Discovering Vulnerabilities in WebAssembly with Code Property Graphs

Pedro Daniel Rogeiro Lopes
pedro.daniel.l@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisors: Professors Nuno Santos and José Fragoso Santos)

Abstract. WebAssembly is a new technology that allows web developers to run native C/C++ on a webpage 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. The goal of this project is to design and implement a new tool that can statically find vulnerabilities in WebAssembly code. Our approach is to use *code property graphs* (CPGs), a program representation that has been successfully applied to the detection of vulnerabilities in high-level languages. We propose to adopt this representation into the realm of WebAssembly programs.

Table of Contents

1 Introduction

The enlargement of the Web led to more complex, sophisticated, and demanding CPU applications such as: games, interactive 3D visualization, audio, and video. By historical accident, JavaScript was the only programming language natively supported by Web browsers and the basic option for the development of such applications. It started as a simple scripting language to bring some interaction to the static Web and today, it evolved to a high-level language where large applications are coded using it. 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 speed and overall performance.

Over the years, browsers implemented optimizations such Just in Time Compile (JIT) [1] and caching, which brought considerable performance improvements [2]. Several additional attempts have been tried to minimize the performance penalization of JavaScript. Adobe has developed and promoted Flash platform [3], Sun's proposed Java applet [4], Microsoft created ActiveX [5], and more recently Google introduced Native Code [6]. However, they all suffer from cross platform compatibility, and did not gain a general acceptance in large scale with exception of Adobe Flash. Also, for being browser plugins, most of them suffered a large number of critical vulnerabilities [7,8], thus, they have been deprecated [9,10] or are in process of being decommissioned [11].

In 2015, engineers from four major browser manufacturers – Google, Firefox, Microsoft, and Apple – collaboratively proposed a new portable, low-level assembly-like language called WebAssembly (Wasm for short) [12]. WebAssembly is designed, essentially, for speed. Its compact binary makes files much smaller in comparison with JavaScript textual files and faster to decode and execute, allowing Wasm programs to run with near-native performance with just only 10% penalty [13]. This is specially important considering that many client-side Web applications execute over slow networks, and on mobile devices and other resources-constrained platforms.

WebAssembly can be a compilation target for other languages such C/C++ and Rust, making these languages able to run also on the Web. However, despite its name, WebAssembly is not quite an assembly language, as it is not meant to any specific architecture. In general, every high-level language gets translated down to machine code. The problem that arrives is the heterogeneity that exists in the hardware world, where different processors implement different instruction sets (Instruction Set Architecture) and thus, have different machine code and consequently different kinds of assembly. At its core, WebAssembly is a virtual ISA. It is made for the browsers with the intuit of being ubiquitous and so, being able to run in any hardware and platform.

WebAssembly is also not meant to substitute JavaScript, but rather to complement it and run alongside Javascript code. So JavaScript can call code from a WebAssembly module and WebAssembly can call JavaScript code. In fact there is no difference between a WebAssembly module and an ES2015 module of JavaScript. Currently, all major browsers already implement WebAssembly and it is available to more than 88% of all users on the Web [14]. Moreover,

the performance gains and the safety mechanisms provided by WebAssembly gained popularity beyond the browser platform. In particular, Wasm has been adopted for uses in server-side runtimes [15,16,17], IoT platforms [18], and in edge computing [19].

However, albeit the existence of security measures provided by the semantics of WebAssembly, there are 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++ that can be imported into WebAssembly code [20]. Typical coding errors and multiple functions in C and C++ that are inherently unsafe, will also be unsafe in WebAssembly.

In this work, we aim at creating Wasmati, a tool that can statically find outstanding vulnerabilities in WebAssembly code. Such vulnerabilities can be exploited in launch specific attacks in the context of the Web, such as cross-site scripting (XSS), and code injection attacks. We propose to achieve this by adopting a recently proposed program analysis technique called *code property graph* (CPG) [21]. 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 in 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 expressed in some representation over another. The search for vulnerabilities can then be reduced to performing simple queries that translates into specific traversals of the program's CPG.

Our tool will receive as input the WebAssembly code and a set of queries, and outputs a list of potential issues that can cause errors at time of execution. However, there are specific challenges in employing a CPG-based approach to the analysis of WebAssembly, stemming from the closeness of Wasm instructions to machine code causing a loss in expressiveness compared to high-level languages. This work will develop a set of techniques to overcome these challenges.

The rest of this report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. Then, section 3 provides some necessary background on WebAssembly and introduces context information about general vulnerabilities in the Web and specific vulnerabilities in WebAssembly. Section 4 provides detailed information about current detection and analysis of code in general and of WebAssembly code in particular. Section 5 presents our proposed solution, covering both its architecture and implementation, and Section 6 outlines our evaluation plan. Finally, Section 7 presents the schedule of future work, and Section 8 concludes the report.

2 Goals

The purpose of this work is to develop a tool for detection of vulnerabilities in client-side Web applications caused by potentially insecure interactions between JavaScript and WebAssembly code. More precisely, we aim to:

Goals: Build a tool that can perform static analysis in WebAssembly code in order to detect and mitigate vulnerabilities present on it.

Our work uses the idea of code property graph that has been applied in other languages to find vulnerabilities. In order to demonstrate the proposed solution, we will implement a prototype of the tool that will receive WebAssembly code and a set of queries and return a list of vulnerabilities. We will perform a set of evaluation experiments to assess performance impact, developer involvement, vulnerability coverage, among others. In summary, the project will produce the following expected results.

Expected results: The work will produce i) a specification of the architecture of Wasmati, a tool to detect vulnerabilities in WebAssembly, ii) an implementation of the proposed architecture, iii) an extensive experimental evaluation using a fully operational prototype.

