Discovering Security Vulnerabilities in WebAssembly with Code Property Graphs

Pedro Daniel Rogeiro Lopes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors:  Prof. Nuno Miguel Carvalho dos Santos
              Prof. José Faustino Fragoso Femenin dos Santos

Examination Committee

  Chairperson:  Prof. Luís Manuel Antunes Veiga
  Supervisor:   Prof. José Faustino Fragoso Femenin dos Santos
  Member of the Committee:  Prof. Pedro Miguel dos Santos Alves Madeira Adão

January 2021
Acknowledgments

First I want to show all my gratitude to my family that supported me through my journey and made sure I had all the tools I needed to succeed. This work was just possible due to their direct effort along the past years. Secondly I would like to thank my supervisors professors Nuno Santos and José Fragoso Santos for all the provided advise and counselling at tackling the scientific challenges in this thesis. I also want to show my gratitude to Xpand IT for the provided financial support during 3 years and for the opportunities they introduced me. Lastly, I want to thank to all my friends prior to Técnicos and to those I had the honour to meet in Técnicos which, in a way or another, made my last 5 years a truly enjoyable journey that I will never forget.

To each and every one of you – Thank you.
Resumo

WebAssembly é uma nova tecnologia que permite que desenvolvedores da Web consigam executar código nativo C/C++ numa página Web com um desempenho próximo ao nativo e, por isso, mais rápido que aplicações típicas de JavaScript. Atualmente é suportado pelos navegadores Web mais populares. WebAssembly traz implicações para a plataforma Web já que permite que aplicações de cliente mais complexas sejam executadas nos navegadores. O formato binário compacto, desempenho e mecanismos de segurança presentes na linguagem teve como consequência o uso da linguagem em contexto fora da Web como codificar servidores, plataformas de IoT e edge computing. Contudo, apesar dos benefícios, a tecnologia de WebAssembly traz consigo algumas preocupações de segurança. Em particular, vulnerabilidades presentes em C e C++ como buffer overflow podem ser importados para WebAssembly. Esta dissertação descreve a arquitetura e implementação de Wasmati, uma framework que consegue detetar vulnerabilidades em código WebAssembly estaticamente fazendo travessias em code property graphs (CPGs). O CPG é a representação de um programa num grafo e que tem sido aplicado com sucesso na detecção de vulnerabilidades em linguagens de alto nível. Através de uma avaliação experimental do nosso sistema, nós observamos a eficiência e eficácia em detetar vulnerabilidades importadas de programas em C e a viabilidade de construir e analisar binários grandes como o compilador GNU GCC em menos de 10 minutos.

Palavras-chave: WebAssembly, Code Property Graph, Análise Estática, Detecção automática de vulnerabilidades
Abstract

WebAssembly is a new technology that allows web developers to run native C/C++ on a Web page with near-native performance and therefore much faster than typical JavaScript applications. Currently supported by the most popular browsers. WebAssembly brings implications for the web platform since it enables more complex client apps to run on the browser. The compact binary format, performance, and safety mechanisms present in the language led it to be used beyond the browser platform, being employed in the context of server-side runtimes, IoT platforms and edge computing. However, in spite of its benefits, WebAssembly technology brings some security concerns attached. In particular, vulnerabilities from C and C++ such buffer overflows can be imported to WebAssembly. This dissertation describes the design and implementation of Wasmati, a framework that can statically find outstanding vulnerabilities in WebAssembly code by querying code property graphs (CPGs). The CPG is a program representation in a graph structure that has been successfully applied to the detection of vulnerabilities in high-level languages. Through experimental evaluation of our system we observe the efficiency and efficacy in finding vulnerabilities ported from C programs and the feasibility to construct and analyse large binaries such the GNU compiler GCC in less than 10 minutes.

Keywords: WebAssembly, Code Property Graph, Static Analysis, Automatic Detection of Vulnerabilities
# Contents

Acknowledgments ................................................................. iii
Resumo ....................................................................................... v
Abstract ..................................................................................... vii
List of Tables ............................................................................. xi
List of Figures ............................................................................. xiii

1 Introduction ........................................................................... 1
1.1 Motivation ............................................................................ 1
1.2 Research Questions .......................................................... 2
1.3 Contributions ....................................................................... 3
1.4 Thesis Outline ...................................................................... 3

2 Background: WebAssembly ..................................................... 5
2.1 Efficiency and performance ............................................... 5
2.2 Runtime environment ....................................................... 6
2.3 Key Concepts ...................................................................... 7
2.4 Generate WebAssembly .................................................... 10
2.5 Security Mechanisms in WebAssembly .............................. 11

3 Related Work ......................................................................... 15
3.1 Studies on WebAssembly .................................................. 15
3.2 Security Enhancements for WebAssembly .......................... 17
3.3 Analysis Tools for WebAssembly ....................................... 18
3.4 Code Property Graphs ...................................................... 21

4 Code Property Graph in WebAssembly ................................. 25
4.1 Property Graph ............................................................... 25
4.2 Abstract Syntax Tree (AST) .............................................. 26
4.3 Control Flow Graph (CFG) ................................................ 28
4.4 Call Graph (CG) .............................................................. 29
4.5 Program Dependence Graph (PDG) ................................. 31
4.6 Code Property Graph (CPG) .............................................. 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Queries</td>
<td>41</td>
</tr>
<tr>
<td>5.1 Format Strings</td>
<td>41</td>
</tr>
<tr>
<td>5.2 Dangerous Functions</td>
<td>43</td>
</tr>
<tr>
<td>5.3 Use After Free and Double Free</td>
<td>44</td>
</tr>
<tr>
<td>5.4 Taint-style Vulnerabilities</td>
<td>45</td>
</tr>
<tr>
<td>5.5 Buffer Overflow</td>
<td>47</td>
</tr>
<tr>
<td>5.6 Limitations</td>
<td>52</td>
</tr>
<tr>
<td>6 Implementation</td>
<td>55</td>
</tr>
<tr>
<td>6.1 Implementation Overview</td>
<td>55</td>
</tr>
<tr>
<td>6.2 Code Property Graph</td>
<td>56</td>
</tr>
<tr>
<td>6.3 WebAssembly Analysis</td>
<td>57</td>
</tr>
<tr>
<td>6.4 Optimisations</td>
<td>60</td>
</tr>
<tr>
<td>7 Evaluation</td>
<td>63</td>
</tr>
<tr>
<td>7.1 Vulnerability Detection</td>
<td>63</td>
</tr>
<tr>
<td>7.2 Generation Performance</td>
<td>67</td>
</tr>
<tr>
<td>7.3 Query Performance</td>
<td>70</td>
</tr>
<tr>
<td>7.4 Key Findings</td>
<td>71</td>
</tr>
<tr>
<td>8 Conclusions</td>
<td>73</td>
</tr>
<tr>
<td>8.1 Conclusions</td>
<td>73</td>
</tr>
<tr>
<td>8.2 Future Work</td>
<td>73</td>
</tr>
<tr>
<td>Bibliography</td>
<td>75</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Most relevant WebAssembly’s features implemented in popular engines. . . . . . . . . . . . 6
2.2 A simple C function on the left and the corresponding WebAssembly’s text format and
    binary encoding on the right side respectively. . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.3 Most popular WebAssembly compilers. . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.1 Tools WebAssembly. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.2 Supported Languages by framework. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

4.1 Node i32.add. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.2 PDG edge. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.3 Transfer Functions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.4 Edges. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.5 Function node. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.6 Module node. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.7 Simple instruction node. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.8 Simple node. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.9 Constant node. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.10 Labelled node. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.11 Block node. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.12 If node. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.13 Load node. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.14 Store node. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.15 Opcode instructions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

6.1 Basic query API. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
6.2 Predicate API. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

7.1 Query vulnerability report by dataset. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
7.2 Report summary. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
7.3 CPG generation times and information from SPEC CPU 2017 binaries. . . . . . . . . . . . 67
7.4 Query execution time (s). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
## List of Figures

2.1 Memory Model of Wasm. ................................................................. 8
2.2 Stack state during execution of the else block in call to factorial(4) represented in Table 2.2. 8
2.3 Compiling code into WebAssembly. .................................................. 10
3.2 Code property graph for code sample in Figure 3.1. ............................ 21
3.1 Taint-style vulnerabilities. ............................................................... 21
4.1 Property graph. .......................................................... 25
4.2 Function addTwo and its WebAssembly’s AST representation. ................ 26
4.3 AST representation. .......................................................... 27
4.4 CFG function foo from Figure 3.1. .................................................. 29
4.5 Polymorphism in C++. ............................................................. 30
4.6 Polymorphism in WebAssembly. ...................................................... 30
4.7 Call graph representations. .......................................................... 31
4.8 Program dependence graph representation. .......................................... 32
4.9 WebAssembly abstract syntax. ......................................................... 36
5.1 Format strings. .......................................................... 41
5.2 AST and PDG of call to printf. ....................................................... 42
5.3 Use after free. ............................................................ 44
5.5 CPG from code in Figures 5.7 and 5.6. .............................................. 46
5.4 Integer overflow OpenSSH 3.3. ....................................................... 46
5.7 XSS activation from a buffer overflow. ............................................. 47
5.6 A call to bof with tainted parameter. .............................................. 47
5.8 Linear memory layout for different compilers. ...................................... 48
5.9 Stack buffer overflow. ............................................................ 48
5.10 Data buffer overflow. ............................................................. 48
5.11 Heap buffer overflow. .............................................................. 49
5.12 Function frame stack allocation from Figure 5.9. ............................... 50
5.13 AST and PDG excerpts from Figure 5.9. .......................................... 51
Chapter 1

Introduction

This dissertation addresses the problem of security vulnerabilities and flaws that can be concealed in WebAssembly code. It studies the propagation of different C vulnerabilities such as format strings, use after free, double free, and buffer overflows into the realm of WebAssembly and consequently to the Web. This thesis proposes an approach based on code property graphs (CPGs) [1], a program representation successfully employed in high-level languages, to assist analysts in statically detecting such flaws.

1.1 Motivation

The enlargement of the Web led to more complex, sophisticated, and demanding CPU applications such as: games, interactive 3D visualisation, audio, and video. By historical accident, JavaScript [2] was the only programming language natively supported by Web browsers and the basic option for the development of such applications. Having started as a simple scripting language to bring some interaction to the static Web and today, JavaScript has evolved into a high-level language that is currently used to write wide spectrum of software applications. However, JavaScript is a dynamically typed and interpreted language. In order to run, the code needs to be downloaded, parsed, compiled, and interpreted, which impairs the speed and overall performance of applications.

In 2015, software engineers from four major browser manufacturers – Google, Firefox, Microsoft, and Apple – have collaboratively proposed a new portable, low-level assembly-like language named WebAssembly (Wasm for short) [3]. WebAssembly is designed, essentially, for speeding up the execution of client-side web code. Its compact binary makes files much smaller in comparison with JavaScript textual files and faster to decode and execute, allowing Wasm programs to run at near-native performance with just only 10% penalty [4]. This is specially important considering that many client-side Web applications execute over slow networks, on mobile devices or other resources-constrained platforms.

Currently, all major browsers already implement WebAssembly and it is available to more than 93% of all users on the Web [5] at the time of writing this document. Moreover, the performance gains and the safety mechanisms provided by WebAssembly have gained popularity beyond the browser platform. It has been adopted for uses in server-side runtimes [6–8], IoT platforms [9], and in edge computing [10].
However, albeit the existence of security measures provided by WebAssembly, 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 [11]. Typical coding errors and multiple functions in C and C++ that are inherently unsafe, will also be unsafe in WebAssembly. Such vulnerabilities can potentially be exploited in the context of the Web, and used for mounting typical Web-based attacks, such as cross-site scripting (XSS) and code injection attacks.

In this work, we aim at creating Wasmatic, a framework to assist analysts in statically finding outstanding vulnerabilities in WebAssembly code. We propose to achieve this by adopting a recently proposed program analysis technique called code property graph (CPG) [1]. CPG creates a canonical representation of code by aggregating information from a program's syntax, control flow, and data dependencies into one single graph. The idea is based on the existence of different graph representations for code, and that patterns in code can be described as classes of graphs. While all these graphs represent the same code, each one is created in a certain context where some properties are easier to express. The search for vulnerabilities can then be reduced to performing simple queries which can be implemented as traversals of the program's CPG.

1.2 Research Questions

In this work, we investigate four main research questions:

• **RQ.1. What are the current available tools to analyse WebAssembly code?** Given the benefits and the importance of WebAssembly in the new Web ecosystem, new tooling is emerging continuously to fulfill the needs. We are interested in assessing existing tools that can analyse WebAssembly code. This will help us gain a deeper insight into the strengths and weaknesses of each technique when designing Wasmatic.

• **RQ.2. What is the WebAssembly’s code property graph model?** Code property graphs have been applied successfully to various high-level languages but never to Wasm. In order to apply CPGs to Wasm, it is necessary to model the code property graph by taking into account the intricacies of the low-level nature of Wasm.

• **RQ.3. How do WebAssembly’s vulnerabilities translate into CPG patterns/traversals?** The search for vulnerabilities reduces to querying the program CPG for patterns that are associated to each vulnerability. We need to study how each vulnerability ported from C is described as a CPG pattern so we can query it.

• **RQ.4. Can CPGs be constructed and analysed for large WebAssembly binaries?** Static analysis of code checks common programming practices, errors, and omissions by scanning the source code or an intermediate representation; this process can be prohibitively slow. For this end, we need to assess if our approach can scale into the analysis of larger binaries in reasonable time.
1.3 Contributions

This thesis analyses, formalises and evaluates a static analysis technique based on code property graphs to detect potential security flaws in WebAssembly code. The main contribution of this thesis is the implementation of Wasmati, a scalable version of a static analysis framework that is able to support the development and execution of queries for detecting security vulnerabilities in Wasm code. More succinctly, this thesis produced the following contributions:

- A full formalisation of Code Property Graph applied to WebAssembly.
- An in-depth analysis of WebAssembly vulnerabilities ported from C.
- An implementation of the framework named Wasmati.
- An extensive experimental analysis of Wasmati and a comparison against other WebAssembly’s data flow analysis tools.

1.4 Thesis Outline

The remaining of this document is organised as follows. Chapter 2 provides some necessary background on WebAssembly. Chapter 3 reviews the related work on current detection, analysis and security enhancements in WebAssembly as well the use of code property graphs in different languages. Chapter 4 presents the full formalisation of code property graph applied to WebAssembly. Chapter 5 dwells over the port of C vulnerabilities and how they translate into patterns in the CPG. Chapter 6 describes the design and implementation of Wasmati. Chapter 7 presents the results from the experimental evaluation of Wasmati. Lastly, Chapter 8 concludes this document outlining the main findings, and unveiling possible directions for future work.
Chapter 2

Background: WebAssembly

This chapter provides some necessary background to understand the focus of our work. We start by introducing WebAssembly by providing some general context about this language, and then specifying in more detail how WebAssembly is generated and what are its built-in security mechanisms.

With the evolution of the Web and its complexity, JavaScript lags behind to create efficient implementations due to the costly interpretation by the browser. To overcome these limitations, in 2013, asm.js [12] was introduced by Mozilla Firefox. It is a strict subset of JavaScript and designed specially for code execution speed. Later, some browsers started to embrace asm.js and added optimisations to gain performance. Despite the considerable performance gains, asm.js did not become a standard and lacks important features for performance, such as native 64-bit integers, as in JavaScript, and consequently in asm.js all numbers are IEEE-764 compliant [13] floating point doubles.

As an alternative, it was proposed a new standard and a new specification for a language called WebAssembly [3], which is a portable low-level byte code and a target compilation for efficient statically-typed code. It is designed to be fast and safe to execute, language-, hardware-, platform-independent, deterministic, compact, easy to validate, decode, generate, to reason about and debuggable, on the Web and off the Web. WebAssembly is compiled from high-level languages. For now, the supported high-level languages are C/C++, and Rust, mainly via the compilers Emscripten [14] and rustc [15], but several ongoing projects aim to support languages like Python [16] and C# [17]. Currently, Awesome WebAssembly Languages [18] lists a dozen of other languages. The resulting WebAssembly code (Wasm for short) is then interpreted and executed in a stacked based machine.

2.1 Efficiency and performance

Given that WebAssembly is more compact than JavaScript, it takes less time to download files. Even when considering modern techniques for JavaScript compression, compressed Wasm binary is still smaller. Also, it is faster to parse and validate because there is no need to generate the abstract syntax tree to transform into an intermediate representation as it is already in that stage. In fact, the parsing performance and compactness of asm.js code are inferior when compared to WebAssembly [19].
Despite being interpreted, due its low-level nature, Wasm code runs at near native speed with just only 10% penalty [4] which is a major improvement compared to JavaScript. Optimisations also become faster, most of them were already performed ahead of time by LLVM [20], the Wasm runtime does not spend time executing code to observe patterns and infer types used, work that is performed by JavaScript just in time compiler (JIT). Sometimes, JIT has to throw away code already optimised and retry it. This usually happens because JIT takes assumptions about the code that do not hold, for example the type of a variable. In WebAssembly, types are explicitly given and thus there is no need to make assumptions about them. For this reason, executing WebAssembly code becomes faster. As many optimisations made by JIT are simple not needed in WebAssembly.

### 2.2 Runtime environment

The first stable Wasm release implemented by the browsers (Firefox [22], Chrome [21], Edge [37] and Safari [23]) aimed at being a Minimal Viable Product (MVP). However, WebAssembly has been gaining new features through a standardization process [38] which are slowly being implemented, e.g., threading and garbage collector. Table 2.1 keeps track of the most relevant WebAssembly’s features in the most popular engines, distinguishing the ones that were accepted through the standardization process, and the ones that are still in progress. It states if a feature 1) is already implemented (✓), 2) is being implemented (✗), 3) is not implemented (⧗), 4) is not applicable to the engine (N/A). Microsoft Edge is not present in the table due to the fact Edge started to use Chromium [39] as its engine, the same engine used by Google Chrome.

