferences. The middleware leverages this information on two main occasions. In one case, it intercepts the HTTP requests to verify the user's credentials and consent preferences and enforce access control accordingly. The middleware also intercepts the database queries of the application to validate if they satisfy the GDPR policy and block them otherwise.

RuleKeeper supports dynamic policies in the sense that, whenever a meaningful contextual change occurs at runtime, e.g., a user changes their consent decision, the manager will accordingly update and propagate a new APD. Moreover, when a policy changes (e.g., due to a business-level decision), these changes can be reflected in a new manifest, checked through static analysis, and propagated by the manager via a new APD to the connected middleware instances. If the application is updated, e.g., with a feature that collects new personal data, the manifest needs to be updated, and the static analysis needs to be rerun to report mismatches.

4.2.3 Validating Policies with Static Analysis

Web developers use RuleKeeper in the offline phase to check if the application code satisfies the restrictions specified in the GDPR manifest. RuleKeeper verifies this using a static code analysis pipeline. After giving an intuition of our approach, we describe each pipeline stage.

Approach overview: Our approach is divided into two steps: first, we generate a model of the web application code which we call Data Processing Graph (DPG), and then we use the DPG to search for inconsistencies in the manifest. The DPG aims to automatically create a picture of the web application that can tell us: i) all the operations that exist in the code, and ii) all the data types accessible to each operation. Based on this information, we can then check if operations' data accesses are legal by comparing them against the respective specifications in the GDPR manifest. Figure 4.4 showcases a DPG sketch that reflects a toy implementation of a "buy ticket" operation. In a nutshell, the DPG: i) identifies each operation with an API endpoint and the code associated with it (e.g., `/buy_ticket`), and ii) identifies the accessed data types based on the database queries performed by the code at the endpoint (e.g., through `Ticket.create`). The static analysis automatically extracts the endpoints and data queries executed at those endpoints

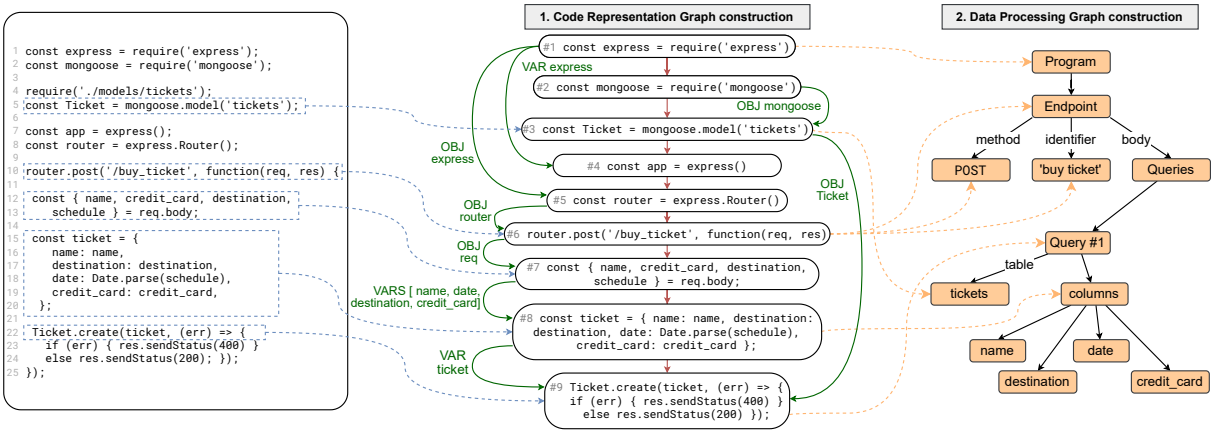

Figure 4.4: Static analysis stages for Webus sample code: the program is modelled into a CRG, which is then processed into a DPG.

based on an intermediate representation that we call Code Representation Graph (CRG). Next, we explain how these data structures are generated during code analysis, and then describe the algorithm for compliance checking.

1. Generating the Code Representation Graph: To build the CRG, we adapted the idea of code property graphs [170, 86, 97] to server-side JavaScript, with support for object dependencies. Thus, the CRG combines the abstract syntax tree (AST), the control flow graph (CFG), and the program dependency graph (PDG) in a single graph structure. In the CRG of Figure 4.4, the AST nodes are represented with numbered rounded boxes. For simplicity, we omit the AST edges and summarize the information in *Statement* nodes. The CFG and PDG edges between AST nodes are represented in red and green, respectively. First, RuleKeeper parses the JavaScript code and builds the corresponding AST. The AST is then traversed to produce the CFG, which creates edges between consecutive code statements and to possible branching operations – `if`, `while`, etc. Finally, the CFG is traversed to determine data dependencies between code statements, producing a PDG. A data dependency can be classified as a variable dependency (`VAR`) or object dependency (`OBJ`). To clarify the differences, consider the example in Figure 4.4. The program declares the variable `express` (line 1) and variable `app` (line 7), which depends on the former. Thus, our analysis creates a data-dependency edge between these statements with the label `VAR express`. On the other hand, the statement at line 8 declares a new variable `router` that also depends on `express`, but this time the content of the `router` variable does not directly depend on the value of the variable `express`, but instead on the value of a property of the `express` object. Consequently, a data-dependency edge between these statements is created with the label `OBJ express`. This analysis is intra-procedural. The output of this step is the CRG, which can then be queried using a graph database querying language to extract information for building the DPG.