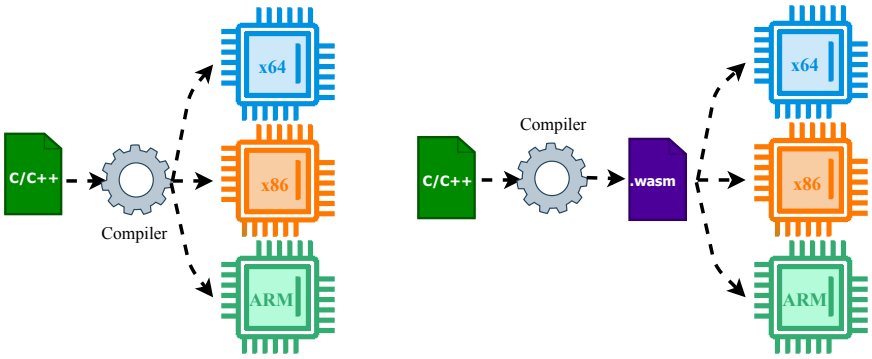
3 Background

This section 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 code is generated and what are its built-in security mechanisms. Then, we provide an overview of some of its potential vulnerabilities, putting them in perspective within the landscape of vulnerabilities in existing Web languages.

3.1 WebAssembly Overview

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* [22] 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 optimizations to gain performance. Despite the considerable performance gains, *asm.js* did not became a standard and lacks in some important features for performance of critical applications with 64-bit integers, in JavaScript and consequently in *asm.js* all numbers are IEEE-764 compliant [23] floating point doubles.

As an alternative, it was proposed a new standard and a new specification for a language called WebAssembly [12], which is a portable low-level byte code and a target compilation for efficient statically typed. It is designed to be fast and safe to execute, language-, hardware-, platform-independent, deterministic, easy to reason about and debug, simple interoperability with the Web, compact, easy to decode, to validate and be generated, 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, but several ongoing projects aim to support languages like Python [24] and C# [25]. Currently, some list includes a dozen of other languages [26]. The resulting WebAssembly code (Wasm for short) is then interpreted and executed in a stacked based machine.



(a) Classic compilation target. (b) Wasm compilation target.

Fig. 1. Compilation target design.

Efficiency and performance: Because WebAssembly is more compact than JavaScript, it takes less time to download files. Even modern techniques of compression of JavaScript where size is reduced significantly, the compressed WebAssembly's binary is still smaller. Also, it becomes faster to parse and validate, there is no need to generate the abstract syntax tree to transform to an intermediate representation as it is already in that stage. In fact, the parsing performance and compactness of *asm.js* code is inferior compared to WebAssembly [27].

Despite being interpreted, due its low-level nature, Wasm code runs at near native speed with just only 10% penalty [13] which is a major improvement compared to JavaScript. Optimizations also become faster, most of them were already performed ahead of time by LLVM [28], it does not spend time running 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 optimized and retry it. This usually happens because JIT takes assumptions about the code that do not hold, for example the type of the 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. Many optimizations made by JIT are simple not needed in WebAssembly.

Runtime environment: The first release implemented by the browsers aims at being a Minimal Viable Product (MVP). This means that many important features are not yet implemented, e.g., threading and garbage collector. The MVP does contain basic features which enable roughly the same functionality as asm.js.

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 and desktop applications like Visual Studio Code are also written in JavaScript powered by Electron. One of the main reasons for the usage of such runtime environments is the portability that they offer. The same code can run

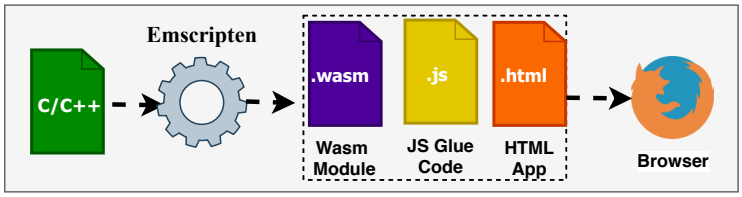


Fig. 2. Compiling code into WebAssembly.

in multiple platforms. WebAssembly can also bring performance gains without loosing portability to those environments.

A new open source group named Bytecode Alliance founded by Mozilla, Intel, Red Hat and Fastly [29] was created to advance the state-of-the-art in WebAssembly runtimes and create/evolve standards such the WebAssembly itself and WebAssembly System Interface (WASI) [17] to give a system interface between WebAssembly runtime and the kernel of the operating system.

3.2 Generation of WebAssembly Code

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, as exemplified in Figure 1(a). In the Web, when delivering code to run in the user's machine, we do not know by advance what architecture the code will run. So, in spite of its name, WebAssembly is not quite an assembly language as it is not meant to a specific architecture as shown in Figure 1(b). 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 set is denominated by virtual instruction set architecture (virtual ISA).

Compiler toolchain: Humans are not meant to program in WebAssembly, since it is a target language. This allows to provide a set of instructions that are closer and ideal to machines and run at near native speed. However, to be debuggable it exists a text format representation of the code [30].

The currently supported compiler tool chain is called LLVM. It is a robust tool and has a various front-ends and back-ends that can be plugged into it. In C for instance, exists a front-end called Clang [31] that go from C source code to LLVM code. From this point, it is in LLVM intermediate representation and many optimizations can be done by LLVM. For last it just needs to implement a back-end to generate WebAssembly code from LLVM.