Even though the main target of WebAssembly are the browsers, there are also many benefits to use it outside the browser. Nowadays, a significant code base for web servers is written in JavaScript powered by runtime environments such Node.js [6] and desktop applications like Visual Studio Code are also written in JavaScript powered by Electron [40]. One of the main reasons for the usage of such

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>JS Bigint to Wasm i64 [25]</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>N/A</td>
<td>N/A</td>
<td>X</td>
</tr>
<tr>
<td>Bulk memory [26]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Multi-value [27]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Import/export globals [28]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reference types [29]</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>❧</td>
</tr>
<tr>
<td>Non-trapping conversions [30]</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Sign-extension operations [31]</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✗</td>
<td>☒</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>☒</td>
<td>✗</td>
<td>✗</td>
<td>☒</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Table 2.1: Most relevant WebAssembly’s features implemented in popular engines.
C program code | Text Representation | Binary
--- | --- | ---
```c
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```
```
;; magic number
(func $factorial (param $n i64) (param $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
```
```
00 61 73 6D 01 00 00 00
01 00 01 60
01 73
06 03 00 01 00 02
0A 00 01 00 00
20 00 50
42 01
20 00 42 01 7D
10 00 7E
0B 0B 15 17
```

Table 2.2: A simple C function on the left and the corresponding WebAssembly's text format and binary encoding on the right side respectively.

runtime environments is the portability that they offer. The same code can run in multiple platforms. WebAssembly can bring performance gains to these execution environments, while at the same time maintaining their portability.

A new open source group named Bytecode Alliance founded by Mozilla, Intel, Red Hat and Fastly [41] was created to advance the state-of-the-art in WebAssembly runtimes, such as Wasmtime [24] and evolve standards such the Wasm standard itself and the WASI [8] standard to give a system interface between WebAssembly runtime and the operating system's kernel to run WebAssembly outside of the Web. Another runtime with the goal of executing WebAssembly outside the Web is Wasmer [7] which enables super lightweight containers that can run anywhere, from Desktop to Cloud and IoT devices or embedded in other programming languages as a library.

### 2.3 Key Concepts

WebAssembly is a portable, low-level, assembly-like language and its main format is a binary that is compact and quick to parse and validate. Since it is a low-level language, WebAssembly is a target language from other high-level languages such C/C++, Java, Python, C# among others. Besides the binary format, Wasm also comes with a text format representation with the goal of making it human-readable and easier to debug. Table 2.2 pictures the basic encoding (in binary) of the function factorial written in C, and the corresponding WebAssembly's text format.

In this section, we will describe some core concepts of WebAssembly in order to understand better how the language is structured and how it behaves. There are nine main basic concepts to know: values, instructions, stack, traps, functions, tables, linear memory, module and embedder. Each concept will be explored with more detail in the in the paragraphs below. In the explanations of the core Wasm concepts, we refer to Figure 2.1, which illustrates how they interact with each other.
Values: There are four primitive types in WebAssembly: i32, i64, f32 and f64. The first two represent integers with 32 and 64 bits respectively, whereas the last two respectively denote 32 and 64 bit floating point data. The integers are not inherently signed or unsigned, it depends on the context, which is itself determined by the operations being applied. The f32 and f64, also known as single and double precision (float and double in C/C++), have the binary representation compliant with the standard IEEE 754-2019 [13].

Instructions: WebAssembly code is a sequence of instructions that are executed in order. It is based in a stack machine, meaning that instructions receive arguments from the stack and return elements onto the stack as depicted in Figure 2.1. Taking the example shown in Table 2.2, we can see the stack representation during the execution of the else block to the function call factorial(4) in Figure 2.2.

Due to their nature, instructions fall into two main categories: simple and control instructions. Simple instructions perform operations on data, they consume their operands from the stack and produce a result that is placed on top of the stack. Control instructions alter the control flow of the code. In WebAssembly the control flow is structured, meaning it is expressed with well-nested structures such as blocks, loops and conditionals.

Even though WebAssembly is specified in terms of a stack machine, browsers do not actually implement stack machines for performance issues, allocating the best registers depending on the architecture of the machine where the code is running.

Traps: Some instructions may produce traps under some conditions. The main consequence of a trap
is the immediate abortion of the execution. Traps are reported to the host environment where they can be caught and handled properly. In case of JavaScript, a trap results in an exception. As of now, WebAssembly does not have the ability to handle traps.

**Functions:** Functions are used to organise Wasm code. Each function can have values as parameters, values as local variables and values as results of its execution (return values). A function must declare explicitly both its argument and return value types. Functions can be exported and can be imported and they can call each other including recursively.

**Tables:** Tables are arrays that indexes a particular element type. Programs can select such values indirectly through a dynamic index operand. In the MVP version, tables can only hold function references. Thereby, a program can call functions indirectly through a dynamic index allowing the emulation of function pointers and polymorphism in OOP languages such C++. The Figure 2.1 shows an indirect call of the index 2. This value can be evaluated in execution time and the function indexed by that value is executed. However, before execute the call, it checks if the signature of the called function matches with the signature declared in the callee.

Tables can be defined in a module (explained in more detail below). In the MVP, at most one table may be defined or imported per module. This restriction may be lifted in future versions.

**Stack:** As explained above, the instructions interact with an implicit stack. The stack is depicted in Figure 2.1 and can contain three different structures:

- **Values:** are the operands of the instructions (see this section above).

- **Labels:** represent the structured control instructions (e.g. blocks and loops). Labels carry the associated branch target, its arity and the address of the next instructions to execute when the branch is taken. A label is pushed onto the stack every time it enters a block or a loop and is popped out of the stack whenever it reaches the end of the respective block or loop, leaving at the top of the stack possible return values.

- **Activation Frames:** is equivalent to stack frames in x86. It contains return arity of the respective function and holds the values of all local variables (including arguments), in the order to their static local indices, and a reference to the function's own module instance. An activation frame is pushed to the stack every time there is a call or a call_indirect and is popped out of the stack when the function returns leaving at the top of the stack its return values.

**Linear memory:** The linear memory is a contiguous, untyped, byte-addressable array that can be read and written. The memory is a multiple of 64Kib which corresponds to a size of a WebAssembly's page. The initial size is defined by the data present in the binary and can be dynamically increased, being always initialised to zero by default. In the browser, the linear memory is a JavaScript's ArrayBuffer and the indices of the array are the addresses of the memory used by the program. A program can load and store values from/to a linear memory at any byte address (including unaligned). A trap occurs if an access is not within the bounds of the current memory size.
The linear memory does not hold global and local variables, the stack, and returned addresses. Managed data resides in the stack and is managed by the VM. Since WebAssembly is restricted to the four primitive types, all non-scalar data such as strings, arrays and other buffers must be stored in linear memory.

**Module:** A module represents the binary format of a WebAssembly that has been compiled. Contains definitions of functions, imports, exports, tables, global variables and memory. Definitions can be imported and can be exported under one or more names. In addition to definitions, a module can initialise data for the linear memory and tables. It can also define a start function that is automatically executed. A module is stateless and can be seen as just a binary large object (Blob).

**Embedder:** A WebAssembly application is executed independently within a sandbox environment (host environment) and it can only escape via dedicated APIs. The job of the embedder is to instantiate the Wasm modules and to handle their imports and exports. In the case of the browser, the host environment is the JavaScript interpreter which provides complementary environment-specific API definitions. A module that has been instantiated contains all the resources it uses at runtime, such as the memory and tables that the program will use.

### 2.4 Generate WebAssembly

Typically, code written in high-level languages gets compiled down to assembly, which represents human-readable machine code. Different processors and architectures define different machine codes and kinds of assembly. In the Web, when delivering code to run in the user’s machine, we do not know in advance in which architecture the code will run. So, in spite of its name, WebAssembly is not quite an assembly language as it does not target a specific architecture. It is a machine code for a conceptual machine and not to an actual physical machine. For this reason, WebAssembly instructions are often called virtual instructions and the whole instruction set is referred to a virtual instruction set architecture (virtual ISA).

Humans are not meant to program in WebAssembly. It is a target language whose goal is to provide a set of instructions that are closer and ideal to machines in order to run at near native speed. However, Wasm includes a text format representation [42] for debugging purposes, which is illustrated in Section 2.3.

The currently supported compiler tool chain is called LLVM [20]. It is a robust tool and has a various front-ends and back-ends that can be plugged into it. The LLVM front-end for the C programming lan-
There are various compilers for compiling C to Wasm, of which the most well-known is Emscripten [14], whose pipeline is illustrated in 2.3. Emscripten is an open-source tool that compiles code written in C/C++ down to WebAssembly. It uses Clang and LLVM to compile C/C++ code into Wasm which takes as input the source code and outputs three files: a Wasm file, a JavaScript file and an HTML file. The Wasm file is the compiled WebAssembly module. The JavaScript file is “glue code”, it instantiates the Wasm module, sets up the memory and imports, and has the runtime routines to execute the code. The HTML file is just a simple interface of the application and imports the JavaScript file for it run when the page is loaded. Functions written in C/C++ can be exported, the exported functions can be called by JavaScript code. Also, Emscripten provides an API and pre-processing directives that developers can use, for instance, have inline JavaScript code within the C/C++ code. This is useful to make changes in the DOM.

Similar to Emscripten, there is a possibility to use Clang/LLVM targeting WebAssembly applying WASI-SDK, compiling the source code to WASI. Then, the binary can run in Wasmtime [24], a WebAssembly runtime outside of the browser, or even in browser using Web Polyfill, which is a web page that implements WASI, a feature that browsers do not yet implement.

WebAssembly support is ever evolving, Table 2.3 shows the most popular compilers for the most popular languages. There are also various more obscure / hobbyist languages that support WebAssembly.

One implication that arrives with multiple compilers from different source languages is that, in addition to the source program, compilers also add their own code to handle host–environment-specific tasks from the standard libraries. For instance, in case of Emscripten in C/C++ source code, there is a need to add the JavaScript implementations of printf in order to print in the browser’s console or in the DOM.

### 2.5 Security Mechanisms in WebAssembly

One of the main goals in the specification of WebAssembly is being safe. In particular, there is a need to protect the user from applications having vulnerabilities due to buggy and/or intentional malicious code, and provide effective mitigation against exploitation. With this in mind, WebAssembly was designed with four main features that work towards such needs: environment protection with sandboxing, secure

<table>
<thead>
<tr>
<th>Compilers</th>
<th>Language</th>
<th>Stability</th>
<th>Vendor</th>
<th>Open-Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emscripten</td>
<td>C/C++, Scala</td>
<td>Stable</td>
<td>Emscripten</td>
<td>✗</td>
</tr>
<tr>
<td>Clang 9 (WASI)</td>
<td>C/C++</td>
<td>Stable</td>
<td>Bytecode Alliance</td>
<td>✗</td>
</tr>
<tr>
<td>Rust</td>
<td>Rust</td>
<td>Stable</td>
<td>Rust Wasm</td>
<td>✗</td>
</tr>
<tr>
<td>Pyodide</td>
<td>Python</td>
<td>Stable</td>
<td>Iodide</td>
<td>✗</td>
</tr>
<tr>
<td>Blazor</td>
<td>C#</td>
<td>Stable</td>
<td>Microsoft</td>
<td>✗</td>
</tr>
<tr>
<td>Go</td>
<td>Go</td>
<td>Experimental</td>
<td>Google</td>
<td>✗</td>
</tr>
<tr>
<td>TeoVM</td>
<td>Java</td>
<td>Experimental</td>
<td>TeaVM</td>
<td>✗</td>
</tr>
<tr>
<td>AssemblyScript</td>
<td>TypeScript</td>
<td>Stable</td>
<td>The AssemblyScript Project</td>
<td>✗</td>
</tr>
<tr>
<td>RemObjects</td>
<td>C#, Java, Swift, Oxygen</td>
<td>Stable</td>
<td>RemObjects Software</td>
<td>✗</td>
</tr>
</tbody>
</table>

Table 2.3: Most popular WebAssembly compilers.
2.5.1 Environment Protection with Sandboxing

As explained in Section 2.3, WebAssembly is sandboxed within the host, each application module is subjected to its security policies such as restrictions on information flow through same-origin policies [44]. Moreover, WebAssembly is restricted in its functionality. An application in WebAssembly has no means to handle peripherals or interact with the system outside of the sandboxed environment. It cannot open sockets and thus, cannot make requests. Functionality within the sandbox environment is also restricted. WebAssembly cannot access the DOM directly for instance. All this functionality must be delegated to host by using existent APIs.

In the case of the host is outside of the browser such as Wasmer, Wasmtime and Nodejs, anything that as to do with system resources, the WebAssembly program must ask the runtime via an API, and the runtime requests the operating system on behalf of the program. In this way, the runtime can limit what a program can do. It may not let the program act with all the permissions the current user of the operating system has. However, this mechanism by itself is not enough to ensure security, because the runtime may give full access to the existent capabilities of the system and in this case there is no improvement in security beyond that given by the semantics of the language. Yet, there is still the possibility of hardening functionality and create a more secure system.

2.5.2 Memory Model Security Measures

Section 2.3 explains that linear memory does not hold global and local variables. Global variables are stored apart in a table named global index space and are fixed-size and addressed by index. Local variables are stored within the protected call stack which is a structure that also holds the return addresses of the function calls. Since data is limited by the basic types of WebAssembly, local variables with unclear static scope such arrays, strings and other buffers existent in C/C++ are stored in linear memory. Buffer overflows, which result from exceeding the boundaries of an object by writing to adjacent memory, do not affect local and global variables. In contrast, data stored in linear memory can be affected since the bound check is performed at linear memory region granularity as stated above.

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 (that exists) can be read and written without any restrictions. This lack of restrictions leads to a possibility of change constant data, such as constant strings or constant values, that should not be changed at any point in the execution of the program. In terms of execution policies, by design, the linear memory is non-executable since the instructions are static and stored apart from linear memory. In addition to custom sections, their execution makes use of virtual memory and paging, which includes page guard security mechanisms.

In the browser, as WebAssembly memory are objects in JavaScript, forgotten cleared memory due to poorly memory management by the programmer, does not result in memory leaks because the JavaScript garbage collector will take care of it. The same does not hold to runtimes like Wasmer
and Wasmtime where memory leaks can occur if not properly handled by the runtime.

### 2.5.3 Control-Flow Integrity

Functions are indexed in a table and in order to be called, the target index must be a valid entry in the function table. This specification does not allow a common attack surface in C code where functions live in memory and function pointers can be corrupted in such a way that can point to a different memory location where malicious code was injected. In the browser, this table is implemented as a JavaScript object named `WebAssembly.Table` which is an array-like structure outside of WebAssembly’s memory. The values are references to functions.

In Wasm, there are two types of function calls: direct and indirect. Indirect calls differ from direct calls in that the index of the function to be called is only computed at runtime (e.g. polymorphism in C++). Indirect call must, however, supply the type signature of the function to be called, which is dynamically. The signature of the called function must match the signature specified at the call site. All the calls happen in the protected call stack. It is protected because it is not possible to overwrite a return pointer, making it invulnerable to buffer overflows. Branches also must point to valid destinations within the enclosing function.

Wasm control flow instructions are designed so that calling an unexpected function is likely going to fail. The expected and unexpected paths of execution are statically analysed at compile time. This hardens the possibility of hijack 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.

### 2.5.4 Compiler Mitigations

State-of-the-art compilers implement default security measures to attenuate or eliminate common vulnerabilities such buffer overflows, pointer subterfuge, division by zero, among others. The compiler tool-chain used to compile high level code to WebAssembly is essentially the same as the one used to compile high level code to native code. This means that extending existing compilation pipelines with support for Wasm is relatively simple. However, some of the security measures do not translate well for WebAssembly as they are not necessary. The control-flow integrity mechanisms and call stack protection prevent direct code injection thus, measures such canaries for stack smashing protection (SSP) [45] and restrict execution of certain sections of memory known as data execution prevention (DEP) [46] are not necessary in the protected call stack.

Address space layout randomization (ASLR) [47] which randomly arranges the code, stack and heap is not currently supported by WebAssembly. Even the support of ASLR would be ineffective since WebAssembly’s linear memory is addressed by 32-bit pointers which does not provide enough entropy for strong protection [48]. Indices of linear memory are deterministic and remain constant between
executions (also between compilation). It is expected that this functionality will be available in future versions of WebAssembly alongside with support to 64-bit addresses.

State-of-the-art compilers (including targeting Wasm) also produce warnings against use of potentially vulnerable functions (like `gets()` and `strcpy()`) and provide control flow integrity checks to protect code compiled to Wasm.

**Summary**

This chapter presented WebAssembly, starting with an overview of its performance benefits when compared to JavaScript as well as of the environments where it can be executed. We then provided a in-depth exposition of WebAssembly’s architecture and its internals instructions, tables, linear memory and stack. We focused our discussion on the existing tool-chains for generating Wasm as well as the existing mechanisms for securing Wasm applications, explaining their flaws and challenges. In the next chapter, we will have a look at the most relevant research work on static analysis for WebAssembly and pattern-based vulnerability detection tools for various programming languages.
Chapter 3

Related Work

This chapter provides an overview of the related work. First, we present the main existing research on WebAssembly focusing on studies, security enhancements, and analysis tools for WebAssembly. Then, we review existing methods for pattern-based vulnerability detection, with special emphasis on code property graphs and its existing applications. CPGs are of special interest to us given that constitute the basic technique that we adopt in the design of Wasmati for finding vulnerabilities in WebAssembly code.

3.1 Studies on WebAssembly

We present the existing WebAssembly studies in two parts. First, we focus on studies that analyse the performance benefits and the current adoption of WebAssembly in the wild. Second, we concentrate on existing works that study the soundness and security properties of this new language for the Web.

3.1.1 Performance and Adoption

Over the years there have been several attempts to mitigate the JavaScript's overhead in the browser and attain a closer performance to native execution. One of the selling points of WebAssembly is its design that enables a better performance when compared to JavaScript with little overhead when compared to native execution. The use of a stack machine similar to Java Virtual Machine (JVM) [49] and Common Language Runtime (CLR) [50] enables compact binaries. Nevertheless, WebAssembly stands out since its stack machine has a structured control flow and does not support objects.

There are two main studies about the performance of Wasm implementations: one comparing it against native code and another comparing different Wasm implementations. The first study is featured in the original paper presenting the WebAssembly specification [3], in 2017. In this work, the authors compare the execution time of the PolyBenchC [51] and SciMark [52] benchmarks between V8 engine (Chromium), SpiderMonkey (Firefox) and native (x86-64). Their findings show that WebAssembly performs very well, with 7 benchmark programs executing only 10% slower than native code, and nearly all of them experiencing at most a $2 \times$ slowdown compared to native code.
The second study was performed by Jangda et al. [4] in 2019. The authors built a framework named Browsix-Wasm. It consists of an extension from Browsix [53] that enables a large-scale evaluation of the performance of WebAssembly against native code and different WebAssembly implementations using various benchmarks. They used three benchmarks: 1) PolyBenchC [51], 2) SPEC CPU2006 [54] and 3) SPEC CPU2017 [55]. They ran the first benchmark to compare with the original work however, they argue that it does not represent typical Wasm execution workload. The last two are C/C++ suites, industry-standardised, CPU intensive that are used extensively to measure and compare computer intensive performance. The authors measured the execution times also in V8, SpiderMonkey and x86-64. Between Chrome and Firefox, the results are very similar with a slight advantage in the execution times to Chrome with 1.53x against 1.54x of Firefox when comparing to native code. This study also shows that WebAssembly implementations improved their performance since the first release in 2017. The authors also make a comparison between WebAssembly and asm.js where WebAssembly outperforms with a mean speedup of 1.54x in Chrome and 1.39x in Firefox with an overall mean speedup of 1.3x, observing similar results as Haas et al. [3] have.

Musch et al. [56] presented a study on the prevalence of WebAssembly in the wild. In this study conducted in 2018, they examined the top 1 million pages of Alexa for WebAssembly binaries. At the time, only 0.17% of the websites were loading WebAssembly modules. They manually categorised the modules according to its usage, to find that about 56% of the web pages used WebAssembly as cryptocurrency mining and, a minority, used as a form obfuscate malicious JavaScript and HTML code. However this study may be already outdated at the time of the writing of this document.

With the rise of WebAssembly's popularity, there are many applications that already run in WebAssembly, mostly graphic applications, 3D environments such Unreal Engine 4 [57] and Unity3D [58]. Many others are now being ported. The WebAssembly Package Manager (WAPM) [59] and Webassembly Open Source Projects [60] have a vast list of projects including OpenSSL [61].

### 3.1.2 Soundness and Security

WebAssembly specifies its operational semantics and type system. Watt, C. [62] mechanised the proof of the soundness of the type system using the Isabelle theorem prover and verified implementations of a type checker and interpreter. In their work, they exposed several issues in the official WebAssembly specification, which were promptly fixed thereby directly influencing the standard.

One of the main design goals of WebAssembly is security. This claim is stated in the initial publication [3] as well as in the WebAssembly’s official website. However, albeit the existence of security measures provided by the semantics of WebAssembly, vulnerabilities that do not exist in the source language can be introduced by compilers and many classes of vulnerabilities such buffer overflows, integer overflows, type confusion, use after free and double free that exist in languages like C and C++ can be imported into WebAssembly code. These type of vulnerabilities were first highlighted by McFadden et al. [11] and more recently by Lehmann et al. [63].

Lehmann et al. gave an in-depth analysis about linear memory and its use by compilers and other tool
chains alongside the flaws of the memory protections that are in place. In accordance with the analysis, they provided a set of primitive attacks that can target the linear memory, mainly buffer overflows. These type of attacks targets the corruption of memory, including constant data. Together with the primitive attacks, it was also provided a set of end-to-end attacks with real application resulting in various attack surfaces and vulnerability classes. Starting with XSS using a vulnerable version of libpng, to remote code execution in Nodejs to arbitrary file write in stand-alone VM despite being advertised as a secure platform to execute C/C++ code. They also showed the feasibility to attack the flow of the program and alter its course of execution in real-world web applications and compiled from large C/C++ programs.

3.2 Security Enhancements for WebAssembly

Beyond the WebAssembly studies referred above, the research community has proposed several ways for enhancing the security of WebAssembly. While some proposals propose solutions at the language specification level (Section 3.2.1), others operate at the runtime execution level (Section 3.2.2).

3.2.1 Extension to Memory Model

There has been early work on improving the safety and security through improved memory protections to avoid the porting of memory related vulnerabilities such as buffer overflow. The first proposal to extend Wasm specification is MS-Wasm [64, 65]. The main goals are to ensure spatial safety (prevent out-of-bounds writes/reads), temporal safety (avoid use after free) and pointer integrity (makes impossible the corruption of a pointer in memory to create a non-valid pointer). The idea to achieve the goals is to add new memory segments in WebAssembly that play alongside linear memory. The life span of each segment is manually managed. The interaction with a segment is performed using handles. The safety comes with the restriction over the handles and the fact that the segments cannot use the read and write instructions that exist in WebAssembly to interact with the linear memory.