2. Generating the Data Processing Graph: To build the DPG, we import the CRG into a graph database to allow for graph traversals, i.e., queries to check how data is being processed within the application. Next, we give an overview of the algorithm used to extract relevant information from the CRG using Figure 4.4 as an example. Our queries expect that the application implements endpoints and database queries using Express.js and Mongoose API calls.

1. First, we query the CRG for the registered endpoints, by searching code statements that register routes through function calls to Express.js's `Router` object. In the example of

Figure 4.4, the query finds the statement that registers the `POST` endpoint (node #6) and checks if it depends on the `Router` object (node #5). Then, we inspect the AST to extract both the path for the endpoint `/buy_ticket` – first argument of the call in node #6 – and the callback function – second argument. This callback is executed when the `/buy_ticket` endpoint is accessed and contains statements that handle user data.

2. Next, we query the CRG for the database queries executed in the context of each endpoint. First, we search for patterns that correspond to `model` calls (node #3), which depend on the Mongoose module (node #2), and extract the model name (`tickets`) by inspecting the AST. We then search for calls using the Mongoose Query API, such as `create` (node #9), that: i) depend on the model call identified before (node #3), and ii) are made inside the scope of the callback registered to the endpoints, such as `/buy_ticket` (node #6).

3. Finally, we search the AST of the Mongoose query calls for the data accessible (read or written) by the query. In this example, the inserted data corresponds to the first argument of the create call (variable `ticket`). For other Mongoose queries, e.g., `findOneAndUpdate`, the inserted data is in the second argument position. Since `ticket` is an object, we know that Mongoose will insert its properties into the database. With this information, we can generate a DPG for the policy compliance verifier. The DPG clearly tells that columns `name`, `credit_card`, `destination`, and `schedule` of table `tickets` are handled whenever the `POST` endpoint with path `/buy_ticket` is accessed, as shown in Figure 4.4.

3. Checking the DPG for policy compliance: From the resulting DPG, our analysis checks if the GDPR manifest reflects how the application processes personal data. The policy compliance verifier parses the manifest into a DPG-like data structure that associates endpoints with the personal data that the endpoints are allowed to process (e.g., endpoint `POST /subscribe` is associated with column "e_mail" of table "newsletters"). This way, one can directly match the information in the manifest with the DPG. At this point, the policy compliance analysis is divided into three validations:

1. *Personal data processing:* Looks for operations that process personal data but have not been declared in the manifest. To this end, RuleKeeper uses a filtered list of endpoints from the DPG that process personal data.

2. *Purpose limitation:* Screens purpose limitation violations. RuleKeeper uses DPG, filtered with queries that process personal data, and checks if any personal data item is processed by an operation for a purpose that is not represented in the DPG-like manifest data structure.

3. *Data minimization:* Similar to 2), but checks if any operation for a given purpose processes more data than the represented in the DPG-like manifest data structure.

4. Debugging inconsistencies: If RuleKeeper's static analysis endorses the GDPR manifest, showing that it reflects the way that the application is processing personal data, then the application is ready for deployment. Otherwise, the developer must fix the detected inconsistencies. These bugs can either stem from an over-permissive application code or an inaccurate GDPR manifest. Our tool reports them by indicating, for each endpoint: which data is expected to be processed (i.e., declared by the manifest), which data is actually being processed by the application, and which of the previous three validations is failing. The developer can either opt to i) update the application code, if she considers the code is over-permissive (i.e., processing more data than acceptable), or ii) update the policy to match the application code.

Application	CRG Size		Execution Time (s)			Accuracy
	Nodes	Edges	CRG	DPG Queries (N,C)		EM-pairs
LEB (257 LoC)	1047	1710	0.201	27.814	21.213	10/11
Amazona (570 LoC)	2238	4508	0.357	41.154	34.520	16/16
Blog (1075 LoC)	4189	8987	0.637	589.669	540.623	31/34

Table 4.2: Static analysis engine metrics.

4.2.4 Implementation of the Static Analysis Tool

We implemented RuleKeeper's static analysis tool using the Esprima v4.0.1 [57] parser with 2111 lines of JavaScript code. This analysis outputs the graph's nodes and edges that are later imported to a graph database. We used Neo4j v4.2.1 [114] as the graph database engine and a custom Python script, with 390 lines of code, to execute 11 custom queries that extract relevant information from the graph.

Portability: Although we target MERN for its popularity [145, 70], RuleKeeper relies on two general techniques that can be adapted to web stacks exposing similar programming abstractions, i.e., model-view-controller and REST APIs: (i) DPG creation in the static analysis, and (ii) middleware hooks in the runtime analysis. For JavaScript-based web frameworks, porting RuleKeeper would require the adaptation of these techniques to new database and hook interfaces. For other programming languages, e.g., PHP, static analysis could leverage already existing tools [127, 9].

4.2.5 Evaluation and Security Considerations of the Static Analysis

Table 4.2 shows several evaluation metrics for the static analysis running across our use cases. We collected the CRG size, execution times, and accuracy metrics. The CRG size increases almost linearly on the application LoC size. We measured the execution times of two distinct stages: i) CRG generation, and ii) CRG querying to extract the DPG. For the second stage, we gauge execution times with (C) and without (N) the Neo4j cache. For larger graphs, the queries' execution time grows to the order of minutes, but it does not affect the application execution time as static analysis runs offline and only occasionally when changes occur in the application code or manifest. Regarding accuracy, our static analysis correctly identified 57 out of 61 endpoint-model (EM) pairs, where EM pairs correspond to the queries executed in the context of an endpoint. The missed EM pairs occur as a limitation of static analysis.