One current robust WebAssembly compiler is Emscripten [32] represented in Figure 2. 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. As represented in Figure 2, it takes C/C++ source code and outputs three files: a Wasm file, a JavaScript file and an HTML file. The Wasm

C program code	Text Representation	Binary

`int addTwo(int a, int b) {`
`  return a + b;`
`}`
	`(func $addTwo`
	`  (param i32 i32)`
	`  (result i32)`
	`  (get_local 0)`
	`  (get_local 1)`
	`  (i32.add))`
		`60`
		`02 7f 7f`
		`01 7f`
		`20 00`
		`20 01`
		`6a`

Table 1. A simple C function on the left and the corresponding WebAssembly's text format and binary encoding on the right side respectively.

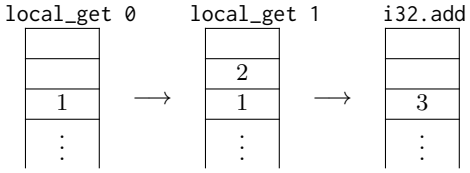


Fig. 3. Stack representation during execution of the program representend in Table 1 with parameters values being 1 and 2 respectively.

file is the compiled WebAssembly module. The JavaScript file is "glue code", it instantiates the Wasm module, sets up the memory, 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 in order to 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 preprocessing directives that developers can use, for instance, program JavaScript code within the C/C++ code. This is important 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 [33], 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.

Structure of Wasm code: Regardless of the toolchain, the end result must be a Wasm file. An example of a C program that is compiled to WebAssembly can be found in Table 1. This examples only shows the basic encoding of each instruction. Of course, there are also other complex instructions and information that go along with these to encode the mechanics of the binary, for instance, the size of the function body. The obtained Wasm file has the binary representation of the code. To understand better how WebAssembly behaves, there are five main basic concepts to know:

– **Stack Machine:** WebAssembly is a stack based machine, meaning that instructions receive arguments from the stack and return elements to the stack. In the example shown in Table 1, the instructions `local_get 0`, `local_get 1`

and `i32.add` are executed in this order. The stack during execution of the call `addTwo(1,2)` can be seen in Figure 3. The instruction `local_get 0` pushes the value of the first argument of the function to the stack and `local_get 1` pushes the value of the second argument. At the end, `i32.add` pops the two values from the stack, executes the operation (sum) and pushes the result onto the stack.

In practice, this reduces the encoding of the `i32.add` instruction to a single byte because it does not need to specify where the arguments of the instruction are. Consequently, reduces the size of a Wasm file.

Even though WebAssembly is specified in terms of a stack machine, the browser does not need nor implement a stack machine due to performance issues. The browser has flexibility to use and allocate the best registers depending in the machine's architecture where the code is supposed to run.

- **Module:** It represents the binary format of a WebAssembly that has been compiled. Contains definitions of functions, imports, exports, tables, global variables and memory. A module is stateless and can be seen as just a binary large object (Blob).
- **Memory:** It is a simple and resizable JavaScript's ArrayBuffer containing the linear array of bytes read and written by WebAssembly's instructions.
- **Table:** An array that indexes references. For instance, the function table, indexes the function's references existent in the module.
- **Instance:** A Module that has been instantiated in JavaScript. It contains all the state it uses at runtime, for instance, the Memory the program will use and the Tables that can be dynamically changed.

Good to mention that the Wasm instructions do not involve any registers of any kind. It operates only with the help of the stack, pushing and popping values. When the function goes out of scope, its return value (if exists) stands at the top of the stack.

3.3 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 mitigations against exploitation. With this in mind, WebAssembly was designed with four main features that work towards such needs:

Environment: It starts with the context of execution within the browser. A WebAssembly application is executed independently within a sandbox environment (JavaScript's environment in case of the browser) and it can not escape without proper APIs. Within the browser, each application module is subjected to its security policies such as restrictions on information flow through same-origin policies [34]. 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 sandbox environment. It cannot open socket 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 JavaScript or using existent APIs.

The same happens outside of the browser. WebAssembly is sandboxed, the runtime is in charge to provide an API that the code can use. 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 fully 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.

Memory model: Unlike normal binary, which compiles to a specific architecture, a WebAssembly module does not have access to all of the memory in its process. It is restricted to a contiguous, untyped, byte-addressable called *linear memory* varying in memory size. This size is always a multiple of a WebAssembly page which is 64KiB. The initial size is defined by the data present in the binary and can be dynamically increased, being always initialized 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. This attenuates common errors in languages such C and C++ involving unsafe pointer usage and undefined behavior. This includes dereferencing pointers to unallocated memory (e.g. NULL) or freed memory locations. Given that the linear memory is, in fact, an ArrayBuffer, the JavaScript VM performs bound-checks on access. It is worth noting that bound-check happens to the region level (whole array) and not at context level. For routines other than the browser, the responsability to check the bounds of linear memory belongs to the runtime.

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 addreses of the function calls. However, 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 [35], 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.

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.

Control-flow integrity: Functions calls cannot be performed to arbitrary addresses. 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 [36]. This table is a JavaScript object named `WebAssembly.Table` which is an array-like structure outside of WebAssembly's memory. The values are references to functions.

When the function call is dynamic and unknown at compile time (e.g. polymorphism in C++), it triggers an indirect function call. The index is pushed to the stack and, before the call succeeds, it is subjected to a type signature. The signature of the called function must match the signature specified at the call site. All the calls happen in a structure called *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.

Control flow is implemented in a way where calling an unexpected function is likely going to fail. The expected and unexpected paths of execution are statically analyzed at compile time. This hardens the possibility of hijack the control flow of the program but does not eliminate the possibility. It is possible to gain program control using code reuse attacks against indirect calls [37]. 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.