A handle can be seen as a pointer to a specific memory location instrumented with additional information such as the start location, end location, offset and a flag. The start and end locations are to ensure that no write/read out of bounds happens, when accessing the memory a validation is made against these values. The offset is the actual location that is calculated with base+offset. The flag prevents a write/read to a segment if is set to true. We need to consider the possible overhead that arrives with load and stores since each one of the operations must be validated before being executed. To avoid the use of segments in the whole memory, can be considered to do in specific sections generated by compilers, for instance, the constant data section could be moved to a segment where the handles restrict the writes.

Other proposal is CT Wasm [66]. This proposal is made to prevent memory leaks and was motivated to ensure the secrecy of cryptographic keys that are used during cryptographic procedures. The idea is to create an additional linear memory and categorise it as secret. A new set of instructions would be necessary to choose which memory is to be read/write (public vs secret). The safety comes by restrict
Table 3.1: Tools WebAssembly.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Category</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>WABT [68]</td>
<td>Toolkit</td>
<td>Static &amp; Dynamic</td>
<td>General</td>
</tr>
<tr>
<td>idawasm [69]</td>
<td>Disassembler</td>
<td>Static</td>
<td>General</td>
</tr>
<tr>
<td>Wasabi [70]</td>
<td>Instrumentation</td>
<td>Dynamic</td>
<td>General</td>
</tr>
<tr>
<td>SEISMIC [71]</td>
<td>Cryptocurrency mining</td>
<td>Dynamic</td>
<td>Security-wise</td>
</tr>
<tr>
<td>POSTER [72]</td>
<td>Cryptocurrency mining</td>
<td>Dynamic</td>
<td>Security-wise</td>
</tr>
<tr>
<td>TaintAssembly [73]</td>
<td>Taint Analysis</td>
<td>Dynamic</td>
<td>Security-wise</td>
</tr>
<tr>
<td>Taint Tracking Wasm [74]</td>
<td>Taint Analysis</td>
<td>Dynamic</td>
<td>Security-wise</td>
</tr>
<tr>
<td>Wassail [75]</td>
<td>Taint Analysis</td>
<td>Static</td>
<td>Security-wise</td>
</tr>
</tbody>
</table>

data usage. For instance, disallow branches from values read from secret linear memory and prevent the use of secret values where public values are expected i.e. writes to public linear memory or parameters from public functions. By default, instruction will only compute over public data unless is explicitly said not to, preventing from leaking unintended information.

### 3.2.2 Sandboxes and Runtimes

One is Gobi [67], a system of compiler changes and runtime support that can sandbox normal C/C++ libraries with Wasm. Another that have already been introduced in Chapter 2 is WASI (WebAssembly System Interface) [8] that has Clang back-end and produces Wasm binaries that use WASI-libc that do not require an embedding runtime (with exception of the browser that needs Web Polyfill).

Lucet [10] is a native Wasm compiler and a runtime, built on top of Cranelift code generator that can generate ELF binaries from a Wasm module. Despite being native code, the compiled version cannot interfere with its environment unless explicitly allowed by its hosts. The same occurs for accessible external functions and files. It allows multiple instances and modules to share memory within the same space. It has some limitations since it does not provide a convenient way to register and unregister callbacks dynamically and presents some performance overhead.

### 3.3 Analysis Tools for WebAssembly

This section addresses research question RQ1. As a result of being a rather new technology, the available tools for analysing binaries and text format in WebAssembly are only now starting to appear. Analogously, there is a lack of documentation to allow an efficient and easy way to analyses it, making a bit like a black-box to a human analyst. Good tooling support becomes necessary as the language evolves. Next, we briefly survey the most relevant work. Table 3.1 summarises the tools presented.

### 3.3.1 Toolkits, Disassemblers and Debuggers

With the creation of a new language there is a need to provide good tooling to interact with it. Furthermore, the low-level nature of WebAssembly, requires the existence of disassemblers. We will now give an overview of the most widely known toolkits, disassemblers and debuggers in WebAssembly.
We start with WebAssembly Binary Toolkit (WABT) [68], a toolkit that is maintained by the official WebAssembly Community. It includes various tools for streamlining static analysis for Wasm. The most important static tools in this toolkit can be used to validate modules, convert between WAT files and Wasm files and decompile Wasm files to a readable C source and header. Simple analysis in the binary include print information about its sections, remove sections or count opcode usage for instructions. It also includes an interpreter which decodes and runs WebAssembly binary file using a stack based machine.

In the same direction, there is already a plugin to Interactive Disassembler (IDA) named idawasm [69] supporting loading and disassembling of WebAssembly modules. It provides a reconstruction of a control flow graph with visual aid, code and data cross references and allows the rename of globals, locals, function parameters to facilitate the analysis. This plugin enables the analysis of WebAssembly using IDA, a full integrated development environment that is the standard for analysis of hostile code and vulnerability research.

Alongside these tools, there are many debuggers. In the front-line are the browsers and their embedded debuggers. They take advantage of what already exists for JavaScript – source maps. The compiler that generated WebAssembly code must also generate debug information in a source map format and in the binary. Emscripten [14] already generates source maps. The source map aggregates information and does a bidirectional map between locations of the source code and the generated code. The browsers’ developer tools use the source maps to symbolicate backtraces, and to implement source-level stepping in debuggers. Clang and LLVM also generate debug information if asked to. This information can be used to debug programs using LLDB or, more recently, GDB.

3.3.2 Instrumentation

Wasabi [70] is a dynamic analysis framework for WebAssembly. It is similar to other tools targeting native binaries such as Pin, Valgrind or Jalangi. It takes the WebAssembly binary and instruments it with callbacks and hooks. Wasabi inserts code in the binary that eventually calls the high-level hooks written in JavaScript giving further information such as the type of the functions in the program in analysis. It can be useful for profiling instructions, basic blocks, branch and instruction coverage. It is possible to find possible dangerous function calls by doing call graph analysis or search for taist-style vulnerabilities by performing dynamic taint analysis. Cryptocurrency mining detection can be achieved by profiling instructions and through the frequency of certain instructions (e.g. xor, add and mul) and memory access tracing it is possible to infer a cryptominer algorithm signature.

However, despite the multitude of analysis that can be done using Wasabi, this tool is not automatic, the human analyst is required to program all the callbacks used to perform the analysis. Also binary is modified to perform the analysis thus, is no longer the original program.
3.3.3 Cryptography and Cryptomining

The security of cryptographic procedures and the prevalence of cryptocurrency mining (cryptomining) motivates the tools that we are going to present. Protzenko et al. [76] define a language that is a subset of $F^*$ that compiles into WebAssembly. Its main goal is to create a programming language and a compiler tool chain that is focused in compactness and auditability. Alongside with the tool chain, they provide a high-assurance cryptographic library named WHACL* that is based in HACL* [77]. Their main focus is to assure the compiled WebAssembly has strong guarantees of safety when using cryptography within a larger security protocol or application. Mainly, guarantees such memory safety, functional correctness, mitigation against side-channel attacks and cryptographic security.

In a different perspective, with the main focus of detecting WebAssembly’s cryptocurrency mining programs, there have been made two proposals named SEISMIC [71] and POSTER [72] to monitor the execution of WebAssembly. The main goal is to detect cryptocurrency mining procedures and terminate them eventually. The difference between them and a classic antivirus is that antiviruses usually try to find a signature of the execution code against a database of hashes, while these two tools tries to infer a cryptocurrency mining program by the frequency, distribution and sequences of key WebAssembly instructions like add, xor, shl and its variations. Using these different metrics they attribute a score from 0 to 1 to evaluate the likelihood that the current code executing is a cryptocurrency mining algorithm. Both implementations achieved a highly precision of almost 100% with very low false-positives (depending in the threshold of the scoring). Furthermore, POSTER that is a more recent study and conducted an evaluation in the Alexa top 100K websites and found a total of 87 websites mining using the CryptoLoot algorithm. These types of monitors add some overhead in the execution of the program.

3.3.4 Taint Analysis

To the best of our knowledge, there are three taint analysis tools for Wasm. The first tool is named TaintAssembly [73] and is a monitor that is implemented in the V8 engine. It provides basic taint tracking functionality to follow specific data throughout the execution of the program that can be considered tainted such as user inputs, network packets or other information that could be important. It also have taint system analysis for linear memory that maps memory addresses to taint values. Alongside with these two methods, they have a probabilistic taint tracking that, as the name suggests, propagates certain values based on a certain probability associated with the result of an operation on the values. According to their results, their tool can have an overhead up to 4x in loading function compared to the unmonitored version. Their approach also brings limitations, explicitly do not assign taint to comparison operators, the probabilistic tainted propagation may not have enough entropy and need to use heavy procedures per operation resulting in a damage of its scaling, and finally, there is a need of a more efficient taint tracking for WebAssembly as it has huge overhead.

Another tool by Szanto et. al [74] implements a JavaScript VM that executes WebAssembly and employs a basic taint propagation of its code. According to their results, the taint propagation has a time overhead ranging from 20% up to 50%. A clear limitation of this tool is that the WebAssembly code must
run in their implemented VM which will be slower than any high-performance WebAssembly’s engine like V8, SpiderMonkey, among others.

The third and last, is a static taint analysis tool named Wassail [75]. It computes information flow for each function in a Wat program from its parameters and global state program state can flow to. The output of the tool are function summaries that are the result of the computation. The information flow analysis is expressed over control flow graphs that they construct from the Wat program and is then visited in order to propagate dependencies at each instruction. Their results states a precision of 64% using the PolyBenchC benchmark with a total time of 56 seconds of analysis.

One limitation of this tool is that the input are only Wat files and does not support the binary version Wasm.

Wasmati can also be employed in taint analysis by making use of the program dependence graph. Since it is a static tool, it differentiates from the first two tools that are dynamic thus, it does not produce overhead. However can be less precise. In comparison with Wassail, the calculated function summaries are equivalent to the edges of the program dependence graph produced by Wasmati that reach the return node of a function. In this perspective, Wasmati has all the features of Wassail augmented with more information about data propagation and the program structure in general making it easier to reason about it. Also, Wasmati can parse WebAssembly binaries and its text format.

### 3.4 Code Property Graphs

In an attempt to aggregate all information from different code structures in a graph, Yamaguchi et al. [1] published the original idea of code property graph (CPG) aimed at searching for vulnerabilities in programs. Currently, CPG is the main component of the only product offered by ShiftLeft [78]. They claim it can inspect and analyse up to 500K lines of source code in less than 10 minutes [79] with high level
Table 3.2: Supported Languages by framework.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Supported Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShiftLeft [78]</td>
<td>C/C++, Go, Java, Scala, JSP, LLVM, Python, Swift, and JavaScript</td>
</tr>
<tr>
<td>Joern [84]</td>
<td>C/C++</td>
</tr>
<tr>
<td>Joern-PHP [85]</td>
<td>C/C++, PHP</td>
</tr>
<tr>
<td>NAVEX [86]</td>
<td>PHP</td>
</tr>
<tr>
<td>CodeQL [87]</td>
<td>C/C++, Go, Java, Python, JavaScript and Typescript</td>
</tr>
<tr>
<td>Plume [88]</td>
<td>Java Bytecode</td>
</tr>
</tbody>
</table>

of accuracy compared with other state-of-the-art tools [80]. CPG can be employed to identify code weaknesses, such as methods with too many parameters, improperly sanitised inputs, duplicate code or inconsistent name conventions in the code. It can have similar functionality as the current lints in the market.

3.4.1 Technical Approach

CPG aggregates information of different code representations in a multi-layered structure that is richer and more comprehensive than most known alternatives. It delivers high insight about the analysed code and vulnerabilities become easier to identify. More specifically, CPG combines the source’s syntax, control-flow and data-flow information in one graph stored in a graph-database. In particular, a CPG is built out of the combination of three representations of a target program: abstract syntax tree (AST), control flow graph (CFG), and program dependence graph (PDG). For instance, Figure 3.2 represents the CFG data structure for the code sample depicted in Figure 3.1. Once the CPG has been generated, security vulnerabilities can be found using graph-queries that inspect the CPG for the patterns associated with the searched vulnerabilities, e.g., checking the existence of some path in the graph connecting nodes that represent data sources and sinks.

In its original idea, CPG does not support inter-procedural analysis. The authors extended the work one year later to support it [81], employing a similar representation of the well known System Dependence Graph [82]. They extended the CPG by explicitly defining data-flow between call sites and their callees introducing edges between the two nodes representing the arguments to parameters of the respective callees and also between the return statement back to the call site. This approach does not encode possible modifications of data that may occur, nor the effects that it can have as data flows back along the call chain. So, they also introduced the idea of using post-dominator trees [83], a classical program representation derivable from the control flow graph. This additional information, by linking both nodes, makes easy to determine if a statement is preceded or followed by another.

3.4.2 CPG Adopters

Besides ShiftLeft, there are numerous projects that adopted code property graphs targeting different languages besides from C. There is work in highly dynamic languages like Python, JavaScript and
In this section we will walk through some of the major works that adopted CPG. Table 3.2 summarises the language supported by each work.

Joern [84] is an open-source platform for analysis of C/C++ code. It employs code property graphs and developed a Scala-based domain-specific query language and its engine enabling the mining of large code bases for vulnerabilities. Joern is the work of Yamaguchi et al. [1] and is “the fundament for the commercial SAST and code exploration products at ShiftLeft”.

Backes et al. [85] extended Joern and applied the concept of code property graphs to PHP code. They also extended it to support inter-procedural analysis by enriching the CPG with information coming from the call graph. Call graphs are just directed graphs whose edges connect call sites with their corresponding function definitions, allowing to reason about control and data flow between functions, i.e., at the inter-procedural level. Subsequently, they identified different types of Web application vulnerabilities by means of programmable graph traversals using the graph database Neo4j [89]. They showed that this approach scales well by analyzing a large amount PHP projects that sit on Github, reporting new found vulnerabilities.

One other tool that makes use of CPGs for the analysis for PHP code is NAVEX [86]. NAVEX combines CPG with dynamic analysis, using CPG to guide their dynamic analysis in search of vulnerabilities in PHP code. Besides the findings of vulnerabilities, they went a step further in the generation of exploits for the newly founded vulnerabilities. They evaluated 26 real-world application which resulted in a construction of 204 exploits where about 96% are a result of taint analysis.

CodeQL [87], originally developed by Semmle, is a static analysis framework that models source code as a database and allows it to be queried in a SQL-like syntax language. The database contains a full hierarchy representation of the source code, including abstract syntax tree (AST), control flow graph (CFG) and data flow graph (DFG). Vulnerabilities, bugs and errors in code are modelled as queries to be executed in the code database. The queries are written in a specially-designed language named QL. It is an object-oriented query language. CodeQL defines dedicated abstractions over the database tables of each of the languages that it supports. As of now, CodeQL supports 7 high-level languages which are: C/C++, Go, Java, Python, JavaScript and Typescript.

Lastly, Plume [88] is a static analysis tool that constructs the code property graph for JVM bytecode. It constructs CPG from the control flow graph generated by Soot [90] which is stored and queried using a library or natively in a graph database. Indirectly, Plume can analyse any language that compiles to JVM bytecode. As of now, Plume is still under development and does not have an official release yet.

As the time of writing this document and to the best of our knowledge, no tool yet exists that implements CPGs for the purpose of finding vulnerabilities in WebAssembly.

**Summary**

In this chapter we presented the most relevant work in WebAssembly. Starting with studies about performance results against native code, general adoption and soundness and security of WebAssembly code that includes verification and validation of its specification. Then we presented work in the area of
security enhancements and highlighted issues unsolved in the specification of WebAssembly that allow
the import of vulnerabilities from C/C++. To mitigate the attack surface, we presented some existing pro-
posals to extend the Wasm memory model. Other works headed to finding the vulnerabilities in Wasm
code with different types of tools and methodologies including sandboxing.

After WebAssembly, we introduced the concept of code property graphs and works that adopted this
technique to analyse code in the search for vulnerabilities. Despite the success of CPG-based tools in
finding vulnerabilities in various programming languages, there is still no such tool for Wasm. In the next
chapter, we propose a novel CPG for Wasm, which enables the efficient detection of common security
vulnerability patterns
Chapter 4

Code Property Graph in WebAssembly

In this chapter we focus on our research question RQ2. CPGs combine in the same structure four different representations of a program: ASTs, CFGs, CGs, and PDGs. We describe how to put together these four different representations of Wasm programs in order to obtain code property graphs that enable the efficient detection of security vulnerabilities in Wasm code. To present the novel CPG representation for Wasm, we will first describe the definition and construction of the various necessary representations for Wasm programs. The chapter will be divided into sections where each section will approach a main component of a CPG. Starting with the core concept of Property Graph in Section 4.1, then we proceed to the Abstract Syntax Tree (AST) in Section 4.2, Control Flow Graph (CFG) in Section 4.3, Call Graph (CG) in Section 4.4 and, lastly, Program Dependence Graph (PDG) in Section 4.5.

4.1 Property Graph

Code property graph has as core concept a property graph which is inspired by the graph models used by graph databases such as Neo4j [89], OrientDB [91] and ArangoDB [92]. A property graph can formally be defined as:

**Definition 4.1 (Property Graph).** A property graph $G$ is a directed, edge-label multi-graph that is defined as a tuple $G = (N, E, \rho, \mu)$ where:

- $N$ is a set of nodes.
- $E$ is a set of directed edges.
- $\rho : E \rightarrow (N \times N)$ is a total function that associates each edge in $E$ with a pair of nodes in $N$.
- $\mu : (N \cup E) \times K \rightarrow V$ is a partial function that associates nodes/edges with properties from the set $K$ to value taken from $V$.

![Property Graph Diagram](image)
int addTwo(int a, int b) {
    return a + b;
}

(module
    (func $addTwo
        (param $p0 i32)
        (param $p1 i32)
        (result i32)
        local.get $p1
        local.get $p0
        i32.add))

Figure 4.2: Function addTwo and its WebAssembly’s AST representation.

• λ : E → Σ is a partial function that maps edges to labels. In the following, we model the
  labelling function simply as a dedicated property of all edges with name label; put formally:
  λ(e) = µ(e, label).

Moreover, to our domain, given two nodes (n₁, n₂) such that ρ(e) = (n₁, n₂), we restrict that n₁ ≠ n₂
and n₁ is the source node and n₂ the target node, meaning that a node cannot have an edge to itself.
Additionally, every node has a unique identifier modelled as a property id that is formally defined as:
for all {n₁,...,nᵢ} ∈ N, there is a property key id ∈ K such that µ(nᵢ, id) = vᵢ where vᵢ ≠ ε and for a
µ(nᵢ, id) = µ(nₖ, id) then nᵢ = nₖ. Figure 4.1 shows an example of a property graph with 4 nodes where
the identifiers are A, B, C, D. The edges are labelled with alphabet Σ = {a, b} and nodes A and D are
assigned with property key k ∈ K to a value from V = {x, w}

4.2 Abstract Syntax Tree (AST)

The abstract syntax tree is the first representation upon which we build our CPGs for Wasm and it is
directly produced by the parser. It represents a program as an hierarchical structure comprising its vari-
ous constituents: literals, expressions, statements, functions, among others. It eases the visualisation
of the source code without the need to look through the actual source code, which is much longer, and
may reveal flaws or weakness in code.

Definition 4.2 (AST). An abstract syntax tree Gₐ is a direct tree that can be defined as a property graph
Gₐ = (Nₐ, Eₐ, ρₐ, µₐ) where:

• Nₐ is the set of nodes given by the tree.

• Eₐ is the corresponding tree edges which is associated with the pair of nodes ρₐ according to the
tree produced by the parser.

• µₐ associates each node with the property type that identifies the type of the node and each edge
is identified with a property type with corresponding value AST; put formally: $\forall e \in E_A, \mu(e, \text{type}) = \text{AST}$

- $\lambda_A$ associates each edge with the empty label $\epsilon$.

In the AST, the order of the children nodes are important. The nodes have an unique identifier that are sequentially assigned at time of creation. The AST nodes’ children are sorted by ascending order of id. This is extremely useful to identify arguments in a call instruction for example.

Figure 4.2 depicts a simplified version of the AST of the code on the left which defines function $addTwo$. For clarity, we illustrate AST edges in green and we do not represent node properties. For instance, the node identified by “func: $addTwo” has several properties, such as: id (unique identifier), type that identifies the node as Function, name of the function ($addTwo$), the index of the function in the WebAssembly’s function table, nargs, nlocals, nresults that are respectively the number of arguments, locals and results of the function and, lastly, isImport, isExport that explicitly declares if the function is imported or exported. It also shows the additional instruction return that is not present in the code sample on the left. The return instruction in the code is implicit, however, it is explicit in the AST as it facilitates the sub sequential construction of PDG and the think process for the queries.

Table 4.1 shows all the properties and possible values for the instruction i32.add that is shown in Figure 4.2. As it is illustrated in Figure 4.2, the WebAssembly’s AST has as its root the node Module. This node is of type Module and can have an optional property name, which is the name of the module. The children of the node Module are Function nodes and represent all module’s functions (including imported). Now, each function has two different sub-trees: the signature tree and the instructions tree. The signature sub-tree encodes important information about the function such as: its parameter names and types, its local names and types and its results types. The instruction sub-tree encodes function’s instruction in a composite manner that will be explained next.

The binary encoding and textual format of WebAssembly translates to a sequential set of simple instructions. Consequently, the abstract syntax tree generated from a WebAssembly’s binary will be a flat tree with maximum depth of 3-4 as we can see in Figure 4.3a. Luckily, one of the properties of WebAssembly is being a stack based machine. This property lets us take one simple instruction (like...
Algorithm 1: AST Folding.

**Input:** Sequencial Instructions  
**Result:** Folded Instructions

1. \( ST := \emptyset \);  \( \triangleright \) Stack for instructions (LIFO)
2. \( RS := \emptyset \);  \( \triangleright \) List for instructions (FIFO)
3. \( \text{foreach } inst \in \text{Instructions do} \)
   4. \((\text{nargs}, \text{nresults}) := \text{GetArity}(\text{inst}); \)
   5. \( \text{foreach } i \in \{1...\text{nargs}\} \text{ do} \)
   6. \( \text{child} := \text{Pop}(\text{ST}) \)
   7. \( \text{AddEdge}(\text{inst}.\text{id}, \text{child}.\text{id}, \text{"AST"}); \)
   8. \( \text{end} \)
9. \( \text{if } \text{nresults} = 0 \text{ then} \)
10. \( \text{Append}(\text{inst}, \text{RS}) \)
11. \( \text{else} \)
12. \( \text{Push}(\text{inst}, \text{ST}) \)
13. \( \text{end} \)
14. \( \text{end} \)
15. \( \text{return } \text{RS} ; \)

add) and express as a composition of other simple instructions. Each WebAssembly instruction has an arity. The arity is a tuple \((\text{nargs}, \text{nresults})\) that states the number of arguments a certain instruction consumes from the stack and a number of results it produces back onto the stack. For example, the instruction \(i32.add\) needs two arguments that must be in the stack when this instruction is executed and returns back one value to the stack. We can fold the instructions and represent as a composition of its arguments which is shown in the abstract syntax tree in Figure 4.3b. The Algorithm 1 shows how to compute the folded abstract syntax tree by folding a set of sequential instructions into a set composed instructions, allowing us to transform Figure 4.3a into Figure 4.3b. This additional work is important as it adds expressiveness and facilitates further search of certain patterns as well finding dependencies between instructions.

### 4.3 Control Flow Graph (CFG)

Control flow graph explicitly describes the order by which instructions are executed as well the conditions that are necessary for a particular execution path to be taken. CFGs summarise control information about the program and show how different program units process information between them in the context of the system. Queries over the CFG can easily locate unreachable code, possible wrong paths a program can take and structures such as loops are easy to find and reason about.

**Definition 4.3.** A control flow graph \(G_C\) is a directed graph that can be defined as a property graph \(G_C = (N_C, E_C, \rho_C, \mu_A)\) where:

- \(N_C\) is the set of instruction nodes extended with an additional return node per function, representing its implicit return instruction, and a \textbf{BeginBlock} node per block, representing its beginning.

\[
N_C = \{ n \in N_A : \mu_A(n, \text{type}) \in \{\text{Instructions, Instruction}\} \} \cup \{ n : \mu_C(n, \text{type}) = \text{Instruction} \wedge \mu_C(n, \text{instType}) = \text{BeginBlock} \}
\]
• $E_C$ is the set of edges that encode the execution path of the instructions.

• $\mu_C$ associates each edge is identified with a property $type$ with corresponding value $CFG$; put formally: $\forall e \in E_C, \mu(e, type) = CFG$

• $\lambda_C$ associates each edge with a label from alphabet $\Sigma = \{\epsilon, false, true, default\} \cup \mathbb{N}_0$.

• The entry node is node $Instructions$ and the exit node is the $return$ instruction.