Illustration of Static Analysis Limitations

In the full paper, we show that our static analysis engine fails to correctly detect one endpoint-model (EM) pair from the LEB use case and three EMs for the Blog use case. At their root cause lies the same limitation. Listing 4.5 shows the code snippet relative to the missed LEB EM pair.

Our static analysis detects the registration of the endpoint `POST /patients/:patientId` and the use of the `findOneAndUpdate` function being called on the `Patient` model, which corresponds to the `patients` table in the database. However, it cannot detect the specific parameters that are being updated, which correspond to the properties of the `data` object. The properties of the `data` object can only be known at runtime, as they depend on the user input sent via the body

```
const express = require('express');
const mongoose = require('mongoose');
const flatten = require('flat');

const patientRouter = express.Router();
const Patient = mongoose.model('patients');
// (...)

/* Update patient data. */
patientRouter.post('/patients/:patientId', function (req, res) {
  const { patientId } = req.params;
  let data = req.body; // data contains the colums to return

  if (data) data = flatten(data);
  const patient = { citizencard: patientId };
  Patient.findOneAndUpdate(patient, data, ...);
});
```

Figure 4.5: Code snippet of LEB application for which static analysis fails to detect columns being updated.

of the request object (`req`). From our experience, sending input objects directly to a database statement, such as `create` or `findOneAndUpdate`, is very uncommon and a security bad practice. Typically, EM pairs resemble the code shown in Listing 4.6, where a new object is created (`user`), whose properties depend on data from the input (`req.body`).

```
const express = require('express');
const mongoose = require('mongoose');

const userRouter = express.Router();
const User = mongoose.model('users');
// (...)

/* Create user. */
userRouter.post('/users/register', function (req, res) {
  const user = {
    username: req.body.username,
    password: hash(req.body.password)
  };
  User.create(user, ...);
});
```

Figure 4.6: Code snippet of a typical EM pair example.

In this example, the static analysis approach can detect the properties of the `user` object, given as a parameter of the `create` Mongoose function, and the structure of this object does not change for different inputs. To handle the lack of knowledge of the object properties, we consider that all data fields (as described in the data model) are being accessed, which can lead to false positives – only known at runtime. This strengthens the need for a second line of defence to prevent GDPR violations, at runtime, described in the full paper.

Policy Enforcement Properties and Limitations

RuleKeeper's policy enforcement relies on a combination of static and dynamic analyses for preventing GDPR compliance violations in web applications. Precision is a key consideration, addressing false positives that may arise due to the dynamic nature of JavaScript. While RuleKeeper's static analysis may report false inconsistencies, it acts as a pre-filter, identifying true violations early and serving as a debugging tool for developers. Soundness, the elimination of false negatives, is crucial for RuleKeeper, ensuring no GDPR violations go undetected. However, the static analysis is not entirely sound, leading to potential false negatives in certain scenarios. To address this, RuleKeeper employs dynamic analysis at runtime to intercept and monitor flows skipped by static analysis, guaranteeing soundness and detection of all non-compliant data processing flows. This combined approach ensures comprehensive enforcement of GDPR policies, provided the manifest is complete and accurately specified. The dynamic analysis applied at runtime does not constitute a contribution of this thesis but was applied in RuleKeeper to overcome some of the static analysis limitations.

Precision: Precision weighs the number of false positives in reporting inconsistencies between the web application and the GDPR manifest. As discussed in prior art [59, 148, 86], due to the dynamic nature of JavaScript, performing a precise and accurate static analysis of JavaScript-based web applications is difficult and may lead to misclassifications. Likewise, RuleKeeper's static analysis may report false inconsistencies between the application code and the GDPR manifest. For instance, CFG and DPG may contain edges that reflect implicitly hidden flows; hence, the resulting CRG may contain seemingly violating edges that may never get triggered in practice. (In Section 4.2.5, we examine the obtained false positives while testing RuleKeeper with real-world applications.) Pruning out false positives requires manually analyzing the application to check if its code is fully compliant with the manifest. Nevertheless, despite the added effort of the developers, RuleKeeper's static analysis brings two key benefits. First, it acts as a pre-filter [59] to preemptively identify true violations that would otherwise be detected at runtime only. Such a deferred detection (and consequent blocking) could impair service availability and increase maintenance costs. Second, RuleKeeper's static analysis can be used as a debugging tool to help developers reason about the privacy implications of their code and early-detect compliance bugs.