Compiler mitigations: Current advanced 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 WebAssembly is essentially the same used to compile to native code giving this for free. However, some of them 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) [38] and restrict execution of certain sections of memory known as data execution prevention (DEP) [39] are not necessary. Address space layout randomization (ASRL) [40] is not currently supported by WebAssembly. 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. Compilers with warnings against unsafe functions (like `gets` and `strcpy`) and provide control flow integrity checks can protect code compiled to WebAssembly.

3.4 Vulnerabilities in Web Languages

Despite the security mechanisms available in WebAssembly, code vulnerabilities are expected to exist as demonstrated by the history of Web languages.

There is a wide range of programming languages used in the Web, both for front-end and back-end development. The most well known are JavaScript, PHP, Java, Python, Ruby, Perl, C#, C, and C++. They are mature languages with long years of use. Nevertheless, many vulnerabilities in code written in those languages have been found over the years. Next, we take a general walk-through to some of them with more focus in C/C++ and PHP.

In the case of C and C++, the developer is responsible for the memory management, which is the cause of the majority of the known vulnerabilities in these languages. Buffer overflows, use after free, double free, memory leaks, and null pointer dereference are just a few. Likewise, type confusion, errors converting types, and poorly thread management such as failure to release locks brought to the table other languages that ease the development.

PHP was one of the first language specifically aimed at web development. It has a combination of garbage collection and reference counting, relieving the developer of the burden of memory management. In a security standpoint, PHP is a poorly designed language and it is widely known by the large quantity of vulnerabilities it causes, which will be summarily explained below. Many functions and directives in the language allows certain functionality that usually developers are unaware, making it easily exploitable.

Many PHP applications are susceptible to remote code execution (RCE) [41] due to the dynamic evaluation of code in an user controlled variable. Database accesses are also exploitable if user input is not properly sanitized allowing code injection, namely SQL injection (SQLi) [42], resulting in disclosure of sensitive information. In the client side, poorly written programs tend to be vulnerable to cross-site scripting (XSS) attacks [43], where an attacker causes malicious code to load in a website visitor's browser and execute. This type of attack can be specialized to a cross-site request forgery (CSRF) attack [44] which occurs when an attacker can create a link and get a site administrator or someone with privileged access to click on that link, thereby triggering unwanted behaviors such as sending the administrator's session cookie to the attacker.

The serialization/marshalling and deserialization of untrusted data constitutes one of the most common vulnerabilities in the Web [45]. In PHP there is a function called `unserialize()` which takes a stored object and converts to an object in memory and this object is stored in a variable that may be user-controlled, making it susceptible to exploits. In this respect, *Python* also had some troubles dealing with object serialization using Pickles library [45].

Dealing with files can lead to remote file inclusion (RFI) and local file inclusion (LFI) [46]. Improper file name sanitization can allow path traversal [47], common across the spectrum of programming languages for the web. RFI affects more PHP as it has an option to load files from URLs. External files from other websites can be executed and attackers can load a sensitive file as PHP code and gain access to the server. Path Traversal is one of the most dangerous vulnerability in the Web. Improper user input sanitization, can allow the user to read arbitrary files on the server which may include credentials, server code, other sensitive operating system files and ultimately gain server control.

```
extern void bof(char *p1, char *p2) {
    char bof1[16];
    char bof2[16];
    strcpy(bof1, p1);
    strcpy(bof2, p2);
    ES_ASM({
        document.getElementById("XSS").innerHTML = (
            Pointer_stringify($0, $1));
    }, bof1, strlen(bof1));
}
```

Fig. 4. XSS activation from a buffer overflow.

3.5 Vulnerabilities in WebAssembly

The introduction of WebAssembly in the Web language landscape raises additional challenges to security. In particular, with the implementation of a new feature, by definition, the attack surface of browsers has increased. Bugs happen and will certainly happen in WebAssembly, which may give the possibility and the opportunity to exploits. In the last few years, a number of vulnerabilities have already been found in WebAssembly implementations written in C++ (V8, ChakraCore and JavaScriptCore) such as type confusion [48], use after free [49], double-free [50,51], integer overflows [52] and not the specification.

Although these vulnerabilities are widely known, from the technical point of view it is relevant to take a closer look at them because their exploitation is not trivial and diverge from their classic exploitation in the Web. For example, having a look to CVE-2017-5116 [48], a type confusion exploit chain, led to the discovery of the fact that the WebAssembly module was backed by a JavaScript `SharedArrayBuffer`, allowing concurrent modifications of an address in multi-threaded systems. This resulted in a race condition (TOCTOU), in which it was possible to change 1 byte of code before it got JIT-ed but after the check of the WebAssembly module, making it possible to use the code to read and write outside of the WebAssembly sandbox.

Some additional reasons include the fact that there is currently no mechanism for code integrity check. This means that there is no way to verify that a certain WebAssembly application has not been tampered with. Also, despite the security mechanism provided by the compiler and the specification of WebAssembly, has been shown that are possible to import vulnerabilities from languages like C/C++ to the web including buffer overflow and format string [20]. Next, we present some examples of representative WebAssembly vulnerabilities:

Buffer overflow: Buffers are stored within linear memory, very similar to what GCC [53] does. Control flow information is never saved in linear memory, only data is stored there. Which means that the return address is not saved, thereby

making it impossible to hijack the control flow of the program in the classic way. Instruction also cannot be corrupted as they are separated from data. However it is possible to corrupt data stored in linear memory that are adjacent. In the example shown in Figure 4 it is possible to see this type of exploit in action.

In this scenario, we assume that `p1` is hardcoded by the developer and `p2` is a user control variable obtained in a GET or POST command. We can see that if `p2` has more than 16 bytes, it will occur a buffer overflow. Since Emscripten places `bof1` and `bof2` contiguously in linear memory (with possibility of having some bytes between them due to alignment), it results that parts of `bof1` 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. This problem holds by the simple usage of the dangerous function `strcpy`.