Unlike typical binary analysis where some jumps in control flow graph must be approximated since are statically unknown (relative jumps), WebAssembly has a structured control flow where each target of jump is always explicit. Control flow is mostly linear except in some control instructions.

Edges originated from the control instruction $br_if$ carry the label $true$ or $false$ denoting the value the instruction must evaluate in order for control be transferred to the destination node. If the instruction’s label belongs to a block, the evaluation to true takes the flow to the end of the block. Otherwise it takes to the next instruction. In case the instruction’s label belongs to a loop, the evaluation to true takes the flow the corresponding loop node.

Edges originated from the control instruction $br_table$, carry the label of an integer or $default$ corresponding to the value it receives. This instruction is equivalent to a switch case in C. Edges originated from any other instruction are labelled with $\epsilon$ which indicates an unconditional control flow.

Giving the sequential list of instructions, the construction of the control flow graph is straightforward. Each instruction will have an edge to the next instruction, the exception cases are the control instructions specified above. Figure 4.4 illustrates the control flow graph for the previously defined function $foo()$ in Figure 3.1. In this case, the code was compiled with the preprocessor directive $\text{MAX}$ being defined with value 30. It showcases the usage of the control instruction $br_if$ with the corresponding paths. For visual aid, the control flow edge is coloured red.

Another form of branching is using function calls. In WebAssembly there are two instructions to call a function: $\text{call}$ and $\text{call_indirect}$. These instructions and this type of branch will be integrated in the Call Graph (CG) and will take a special attention in the next section.

4.4 Call Graph (CG)

Call graphs were not considered in the original work on CPGs [1]. They were added to code property graphs by other more recent works [81, 85] to cater for inter-procedural analyses. A call graph is a simple directed graph that connects call nodes (i.e., nodes representing call instructions) to the root nodes of
struct Animal {
  virtual int lives() = 0;
};

struct Cat : Animal {
  int lives() override {
    return 7;
  }
};

struct Dog : Animal {
  int lives() override {
    return 1;
  }
};

// Can be either a Cat or a Dog
int howManyLives(Animal* a) {
  return a->lives();
}

Figure 4.5: Polymorphism in C++.

(module $t0 (func (param i32)
  (result i32)))

(elem $e0 (i32.const 1)
  $Cat::lives__ $Dog::lives__)
(func $Cat::lives__ (type $t0)
  i32.const 7)
(func $Dog::lives__ (type $t0)
  i32.const 1)
(func $howManyLives_Animal*_{
  (type $t0)
  local.get $p0
  local.get $p0
  i32.load
  i32.load
  call_indirect (type $t0))

Figure 4.6: Polymorphism in WebAssembly.

the corresponding functions. This enables the analyst to reason about the control and data flow at an inter-procedural level.

Definition 4.4. A call graph $G_G$ is a directed graph that connects call nodes to the root nodes of the corresponding functions; it can be defined as a property graph $G_G = (N_G, E_G, \rho_G, \mu_G)$ where:

- $N_G$ contains all the function nodes and call nodes of the program’s AST:

  $$N_G = \{ n \in N_A : \mu_A(n, type) = Function \} \cup \{ n \in N_A : \mu_A(n, type) = Instruction \land \mu_A(n, instType) \in \{ call, call_indirect \} \}$$

- $E_G$ is the set of edges that connect the call nodes to their corresponding function nodes.

- $\lambda_G$ associates each edge with the empty label $\epsilon$.

- $\mu_G$ associates each edge in the call graph with the property $type$ with value $CG$, to indicate that the edge belongs to the call graph; put formally: $\forall e \in E_G, \mu(e, type) = CG$.

The construction of the call graph for the instruction call is trivial. It just needs to insert an edge connecting the call node to the Function node that has the same name as the label of the call node. Figure 4.7a illustrates the call graph of the program given in Figure 3.1. Unsurprisingly, the call graph has edges from the calls to functions $source$ and $sink$ to their respective function nodes. For clarity, the call graph’s edges are coloured pink.

Dynamic calls are more difficult to analyse given that the index of the function being called is computed at runtime. However, in order to execute a dynamic call successfully two conditions must be met: 1) the function must be indexed by the function table and 2) the signature of the indexed function must coincide with the one supplied to the call instruction. Hence, the call graph connects every indirect function call to the function nodes of all indexed functions whose signatures coincide with the signature supplied to the call indirect instruction.
In order to better understand how the call graph is constructed when considering indirect calls, let us know analyse the C++ program shown in Figure 4.5, in which a call site may correspond to two different functions. More concretely, the call site at line 15 might call either the method lives of Cat or the method lives of Dog. In Figure 4.6, on the right, we show the stylised compilation of this C++ program to Wasm. In lines 4 and 5, the Wasm program initialises the function table, declaring the starting index to be 1 and adding two functions: $Cat::lives__ and $Dog::lives___. Note that one only needs to add to the function table the functions that are going to be called indirectly. Importantly, both functions share the same signature, which is named $t0 and maps an argument of type i32 to a result of type i32 (lines 2 and 3). In line 14, there is a call_indirect instruction, which states that the function to be called must have signature $t0. From the function table, there are two functions that match this signature. Consequently, in the construction of the call graph, we add connecting this call site to both functions in the function table. Figure 4.7b depicts the generated call graph.

In line 14, there is a call_indirect instruction that states it wants to call a function that has the type signature equal to $t0. From the function table, there are two functions that match this signature. Consequently, to construct the call graph, is inserted an edge for each function in the function table that match the signature provided in the instruction. Figure 4.7b depicts this case with a call graph edge to all possible functions that are allowed to be called: $Cat::lives__ and $Dog::lives___.

### 4.5 Program Dependence Graph (PDG)

Program dependence graphs (PDGs) explicitly represent data and control flow dependencies between the instructions of a program and are constructed by traversing its control flow graph. The original paper on CPGs makes use of PDGs constructed using a standard reaching definitions analysis [93] for computing data dependencies. In a nutshell, an instruction $inst_2 \in I$ data-depends on another instruction $inst_1 \in I$, if $inst_2$ uses a variable variable defined by $inst_1$. Here, we enrich the information present on the PDG of the analysed program with the result of a constant and function call propagation analyses. To this end, we instrument PDG edges with additional information pertaining to the type of dependency being expressed; for instance, data dependencies on variables with a constant value are represented by edges annotated with that constant value. Below we give the formal definition of PDGs as property graphs.

**Definition 4.5.** A program dependence graph $G_P$ is a directed graph whose edges encode data and
control dependencies and can be defined as a property graph $G_P = (N_P, E_P, \rho_P, \mu_P)$ where:

- $N_P$ is the set of nodes that contains all nodes from the CFG: $N_P = N_C$.
- $E_P$ is the set of edges that encode the dependencies between instructions.
- $\lambda_P$ maps each edge to a label taken from the set:

$$\Sigma_P = \{\text{false, true, default}\} \cup \mathbb{R} \cup \text{FunctionName} \cup \text{VarNames}$$

Edges annotated with a value in the set \{false, true, default\} represent control-dependencies. Edges annotated with a constant number in $\mathbb{R}$ represent data-dependencies on constant values. Edges annotated with function names in the set FunctionName represent data-dependencies on the return value of a function. And, finally, edges annotated with local or global variable names represent the standard data dependencies on variable definitions.

- $\mu_P$ associates each edge in the program dependence graph with the properties type and pdgType, the former indicating that the edge is a PDG edge and the latter indicating the type of dependency. Formally, the property pdgType takes its values from the set:

$$\Sigma_T = \{\text{Global, Local, Const, Control, Function}\}$$

whereas the property type has value PDG for all PDG edges, put formally: $\forall e \in E_P, \mu(e, \text{type}) = \text{PDG}$. Constant PDG edges have two additional properties: valueType and value, respectively indicating the type and value of the constant; valueType takes values from $\Sigma_V = \{\text{i32, i64, f32, f64}\}$, whereas value takes values from $\mathbb{R}$. 
All the information about the dependencies will be held in the PDG edges. Table 4.2 summarises the properties of a PDG edge which can later be queried when searching for patterns.

### 4.5.1 Constructing the PDG

In order to compute the PDG, we follow the so called monotone framework [93], associating both local and global variables with *abstract values* that capture their corresponding data dependencies. Accordingly, concrete states are lifted to abstract states, containing the dependencies of local and global variables as well as of the elements of the stack. Finally, we define the behaviour of each Wasm instruction with respect to abstract states, associating each Wasm instruction with a transfer function that describes how that instruction propagates data dependencies. We represent data dependencies as triples consisting of:

- a *type* field, recording the type of data dependency. We consider four types of data dependencies: (1) the *global* type describes dependencies on global variables, (2) the *local* type describes dependencies on local variables, (3) the *function* type describes dependencies on results of function calls, and (4) the *const* type describes dependencies on constant values.

- an *id* field, recording the unique identifier of the node that generated the data dependency. Data dependencies are introduced via instructions corresponding to variable definitions or instructions that push new values onto the stack; a data dependency keeps track of the id of the instruction that generated it.

- a *value* field, recording additional information pertaining to the type of data dependency. Global and local data dependencies require the name of the respective variables; function data dependencies require the name of the function; and constant data dependencies require the type and value of the constant.

To better understand how dependencies are modelled, let us consider the Wasm program given in Figure 4.8. Each node’s identifier is displayed on its upper-left corner. Let us consider the dependencies of the top of the stack after the execution of line 8. In this case, the top of the stack depends on the value of the local variable \(y\). This dependency is represented as the tuple \((\text{Local}, 4, y)\). Analogously, after the execution of line 9, the top of the stack depends on a constant value 2 and the dependency is represented with the tuple \((\text{Const}, 5, (\text{i32}, 2))\).

**Definition 4.6 (Abstract State).** An abstract state \(\sigma \in \mathcal{S}\) is a tuple \((g, l, st, lb)\) where:

- \(g : G \to \mathcal{P}(D)\) is a total function that associates each global variable with a set of abstract values\(^1\).

- \(l : F \times L \to \mathcal{P}(D)\) is a partial function that maps pairs of function identifiers and local variables to sets of abstract values from \(\mathcal{P}(D)\).

- \(st\) is a list of sets of abstract values.

\(^1\)We use \(\mathcal{P}(D)\) to denote the set of all subsets of \(D\).
For instance, if we have $F =$ \{$\mathtt{tes}t\}$, $G =$ \{$\mathtt{g}0\}$, $L =$ \{$\mathtt{p}0, \mathtt{p}1\}$, the global dependency $g(\mathtt{g}0) =$ \{\{Const, 20, (i32, 3)\}\} means that $\mathtt{g}0$ has constant value 3 and was assigned in node 20, whereas the local dependency $l(\mathtt{tes}t, \mathtt{p}1) =$ (Local, 3, $\mathtt{p}0$) means that the local variable $\mathtt{p}1$ of function $\mathtt{tes}t$ depends on the value of $\mathtt{p}0$ because, in instruction 3, the value assigned to $\mathtt{p}1$ was calculated using $\mathtt{p}0$.

To calculate the data dependencies at each execution point, we traverse the CFG of the program to be analysed, propagating dependencies in a forward manner. More concretely, we define, for each Wasm 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, a general transfer function $\mathcal{T} : I \times S \rightarrow S$ that computes an output abstract state given an instruction in $I$ and an input abstract state. The function $\mathcal{T}$ is defined by rules in Figure 4.3.

We will now illustrate how the transfer functions are used to propagate dependencies through the

| $T(t_\text{const} \ c, (g, l, st, lb))$ | $\leftrightarrow$ | $(g, l, st : (\{\text{Const}, \text{id}, (t, c)\}), lb)$ |
| $T(t_\text{unop}, (g, l, st : v_0, lb))$ | $\leftrightarrow$ | $(g, l, st : v_0, lb)$ |
| $T(t_\text{binop}, (g, l, st : v_0 : v_1, lb))$ | $\leftrightarrow$ | $(g, l, st : v_0 \cup v_1, lb)$ |
| $T(t_\text{relop}, (g, l, st : v_0 : v_1, lb))$ | $\leftrightarrow$ | $(g, l, st : v_0 \cup v_1, lb)$ |
| $T(t_\text{cvtop}_t \_\text{sx}, (g, l, st : v_0, lb))$ | $\leftrightarrow$ | $(g, l, st : v_0, lb)$ |
| $T(\text{drop}, (g, l, st : v_0, lb))$ | $\leftrightarrow$ | $(g, l, st : v_0, lb)$ |
| $T(\text{select}, (g, l, st : v_0 : v_1 : v_2, lb))$ | $\leftrightarrow$ | $(g, l, st : v_0 \cup v_1, lb)$ |
| $T(\text{local} \_\text{get} \ x, (g, l, st, lb))$ | $\leftrightarrow$ | $(g, l, st : l(f, x), lb)$ |
| $T(\text{local} \_\text{set} \ x, (g, l, st : v_0, lb))$ | $\leftrightarrow$ | $(g, l[x \mapsto v_0], st, lb)$ |
| $T(\text{local} \_\text{tee} \ x, (g, l, st : v_0, lb))$ | $\leftrightarrow$ | $(g, l[x \mapsto v_0], st : v_0, lb)$ |
| $T(\text{global} \_\text{get} \ x, (g, l, st, lb))$ | $\leftrightarrow$ | $(g, l, st : g(x), lb)$ |
| $T(\text{global} \_\text{set} \ x, (g, l, st : v_0, lb))$ | $\leftrightarrow$ | $(g[x \mapsto v_0], l, st, lb)$ |
| $T(\text{t} \_\text{load}, (g, l, st : v_0, lb))$ | $\leftrightarrow$ | $(g, l, st, lb)$ |
| $T(\text{t} \_\text{store}, (g, l, st : v_0 : v_1, lb))$ | $\leftrightarrow$ | $(g, l, st, lb)$ |
| $T(\text{memory} \_\text{size}, (g, l, st, lb))$ | $\leftrightarrow$ | $(g, l, st : \{\}, lb)$ |
| $T(\text{memory} \_\text{grow}, (g, l, st : v_0, lb))$ | $\leftrightarrow$ | $(g, l, st : \{\}, lb)$ |
| $T(\text{nop} \mid \text{br} \ b \mid \text{return}, (g, l, st, lb))$ | $\leftrightarrow$ | $(g, l, st, lb)$ |
| $T(\text{unreachable}, (g, l, st, lb))$ | $\leftrightarrow$ | $(g, l, st, lb)$ |
| $T(\text{begin} \_\text{Block} \ b, (g, l, st, lb))$ | $\leftrightarrow$ | $(g, l, st, lb : b)$ |
| $T(\text{end} \_\text{Block} \ b, (g, l, st : v_0 : v_1 : v_2, lb : b : b_k))$ | $\leftrightarrow$ | $(g, l, st : v_0 : v_2, lb)$ |
| $T(\text{loop} \ b, (g, l, st, lb))$ | $\leftrightarrow$ | $(g, l, st, lb : b)$ |
| $T(\text{if}, (g, l, st : v_0, lb))$ | $\leftrightarrow$ | $(g, l, st, lb)$ |
| $T(\text{br} \_\text{if} \ b, (g, l, st : v_0, lb))$ | $\leftrightarrow$ | $(g, l, st, lb)$ |
| $T(\text{br} \_\text{table} \ b^x \ b_k, (g, l, st : v_0, lb))$ | $\leftrightarrow$ | $(g, l, st, lb)$ |
| $T(\text{call} \ x, (g, l, st : a_0 : a_1, lb))$ | $\leftrightarrow$ | $(g, l, st : v_j^x, lb)$ |
| $T(\text{call} \_\text{indirect} \ t, (g, l, st : a_0 : a_1 : c, lb))$ | $\leftrightarrow$ | $(g, l, st : v_j^t, lb)$ |

Table 4.3: Transfer Functions.

• lb is a list of block labels representing the enclosing blocks.
Algorithm 2: PDG data flow generation for function $f$.

Input: Entry Node: Node Instructions of function $f$
Result: Map (instruction, rdef) for every instruction.

1. $DEP = \{\}$
2. $g$ ; $\triangleright$ Map for global definitions.
3. $l$ ; $\triangleright$ Map for local definitions.
4. $st := \emptyset$ ; $\triangleright$ Lists of sets of definitions.
5. $lb := \emptyset$ ; $\triangleright$ Lists of labels.
6. $ST = \{(\text{Insts}, DEP)\}$ ; $\triangleright$ Stack (node, rdef) (LIFO)
7. $res = \emptyset$ ; $\triangleright$ Map (instruction, inDep) to return.
8. while $ST$ not Empty do
9. 10. $(\text{inst}, \text{inDep}) := \text{Pop}(ST)$;
11. 12. $DEP := T(\text{inst}, \text{inDep})$;
13. if $\text{instType} = \text{Loop}$ and $(DEP \cup \text{res}[\text{inst}]) = \text{res}[\text{inst}]$ then
14. 15. continue;
16. else
17. 18. $\text{res}[\text{inst}] := DEP$;
19. foreach $\text{child} \in \text{children}(\text{inst}, CFG)$ do
20. 21. $\text{Push}(ST, (\text{child}, RD))$;
22. end
23. end
24. end
25. $\text{return } res$;

CFG in Figure 4.8 by appealing to three simple examples:

- The instruction with identifier 4, $\text{local.get } y$, pushes the value of the local variable $y$ onto the stack. Likewise, the transfer function $T$ pushes the abstract-value set describing the dependencies of $y$ onto the stack. Given that the dependencies of $y$ have not been updated since the entry point of the function, we simply push the dependency $\{(\text{Local}, 4, y)\}$ onto the stack. If the program contained a set instruction to update the value of $y$, say $\text{local.set } y$, the corresponding transfer function would update the dependencies of $y$ accordingly. More concretely, given the input abstract state $(g, l, st :: v_0, lb)$, we would obtain the output abstract state $(g, [(test, y) \to v_0], st, lb)$, which coincides with the input state except that it pops out the top of the stack, assigning it to the corresponding local variable.

- The instruction with identifier 5, $\text{i32.const } 2$, pushes the value 2 of type i32 onto the stack. The transfer function $T$ extends the input stack $st$ with a constant dependency $\{(\text{Const}, 5, (\text{i32}, 2))\}$, leaving the other components of the abstract state unchanged. In the rule for constants in Figure 4.8, we use “::” to denote list concatenation.

- The instruction with identifier 6, $\text{i32.add}$, takes two i32 integers from the stack and pushes their sum onto the stack. Analogously, the transfer function $T$ also takes the two top abstract-value sets from the stack and push back their union. However, we do not propagate constant dependencies if the cardinality of the resulting union is greater than one. In the case of the example, we therefore obtain the dependency: $\{(\text{Local}, 6, y)\}$.

Algorithm 2 shows the pseudo-code for calculating the data dependencies of a given Wasm function given its entry point node and CFG. More concretely, the goal of the algorithm is to compute a map
unop, N := \texttt{c tz} | \texttt{ctz} | \texttt{popcnt}
unop, N := \texttt{neg} | \texttt{abs} | \texttt{ceil} | \texttt{floor} | \texttt{trunc} | \texttt{nearest} | \texttt{sqrt}
binop, N := \texttt{add} | \texttt{sub} | \texttt{mul} | \texttt{div} | \texttt{rem} | \texttt{sx} | \texttt{xor} | \texttt{shr} | \texttt{shr, sx} | \texttt{rotl} | \texttt{rotr}
binop, N := \texttt{add} | \texttt{sub} | \texttt{mul} | \texttt{div} | \texttt{min} | \texttt{max} | \texttt{copysign}
testop, N := \texttt{eq} | \texttt{ne} | \texttt{lt} | \texttt{le} | \texttt{gt} | \texttt{ge} | \texttt{select}
cutop, N := \texttt{extend} | \texttt{trim} | \texttt{convert} | \texttt{demote} | \texttt{promote} | \texttt{reinterpret}
sx := \$ u

(res), assigning each node in the CFG to the abstract state that describes the dependencies after the execution of the corresponding instruction.

The initial abstract state is set in lines 1-6; it contains the initial dependencies: \(DEP = (g_0, l_0, 0, 0)\), with \(g_0\) and \(l_0\) simply stating that each global/local variable depends on itself. Put formally, \(g_0\) maps each global variable \(g_i\) to the dependency \(\{ (\text{Global}, \text{id}_{DEP}, g_i) \}\), where \(\text{id}_{DEP}\) is the identifier of the entry point node. Likewise, \(l_0\) maps each local variable \(s_1\) to the dependency \(\{ (\text{Local}, \text{id}_{DEP}, s_1) \}\).

After computing the initial abstract state (lines 1-6), we perform a modified DFS [94] on the given CFG starting from the supplied entry point node (lines 7-20). Our DFS stack, \(ST\), pairs up the instructions to be visited with the input abstract state that describes the dependencies immediately before their execution. For each visited instruction, we proceed as follows:

- we apply the corresponding transfer function to the input abstract state (line 11); and
- we pair up the output abstract state with all the successors of the current instruction and push the obtained pairs onto the DFS stack (lines 16-18).

Loop instructions are only re-visited if their input dependencies change. Note that we guaranteed not to loop forever because the sets \(F, I, G\) and \(L\) are finite, meaning that, in the worst case, a loop might depend on all function calls, constant instructions, global and local variables.

Given the computed dependencies, it is trivial to generate the PDG. We simply have to connect each node to all the nodes on which it depends by inspecting the \textit{id} field of its corresponding dependencies. Each edges is then labelled with the information provided by the fields \textit{type} and \textit{value} of the corresponding dependency. Figure 4.8 gives a complete example of a Wasm function and its corresponding PDG. Besides data dependencies, the presented PDG also contains control dependencies. The calculation of control dependencies is, however, trivial to perform given that Wasm has structured control flow. More specifically, in WebAssembly there are only three instructions that can branch: \texttt{br_if}, \texttt{br_table} and \texttt{if}. Given the CFG, we create control-PDG edges from these instructions to all instructions of their

![Figure 4.9: WebAssembly abstract syntax.](image-url)
corresponding branches. For instance, in Figure 4.8, we have a control-PDG edge connecting the if instruction with id 3 to the instruction with ids 4, 5, and 6.

4.6 Code Property Graph (CPG)

The code property graph is the combination of all previously defined graphs: AST, CFG, CG and PDG. Its definition is the simple union of all those graphs.

Definition 4.7. A code property graph $G$ is a directed, multi-graph that can be defined as a property graph $G = (N, E, \rho, \mu)$ where:

- $N = N_A \cup N_C$
- $E = E_A \cup E_C \cup E_G \cup E_P$
- $\rho = \rho_A \cup \rho_C \cup \rho_G \cup \rho_P$
- $\mu = \mu_A \cup \mu_C \cup \mu_G \cup \mu_P$

Figure 4.9 presents WebAssembly’s syntax tree defined in WebAssembly’s official specification [95], fully supported by Wasmapi. We now define all CPG nodes and edges from $N$ and $E$ respectively. Each of the following tables represents a node or a group of nodes that share the same set of key properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>AST</td>
</tr>
<tr>
<td></td>
<td>(a) AST edge.</td>
</tr>
<tr>
<td>type</td>
<td>CG</td>
</tr>
<tr>
<td></td>
<td>(b) CG edge.</td>
</tr>
<tr>
<td>label</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td></td>
<td>(c) CFG edge.</td>
</tr>
<tr>
<td>pdgType</td>
<td>$\Sigma_P$</td>
</tr>
<tr>
<td>valueType</td>
<td>$\Sigma_T$</td>
</tr>
<tr>
<td>value</td>
<td>$\Sigma_V$</td>
</tr>
</tbody>
</table>

Table 4.4: Edges.

$\Sigma = \{\text{false, true, default}\} \cup \mathbb{N}_0$

$\Sigma_P = \Sigma \cup \text{FunctionName}$

$\Sigma_T = \{\text{Global, Local, Const, Control, Function}\}$

$\Sigma_V = \{\text{i32, i64, f32, f64}\}$
Table 4.5: Function node.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>N₀</td>
</tr>
<tr>
<td>type</td>
<td>Module</td>
</tr>
<tr>
<td>name</td>
<td>[string]</td>
</tr>
<tr>
<td>index</td>
<td>N₀</td>
</tr>
<tr>
<td>nargs</td>
<td>N₀</td>
</tr>
<tr>
<td>nlocals</td>
<td>N₀</td>
</tr>
<tr>
<td>nresults</td>
<td>N₀</td>
</tr>
<tr>
<td>isImport</td>
<td>{false,true}</td>
</tr>
<tr>
<td>isExport</td>
<td>{false,true}</td>
</tr>
</tbody>
</table>

Table 4.6: Module node.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>N₀</td>
</tr>
<tr>
<td>type</td>
<td>Module</td>
</tr>
<tr>
<td>name</td>
<td>[string]</td>
</tr>
</tbody>
</table>

Table 4.7: Simple instruction node.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>N₀</td>
</tr>
<tr>
<td>type</td>
<td>Instruction</td>
</tr>
<tr>
<td>instType</td>
<td>Σₜ</td>
</tr>
<tr>
<td>label</td>
<td>{string}</td>
</tr>
<tr>
<td>nresults</td>
<td>N₀</td>
</tr>
</tbody>
</table>

Table 4.8: Simple node.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>N₀</td>
</tr>
<tr>
<td>type</td>
<td>Instruction</td>
</tr>
<tr>
<td>instType</td>
<td>Block</td>
</tr>
<tr>
<td>label</td>
<td>{string}</td>
</tr>
<tr>
<td>value</td>
<td>R</td>
</tr>
</tbody>
</table>

Table 4.9: Constant node.

Σₜ = {FunctionSignature, Parameters, Locals, Results, Else, Trap, Start}.

Σₛ = {Nop, Unreachable, Return, BrTable, Drop, Select, MemorySize, MemoryGrow, CallIndirect}.

Table 4.10: Labelled node.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>N₀</td>
</tr>
<tr>
<td>type</td>
<td>Instruction</td>
</tr>
<tr>
<td>instType</td>
<td>Σₜ</td>
</tr>
<tr>
<td>label</td>
<td>{string}</td>
</tr>
</tbody>
</table>

Table 4.11: Block node.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>N₀</td>
</tr>
<tr>
<td>type</td>
<td>Instruction</td>
</tr>
<tr>
<td>instType</td>
<td>If</td>
</tr>
<tr>
<td>label</td>
<td>{string}</td>
</tr>
<tr>
<td>hasElse</td>
<td>{false,true}</td>
</tr>
</tbody>
</table>

Table 4.12: If node.

Σₜ = {Br, BrIf, GlobalGet, GlobalSet, LocalGet, LocalSet, LocalTee, Call, BeginBlock}.

Table 4.13: Load node.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>N₀</td>
</tr>
<tr>
<td>type</td>
<td>Instruction</td>
</tr>
<tr>
<td>instType</td>
<td>Binary</td>
</tr>
<tr>
<td>opcode</td>
<td>{binop}</td>
</tr>
</tbody>
</table>

(a) Binary node.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>N₀</td>
</tr>
<tr>
<td>type</td>
<td>Instruction</td>
</tr>
<tr>
<td>instType</td>
<td>Compare</td>
</tr>
<tr>
<td>opcode</td>
<td>{relop}</td>
</tr>
</tbody>
</table>

(b) Compare node.

Table 4.14: Store node.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>N₀</td>
</tr>
<tr>
<td>type</td>
<td>Instruction</td>
</tr>
<tr>
<td>instType</td>
<td>Convert</td>
</tr>
<tr>
<td>opcode</td>
<td>{cutop}</td>
</tr>
</tbody>
</table>

(a) Convert node.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>N₀</td>
</tr>
<tr>
<td>type</td>
<td>Instruction</td>
</tr>
<tr>
<td>instType</td>
<td>Unary</td>
</tr>
<tr>
<td>opcode</td>
<td>{unop}</td>
</tr>
</tbody>
</table>

(b) Unary node.

Table 4.15: Opcode instructions.
Summary

In this chapter we formalised the main components of the code property graphs and its construction. We started by the main concept of property graph. Next, we brought the abstract syntax tree to the domain of a property graph and explained how to generate folded instructions for a sequential list. We exemplified the control flow graph and the intricacies of static and dynamic calls in WebAssembly to generate the call graph. In the end, we defined the concepts of abstract value, abstract state and transfer functions in order to produce a data flow analysis to construct the program dependence graph. In the next chapter we will define traversals and study patterns to find vulnerabilities in WebAssembly code.
Chapter 5

Queries

With the code property graph, we attempt to transform security flaws present in WebAssembly’s binary or its text format into graph patterns. A query is an attempt to find an explicit pattern within the code property graph, that is, a subset of nodes or edges that have certain properties. For instance, a simple query could be to find every node that is an instruction call to `printf` and is called in function `f`.

In this chapter we will look closely into our research question RQ3 pertaining to the modelling of security vulnerabilities as CPG queries. We will provide various examples of CPG queries aimed at detecting classic C vulnerabilities in Wasm code, including format strings, use of inherently dangerous functions, use after free, double free, taint vulnerabilities and buffer overflows.

5.1 Format Strings

Format string vulnerabilities refer to the format parameter used by the `printf` function family in the standard library of C and many others programming languages. The string, usually given as first parameter of such functions, is written in a simple template language where certain characters preceded by `%` are called format specifiers. Format specifiers indicate the location and method to show a piece of data, such as numbers and strings. Wrong syntax or invalid conversion specifications lead to undefined behaviour, and can cause program termination, leak of sensitive information, write in arbitrary memory and/or hijack of the program control flow.

Usually, the cause of this kind of vulnerabilities is the misuse of these functions, with mistakes such as: 1) the template syntax is not a constant string and is controlled by the input, 2) wrong template syntax, 3) there are too few function arguments to supply the specification present in the template or 4) the types of the argument are different from those specified in the template. Figure 5.1 exemplified the first case and the most common case, the call to `printf` does not specify a constant format and is