Soundness: A sound analysis eliminates false negatives, i.e., RuleKeeper will not fail to report an existing inconsistency between the application and the GDPR manifest. This is the most critically desired property for RuleKeeper, as it must not tolerate GDPR compliance violations to go past undetected. Unfortunately, RuleKeeper's static analysis is not sound and thus can have false negatives. For instance, it relies on the AST to learn the path and the callback function of each endpoint of the target application. However, this information may not be statically accessible in the AST; for example, in one use case application studied in the Evaluation, the path is the return value of a function invocation and not a static string, rendering it impossible to annotate the DPG with endpoint information for this application. JavaScript is also known for its dynamic code generation capabilities, such as the `eval` function, which can result in missing function calls and lead to an incomplete CFG. In such cases, the resulting CRG may lack some edges that can result in violations of the GDPR manifest. To prevent false negatives, we leverage RuleKeeper's existing runtime infrastructure with a dynamic policy enforcement mechanism. At runtime, RuleKeeper can intercept all the concrete execution flows and monitor the observable violating flows skipped by the static analysis. Therefore, complementing static with dynamic analysis guarantees soundness, ensuring that all non-compliant data processing flows are detected as long as the manifest is complete. If the manifest is incomplete, e.g., missing the definition of a personal

data type, the verified properties may not be the ones intended by the developer. To avoid this problem, manifests can be specified with the aid of automatic personal data classification tools [113], cross-validation by several developers, and cooperation with DPO.

4.2.6 Conclusions

RuleKeeper is introduced as a GDPR-aware policy compliance system designed for full-stack web development frameworks, specifically implemented for the MERN web stack. The system features a policy specification language and leverages static analysis to enforce policy compliance by detecting privacy-sensitive data flows while maintaining acceptable performance overheads. Future work includes enhancing the accuracy of static analysis and optimizing performance through methods like query rewriting for the offline phase.

Acknowledgments. We thank our shepherd and the anonymous reviewers for their comments and insightful feedback. We also thank all the participants of the usability study. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) via the 2021.06134.BD and SFRH/BD/146698/2019 grants, and the UIDB/50021/2020, PTDC/CCI-COM/32378/2017 (INFOCOS), and CMU/TIC/0053/2021 (DIVINA) projects.

Bibliography

[1] Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., and Shihab, E. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2017), ACM.

[2] Abraham, A. NodeJsScan. https://github.com/ajinabraham/njsscan. (visited on 07/01/2024).

[3] Abraham, A. NodeJsScan Eval SemGrep Rule. https://github.com/ajinabraham/njsscan/blob/master/njsscan/rules/semantic_grep/eval/eval_node.yaml. (visited on 07/01/2024).

[4] Abraham, A. NodeJsScan Rule Repository. https://github.com/ajinabraham/njsscan/blob/master/njsscan/rules/semantic_grep. (visited on 07/01/2024).

[5] Advisory 315 Page. https://www.npmjs.com/advisories/315. (visited on 07/01/2024).

[6] Advisory Report Procedure. https://docs.npmjs.com/reporting-a-vulnerability-in-an-npm-package. (visited on 07/01/2024).

[7] Aether. http://aetherjs.com. (visited on 07/01/2024).

[8] Alhuzali, A., Eshete, B., Gjomemo, R., and Venkatakrishnan, V. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the Conference on Computer and Communications Security (CCS)* (2016), ACM.

[9] Alhuzali, A., Gjomemo, R., Eshete, B., and Venkatakrishnan, V. NAVEX: Precise and scalable exploit generation for dynamic web applications. In *Proceedings of the USENIX Security Symposium* (2018), USENIX.

[10] Andreianu, G. Protecting your e-commerce business. analysis on cyber security threats. In *Proceedings of the International Conference on Cybersecurity and Cybercrime (IC3)* (2023).

[11] ApplicationInspector. https://github.com/microsoft/ApplicationInspector. (visited on 07/01/2024).

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

[13] Ashkenas, J. http://coffeescript.org/. (visited on 07/01/2024).

[14] Austin, T. H., and Flanagan, C. Permissive dynamic information flow analysis. In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)* (2010), ACM.

[15] https://github.com/dseguy/awesome-static-analysis. (visited on 07/01/2024).

[16] https://github.com/creinartz/awesome-static-analysis. (visited on 07/01/2024).

[17] https://web.archive.org/web/20230113040016/https://github.com/codefactor-io/awesome-static-analysis. (visited on 07/01/2024).

[18] Webassembly Open Source Projects. https://awesomeopensource.com/projects/webassembly. (visited on 07/01/2024).

[19] Backes, M., Rieck, K., Skoruppa, M., Stock, B., and Yamaguchi, F. Efficient and flexible discovery of php application vulnerabilities. In *Proceedings of the European Symposium on Security and Privacy (EuroS&P)* (2017), IEEE.

[20] Balzarotti, D., Cova, M., Felmetsger, V. V., and Vigna, G. Multi-module vulnerability analysis of web-based applications. In *Proceedings of the Conference on Computer and Communications Security (CCS)* (2007), ACM.

[21] Beyer, D. Advances in Automatic Software Testing: Test-Comp 2022. In *Proceedings of Fundamental Approaches to Software Engineering (FASE)* (2022), E. B. Johnsen and M. Wimmer, Eds.

[22] Beyond Security SAST. https://beyondsecurity.com/solutions/besource.html. (visited on 07/01/2024).

[23] Bhuiyan, M. H. M., Parthasarathy, A. S., Vasilakis, N., Pradel, M., and Staicu, C.-A. Secbench. js: An executable security benchmark suite for server-side javascript. In *Proceedings of the International Conference on Software Engineering (ICSE)* (2023), IEEE.

[24] Brito, T., Ferreira, M., Monteiro, M., Lopes, P., Barros, M., Santos, J. F., and Santos, N. Study of javascript static analysis tools for vulnerability detection in node.js packages. *IEEE Transactions on Reliability* (2023).