Integer overflow: 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 classify 32 and 64 bit floating point data. The integers are not inherently signed or unsigned, it depends on the context that is determined by the individual operations. 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 [23].

However, numeric values in JavaScript are only represented by the *Number* primitive [54] defined with double-precision 64-bit binary format IEE 754-2019 and can take any value between -2^{53} and 2^{53}. On the other hand, a 32-bit integer can take any value between -2^{31} and 2^{31} if signed and up to $2^{33}-1$ unsigned. An integer overflow happens when an operation tries to create a numeric value outside of the range the representation can handle. The different number encodings, different ranges and the interoperability between WebAssembly and JavaScript gives opportunity to the existence of integer overflow. A scenario where there is a counter in JavaScript with the value $2^{33}-1$ and is passed to a WebAssembly function expecting a 32-bit integer that returns the increment of the received integer, would cause the return to be zero.

Integer overflows can be dangerous, the resetting of counters can be one of the consequences, but there are other scenarios where integer overflows can happen in sensitive memory allocation functions and can be leverage to exploit a buffer overflow or heap corruption. In the case of CVE-2018-6551 [55], a call to `malloc` with argument close to the maximum integer the representation can handle, could return a pointer to a heap region smaller than requested, eventually, leading to heap corruption.

Malware: Despite the goal to protect user against malicious applications, attackers still have a lot of opportunities. Cryptocurrency mining has become a popular activity for malicious actors, leveraging a victim's CPU and electricity to make money. There is a prevalence of WebAssembly [56] in this kind of activities, which is normal due to a higher return on performance boost comparing to

JavaScript. It calculates more hashes per second than the same implementation in JavaScript. Also, WebAssembly can be the perfect trojan horse and evade Web application firewall (WAF). With the use of Emscripten, it becomes trivial to obfuscate JavaScript in C and be `eval`-ed at run-time.

Side channel attacks: Another opportunity for exploitation is the usage of side channel attacks. In 2018 in response to the CPU vulnerabilities Spectre and Meltdown [57,58], it was mitigated in browser by lowering the precision of timers and disable `SharedArrayBuffer` in JavaScript [59]. However, in the future is expected that WebAssembly supports threads with shared memory and very accurate timers. That will make current browser mitigations of certain CPU side channel attacks useless.

4 Related Work

The existence of a new source of vulnerabilities in Web applications due to the usage of WebAssembly motivates our work, which aims at building adequate tools for detecting and mitigating WebAssembly vulnerabilities. In this section, we discuss the related work. We begin by providing a birds-eye view on the two main approaches for verifying code: dynamic and static analysis. Then, we concentrate on a specific static analysis technique that we find to be most promising for detecting vulnerabilities in WebAssembly: *code property graphs*. Lastly, we present some existing efforts in analysing WebAssembly code.

4.1 Dynamic and Static Analysis

There are many approaches and techniques for detecting vulnerabilities. Depending on their nature, existing techniques can be classified as *dynamic* or *static* [60]. Dynamic analysis is where common errors that lead to vulnerabilities are checked at runtime. Dynamic analysis tools try to execute the program over some inputs and observe its execution while static analysis tools try to build a model of the program state in order to reason over possible behaviors. Static analysis of code checks common programming practices, errors, and omissions by scanning the source code or an intermediate representation.

When compared with each other, we can highlight some key differences. Dynamic analysis can be: fast, if executed over a test suite, but slow if exhaustive; precise as there is no abstraction or approximation; and unsound, meaning that results may not generalize for future runs. In contrast, static analysis can be: slow, depending how well it tries to approximate; usually follows a conservative approach, thereby is sound. However, due to imperfect models, static analysis often produce false positives, hence the necessity of human analysis. Table 2 summarizes the explained techniques. Next, we briefly introduce some known techniques for each realm of program analysis approaches.

Dynamic Analysis	Static Analysis
Testing	Lexical
Fuzzing	Flow Analysis
Taint-based fuzzing	Graph analysis
Symbolic execution	Code property graph

Table 2. Dynamic and static analysis techniques respectively.

Dynamic analysis: Dynamic approaches can be split in two main categories: *concrete* and *symbolic execution*. The most relevant applications of dynamic concrete execution are: *fuzzing* [61,62], in which malformed input is provided in an attempt to trigger a crash, and *taint-based fuzzing* [63,64] which analyzes how the program processes input in order to understand what parts of the input should be modified in future runs. Unfortunately, we can only find vulnerabilities that are triggered during execution. This means that unusual and rare paths may unveil security flows that are not discovered by those techniques. Taint-based fuzzer can understand what part of input to change but not how to change it. To counteract this result, another approach is symbolic execution, which bridges the gap between static and dynamic analysis, providing a solution to cope with the limited semantic insight of fuzzing.

There is a lot of work in *symbolically-guided fuzzers* [61,65,66,67], which modify inputs identified by the fuzzing component by processing them in a dynamic symbolic execution engine. This is a costly technique as its execution is made over the abstract domain of symbolic variables and track the state of memory and constraints on those variables throughout the program execution, whenever there is a branch, execution forks and follows both paths. This results in a potentially exponential number of paths to consider.

Static analysis: Within static analysis, there are a multitude of approaches. One is *lexical* analysis, scanning the source code for patterns or abstract the syntax. It was one of the first techniques to find vulnerabilities and there is a long list of vulnerability scanners used in practice as it is the case of PScan [68], Microsoft PREfast [69] and JSLint [70], enforcing good programming practices. Those are valuable tools, specially in development environment, however, they fail when looking for more complex and subtle vulnerabilities. One extension to this technique is allowing analysts to provide annotations in code. Splint [71] finds potential vulnerabilities by checking to see that source code is consistent with the properties implied by annotations. Microsoft extended PREfast with annotations using macros and other C/C++ preprocessor directives. Its drawback becomes visible as the code base increases, it is hard to scale because it needs manually annotated code.