```c
int main(int argc, const char **argv) {
    char bof[] = "AAAA";
    printf(argv[1]);
    return 0;
}
```

Figure 5.1: Format strings.
dependent on user input (program argument). A user can provide a crafted format to crash the program, leak memory and write in arbitrary memory. By considering the C code listing shown in Figure 5.1 as a running example. For the sake of illustration, it suffers from a trivial format string vulnerability. The buffer bof and argv[1] is stored in the stack. The printf function uses the stack to supply arguments to the format template.

**Crash the program:** The simplest format string attack is to force the program to crash which is useful to provoke denial of service. By providing the format string with a certain number of %s we can easily trigger some invalid pointer access resulting in a trap in WebAssembly. The printf starts to iterate over the stack considering each memory position as a pointer to a string to be print and starts printing from the given pointer until it finds a null value. If a value in the stack pointer is larger than the size allocated, printf will try to access this pointer and trigger a trap since that address does not exist.

**Write in arbitrary memory:** The format specifier %n writes in the specified location the number of characters written until %. The string argument argv[1] in Figure 5.1 will be placed in the stack at the address m. The idea is to provide the address in the template, for instance, to write in address 0x08480110 we encode it in little endian (since WebAssembly stores data that way) “\x10\x01\x48\x08” and provide the argument ”\x10\x01\x48\x08_%x%x%x%x%x_%n_” to printf, resulting in a write of value 11 to the address 0x08480110. The format specifier %x will print the contents from the memory positions in the stack until it reaches the memory position m, which represents the beginning of the exploit payload (address). The specifier %n will take as parameter the address 0x08480110 and write on it he number of characters printed (which is 11). In WebAssembly this can also alter the flow of the program if there are control instructions depending on the data stored in the linear memory.

**Memory Leak:** We can trigger a memory leak anywhere. In the stack, in example shown in Figure 5.1, by providing ”%x%x%x%x%x” will result in the leak of the contents stored in bof that could be sensitive information like passwords or cryptographic keys. It is also possible to craft a format string template to read memory from any address using %. With the previous payload, it is possible to build a chain of format to change the position in memory where format specifier %s uses to read, to point to any address of memory and dump its contents until it finds a null byte.
Query 1: Format strings.