[25] Brito, T., Lopes, P., Santos, N., and Santos, J. F. Wasmati: An efficient static vulnerability scanner for webassembly. *Computers & Security* (2022).

[26] Bytecode Alliance. Wasmtime: a WebAssembly Runtime. https://wasmtime.dev/. (visited on 07/01/2024).

[27] Chakraborty, S., Krishna, R., Ding, Y., and Ray, B. Deep learning based vulnerability detection: Are we there yet. *Transactions on Software Engineering* (2021).

[28] Checkmarx SAST. https://www.checkmarx.com/products/static-application-security-testing. (visited on 07/01/2024).

[29] Chudnov, A., and Naumann, D. A. Inlined information flow monitoring for javascript. In *Proceedings of the Conference on Computer and Communications Security (CCS)* (2015), ACM.

[30] Clark, L. Standardizing WASI: A system interface to run WebAssembly outside the web. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, 2019. (visited on 07/01/2024).

[31] Coala. https://github.com/coala/coala. (visited on 07/01/2024).

[32] Codeburner. https://github.com/groupon/codeburner. (visited on 07/01/2024).

[33] CodeSonar. https://web.archive.org/web/20230828210142/https://www.grammatech.com/our-products/codesonar/. (visited on 07/01/2024).

[34] CoolerVoid. CodeWarrior. https://web.archive.org/web/20230316184859/https://github.com/CoolerVoid/codewarrior. (visited on 07/01/2024).

[35] CoolerVoid. Mosca. https://web.archive.org/web/20230113040008/http://github.com/CoolerVoid/Mosca. (visited on 07/01/2024).

[36] Coverty Scan. https://scan.coverity.com. (visited on 07/01/2024).

[37] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium* (1998), USENIX.

[38] CVE-2018-14550. https://www.cvedetails.com/cve/CVE-2018-14550/. (visited on 07/01/2024).

[39] CVE-2023-40582. https://nvd.nist.gov/vuln/detail/CVE-2023-40582. (visited on 07/01/2024).

[40] CWE-242: Use of Inherently Dangerous Function. https://cwe.mitre.org/data/definitions/242. (visited on 07/01/2024).

[41] CWE-658: Weaknesses in Software Written in C. https://cwe.mitre.org/data/definitions/658.html. (visited on 07/01/2024).

[42] Cypher. https://neo4j.com/developer/cypher/. (visited on 07/01/2024).

[43] Dahse, J., and Holz, T. Static detection of second-order vulnerabilities in web applications. In *Proceedings of the USENIX Security Symposium* (2014), USENIX.

[44] Davis, J. C., Coghlan, C. A., Servant, F., and Lee, D. The impact of regular expression denial of service (redos) in practice: An empirical study at the ecosystem scale. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2018), ACM.

[45] DeepScan. https://deepscan.io. (visited on 07/01/2024).

[46] Doupé, A., Cavedon, L., Kruegel, C., and Vigna, G. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proceedings of the USENIX Security Symposium* (2012), USENIX.

[47] Duan, R., Alrawi, O., Kasturi, R., Elder, R., Saltaformaggio, B., and Lee, W. Towards measuring supply chain attacks on package managers for interpreted languages. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2021), The Internet Society.

[48] Durieux, T., Ferreira, J. F., Abreu, R., and Cruz, P. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the International Conference on Software Engineering (ICSE)* (2020), IEEE.

[49] Damn Vulnerable Node Application. `https://github.com/appsecco/dvna`. (visited on 07/01/2024).

[50] ECMA Script Standard. `http://www.ecma-international.org/ecma-262/11.0/index.html`. (visited on 07/01/2024).

[51] Emscripten. `https://emscripten.org`. (visited on 07/01/2024).

[52] ESComplex. `https://github.com/escomplex/escomplex`. (visited on 07/01/2024).

[53] ESFlow. `https://www.npmjs.com/package/esflow`. (visited on 07/01/2024).

[54] ESLint. `https://eslint.org/`. (visited on 07/01/2024).

[55] ESLint Security Plugin. `https://github.com/nodesecurity/eslint-plugin-security`. (visited on 07/01/2024).

[56] ESLint Security Scanner Configs. `https://github.com/Greenwolf/eslint-security-scanner-configs`. (visited on 07/01/2024).

[57] ECMAScript parsing infrastructure for multipurpose analysis. `https://esprima.org/index.html`. (visited on 07/01/2024).

[58] European Parliament. Regulation (EU) 2016/679 of the European Parliament and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union* (2016).

[59] Fass, A., Backes, M., and Stock, B. JStap: A static pre-filter for malicious javascript detection. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2019), ACM.

[60] Felmetsger, V., Cavedon, L., Kruegel, C., and Vigna, G. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the USENIX Security Symposium* (2010), USENIX.

[61] Ferreira, M., Brito, T., Santos, J. F., and Santos, N. Rulekeeper: Gdpr-aware personal data compliance for web frameworks. In *Proceedings of the Symposium on Security and Privacy (SP)* (2023), IEEE.

[62] Ferreira, M., Monteiro, M., Brito, T., Coimbra, M. E., Santos, N., Jia, L., and Santos, J. F. Efficient static vulnerability analysis for javascript with multiversion dependency graphs. In *Proceedings of the Programming Language Design and Implementation (PLDI)* (2024), ACM.