Another program analysis technique is *taint analysis*, variables controlled by the user are marked as tainted. From that point on, it traces them to possible vulnerable functions called sinks, if tainted values reach the sinks without

proper sanitation, the analyzer flags a vulnerability. Huang et. al [72] were the first to introduce a lattice-based algorithm derived from type systems and type-state in the context of PHP code, creating a tool named WebSSARI. One of its limitations is the impossibility of inter-procedural analysis. To address this limitation, new techniques were proposed based in data flow analysis where is collected information about the way variables behave, and gather information about the possible set of values that can be assigned to them. RIPS [73] is a tool that performs fine-grained analysis of the interaction between different types of sanitization, encoding, sources and sinks also in PHP. This approach is highly tight to the language in question.

There were several attempts focusing certain vulnerabilities in the Web provided by PHP applications, as is the case of SQLi and XSS. Xie and Aiken presented an algorithm to find SQLi using block and function summaries [74]. At the same time it was also presented Pixy [75] written in Java focusing also in XSS, using flow-sensitive, interprocedural and context-sensitive data flow analysis. Balzarotti et al. [76] extended Pixy and named their solution Saner improving the detection of user-defined sanitization by providing predefined test-cases to check sanitization routines. There is also some work in string analysis for checking the correctness of sanitization routines without the use of dynamic analysis. The main drawback of Saner is that is only good as their test-cases are. For instance, if it is configured to a certain pattern, other attack pattern could exist that bypass the sanitization undetected.

Alternatively, one can use flow analysis as Pixy. Control flow determines the flow and possible paths that program can take by explicitly describing the execution order of each instruction. Data flow determines the flow of data and its dependencies. It relies on building a model of a bug, represented by a set of nodes in a control-flow or dependency graph and identify bugs through traversals of the model. Control-flow graph has become a standard in activities such reverse engineering in an attempt to understand better the program. Moreover, it has been an useful tool to aid in detect variants of malicious applications [77] and guide fuzz-testing tools [78]. However, control-flow graphs do not easily identify statements that process data influenced by an attacker as it has no information regarding data flow.

Ferrante et al. [79] introduced the concept of program slicing called *program dependence graph*. It exposes all statements and predicates of a program and make possible to statically analyze the propagation of attacker-controller data. It is structured in two types of edges: control dependency which indicate that execution of a statement depends on a predicate and data dependency which indicate that a variable defined at the source statement is subsequently used at the destination statement. Data dependencies are calculated by solving reaching definitions, a known problem in data-flow analysis [80].

4.2 Code Property Graphs

In an attempt to aggregate all information in a graph, Yamaguchi et al. [21] published the original idea of *code property graph* (CPG) aimed at searching for

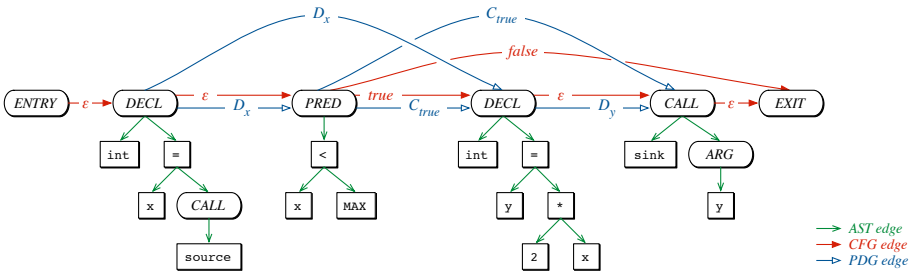


Fig. 6. Code property graph for code sample in Figure 5

vulnerabilities in programs. Currently, CPG is the main component of the only product offered by ShiftLeft, they claim that it can inspect and analyze up to 500K lines of source code in less than 10 minutes [81] with high level of accuracy compared with other state-of-the-art tools [82]. CPG can also be employed to identify code weakness, such as methods with too many parameters, improperly sanitized inputs, duplicate code and inconsistent name conventions present in the construction of code. It can have similar to the current lints in the market.

Technical approach: CPG aggregate 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 analyzed code and vulnerabilities become easier to identify. More specifically, CPG tries to gain joint power of the source's syntax, control-flow and data-flow information in one graph stored in an 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 6 represents the CFG data structure for the code sample depicted in Figure 5. Once the CPG has been generated, the search for vulnerabilities in the program can be translated into querying for certain patterns via graph-database queries, e.g., checking the existence of some path in the graph connecting nodes that represent data sources and sinks.

```
void foo() {
    int x = source();
    if (x < MAX) {
        int y = 2 * x;
        sink(y);
    }
}
```

Fig. 5. Exemplary C code sample.

In its original idea, CPG does not support inter-procedural analysis. The authors extended the work one year later to support it [83], employing a similar representation of the well known System Dependence Graph [84]. They extended the CPG by explicitly defining data-flow between call sites and their callees intro-

ducing 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* [85], 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.

At the same time, Backes et al. [86] applied the concept of code property graph in PHP code, and extended it to support inter-procedural analysis. They merged another graph named *call graph*. Call graphs are just directed graphs whose links connect nodes between the call sites of functions and the corresponding function definition, allowing to reason about control and data flow between functions, i.e., at the inter-procedural level.

Adopting CPG for WebAssembly: The CPG data structure can be seen as liberal definition, as it asks only for certain structures to be merged while leaving the graph schema open. In consequence, when someone creates this graph to a targeting programming language, different implementations result in an highly different graphs between them. Different languages usually have different syntax and different semantics thus, generates different code property graphs and also require different queries when searching for patterns.