```java
1 foreach func ∈ prog do
2     nodes := [ n | n ∈ func.instructions and n.instType = call and n.label = $printf];
3 foreach node ∈ nodes do
4     child := child(node, 1, AST);
5     if PDGEdge(child, node, Const) = nil then
6         vulnerability
7     end
8 end
9 end
```

Query 2: Dangerous functions.

```java
1 foreach func ∈ prog do
2     nodes := [ n | n ∈ func.instructions and n.instType = call and n.label ∈ DANGEROUS];
3     if nodes not Empty then
4         vulnerability
5     end
6 end
```

5.1.1 Pattern

As stated previously, the most common cause of format string vulnerabilities is the lack of a constant format template in a call to a `printf`-like function. We can traverse the AST to search for calls to these functions, checking if the argument that is supposed to hold the format template is dependent on a constant or not.

Figure 5.2a shows a fragment of a CPG representing a call to `printf` with a constant format template. The first argument from `printf` is the instruction `i32.const 1024` and contains a constant PDG edge with value `1024`. In contrast, Figure 5.2b shows a call to `printf` from line 3 of the code in Figure 5.1. We can see that the first argument is a composed instruction `i32.load` that takes the value from the function’s parameter `$p1`. In this case no PDG edge exists from the first argument node (`i32.load`) to the call node.

Query 1 traverses the CPG of a Wasm program to find format string vulnerabilities, such as those described above. For each function in the program, it first obtains all nodes corresponding to calls to the `printf` function by checking the properties `instType` and `label`. Then, for each call node, it takes the first child of that node in the AST graph and verifies if there is a constant PDG edge to the call node. If such an edge does not exist, a vulnerability is flagged.

5.2 Dangerous Functions

There are certain functions in C that behave dangerously regardless of the context in which they are executed. There are never guarantees that such calls to those functions will work safely. These type of functions were implemented without security concerns. For instance, the function `gets()` is a dangerous function since it does not perform any boundary on the number of characters read to the buffer given as an argument. An attacker can easily trigger a buffer overflow by sending an arbitrarily-sized input.
There are many other functions that can be considered dangerous. A group of unsafe functions that may lead to buffer overflows are: `gets()`, `sprintf()`, `strcat()`, `strcpy()`, `strncpy()` and `vprintf()`. Other group of unsafe functions are those which can result in undefined behaviour in case of overflow, which include: `atoi`, `atof`, and `atol`.

The look for these security flaws in code property graphs is straightforward and it is showcased in Query 2. For each function in the program, we simply obtain the set of nodes corresponding to calls to dangerous functions. If such set is different from empty, there is a vulnerability.

### 5.3 Use After Free and Double Free

Use after free occurs when one uses a reference to a memory location that has already been freed. Similarly, double free occurs when one frees a reference that has already been freed. The misuse of heap allocated memory after it has been freed or deleted leads to undefined system behaviour, crash of the program and, in many cases, to a write-what-where condition.

Like double free errors and memory leaks, use after free errors have two common and sometimes overlapping causes: 1) error conditions / other exceptional circumstances and 2) confusion over which part of the program is responsible for freeing the memory.

The use of previously freed memory can have various adverse consequences; it can, for instance, compromise the integrity and/or availability of the system by causing it to crash and it can also lead to the corruption of valid data when the memory area in question has already been re-allocated.
In Figure 5.3, when err is true, the pointer is immediately freed. However, this pointer is later incorrectly used in the logError() function since abrt is set to true, causing the then-branch of the if statement in line 9 to be taken.

The search for both types of vulnerabilities can be achieved by traversing the control flow graph. Our goal is to write a query that finds a control flow path from the instruction malloc() in line 2 to both lines 6 and 10. Query 3 depicts the query’s pseudo-code. We iterate over each function in the program and filter all nodes that are calls to $malloc. Then, for each call to $malloc, we retrieve its control flow graph descendants, further filtering these nodes to only obtain the calls to $free that depend on the specific call to $malloc (by imposing that reachesPDG(callMalloc, n, Function, $malloc)). Then, for each call node to $free, we query its CFG descendants for nodes which depend on the initial $malloc. If the obtained set of nodes is different from empty, a vulnerability can be flagged, since the query found a use after free.

The query to find double free vulnerabilities is similar to the one presented for use after free vulnerabilities, except that we would just need to additionally filter the obtained nodes in line 6 (uafs) in order to keep only those corresponding to calls to $free.

### 5.4 Taint-style Vulnerabilities

The term taint-style vulnerabilities has its roots in taint analysis, a technique for tracing the propagation of data through a program. One goal of taint analysis is to identify data flows from attacker-controlled sources to security-sensitive sinks that do not undergo sanitisation. This procedure requires the definition of 1) appropriate sources, 2) corresponding sinks and 3) sanitisation rules. While at first, it may seem that only a handful of flaws can be described in this way, this vulnerability class fits many common security defects well, including different types of buffer overflows and other memory corruption flaws, such as SQL, XSS and command injection, as well as missing authorisation checks.

Figure 5.4 shows an example of an integer overflow that chains into an arbitrary code execution in an older version of OpenSSH 3.3 with assigned CVE-2002-0639 [96]. The variable nresp is assigned with a value retrieved from the network by the function packet_get_int(). The value is then used in function xmalloc() allocating the necessary memory. We can say that a sensitive value (source) reaches a possible dangerous function (sink) that can trigger any type of other vulnerabilities, from buffer overflows to command injection. In this example, sensitive data does not undergo validation. The
value retrieved from the network can be high enough that when multiplying by the size of the pointer (in case of WebAssembly is 4), will trigger an integer overflow and, as consequence, allocate less memory than it is expected leading to a heap buffer overflow which may, in turn, be used by an attacker to execute arbitrary code.

The sanitisation of payloads is not trivial and difficult to analyse without deep knowledge of the code base. To describe taint-style vulnerabilities in enough detail and to search for their occurrence in software, it is necessary to inspect how information propagates from one statement to another as well as how this flow is controlled by conditions.

We can construct a query for finding simply taint-style vulnerabilities by using exclusively the PDG. In Query 4, for each function, we query its instructions for nodes that are calls to functions in the set SINKS. In this case $xmalloc$ is a sink and is present in the set SINKS. Next, for each sink call, we verify if there is an incoming function PDG edge from a source function, i.e. the sink call depends on a value returned by a source function in the set SOURCES. If such dependency exists, we flag a vulnerability.

Neither a source nor a sink need to be a function. A source can be a function parameter and a sink could be any instruction or a specific memory location. For instance, in Figure 5.7, we show a code snippet that was compiled using Emscripten. We can see that the parameter $p2$ is user-controlled when called from the code in Figure 5.6. If $p2$ has more than 32 bytes, there will be a buffer overflow. Since Emscripten arranges the buffers $bof1$ and $bof2$ contiguously in linear memory (with $bof2$ before $bof1$ and possibly some bytes between them due to alignment constraints), we will have that parts of $bof1$ (containing HTML code) will be corrupted by the data contained in $p2$. The string $bof1$ that was assumed to be static can be overwritten by the user input. When line 7 is executed, the string saved in $bof1$ will be written to the DOM, allowing a possible XSS. The code generated by Emscripten in lines 6 to 8, is a call to an external JavaScript function. If we take external functions as sinks, we can tweak Query 4 in line 4, to look for a local PDG edge that is dependent on parameter $p2$.  

---

// [...]  
nresp = packet_get_int();  
if (nresp > 0) {  
    response =  
        xmalloc(nresp*sizeof(char*));  
    for (i = 0; i < nresp; i++)  
        response[i] =  
            packet_get_string(NULL);  
}  
// [...]  

Figure 5.4: Integer overflow OpenSSH 3.3.
```c
extern void bof(char *p1, char *p2) {
    char bof1[32];
    char bof2[32];
    strcpy(bof1, p1);
    strcpy(bof2, p2);
    ES_ASM({
        document.getElementById("XSS").innerHTML = (Pointer_stringify($0, $1));
    }, bof1, strlen(bof1));
}
```

Figure 5.7: XSS activation from a buffer overflow.

The parameters from `main` are always tainted. We can create another query to recursively propagate tainted variables through the use of call graph edges. Figure 5.5 shows some important edges to consider. In function `main`, both parameters `$p0` and `$p1` are tainted. The first argument in the call to `bof` depends on the global variable `$g0` and, in the second argument, the load instruction depends on the `main` local variable `$p1`. We can propagate the dependency of `$p1` in the second argument and state that the second parameter of `bof (p2)` is tainted. The vulnerability is flagged when the parameter `$p2` reaches the argument of external JavaScript functions (lines 6-8 from Figure 5.7).

### 5.5 Buffer Overflow

Before we dive into the definition of buffer overflow and possible attack surface, we need first, to grasp the concept of linear memory layout and how compilers arrange it.

#### 5.5.1 Memory Layout

Despite the non-existence of sections in WebAssembly’s linear memory, compilers of C/C++ Emscripten and Clang (WASI), divide the linear memory in three main regions: data, stack and heap. The data section is used to save static constant data, zero-initialised data and global data structures/arrays. The stack is used to save function’s local variables with unclear static scope such as arrays. Finally, the heap, is where dynamic allocated memory will be placed, that is, the example call to `malloc` will return memory from the heap section. To avoid confusion, from hereafter, we will refer to the stack as the stack created by the compiler in linear memory to distinguish from the protected evaluation stack, which is a structure managed by the WebAssembly’s runtime and explained in Section 2.3.

The sections are arranged in linear memory depending in the compiler that generates the code and the flags given to the compiler. However, the heap must be placed at the end of the linear memory,
so it can grow towards higher addresses. The Emscripten and Clang use LLVM, which recently has gained its own in-tree WebAssembly back-end, and follow the same layout as x86 and ARM where stack grows towards lower addresses. Figure 5.8 shows how the different sections are arranged depending of the compiler and the flags given as argument. Prior to Emscripten 1.39.7, the stack grew upwards, towards the high addresses and consequently, the heap. This version is deprecated since October of 2019. Figure 5.8a is the most common and the figure 5.8b is obtained in Clang 9 (WASI) by providing the flag --stack-first at compile time.

Depending in the layouts generated, is also possible to see what are the consequences in write beyond the limits of a buffer in the stack. In both cases, a large enough write can reach and alter the heap. But, in the second case, a overflow can potentially override the data section that contains constant data like strings and values that should never be altered in the course of the program execution.

### 5.5.2 Definition and Attack Surface

```c
int main(int argc, char** argv) {
    char buffer[64];
    // Here is the buffer overflow.
    read(STDIN_FILENO, buffer, 128);
    return 1;
}
```

Figure 5.9: Stack buffer overflow.

```c
char buffer[64];
int main(int argc, char** argv) {
    // Here is the buffer overflow.
    read(STDIN_FILENO, buffer, 128);
    return 1;
}
```

Figure 5.10: Data buffer overflow.

Buffer overflow is an old vulnerability that has been widely exploited over the years. Many mitigation techniques are in place to avoid buffers be overrun. One technique that modern compilers apply is the use stack guards, commonly known by stack canaries or stack cookies above local data. The exploitation range between corruption of data and hijack-control of the program.

In WebAssembly, it is impossible to overtake program’s control by overriding the return address in the stack since the return address is stored in the protected stack managed by the WebAssembly’s runtime. However corruption of data is possible and the extent of the exploit depends on: 1) where the buffer is
Query 5: Static buffer sizes in function $f$.

```c
// 1) Get function frame size.
alloc := 0 subNodes := [ n | n ∈ $f$.instructions and n.instType = Binary and n.opcode = i32.sub ];
foreach node ∈ subNodes do
    pdgEdge := inPdgEdge(node, Const, i32); if inPdgEdge(node, Global, $g0$) not nil and pdgEdge not nil then
        alloc := pdgEdge.value;
        break;
end
// 2) Find buffer locations.
buffers := [0, alloc]; addNodes := [ n | n ∈ $f$.instructions and n.instType = Binary and n.opcode = i32.add ];
foreach node ∈ addNodes do
    pdgEdge := inPdgEdge(node, Const, i32); if inPdgEdge(node, Global, $g0$) not nil and pdgEdge not nil then
        Append(buffers, pdgEdge.value);
end
// 3) Difference between buffer locations.
buffers := sort(buffers);
reduction := reductionDiff(buffers);
buffSizes := map(buffers, reduction);
```

stored and 2) the memory layout.

The code in Figure 5.9 shows a classic buffer overflow of $buffer$ by using the function `read` where expects to read 128 bytes from the standard input while the destination buffer only has 64 bytes allocated. Since $buffer$ is declared inside the function `main`, it will be allocated in the stack when the function `main` is called. In case the compiler generates the layout represented in Figure 5.8b, a possible exploit could be override constant data from the data section. For instance, if there is a constant string of JavaScript code that will be later evaluated, an attacker can corrupt the memory where the JavaScript's code is stored with code of his own resulting in arbitrary code execution. The string can be also be HTML code that will later end up in the DOM and in this case it is possible to trigger a cross-site scripting. There is also the possibility of hijacking control of the program if the code takes use of indirect calls using data that is stored in the data section by corrupting that data.

The examples in Figures 5.10 and 5.11 are similar. The differences lay down where the buffer is stored. In the first case, $buffer$ is a global array, thus, it will be stored in the data section. In the second case, $buffer$ is allocated in the heap after the call to `malloc`. The possible exploits are similar to the stack buffer overflow, with the clearly restrictions in the heap given the its location within the linear memory, making it impossible to override the stack or data segments. However it exposes possible safe unlinking in the heap triggered by calls to `free` in corrupted heap metadata.