[63] FireEye. IDA Pro loader and processor modules for WebAssembly . `https://github.com/fireeye/idawasm`. (visited on 07/01/2024).

[64] Fortify SAST. `https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview`. (visited on 07/01/2024).

[65] Fu, W., Lin, R., and Inge, D. Taintassembly: Taint-based information flow control tracking for webassembly. *arXiv preprint* (2018).

[66] Gauthier, F., Hassanshahi, B., and Jordan, A. Affogato: Runtime detection of injection attacks for node.js. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (2018).

[67] GitHub. CodeQL. `https://github.com/github/codeql`. (visited on 07/01/2024).

[68] GitHub. CodeQL Eval Taint Example. `https://github.com/github/codeql/blob/main/javascript/ql/examples/queries/dataflow/EvalTaint/EvalTaint.ql`. (visited on 07/01/2024).

[69] GitHub. CodeQL JavaScript Rule Repository. `https://github.com/github/codeql/tree/main/javascript`. (visited on 07/01/2024).

[70] GitHub. The 2021 state of the octoverse top languages. `https://octoverse.github.com/`, 2021. (visited on 07/01/2024).

[71] GitLab's CI/CD. `https://docs.gitlab.com/ee/user/application_security/sast/index.html`. (visited on 07/01/2024).

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

[73] Gotovchits, I., Van Tonder, R., and Brumley, D. Saluki: finding taint-style vulnerabilities with static property checking. In *Proceedings of the NDSS Workshop on Binary Analysis Research* (2018), The Internet Society.

[74] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. Bringing the web up to speed with webassembly. In *Proceedings of Programming Language Design and Implementation (PLDI)* (2017), ACM.

[75] Hall, A., and Ramachandran, U. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation (IoTDI)* (2019).

[76] Hedin, D., Birgisson, A., Bello, L., and Sabelfeld, A. Jsflow: Tracking information flow in javascript and its apis. In *Proceedings of the Symposium on Applied Computing (SAC)* (2014), ACM.

[77] Hedin, D., and Sabelfeld, A. Information-flow security for a core of javascript. In *Proceedings of the Computer Security Foundations Symposium (CSF)* (2012), IEEE.

[78] Hilbig, A., Lehmann, D., and Pradel, M. An empirical study of real-world webassembly binaries. In *Proceedings of the Web Conference (WWW)* (2021), ACM.

[79] HTML Living Standard. `https://html.spec.whatwg.org/multipage/urls-and-fetching.html#cors-same-origin`. (visited on 07/01/2024).

[80] InsiderSec. `https://github.com/insidersec/insider`. (visited on 07/01/2024).

[81] International Organization for Standardization. ISO/IEC 9899:2011. Standard, International Organization for Standardization, 2011. (visited on 07/01/2024).

[82] Jensen, S. H., Møller, A., and Thiemann, P. Type analysis for javascript. In *Proceedings of the International Static Analysis Symposium (SAS)* (2009).

[83] Joern. https://github.com/joernio/joern. (visited on 07/01/2024).

[84] Jordan, H., Scholz, B., and Subotić, P. Souffle: On synthesis of program analyzers. In *Proceedings of the International Conference on Computer Aided Verification (CAV)* (2016).

[85] JsHint. https://github.com/jshint/jshint. (visited on 07/01/2024).

[86] Khodayari, S., and Pellegrino, G. JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals. In *Proceedings of the USENIX Security Symposium* (2021).

[87] Kieyzun, A., Guo, P. J., Jayaraman, K., and Ernst, M. D. Automatic creation of sql injection and cross-site scripting attacks. In *Proceedings of the International Conference on Software Engineering (ICSE)* (2009).

[88] Kim, H. Y., Kim, J. H., Oh, H. K., Lee, B. J., Mun, S. W., Shin, J. H., and Kim, K. Dapp: automatic detection and analysis of prototype pollution vulnerability in node. js modules. *International Journal of Information Security* (2022).

[89] Kiuwan SAST. https://www.kiuwan.com/code-security-sast. (visited on 07/01/2024).

[90] Lane, C. A. Drek. https://github.com/chrisallenlane/drek. (visited on 07/01/2024).

[91] Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., and Kirda, E. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2017), The Internet Society.

[92] Lee, H., Won, S., Jin, J., Cho, J., and Ryu, S. Safe: Formal specification and implementation of a scalable analysis framework for ecmascript. In *Proceedings of the International Workshop on Foundations of Object-Oriented Languages (FOOL)* (2012).

[93] Lehmann, D., Kinder, J., and Pradel, M. Everything old is new again: Binary security of webassembly. In *Proceedings of the USENIX Security Symposium* (2020), USENIX.

[94] Lehmann, D., and Pradel, M. Wasabi. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2019).

[95] Lekies, S., Stock, B., and Johns, M. 25 million flows later: Large-scale detection of dom-based xss. In *Proceedings of the Conference on Computer and Communications Security (CCS)* (2013).

[96] Li, S., Kang, M., Hou, J., and Cao, Y. Detecting node.js prototype pollution vulnerabilities via object lookup analysis. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2021).

[97] Li, S., Kang, M., Hou, J., and Cao, Y. Mining node.js vulnerabilities via object dependence graph and query. In *Proceedings of the USENIX Security Symposium* (2022).

[98] Marcussen, E. Graudit. `https://github.com/wireghoul/graudit`. (visited on 07/01/2024).