Consider the example presented in Table 3 by applying code property graph in WebAssembly code that had been compiled from a C source file. The syntax is clearly different thus, the generated graph will have different nodes. The first program shows an insecure argument, vulnerable to format string. Being vulnerable to a format string does not necessarily mean that the exploit can be triggered. If the string passed as an argument was not user-controlled, it can never trigger this vulnerability. However, we can flag it because there are few reasons to ever supply a non-constant string to the `printf` function. The simple check is to search nodes which `call $printf` with a non-constant first argument. It can trivially be done for the C source code as for the WebAssembly's code.

This is a task that a scanner could flag, however, if we would like to do a more fine-grained search using data dependencies and model an attacker controller variable, a simple scanner analysis will not do. Instead we can model an attacker controlled-variable using code property graph. In the first case displayed in Table 3, both in C as in WebAssembly, the code property graph would give us a direct dependence between the call and the given parameters, provided by the program dependence graph. Yet, while in the second case the complexity in the C code is the same, in the WebAssembly is not. In the case of WebAssembly, the compiler stores the pointer given as a parameter in a memory address in linear memory (index 70240) alongside with constant string in another position (index 1024). Both indexes (can be seen as pointers to memory addresses) are the ones given as parameters to `printf`. The code property graph does not give that direct dependency between the variable which is user-controller (`msg`) and

C program code	Text Representation

`void log(char* msg) {`
`  printf(msg);`
`}`
	`(func $log (param $msg i32)`
	`  get_local $msg`
	`  i32.const 0`
	`  call $printf`
	`  drop)`

`void log(char* msg) {`
`  printf("%s\n",msg);`
`}`
	`(func $log (param $msg i32)`
	`  i32.const 70240`
	`  get_local $msg`
	`  i32.store`
	`  i32.const 1024`
	`  i32.const 70240`
	`  call $printf`
	`  drop)`

Table 3. Two printf wrapper functions: one vulnerable to format string (first row) and the other not (second row). On the right it is the corresponding Wasm text format.

the one that is given as input to the sink (`printf`). To the best of our knowledge no single work to date has focused on CPG generation for WebAssembly code.

4.3 WebAssembly Analysis

From a research and analysis point of view, WebAssembly is significantly more laborious language to analyze comparing to high-level language such JavaScript due to the binary format. Binary analysis has been known as an hard challenge, asking "will it crash?" is no different from the halting problem [87]. Bugs are not created equal. It depends on the scope, many tools analyze a specific scope of the program or widen the analysis to the whole application. Low semantics or the lack of it, hardens the problem and the ability to reason about the program. Many challenges in classical binary analysis suffer from cross-platform dependence as each processor has its set of instruction set. Luckily, WebAssembly is different in that way, since it is meant to run in multiple platform. Moreover, compilers are not bug free and can introduce new bugs not present in the source code. Also, different compilers generate different WebAssembly code from the same source file, consequently generating different code patterns.

As a result of being a rather new technology, there are not many available tools for analyzing binaries and text format in WebAssembly. Analogously, there is a lack of documentation to allow an efficient and easy way to analyze 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.

Browsers and embedded debuggers: In the front-line are the browsers and the embedded debuggers. They take advantage of what already exists for JavaScript – *source maps*. The compiler used to generate WebAssembly code must also generate debug information in a source map format. Emscripten 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.

External runtimes: Outside of the browser, Wasmtime [33], which is a standalone runtime for WebAssembly, and WASI [17], takes advantage of a modern, high-performance debugger in the LLVM project that is built on top of Clang/LLVM called LLDB. This allows developers to examine their programs execution and makes it easier to catch and diagnose bugs. Certain bugs classes may arise in WebAssembly and not in a native build of the same code. These two tools can be handful to develop code. However, it is required to have the source code. When delivering code to the client, specially in the browser, the source code is not made available, only the compiled binary. To analyze raw WebAssembly binary, one needs reverse engineering tools that produce WebAssembly textual form from WebAssembly binary.

Reverse engineering tools: We describe two tools that include reverse engineering of Wasm code. The first is maintained by the official WebAssembly Community Group called WebAssembly Binary Toolkit (WABT) [88]. It includes tools to validate modules, convert WAT files to Wasm files and vice-versa, convert Wasm files to a C source and header, print information about the binary, remove sections of a binary file and an interpreter which decodes and runs WebAssembly binary file using a stack based machine. In the same direction, there is already an IDA plugin named *idawasm* [89] supporting loading and disassembling of WebAssembly modules. Another dynamic analysis tool is Wasabi [90]. It takes the WebAssembly binary and does instrumentation using 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, call graph analysis, dynamic taint analysis, cryptominer detection through the frequence of certain instructions (e.g. `xor`, `add` and `mul`) and memory access tracing. 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. Most existing tooling is made for dynamic and manual analysis.

5 Architecture

This section introduces Wasmati, our proposed tool for analyzing vulnerabilities in WebAssembly code. We begin by providing an overview of our tool, and then discuss some of its most interesting technical details.