```c
int main(int argc, char** argv) {
    char* buffer = malloc(64 * sizeof(char));
    // Here is the buffer overflow.
    read(STDIN_FILENO, buffer, 128);
    free(buffer);
    return 1;
}
```

Figure 5.11: Heap buffer overflow.
Static buffers

In the three shown examples, the problem relies in the possibility of read more characters than it is allocated. A good approach to find this vulnerability is by checking if the third argument from the function call read() is less or equal than the buffer size in the second argument. In the C source code, both the buffer pointer (C identifier) and the static buffer size is explicit, however, in WebAssembly is not so trivial. Regarding Figure 5.9, in WebAssembly, when the function main() is called, a function frame is allocated in the stack taking into account the space needed for the buffer. Figure 5.12 is an excerpt from the wasm code compiled from Figure 5.9 and corresponds to frame allocation where $g0$ is a global variable holding the stack pointer. The way it is allocated depends in the memory layout produced by the compiler. For instance, if the stack grows downwards (into lower addresses), an allocation is made by the difference between the stack pointer and the size allocated ($g0 - <\text{size\_allocated}>$) thus, the instruction i32.sub is present in line 3. As we can see, the new value of the stack pointer is saved both in the local variable $l2$ and in global variable $g0$. This is useful to refer to the buffer relative to the stack pointer. In this case, the buffer is referenced by $l2 + 32$. As stated previously in Section 5.5.1, by default, the stack grows downward so, for clarity we will look just into this pattern as the other option is similar, but with different instructions.

When the buffer is stack-based, we have enough information to infer its size. Query 5 shows the pseudo-code of how to calculate static buffers. The idea can be divided into three parts: 1) get function frame size, 2) find all referenced buffers and 3) make a difference between buffer locations to infer its size. For instance, in the example from Figure 5.12, a total of 96 bytes is allocated. This number can be retrieved by querying the function’s instructions for i32.sub instructions that depend in the global variable $g0$ and a constant as depicted in Figure 5.13a. Secondly we query for buffer positions by looking for i32.add instructions that also depend on $g0$ and a constant which represents the offset. In the CPG exemplified in Figure 5.13b, buffer is located at $l2 + 32$ (@32). Lastly, its size can be calculated by $96 - 32 = 64$.

Looking more closely to Query 5, if we had two buffers (bof1 and bof2) of size 64 each, a total allocated space would be 160 bytes which is assigned to alloc in line 5. The buffer locations for bof1 would be @32 and bof2 @96. This results that the list buffers, initialised with values [0,160], at the end of the for each loop in line 16 constrains the values [0,160,32,96]. The calculations of the buffer sizes occur after the list being sorted in line 17 ([0,32,96,160]) and perform a difference reduction assigning the list reduction := [32,64,64] which is calculated by $32 - 0 = 32$, $96 - 32 = 64$ and $160 - 96 = 64$. For the last instruction in line 19, we map the buffer positions against the result of the reduction: {0: 32, @32: 64, @96: 64}.

Having the buffer sizes, flag the vulnerability becomes trivial. As depicted in Figure 5.13b, we query the second argument for the buffer position, which is @32, and the third argument that depends on a
constant with value 128. Since we calculated that buffer @32 is 64, we can say that there is a possible vulnerability as 64 is lesser than 128.

**Buffer overflow in loops**

Buffer overflows can also happen if the programmer uses buffers inside loops in a careless manner. This is a common mistake specially, when handling with strings. Figure 5.14 exemplifies a buffer overflow inside a loop in a PNM decoding procedure in pnm2png from libpng library (CVE-2018-14550) [97]. For each iteration of the loop, the token buffer’s index is incremented and is assigned a character read from a file. The loop continues until a read character is one of the specific characters in line 7 meaning the loop can have an arbitrarily large number of iterations. We can craft a file with a string large enough without the specific characters in line 7 to trigger a buffer overflow and exploit the vulnerability.

The idea to find this vulnerability in WebAssembly’s CPG is to query loop’s instructions where in its AST descendants (loop’s block and condition) a local variable $l$ representing the index 1) is being incremented, 2) a store instructions depends on $l$ (assignment) and 3) there is no exit br_if whose condition test verifies the boundaries of $l$. For the first, we look for instructions i32.add that have incoming local PDG edge for $l$ and a constant PDG edge (which is the increment value). In the second, we simple look for store instruction with incoming local PDG edge for $l2$. And lastly, we search for br_if instructions and query its AST descendants (representing the composite condition) for the existence of comparison instruction which incoming local PDG edge for $l$.
5.6 Limitations

Static analysis tools always carry some limitations that directly affect their precision. A tool can produce false negatives (the program has bugs that the tool does not report) and can also produce false positives (the tool reports bugs that the program does not contain). Wasmati is a framework that generates CPG and allows the analyst to write and execute queries over the generated CPG. The precision of the results are limited by the precision of the queries that generate them. For instance, the simple query described in Query 4 completely disregards sanitisation procedures between the source and the sink where the sink depends directly in the value from the source. This may produce false positives in a certain code base however, that also might be the intention of the analyst, who might opt for a more conservative analysis for security-critical applications, in which case the flagged vulnerabilities need to be checked by hand.

The language of the source code and compiler tool chains used to obtain the Wasm program and the way it was compiled influences the precision of the queries as they affect the CPG’s structure.

**Memory Layout:** the way compiler rearranges the linear memory impacts the code generated. For instance, the calculation of the buffer sizes in Query 5 is slightly different if the stack grows upwards (into higher addresses). Instead of querying for `i32. sub` when looking for memory allocation, we must query for `i32. add` and vice-versa when looking for the buffer positions. Also, the position of buffers in different memory locations may difficult the search for certain vulnerabilities. Looking for the format strings query, a global static buffer is allocated in the data section where also constant strings are stored. This makes so that a global static buffer is indistinguishably from a constant string thus, Query 1 does not flag the vulnerability `printf(buf)` when `buf` is a global static buffer. The same occurs to the impossibility to check its size hence, the vulnerability from Figure 5.10 is never flagged.

**Optimised code:** since WebAssembly is a target language, many optimisations occur ahead-of-time during compilation. The application of different optimisation techniques such as loop unrolling, function inlining, removal of code, among others may result in a multitude of patterns for the same source code. This can have a profound effect in the produced AST which may difficult the work of the analyst when writing queries since it may produce different results, for the same source code, compiled with different optimisations.

**Debug Information:** compilers can produce a lot of debug information into the WebAssembly binary or text format. For the analyst, the more information the better. However, it is expected that most of the binaries produced are stripped from debug information for different valid reason such as decrease its file size or obfuscation. This can have an impact when analysing taint-style vulnerabilities where we rely in function names to describe sources and sinks. Nonetheless, the names of exported and imported functions are preserved, for instance, function calls to APIs, such as malloc, are preserved.
Summary

In this chapter we described different C security flaws including format strings, use of dangerous functions, use after free, double free, taint-style vulnerabilities and buffer overflow that can be ported to WebAssembly. We further showed how these vulnerabilities can be exploited at the Wasm level and how they can be identified via our CPGs for Wasm using CPG queries. We have given several examples of such queries. We closed the chapter with a discussion about the limitations of the proposed methodology. Next, we present the implementation details of our tool.
Chapter 6

Implementation

This chapter addresses the implementation details of the Wasmati framework. Section 6.1 presents an overview of our implementation goals and decisions taken. The graph structure and generation of the code property graph is described in Section 6.2. Lastly, Section 6.3 addresses the WebAssembly analysis, mainly the native queries and implemented query engine.

6.1 Implementation Overview

We have implemented Wasmati using C++11 using Cmake as its build system. Figure 6.1 represents the internal components of our tool. Wasmati is comprised of two main components: 1) CPG generator and 2) the query engine. We took full advantage from WebAssembly Binary Toolkit (WABT) which is an open-source project maintained by the official WebAssembly community to parse the WebAssembly’s binary and text format. These choices were taken having in account the following goals:

1. **Fast generation:** The generation of CPGs is CPU intensive and can take some time for real world applications. C++ is a mid level language and well known for its speed. It can employ multiple programming paradigms and has a vast and valuable function library.

2. **Portable:** C++11 is the standard for a couple of years and is well supported by multiple compilers and different platforms. The setup of Wasmati’s Cmake enables its compilation targeting Linux, MacOS, Windows and even the browser using WebAssembly. It supports different compilers GCC, Clang, MSVC, MinGW and Emscripten.

3. **Easy integration:** One of the main reason to choose C++11 was to incorporate the WABT parser and extend from that. WABT is built using C++11 and Cmake which eased its integration into Wasmati.

4. **Wasmati as library:** Wasmati can be included and used as a library in any C++ program by adding it to Cmake’s property list.

5. **MVP:** All browsers and WebAssembly’s runtimes already implemented the MVP version for Web-Assembly. The newly proposed features, despite some being already standardised, are not yet imple-
mented by all runtimes. Taking this into account, Wasmati targets the MVP version of WebAssembly, supporting all its instructions.

### 6.2 Code Property Graph

In this section we provide some additional implementation details on how the graph structure of code property graph is represented and how it this data structure is generated.

**Graph structure:** The graph, i.e., nodes and edges, were implemented following the object-oriented paradigm. Graph, node and edge are classes representing the base class for each entity. Figure 6.2 exemplifies the C++ code for the three classes. A graph contains a set of node pointers sorted by id and each node is comprised of two set of edge pointers which are 1) the incoming edges and 2) the outgoing edges. This enhances performance in back tracing during queries in detriment of having each type of node and edge represented by a different class. For instance, the CFG edge is a class which extends the Edge class. The same goes for the nodes. The BaseNode class is a template to generate definitions of other nodes. Figure 6.3 exemplifies the definition of an abstract class Instruction, which must be extended by all instruction nodes, and the definition of simple instruction node from Table 4.7 for every instruction in $\Sigma_S$.

**Graph generation:** Using Figure 6.1 as reference, Wasmati can receive a WebAssembly’s binary or text format module which will be fed into the WABT parser. The parser produces a list of functions, each one comprised with sequential list of instructions from the program input. The function list is given next as input to the AST builder which will create the AST structure containing the module node, function nodes, signatures and the corresponding instructions organised in a hierarchical construction obtained
typedef std::set<Node*> NodeSet;
typedef std::set<Edge*> EdgeSet;

class Node {
  const uint64_t _id;
  EdgeSet _inEdges;
  EdgeSet _outEdges;
  // [...]
}

template <NodeType t>
class BaseNode : public Node {
  // [...]
}

class Edge {
  Node* const _src;
  Node* const _dest;
  const EdgeType _type;
  // [...]
}

class Graph {
  Module* _module;
  NodeSet _nodes;
  // [...]
}

class Instruction :
  public BaseNode<NodeType::Instruction> {
    const ExprType _instType;
    // [...]
  }

template <ExprType exprType>
class SimpleInst : public Instruction {
  // [...]
}

typedef SimpleInst<ExprType::Nop> NopInst;
typedef SimpleInst<ExprType::Unreachable> UnreachableInst;
typedef SimpleInst<ExprType::Return> ReturnInst;
typedef SimpleInst<ExprType::BrTable> BrTableInst;
typedef SimpleInst<ExprType::Drop> DropInst;
typedef SimpleInst<ExprType::Select> SelectInst;
typedef SimpleInst<ExprType::MemorySize> MemorySizeInst;
typedef SimpleInst<ExprType::MemoryGrow> MemoryGrowInst;
// [...]

Figure 6.2: Graph structure.

Figure 6.3: Definition of Simple Instructions.

by folding instructions as described in Algorithm 1. During the execution of the folding algorithm, auxiliary structures are created to later aid the construction of the control flow graph. One example is a map that maps the instruction’s object from the list generated by the WABT parser to the corresponding newly created node. After the AST generation, it goes into the CFG builder. The CFG is generated by making another iteration on the function list generated by the WABT parser. With the help of the map created in the AST builder, the CFG builder connects with a CFG edge each instruction node to the next instruction, according to the list order and branching. The call graph is generated in this component. When a call instruction node is reached, the necessary call edges are created to link the call and the function.

The last CPG generator component is the PDG builder which is responsible for the data flow propagation used to create the program dependence graph. It is a straightforward implementation from Algorithm 2. An iterative version of DFS over the CFG performed applying the transfer function for each visited node. PDG edges are created according to the procedure described in Section 4.5. Depending on the input option given to Wasmati, when the CPG generation terminates, Wasmati can query the newly generated CPG, export the CPG to a file to later be queried or both. There are multiple options to export the CPG. We implemented different serialisation formats of the CFG. As of now, it can output the CPG in a CSV (compressed with gzip), Json, Soufflé’s datalog CSV [98], DOT and Neo4j’s CSV [89].

6.3 WebAssembly Analysis

Wasmati is flexible with respect to the querying engine used to find vulnerabilities in the constructed CPGs. As stated in the previous section, a CPG can be exported to a file in different formats. With this feature, it is possible to export the CPG and then import into another query platform. For instance, using
the exported Souflé’s datalog CSV, we can write queries in datalog and execute them in Souflé’s engine or we can import the generated Neo4j’s CSV into the graph database Neo4j to make full use of the SQL syntax-like query provided.

For performance reasons, in this thesis, we focus on the native query engine that we implemented as part of Wasmati with which we can easily write native C++ queries for inspecting our CPGs and execute them at nearly optimal speed. The query engine component provides three main entities to perform a query: 1) node set, 2) edge set and 3) predicate. Alongside with these three entities, the engine also provides a list of basic queries that can be composed with each other as is described in Table 6.1. A basic query filters the set of nodes/edges returning those corresponding to the supplied type. Some basic queries additionally accept a predicate to further filter the obtained set of nodes. For instance, the basic query `filter()`, receives a set of nodes and a predicate and iterates over the set applying the supplied predicate, returning the set of nodes for which the predicate holds. A simple query can be composed of basic queries. An example would be a query to retrieve all instructions from a function. This is achieved by composing the basic query `descendants()` following the AST edges and then `filter()` the nodes where the property `type` has the value `Instruction`. The `descendants()` procedure receives a set of nodes and performs a BFS at each node returning a node set containing all nodes visited during the BFS. By composing both, from the node set containing function nodes we retrieve a node set comprised from its instructions.

We implemented node streams and edge streams to allow for the composition of queries in a compact manner. Each stream is a class which stores a set of the respective type i.e. node set or edge set, with the same methods depicted in Table 6.1 which applies the respective query API to its stored set and assigning the result. The method returns the stream class instance. For example, to retrieve all instructions from the program we can simply query `NodeStream().functions().instructions(Query::ALL_NODES)`. The node stream starts with an empty set, queries for the function nodes and then queries for all instructions. The argument `Query::ALL_NODES` is a predicate that returns true for all nodes.

The predicates can be applied via Predicate class or as lambda function to allow a maximum flex-

<table>
<thead>
<tr>
<th>NodeSet functions()</th>
<th>(O(1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns all the function nodes.</td>
<td></td>
</tr>
<tr>
<td>NodeSet child(NodeSet set, Index n, const EdgeType type)</td>
<td>(O(V))</td>
</tr>
<tr>
<td>Returns the n'th child according to the edge type.</td>
<td></td>
</tr>
<tr>
<td>NodeSet children(NodeSet set, EdgeCondition&amp; cond)</td>
<td>(O(V))</td>
</tr>
<tr>
<td>Returns all the children following the edges according to an edge condition.</td>
<td></td>
</tr>
<tr>
<td>NodeSet parents(NodeSet set, EdgeCondition&amp; cond)</td>
<td>(O(V))</td>
</tr>
<tr>
<td>Returns all the parents following the edges according to an edge condition.</td>
<td></td>
</tr>
<tr>
<td>NodeSet filter(NodeSet set, Predicate&amp; pred)</td>
<td>(O(V))</td>
</tr>
<tr>
<td>Filters the node set according to the given predicate.</td>
<td></td>
</tr>
<tr>
<td>NodeSet descendants(NodeSet set, Predicate&amp; pred, EdgeCondition&amp; cond)</td>
<td>(O((N + E)/V))</td>
</tr>
<tr>
<td>Returns all descendants nodes following an edge condition and a predicate.</td>
<td></td>
</tr>
<tr>
<td>NodeSet instructions(NodeSet set, Predicate&amp; pred)</td>
<td>(O(N/V))</td>
</tr>
<tr>
<td>Returns all instruction nodes from the function nodes in the set.</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Basic query API.
Predicate& <keyProperty>(Value value, bool equal = true)
  Compares the value from property <keyProperty> against the provided value. $O(1)$

Predicate& inEdge(EdgeType type, std::string label, bool equal = true)
  Check if node contains and incoming edge with given type and label. $O(E)$

Predicate& inPDGEdge(PDGType type, std::string label, bool equal = true)
  Check if node contains and incoming PDG edge with given PDG type and label. $O(E)$

Predicate& outEdge(EdgeType type, std::string label, bool equal = true)
  Check if node contains and out-coming edge with given type and label. $O(E)$

Predicate& outPDGEdge(PDGType type, std::string label, bool equal = true)
  Check if node contains and ot-coming PDG edge with given PDG type and label. $O(E)$

Predicate& reachesIn(Node* src, EdgeCondition& cond)
  Check if there is a patch from src following an edge condition. $O(N+E)$

Predicate& reachesOut(Node* dest, EdgeCondition& cond)
  Check if there is a patch to dest following an edge condition. $O(N+E)$

Predicate& test(std::function<bool(Node*)> func)
  Tests the condition given by lambda function func. $O(func)$

Table 6.2: Predicate API.

Table of contents: The Predicate class is an helper class that construct a list of lambda functions that will then be applied sequentially to each node during a query (i.e. filter()), returning either true or false. The result of a predicate execution is the conjunction of the results from each lambda function call. Our main goal was to create an API to query using C++ which is simple, easy to write, readable and fast. Table 6.2 describes the API to construct predicates. To illustrate how this API is used, we will now build a query for identifying all calls to gets() inside the function $main$. We start by defining the predicate $p1$=Predicate().name($main) that will filter $main$ among the function nodes. The second predicate $p2$=Predicate().instType(ExprType::Call).label($gets$) will filter call instructions with label $gets$. The query NodeStream().functions().filter($p1$).instructions($p2$) returns all call nodes with label $gets$ that are present in function $main$.

The component Native Queries depicted in Figure 6.1 is a list of C++ files that implement the queries described in Chapter 5. The queries can use information from a JSON configuration file to query certain values. Figure 6.4 shows part of the configuration file that the implementation of the format strings uses in Query 1. It contains some functions susceptible to format strings vulnerabilities together with the index of the argument corresponding to the format template. Figure 6.5 shows a simplified implementation of the Query 1 in C++, using both the node stream and the predicate previously explained. In order to add a new query, we simply need to add the corresponding C++ file, compile and link.

To put the native query in perspective, Figure 6.6 shows the implementation of the same query in Soufflé’s flavoured datalog. The predicate reachesFunc() returns the function which the instruction $X$ integrates. Instead of a JSON config, the declaration of functions susceptible to format strings is made through facts. An example of a fact can be seen in line 2 for function $printf$. The query includes the file base.dl containing the graph definition and basic predicates. To execute, we
for (Node* func : Query::functions()) {
    NodeStream(func).instructions(Predicate()
        .instType(ExprType::Call)
        .TEST(config["fs"].contains(node->label()))
        .child(config["fs"][node->label()], Predicate()
            .outPDGEdge(PDGType::Const, "," false), Query::AST_EDGES)
    .forEach([](Node* node) {
        std::cout << "Vulnerability" << std::endl;
    });
}

Figure 6.5: Native format string query.

#include "base.dl"
fsFunc("$printf", 0).
.decl fs(funcName:symbol, id:unsigned, callName:symbol)
fs(FUNC_NAME, X, NAME) :-
    fsFunc(NAME, IDX), call(X, NAME, _, _), reachesFunc(FUNC_NAME, X),
    child(X, BUFFER_ARG, IDX, "AST"),
    !pdgEdge(BUFFER_ARG, X, _, "Const", _).

Figure 6.6: Datalog format string query.

import the query file into Soufflé’s alongside with the exported CPG generated by Wasmikit.

6.4 Optimisations

During the implementation a few design decisions were made in order to speed the CPG generation. The most impactful are briefly explained below. **Call edges:** The calculations for possible function calls in case of a call indirect are performed beforehand to avoid recalculations every time a call indirect node is reached. This proved to have a significant performance boost of about 60-80% during the generation of the CFG + CG.

**PDG iterative version:** The first implementation of Algorithm 2 performed a recursive version of DFS over the CFG. This approach had some residual overhead in execution time due to many function calls but, more importantly, it did not scale since the stack has usually no more than a few megabytes of space, larger programs resulted in a stack overflow terminating the program. The iterative version of the DFS described in Algorithm 2 proved to be scalable since it allocates space in the heap as necessary.

**PDG loops:** As stated before, when calculating dependencies in loops, it is necessary to traverse the body of loops multiple times until the dependencies of the loop’s entry point stabilise. A problem that arises is that, until the condition is met, instructions that follow the loop from out of its body, also re-execute which impaired performance in larger programs. To counter this, we restricted the calculation to just the instructions of the loop’s body by creating a mechanism to refuse the calculation if an instruction does not belong to the loop’s body while the loop calculation is still in progress.

**PDG loop caching:** When analysing WebAssembly programs we found that there is a high frequency of loops inside loops where sometimes can go up to a depth larger than 8 enclosing loops. This resulted in
high number of iterations due to values being assigned changing the abstract state given as input from outside the loop. To avoid recalculations, every incoming abstract state from outside of loop is cached. If an incoming abstract state is present in cache, the program follows with the last calculation made. This reduced the time of PDG generation in about 20%.

**PDG control edges:** Following some finding from the last paragraph, where WebAssembly programs have usually a large block / loop depth, resulted in a large quantity of control PDG edges and high memory usage. In some programs, the control PDG edges accounted up to 98% of total edges of the CPG. We decided to not generate them since they can be easily queried afterwards using the CFG where they do not deem high query time.

**Summary**

This chapter has described the implementation of Wasmati. It explained the graph, nodes and edges structures alongside with the detailed information about the generation of CPG in the different stages. The Wasmati's implementation includes a query engine with an high level query API aided with node/edge stream and predicates to facilitate the writing and readability of queries. This deemed possible the implementation of several native queries following the specification from the previous Chapter 5. If desirable, the implementation gives flexibility to export the CPG to an external querying platform. The chapter ends by elucidating the most important decisions taken which contributed for an higher performance of Wasmati. The next chapter presents a comprehensive experimental evaluation of the framework.
Chapter 7

Evaluation

This chapter presents the experiments that were carried in order to evaluate Wasmati. We aim to answer our research question RQ4 by showing that Wasmati is a viable tool to assist analysts in detecting security vulnerabilities in WebAssembly code. In order to evaluate the framework, we divide the evaluation in three parts. The first part aims to assess the feasibility of detecting vulnerabilities (Section 7.1). This is achieved by executing the described queries in Chapter 5 over four different datasets containing C programs with known vulnerabilities. The second part is presented in Section 7.3 and evaluates the CPG generation according to its scalability, performance, and resource usage using PolybenchC [51] and CPU SPEC 2017 [55] benchmarks. We also compare our generation times against Wassail [75]. Lastly, Section 7.2, portrays the performance evaluation of the native queries using the same benchmarks. In summary, our evaluation aims at answering the following main questions:

1. Can security vulnerabilities in WebAssembly be modelled using CPG?
2. How well does Wasmati scale when generating CPGs from large real world applications?