[99] Marcussen, E. Graudit Rule Repository. `https://github.com/wireghoul/graudit/tree/master/signatures`. (visited on 07/01/2024).

[100] Mcallister, S., Kirda, E., and Kruegel, C. Leveraging user interactions for in-depth testing of web applications. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)* (2008), Springer.

[101] McFadden, B., Lukasiewicz, T., Dileo, J., and Engler, J. Security chasms of wasm. *BlackHat US-18* (2018). (visited on 07/01/2024).

[102] Melicher, W., Das, A., Sharif, M., Bauer, L., and Jia, L. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2018), The Internet Society.

[103] MERN Stack Explained. `https://www.mongodb.com/mern-stack`. (visited on 07/01/2024).

[104] Microsoft. DevSkim. `https://github.com/microsoft/DevSkim`. (visited on 07/01/2024).

[105] Microsoft. Data Execution Prevention. https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention, 2004. (visited on 07/01/2024).

[106] Microsoft. TypeScript language specification Version 1.8. Tech. rep., Microsoft, 2016.

[107] MITRE. Common Weakness Enumeration. https://cwe.mitre.org/. (visited on 07/01/2024).

[108] MITRE. CWE-OWASP Map. https://cwe.mitre.org/data/definitions/1344.html. (visited on 07/01/2024).

[109] Mozilla. ScanJS. https://github.com/mozilla/scanjs. (visited on 07/01/2024).

[110] Musch, M., Steffens, M., Roth, S., Stock, B., and Johns, M. Script-protect: Mitigating unsafe third-party javascript practices. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS)* (2019), ACM.

[111] Musch, M., Wressnegger, C., Johns, M., and Rieck, K. New kid on the web: A study on the prevalence of webassembly in the wild. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2019), Springer.

[112] Møller, A., and Schwartzbach, M. I. *Static Program Analysis*. Aarhus University, 2020.

[113] Nagpal, A., Dasgupta, R., and Ganesan, B. Fine grained classification of personal data entities with language models. In *Proceedings of the International Conference on Data Science and Management of Data (CODS-COMAD)* (2022), ACM.

[114] Neo4J. https://neo4j.com. (visited on 07/01/2024).

[115] Node.js. https://nodejs.org. (visited on 07/01/2024).

[116] NPM Advisories. https://www.npmjs.com/advisories. (visited on 07/01/2024).

[117] NPM Audit. https://docs.npmjs.com/cli/v7/commands/npm-audit. (visited on 07/01/2024).

[118] Ntantogian, C., Bountakas, P., Antonaropoulos, D., Patsakis, C., and Xenakis, C. Nodexp: Node. js server-side javascript injection vulnerability detection and exploitation. *Journal of Information Security and Applications (JISA)* (2021).

[119] Nunes, P., Medeiros, I., Fonseca, J., Neves, N., Correia, M., and Vieira, M. An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios. In *Proceedings of the European Dependable Computing Conference (EDCC)* (2019), Springer.

[120] Nunes, P., Medeiros, I., Fonseca, J. C., Neves, N., Correia, M., and Vieira, M. Benchmarking static analysis tools for web security. *Transactions on Reliability* (2018).

[121] OWASP. App Security Tools. https://owasp.org/www-community/Free_for_Open_Source_Application_Security_Tools. (visited on 07/01/2024).

[122] OWASP. Scanning Tools. https://owasp.org/www-community/Vulnerability_Scanning_Tools. (visited on 07/01/2024).

[123] OWASP. Top 10 Web Security Risks. https://owasp.org/www-project-top-ten. (visited on 07/01/2024).

[124] Parity. Blockchain Infrastructure for the Decentralised Web. https://www.parity.io. (visited on 07/01/2024).

[125] Pat Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime. (visited on 07/01/2024).

[126] Patnaik, N. D. JsPrime. https://github.com/dpnishant/jsprime. (visited on 07/01/2024).

[127] Pellegrino, G., Johns, M., Koch, S., Backes, M., and Rossow, C. Deemon: Detecting csrf with dynamic analysis and property graphs. In *Proceedings of the Conference on Computer and Communications Security (CCS)* (2017), ACM.

[128] Soot - A framework for analyzing and transforming Java and Android applications. https://soot-oss.github.io/soot/. (visited on 07/01/2024).

[129] PMD. https://github.com/pmd/pmd. (visited on 07/01/2024).

[130] Port Swigger. Burp Suite. https://portswigger.net/burp. (visited on 07/01/2024).

[131] Pouchet, L. N. PolyBenchC: the polyhedral benchmark suite. http://web.cs.ucla.edu/~pouchet/software/polybench/. (visited on 07/01/2024).

[132] Powers, B., Vilk, J., and Berger, E. D. Browsix: Bridging the gap between unix and the browser. *ACM SIGPLAN Notices* (2017).

[133] Rossberg, A. WebAssembly Core Specification. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. (visited on 07/01/2024).

[134] Rus, C., Sarmah, D. K., and El-Hajj, M. Defeating magecart attacks in a naiss way. In *Proceedings of the International Conference on Security and Cryptography (SECRYPT)* (2023).

[135] Rust Language. https://www.rust-lang.org/. (visited on 07/01/2024).

[136] SemGrep. https://semgrep.dev. (visited on 07/01/2024).

[137] ShiftLeft. https://www.shiftleft.io/. (visited on 07/01/2024).