3. How well does the graph querying scale over CPGs generated from large real world applications?

7.1 Vulnerability Detection

The first part from our evaluation is described in this section. Our goal is to assert the feasibility of modelling security vulnerabilities in WebAssembly using code property graphs. To achieve the proposed goal, we executed the queries described in Chapter 5 over four datasets containing programs written in C with known vulnerabilities. The datasets were compiled to WebAssembly using Emscripten 2.0.9 [14] with level 1 optimisation and debug information. This proved to be the most balanced option to manually analyse binaries since it does not perform function inlining.

7.1.1 Datasets

Next we describe the four datasets used in our evaluation 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 phases of Wasmati. It contains all the code snippets with vulnerabilities present in this document, mainly in Chapter 5.

2. **Lehmann**: is comprised of 7 programs from Lehman et al. [63]. 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 5.14, a remote execution code in NodeJS and arbitrary file write.

3. **STT**: is a repository of 47 C vulnerable programs from Binary Analysis School maintained by the Security Team @Tecnico (STT) [99]. 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. 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. The `sizeof()` procedure does not exist in WebAssembly since it is replaced at compile time with the its corresponding value.

### 7.1.2 Methodology

We considered a set of 10 queries targeting: format strings, dangerous functions, use after free, double free and different variation of tainted-style vulnerability’s and buffer overflows to evaluate the effectiveness and precision in finding all the vulnerabilities in the chosen datasets with the least false positives. In order to maintain a low percentage of false positives, we followed a conservative approach for each query in order to produce as few false vulnerability reports as possible.

Regarding dangerous functions, we selected only two functions which we considered inherently dangerous independently of the context is used. The two functions are `gets()` and `strcat()`. We decided to exclude the other functions described in Section 5.2 since it would include cases where they are not considered a security vulnerability. Also, many of the excluded functions are taken into account as part of other queries; for instance, the `strcpy()` is a function reason in buffer overflow queries.

With respect to taint-style vulnerabilities, three different queries were performed: Tainted CallIndirect, Tainted Func-to-Func, and Tainted Local-to-Func. The query Tainted CallIndirect inquires the taint state of the last argument of instruction `call_indirect` which contains the index of the function to be called. The argument controlled by the user can result in arbitrary code execution within the restrictions of the WebAssembly semantics. The query Tainted Func-to-Func follows the classic pattern of taint-style vulnerabilities by tracking a value returned from a source call to a sink call. The query Local-to-Func is similar but tracks the function parameters from the start of the function to a sink. It follows an inter-procedural propagation by recursive backtracking using call edges. We only follow call edges whose source nodes are call instructions, discard in the process `call_indirect` calls. This tries to avoid an exponential explosion of paths degrading the query performance. We configured the queries to consider imported functions as sources and sinks disregarding all runtime APIs.
<table>
<thead>
<tr>
<th>#</th>
<th>Query Description</th>
<th>LOC</th>
<th>Basic TP</th>
<th>FP</th>
<th>P</th>
<th>Lehm TT TP</th>
<th>FP</th>
<th>P</th>
<th>STT TP</th>
<th>FP</th>
<th>P</th>
<th>CWE TP</th>
<th>FP</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Format Strings</td>
<td>24</td>
<td>5</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>8</td>
<td>11</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Dangerous Function</td>
<td>14</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Use After Free</td>
<td>67</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Double Free</td>
<td>60</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Tainted CallIndirect</td>
<td>55</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Tainted Func-to-Func</td>
<td>55</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Tainted Local-to-Func (recursive*)</td>
<td>124</td>
<td>16</td>
<td>0</td>
<td>16</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>Buffer Overflow - Static Buffer</td>
<td>141</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>12</td>
<td>0</td>
<td>15</td>
<td>8</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>Buffer Overflow - Static Buffer (malloc)</td>
<td>86</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Buffer Overflow - Loops</td>
<td>94</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Total: 720 35 0 38 6 0 7 40 8 43 19 0 20

Table 7.1: Query vulnerability report by dataset.

Lastly, in the buffer overflows we targeted two main categories: 1) memory copy to static buffers with wrong size and 2) unbounded assignment/reads of buffers in loops. Both queries 8 and 9 from Table 7.1 fits into the first category. It looks for local buffers and for heap buffers allocated with a constant sized argument by malloc. Memory copy procedures considered are `fgets()`, `read()`, `memcpy()`, `strcpy()` and `strncpy()`. The query number 10 falls into the second category and it queries for reads or assignments in buffers during the execution of loops where the exit condition does not check index boundaries.

### 7.1.3 Results

The security vulnerabilities reported by each query can be seen in Table 7.1. It presents the lines of code (LOC) of each query averaging 72 LOC per query. Findings are aggregated by dataset describing the absolute number of true positives (TP), false positives (FP) and the actual present vulnerabilities in the code (P). We count a true positive as a correctly reported vulnerability by the query and a false positive otherwise. We also consider a present vulnerability if the vulnerability is represented both in the C source file and in the WebAssembly compiled code. For instance, in some cases, pieces of code containing the vulnerability were removed by the compiler at compile time, this took special effect in some double free vulnerabilities and memory copies which result in buffer overflows. Each present vulnerability is categorised accordingly to its nature and is assigned to the query that best matches its pattern.

Looking closely at the results, the queries performed well, given that, for the majority of benchmarks, they reported all existing vulnerabilities while at the same time reporting no false positives. The exceptions are the format strings (1) and buffer overflows in static buffers (8).

Starting with format string vulnerabilities, in the basic dataset, there were two vulnerabilities the query did not report. The main reason arises from calls to `printf()` using global static buffers as its first argument. This obviously constitutes a vulnerability however, in WebAssembly, both constant data strings and global static buffers are indistinguishable from each other and, as result, is not reported as such. In the STT dataset a total of 8 false positives were reported by the query. Analysing the binary, we find that when there are multiple `printf` calls sequentially, the WebAssembly compiler stores...
contiguously the constant format templates and each template is referenced relative to the first position. The query wrongly perceives this logic as a variable format template despite it in fact being constant. The relative reference does not happen in code compiled with level 3 optimisation flag on.

In the case of the buffer overflows in static buffers, a total of 4 were not reported. All of them were overflows in global static buffers. The buffer is stored in the data section and is referred by a constant pointer. The query cannot infer the size of a global static buffer so, it follows a conservative heuristic to not report.

Table 7.2 shows the summary of the results. In a total of 108 vulnerabilities from 110 programs, 100 were correctly reported and 8 were false positives. Alongside these values, it provides three other metrics. The FPR stands for false positive rate and describes the proportion of total results corresponding to incorrect reports:

$$FPR = \frac{FP}{P} \quad (7.1)$$

The true positive rate (TPR) represents the proportion of total results corresponding to correct reports:

$$TPR = \frac{TP}{P} \quad (7.2)$$

Finally, the precision from the set of queries accesses the proportion of true positives in respect to the total reports made by the queries:

$$Precision = \frac{TP}{TP + FP} \quad (7.3)$$

Overall, the presented set of queries have a precision of 92.6% over the four databases. Clearly, precision can be improved by trying to ameliorate the format strings query. The simple format string query presented and described in Chapter 5 does not account for some correct patterns in WebAssembly’s code and flag them as a security vulnerabilities.
Table 7.3: CPG generation times and information from SPEC CPU 2017 binaries.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Source</th>
<th>Instruct. (k)</th>
<th>Size (KiB)</th>
<th>Nodes</th>
<th>Edges</th>
<th>Memory (MiB)</th>
<th>Time</th>
<th>Exported (MiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500.perlbench.r</td>
<td>C</td>
<td>837.8k</td>
<td>1,964</td>
<td>879.0k</td>
<td>2.5M</td>
<td>175.87</td>
<td>45.34s</td>
<td>14.00</td>
</tr>
<tr>
<td>502.gcc_r</td>
<td>C</td>
<td>2.9M</td>
<td>6,964</td>
<td>3.1M</td>
<td>9.6M</td>
<td>642.24</td>
<td>4min15</td>
<td>53.00</td>
</tr>
<tr>
<td>505.mcf_r</td>
<td>C</td>
<td>27.4k</td>
<td>56</td>
<td>30.0k</td>
<td>89.0k</td>
<td>6.14</td>
<td>1.20s</td>
<td>0.48</td>
</tr>
<tr>
<td>508.namd_f</td>
<td>C++</td>
<td>323.0k</td>
<td>636</td>
<td>343.0k</td>
<td>813.0k</td>
<td>64.05</td>
<td>7.62s</td>
<td>4.70</td>
</tr>
<tr>
<td>510.parest_f</td>
<td>C++</td>
<td>1.0M</td>
<td>2,190</td>
<td>1.1M</td>
<td>3.5M</td>
<td>226.47</td>
<td>33.51s</td>
<td>19.00</td>
</tr>
<tr>
<td>511.povray_f</td>
<td>C++</td>
<td>385.4k</td>
<td>909</td>
<td>406.5k</td>
<td>1.4M</td>
<td>90.06</td>
<td>4min19</td>
<td>7.30</td>
</tr>
<tr>
<td>519.ibm_r</td>
<td>C</td>
<td>13.4k</td>
<td>29</td>
<td>14.6k</td>
<td>55.2k</td>
<td>3.36</td>
<td>1.39s</td>
<td>0.26</td>
</tr>
<tr>
<td>520.omnetpp_f</td>
<td>C++</td>
<td>379.3k</td>
<td>98</td>
<td>379.3k</td>
<td>1.3M</td>
<td>125.04</td>
<td>34.24s</td>
<td>20.90</td>
</tr>
<tr>
<td>523.xalancbmk_f</td>
<td>C++</td>
<td>3.2M</td>
<td>7,944</td>
<td>3.4M</td>
<td>13.7M</td>
<td>444.67</td>
<td>6.44s</td>
<td>3.40</td>
</tr>
<tr>
<td>525.x264_f</td>
<td>C</td>
<td>283.6k</td>
<td>592</td>
<td>282.1k</td>
<td>264.1k</td>
<td>58.65</td>
<td>8.88s</td>
<td>4.60</td>
</tr>
<tr>
<td>526.blender_f</td>
<td>C++</td>
<td>3.2M</td>
<td>9,944</td>
<td>3.4M</td>
<td>44.1M</td>
<td>1,735.99</td>
<td>3min45</td>
<td>184.00</td>
</tr>
<tr>
<td>531.deepsjeng_f</td>
<td>C</td>
<td>53.0k</td>
<td>112</td>
<td>56.6k</td>
<td>158.0k</td>
<td>11.30</td>
<td>2.15s</td>
<td>0.88</td>
</tr>
<tr>
<td>538.imagick_f</td>
<td>C</td>
<td>517.5k</td>
<td>1,216</td>
<td>552.5k</td>
<td>1.6M</td>
<td>112.10</td>
<td>35.74s</td>
<td>8.90</td>
</tr>
<tr>
<td>541.leela_f</td>
<td>C++</td>
<td>118.8k</td>
<td>272</td>
<td>135.5k</td>
<td>384.0k</td>
<td>27.23</td>
<td>3.08s</td>
<td>2.10</td>
</tr>
<tr>
<td>544.nab_f</td>
<td>C</td>
<td>55.6k</td>
<td>122</td>
<td>60.6k</td>
<td>172.6k</td>
<td>12.13</td>
<td>2.85s</td>
<td>0.94</td>
</tr>
<tr>
<td>557.xz_f</td>
<td>C</td>
<td>53.3k</td>
<td>136</td>
<td>57.8k</td>
<td>185.8k</td>
<td>12.29</td>
<td>10.37s</td>
<td>0.94</td>
</tr>
</tbody>
</table>

Average per binary: 713.0k 1,797 747.6k 4.9M 236.17 57.79s 22.44
Total: 12.1M 30,547 12.7M 83.9M 4014.91 16min22.36s 381.50

### 7.2 Generation Performance

This section comprises the second part from our evaluation. 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 [51] and SPEC CPU 2017 [55]. The experimental evaluation was performed in a 64bit Ubuntu 18.04LTS with 16GB RAM and 4 Intel Core i7-4700HQ 2.40GHz CPUs with all runs using the same configuration JSON file.

The aforementioned PolybenchC suite is composed of 30 C programs designed to measure the effect of polyhedral loop optimisations in compilers. All programs in the suite are small scientific computing programs with 100 LOC on average. Wassail [75] used PolybenchC to evaluate the generation of function summaries. The summaries computed by Wassail are equivalent to the PDG edges targeting the return instruction generated by Wasmati. To that end, they generate the control flow graph for the requested function followed by a data flow analysis propagation over the CFG. Wasmati in comparison to Wassail, also generates the CFG and performs data flow analysis in order to calculate dependencies, however, Wasmati generates the programs AST which Wassail does not. As we have already argued, the PolybenchC are small programs and do not represent larger applications to assess Wasmati's scalability. Nonetheless, it is still valuable to us generate the CPG using Wasmati to put in perspective with Wassail [75]. Figure 7.1 shows the time comparison between Wasmati and Wassail demonstrating that our framework is far superior in terms of execution time. On average, Wasmati is 9.1x faster than Wassail.

To assess the scalability of Wasmati, we generated CPGs for some of the applications in 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 7.3. On average, the binaries are composed of 713k instructions with 502.gcc and 526.blender peaking in 2.9M and 3.2M instructions respectively. Despite
the large quantity of instructions, the binaries are no larger than 10KB. 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 its CPG in less than 4min30s using up to 1.73 GB of RAM.

Figure 7.2 presents a more detailed analysis of time taken to construct the CPG per benchmark, showing how much time is taken 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 PDG. As expected, the PDG 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. We believe this is due to the high number of call indirect instructions occurring in these benchmarks. The number of nodes per generated graph is not much different from its number of instructions. Besides instructions, CPG nodes additionally represent function parameter, local variables and return values. As the number of extra nodes is considerably smaller than the number of instructions, we can state that the number of nodes in the generated graphs grows linearly (with a constant factor very
close to 1) with the number of instructions contained in the corresponding Wasm program.

Regarding the number of edges, the data shows high variance. The average is heavy weighted by the single program 526.blender where, with its 44.1M, constitutes 53% of total edges. Removing it would low the average of 2.5M, still with high variance. There is only one incomming AST edge (except module node) for each node, making the AST edges grow linearly with the number of instruction. In the CFG, if all nodes are reachable, there are at least one incomming CFG edge for each instruction. Since branch are rare and apart from br_table witch can branch indefinitely, we take the average of 2. In this case the CFG edges also grow linearly with the number of nodes. The PDG and the CG are dependent in the nature of the program thus, provoking the variance. In Figure 7.3 shows the percentage of AST edges and CFG edges are even out while the PDG and CG varies a lot depending in the program. Figure 7.4 depicts an even edge distribution by type over all 158 binaries composed from the 4 datasets, PolybenchC and SPEC CPU 2017.

Figure 7.5 represents the time spent constructing the CPG in terms of the graph size, total number of nodes + total number of edges. The regression follows a power series with $R^2 = 0.929$. This demonstrates that the PDG construction is polynomial in function of the graph’s size and can scale to larger programs. The last column in Table 7.3 exhibits the size of the compressed exported file representing
Table 7.4: Query execution time (s).

<table>
<thead>
<tr>
<th>Binary</th>
<th>Query</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>500.perlbench_r</td>
<td></td>
<td>2.73</td>
<td>2.73</td>
<td>2.71</td>
<td>2.72</td>
<td>2.71</td>
<td>2.67</td>
<td>2.76</td>
<td>10.28</td>
<td>2.72</td>
<td>5.23</td>
<td>37.25</td>
</tr>
<tr>
<td>502.gcc_r</td>
<td></td>
<td>9.95</td>
<td>10.04</td>
<td>9.93</td>
<td>9.87</td>
<td>9.84</td>
<td>9.70</td>
<td>9.89</td>
<td>33.98</td>
<td>10.10</td>
<td>15.67</td>
<td>128.96</td>
</tr>
<tr>
<td>505.mcf_r</td>
<td></td>
<td>0.09</td>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
<td>0.09</td>
<td>0.11</td>
<td>0.10</td>
<td>0.33</td>
<td>0.09</td>
<td>0.15</td>
<td>1.25</td>
</tr>
<tr>
<td>508.namd_r</td>
<td></td>
<td>1.07</td>
<td>1.09</td>
<td>1.08</td>
<td>1.08</td>
<td>1.07</td>
<td>1.06</td>
<td>1.08</td>
<td>3.46</td>
<td>1.08</td>
<td>2.25</td>
<td>14.31</td>
</tr>
<tr>
<td>510.parest_r</td>
<td></td>
<td>3.35</td>
<td>3.36</td>
<td>3.44</td>
<td>3.43</td>
<td>3.36</td>
<td>3.34</td>
<td>3.62</td>
<td>12.26</td>
<td>3.37</td>
<td>5.79</td>
<td>45.31</td>
</tr>
<tr>
<td>511.povray_r</td>
<td></td>
<td>1.29</td>
<td>1.29</td>
<td>1.29</td>
<td>1.29</td>
<td>1.30</td>
<td>1.27</td>
<td>1.29</td>
<td>4.70</td>
<td>1.28</td>
<td>2.24</td>
<td>17.25</td>
</tr>
<tr>
<td>519.lbm_r</td>
<td></td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.17</td>
<td>0.05</td>
<td>0.08</td>
<td>0.61</td>
</tr>
<tr>
<td>520.omnetpp_r</td>
<td></td>
<td>2.46</td>
<td>2.45</td>
<td>2.44</td>
<td>2.44</td>
<td>2.57</td>
<td>2.47</td>
<td>2.60</td>
<td>8.60</td>
<td>2.43</td>
<td>3.30</td>
<td>31.76</td>
</tr>
<tr>
<td>523.xalancbmk_r</td>
<td></td>
<td>6.47</td>
<td>6.45</td>
<td>6.44</td>
<td>6.82</td>
<td>7.15</td>
<td>6.49</td>
<td>7.03</td>
<td>21.44</td>
<td>6.46</td>
<td>8.42</td>
<td>83.16</td>
</tr>
<tr>
<td>525.ldecod_r</td>
<td></td>
<td>0.68</td>
<td>0.67</td>
<td>0.68</td>
<td>0.68</td>
<td>0.68</td>
<td>0.67</td>
<td>0.69</td>
<td>2.57</td>
<td>0.72</td>
<td>1.12</td>
<td>9.14</td>
</tr>
<tr>
<td>525.x264_r</td>
<td></td>
<td>0.90</td>
<td>0.89</td>
<td>0.90</td>
<td>0.90</td>
<td>0.91</td>
<td>0.88</td>
<td>0.90</td>
<td>3.19</td>
<td>0.90</td>
<td>1.30</td>
<td>11.66</td>
</tr>
<tr>
<td>526.blender_r</td>
<td></td>
<td>18.58</td>
<td>18.23</td>
<td>18.81</td>
<td>18.78</td>
<td>20.94</td>
<td>18.78</td>
<td>18.64</td>
<td>64.63</td>
<td>18.52</td>
<td>28.95</td>
<td>244.85</td>
</tr>
<tr>
<td>531.deepsjeng_r</td>
<td></td>
<td>0.18</td>
<td>0.19</td>
<td>0.19</td>
<td>0.18</td>
<td>0.19</td>
<td>0.17</td>
<td>0.19</td>
<td>0.65</td>
<td>0.17</td>
<td>0.29</td>
<td>2.39</td>
</tr>
<tr>
<td>538.imagick_r</td>
<td></td>
<td>1.78</td>
<td>1.77</td>
<td>1.78</td>
<td>1.77</td>
<td>1.79</td>
<td>1.77</td>
<td>1.79</td>
<td>6.74</td>
<td>1.75</td>
<td>3.98</td>
<td>24.93</td>
</tr>
<tr>
<td>541.leela_r</td>
<td></td>
<td>0.39</td>
<td>0.39</td>
<td>0.38</td>
<td>0.39</td>
<td>0.38</td>
<td>0.37</td>
<td>0.40</td>
<td>1.37</td>
<td>0.38</td>
<td>0.53</td>
<td>4.98</td>
</tr>
<tr>
<td>544.nab_r</td>
<td></td>
<td>0.18</td>
<td>0.18</td>
<td>0.18</td>
<td>0.18</td>
<td>0.17</td>
<td>0.17</td>
<td>0.18</td>
<td>0.62</td>
<td>0.17</td>
<td>0.32</td>
<td>2.35</td>
</tr>
<tr>
<td>557.xz_r</td>
<td></td>
<td>0.18</td>
<td>0.18</td>
<td>0.18</td>
<td>0.18</td>
<td>0.18</td>
<td>0.17</td>
<td>0.18</td>
<td>0.59</td>
<td>0.18</td>
<td>0.29</td>
<td>2.28</td>
</tr>
<tr>
<td><strong>Average per binary</strong></td>
<td></td>
<td><strong>2.96</strong></td>
<td><strong>2.94</strong></td>
<td><strong>2.97</strong></td>
<td><strong>2.99</strong></td>
<td><strong>3.14</strong></td>
<td><strong>2.95</strong></td>
<td><strong>3.02</strong></td>
<td><strong>10.33</strong></td>
<td><strong>2.96</strong></td>
<td><strong>4.70</strong></td>
<td><strong>38.97</strong></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>50.30</strong></td>
<td><strong>50.04</strong></td>
<td><strong>50.57</strong></td>
<td><strong>50.85</strong></td>
<td><strong>53.38</strong></td>
<td><strong>50.12</strong></td>
<td><strong>51.38</strong></td>
<td><strong>175.57</strong></td>
<td><strong>50.35</strong></td>
<td><strong>79.88</strong></td>
<td><strong>662.44</strong></td>
</tr>
</tbody>
</table>

the CPG serialised sizing about 22MiB per binary.

### 7.3 Query Performance

In the third and last stage of our evaluation, we measure query execution time over the SPEC CPU 2017 benchmark in order to assess its scalability. We attempted to put the execution time of native queries in perspective with Soufflé’s datalog engine. However, we did not obtain reasonable times using datalog. For smaller examples, native queries surpassed the execution times by a factor of 100 which prevented us from running the datalog queries on the applications of the SPEC CPU 2017 benchmark.

Table 7.4 shows the execution times (in seconds) of the native queries first described in Table 7.1 following the same enumeration. The first six queries are all similar in its complexity. This is reflected with similar time execution. On average, a binary took less than 39 seconds to perform all the ten queries totalling an execution time of 11 minutes for all binaries. The larger binary, 526.blender with 3.2M nodes and 44.1M edges, took about 4 minutes to complete all queries. The query time is dependent of the complexity of the queries. We analysed the complexity of each query from the set of queries described in Table 7.1. The complexity of a query is calculated taking the number of basic API queries described in Table 6.1 together with the predicates API in Table 6.2 calls. Our calculations state a polynomial complexity, meaning execution time grows polynomial with the number of nodes and edges of the CPG.

To evaluate the validity of our analysis, we performed an experimental evaluation by measuring the total query execution time over the 158 binaries. Figure 7.6 plots query times in function of the sum between number of nodes and edges. The regression also follows a power series with $R^2 = 0.902$,
proving our analysis. These particular queries demonstrate a reasonable execution time over the large binaries in SPEC CPU 2017 benchmark (average 39 seconds per binary) and a polynomial execution time in function of the graph’s size. Our queries try to find simple security vulnerabilities and we wrote them taking performance considerations. An analyst can write arbitrarily complex queries to find more complex vulnerabilities which may take exponential time due to the intrinsic nature of the query or just due to being poorly written. However, we believe that our results show that Wasmati supports the writing of performant queries which can run efficiently even when given real-world projects with millions of instructions.

7.4 Key Findings

Our evaluation aimed to address three main research questions. We revisit these questions and summarise our findings:

1) Can security vulnerabilities in WebAssembly be modelled using CPGs? We found 100 vulnerabilities out of 108 present in four datasets with known vulnerabilities determining a precision of 92.6%.

2) How well does Wasmati scale when generating CPGs from large real world applications? Comparing to Wassail, Wasmati is 9.1x faster. The construction time averaged 58 seconds per binary using SPEC CPU 2017 and the regression of execution time in function of the number nodes + edges follows a power series. This concludes the constructions is polynomial in time given the graph’s size.

3) How well does the graph querying scale over CPGs generated from large real world applications? We executed the 10 queries in the generated CPGs of SPEC CPU 2017 with an average execution time of 39 seconds. Our analysis of the queries found that the execution time of the written 10 queries grows polynomially given the graph’s size. This was supported by our experimental evaluation over 158 C programs.
Summary

In this chapter we started by analysing four datasets of C programs containing known vulnerabilities. We evaluated the precision of 10 queries by executing them over the generated CPGs of the four datasets. We then proceeded to a in-depth performance analysis by first comparing Wasmati’s construction time against Wassail using PolybenchC, we then proceeded to a more fine-grained evaluation over SPEC CPU 2017 which is comprised of large C and C++ programs compiled to WebAssembly. Finally, we measured the execution time of each query over the SPEC CPU 2017 benchmark CPGs generated in the previous step and ended with a discussion on the key findings in the evaluation. Next chapter concludes this dissertation by presenting the conclusion and directions for future work.
Chapter 8

Conclusions

8.1 Conclusions

WebAssembly is a new technology that allows web developers to run native C/C++ on a web page with near-native speed. Despite the mechanism provided by its original specification, it is possible to import common vulnerabilities from C and C++ code, such as format strings, buffer overflows, use after free, double free. These vulnerabilities can be leveraged by an adversary for performing exploits in the Web, such as cross-site scripting and SQL injection.

In this dissertation, we have addressed the lack of available tools in WebAssembly to assist analysts in finding vulnerabilities by providing a framework named Wasmati which relies on the construction and querying of code property graphs. We have shown how to construct CPGs for Wasm applications and how to query those graphs to find security vulnerabilities ported from C applications. We also described how C vulnerabilities can be ported to Wasm and how these vulnerabilities can be found by querying the CPGs that we generate. We have presented the design, implementation, and evaluation of Wasmati.

In the experimental evaluation, we have applied our queries to Wasm binaries resulting from the compilation of C programs with known vulnerabilities, demonstrating that our queries exhibit a high true positive rate and precision. We also measured construction and query execution times in large binaries compiled from SPEC CPU 2017. Our results show that Wasmati can construct the CPGs of Wasm binaries with millions of instructions in a timely manner, appearing to take polynomial time with respect to the size of the generated graph. Furthermore, we compared construction times using PolybenchC against Wassail proving Wasmati to be 9.1x faster.

8.2 Future Work

The work in this thesis focused primarily on the construction of code property graphs and on setting up a robust infrastructure for implementing efficient queries for finding security vulnerabilities in the generated CPGs. It would be interesting to explore in more detail vulnerabilities in WebAssembly ported from C and other languages which can be compiled into Wasm code. To this end, it would be necessary to
analyse patterns in the code property graph generated from different compilers which also implicates a more thorough knowledge to write queries.

Another interesting direction for future work would be to explore different alternatives and platforms to query the generated code property graph considering readability, ease of writing and speed. One possibility is to design and implement a domain specific language (DSL) and integrate it with our engine. Another approach would be to assess graph databases such as Neo4j, ArangoDB or OrientDB.

Due to time constraints, it was not possible to perform a large scale evaluation over WebAssembly binaries in the wild in order to identify their vulnerabilities. It would be relevant to perform such evaluation to study the prevalence of security vulnerabilities in Wasm code in the wild.

Wasm code often interacts with its host JavaScript program via a number of APIs. In the future, it would also be interesting to integrate Wasm CPGs with JavaScript CPGs into one unified structure that could be analysed using queries. The search for vulnerabilities would take in account both domains.
Bibliography


[99] STT. We are STT. https://sectt.github.io/. Accessed: 2020-12-09.