[138] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the Symposium on Security and Privacy (SP)* (2016), IEEE.

[139] Simic, H. find-exec package v1.0.3. https://www.npmjs.com/package/find-exec/v/1.0.3. (visited on 07/01/2024).

[140] Snyk. https://snyk.io/. (visited on 07/01/2024).

[141] SNYK-JS-IBMDB-459762. https://snyk.io/vuln/SNYK-JS-IBMDB-459762. (visited on 07/01/2024).

[142] SonarQube. https://github.com/SonarSource/sonarqube. (visited on 07/01/2024).

[143] Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., and Franz, M. Sok: Sanitizing for security. In *Proceedings of the Symposium on Security and Privacy (SP)* (2019), IEEE.

[144] SPEC CPU® Benchmark 2017. https://www.spec.org/cpu2017/. (visited on 07/01/2024).

[145] Stack Overflow. Stack overflow developer survey results 2021. https://insights.stackoverflow.com/survey/2021, 2022. (visited on 07/01/2024).

[146] Staicu, C.-A., and Pradel, M. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *Proceedings of the USENIX Security Symposium* (2018), USENIX.

[147] Staicu, C.-A., Pradel, M., and Livshits, B. Synode: Understanding and automatically preventing injection attacks on node. js. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2018), The Internet Society.

[148] Staicu, C.-A., Schoepe, D., Balliu, M., Pradel, M., and Sabelfeld, A. An empirical study of information flows in real-world javascript. In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)* (2019), ACM.

[149] Staicu, C.-A., Torp, M. T., Schäfer, M., Møller, A., and Pradel, M. Extracting taint specifications for javascript libraries. In *Proceedings of the International Conference on Software Engineering (ICSE)* (2020), IEEE.

[150] Steffens, M., Rossow, C., Johns, M., and Stock, B. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2019), The Internet Society.

[151] Stiévenart, Q., and De Roover, C. Compositional information flow analysis for webassembly programs. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2020), IEEE.

[152] Stock, B., Johns, M., Steffens, M., and Backes, M. How the web tangled itself: Uncovering the history of client-side web (in) security. In *Proceedings of the USENIX Security Symposium* (2017), USENIX.

[153] Stock, B., Lekies, S., Mueller, T., Spiegel, P., and Johns, M. Precise client-side protection against dom-based cross-site scripting. In *Proceedings of the USENIX Security Symposium* (2014), USENIX.

[154] Security Team @ Técnico (STT). https://sectt.github.io/. (visited on 07/01/2024).

[155] Szanto, A., Tamm, T., and Pagnoni, A. Taint tracking for webassembly. *arXiv preprint* (2018).

[156] Thunderscan. https://web.archive.org/web/20210917212650/https://www.defensecode.com/thunderscan-sast/. (visited on 07/01/2024).

[157] https://web.archive.org/web/20231003202952/https://www.softwaresecured.com/top-sast-tools-for-developers/. (visited on 07/01/2024).

[158] https://medium.com/@manjula.aw/nodejs-security-tools-de0d0c937ec0. (visited on 07/01/2024).

[159] https://www.softwaretestinghelp.com/tools/top-40-static-code-analysis-tools/. (visited on 07/01/2024).

[160] Veracode SAST. https://www.veracode.com/products/binary-static-analysis-sast. (visited on 07/01/2024).

[161] WALA. https://web.archive.org/web/20230723160112/https://wala.sourceforge.net/wiki/index.php/Main_Page. (visited on 07/01/2024).

[162] Wasmer: An open-source runtime for executing WebAssembly on the Server. https://wasmer.io/. (visited on 07/01/2024).

[163] Watt, C. Mechanising and verifying the webassembly specification. In *Proceedings of the International Conference on Certified Programs and Proofs (CPP)* (2018), ACM.

[164] Watt, C., Maksimovic, P., Krishnaswami, N. R., and Gardner, P. A program logic for first-order encapsulated WebAssembly. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (2019).

[165] Watt, C., Renner, J., Popescu, N., Cauligi, S., and Stefan, D. Ct-wasm: Type-driven secure cryptography for the web ecosystem. *Proc. ACM Program. Lang.* (2019).

[166] WebAssembly Community Group. The WebAssembly Binary Toolkit. https://github.com/WebAssembly/wabt. (visited on 07/01/2024).

[167] Weiser, M. Program slicing. In *Proceedings of the International Conference on Software Engineering (ICSE)* (1981), IEEE.

[168] WhiteHat SAST. `https://www.whitehatsec.com/platform/static-application-security-testing`. (visited on 07/01/2024).

[169] Xiao, F., Huang, J., Xiong, Y., Yang, G., Hu, H., Gu, G., and Lee, W. Abusing hidden properties to attack the node. js ecosystem. In *Proceedings of the USENIX Security Symposium* (2021).

[170] Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the Symposium on Security and Privacy (SP)* (2014), IEEE.

[171] Yamaguchi, F., Maier, A., Gascon, H., and Rieck, K. Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of the Symposium on Security and Privacy (SP)* (2015), IEEE.

[172] Yan, L. K., and Yin, H. Sok: On the soundness and precision of dynamic taint analysis. *Formal. Taint* (2017).

[173] Zimmermann, M., Staicu, C.-A., Tenny, C., and Pradel, M. Small world with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the USENIX Security Symposium* (2019), USENIX.