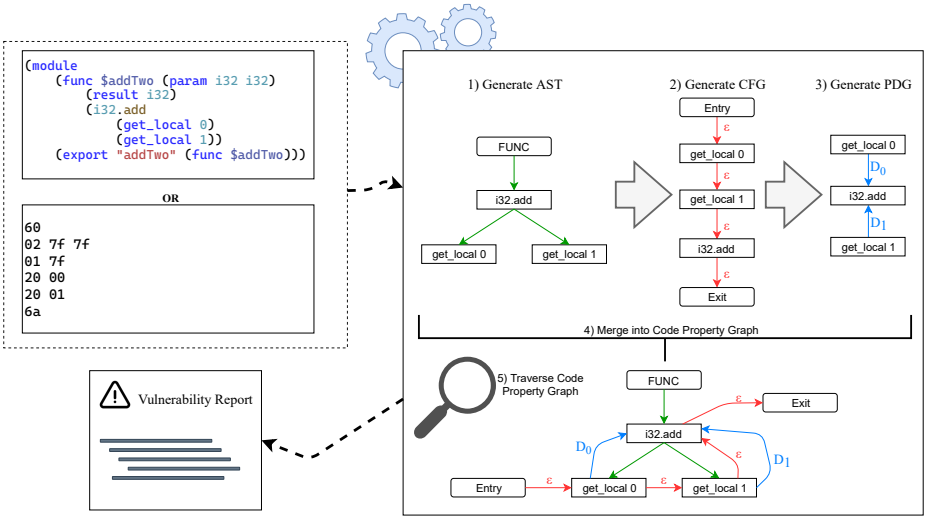


Fig. 7. Wasmati system architecture.

5.1 Overview

Figure 7 represents the general architecture of Wasmati, the system we propose to develop for finding vulnerabilities in WebAssembly. It incorporates different stages which will each be responsible for a specific task:

1. AST generation: The first stage as represented in Figure 7 by the number 1, is to parse a valid binary or textual format file and generate an *abstract syntax tree* (AST) which is a representation of the structure of source code.

2. CFG generation: The second stage is made doing several traverses in the abstract syntax tree to generate the *control flow graph* (CFG), which explicitly describes the order by which instruction are executed as well conditions that are necessary for a particular path of execution.

3. PDG generation: For the third stage and with the information gathered with the previous graphs, we will generate the *program dependence graph* (PDG), that explicitly represents dependencies of data and flow between instructions.

4. CPG generation: Having the three graphs available, the code property graph is generated by merging all previous graphs. At this point can be made a decision to also merge other structures such call graph or add link between the arguments of the call site and the parameters in the function definition similar to what was explained in Section 4.2, this will allow inter-procedural analysis. Some nodes will be added and redundant ones are removed. For common nodes, links are added according to the existing links in each graph.

5. CPG traversal: For the fifth and final stage, several traversals will be done to perform the queries and output a vulnerability report. A query represents the way of traverse of the graph in the pursue of certain properties of the underlying software. CPG allows the expression of multiple programming patterns. Several well-known types of vulnerabilities can be identified by specific queries that translate to a specific traverse of the graph in search of certain properties. When such property is found, it is flagged as a vulnerability. The system can have a set of predefined queries configured in advance or, have the CPG generated, receive as input a set of other queries and perform them over the code property graph.

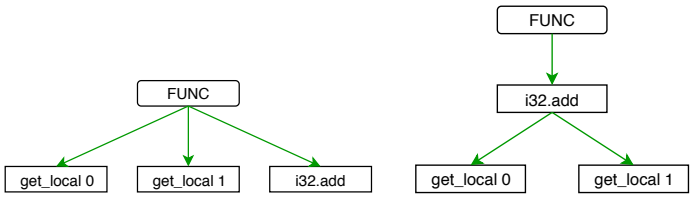
5.2 Challenges in CPG Generation and Traversal

There are two big challenges to solve in order to generate a CPG for WebAssembly code. One of them is when generating the abstract syntax tree and the other is generating the program dependence graph. Next, we discuss these issues along with additional other challenges, e.g., in the traversal of CPG.

Challenges in generating the AST: The abstract tree is the first representation and is directly produced by parser of a compiler. It is a field already well investigated [80]. However, as explained below, for being a low-level language, WebAssembly loses some expressiveness when compared with the high-level ones.

The binary encoding 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 8(a). Luckily, one of the properties of WebAssembly is being a stack based machine. This property lets us take one simple instruction (like add) and express as a composition of other simple instructions. For example, the instruction `i32.add` needs two arguments and they must be in the stack when this instruction is executed. The control flow graph in Figure 7 shows what instructions are executed and the order in which are executed and encodes perfectly how it is in the binary format. Nevertheless, we can fold the instructions and represent as they are in the textual format and shown in the abstract syntax tree in Figure 8(b). The challenge generating the abstract syntax tree comes by folding a set of sequential instructions into an instruction composed of other expressions, which is, converting from Figure 8(a) to Figure 8(b). This additional work is important as it adds expressiveness and facilitates further search of certain patterns as well finding dependencies between instructions.

Challenges in generating the PDG: In generating the program dependence graph, it is straightforward to see that an instruction that is composed of other instructions has, by definition, a dependence with the other instructions. Finding dependencies between instruction is direct. However, there are some difficult challenges finding data dependencies through the flow of the program. For instance, we have the code `a=userInput(); b=a; c=b;`, we must derive that the third instruction depend on the first because `c` is dependent of value stored in `a`. To succeed, firstly, using the control flow graph previous generated, it is



(a) Flat AST (b) Folded AST

Fig. 8. AST representation.

necessary to find the set of variables defined and the set of variables used by each instruction and calculating reaching definitions needs to formally define the transfer functions for each instruction. There is not yet a work in reaching definitions for the WebAssembly language, which means it will be necessary to formally define them.

Challenges in CPG traversal and others: We will also have to define new traversals in CPG. Code property graphs are different for each language and present different code patterns. Different compilers generate different code and different patterns, there is a need to study the code generated by each one of them. Some traversals can be directly applied by what is implemented in other languages like C, however, some are not trivial. The example in Table 3 is a good example of it, the data-flow generated by the code property graph is not enough to do a proper data-flow analysis in the code. We must try to extend the code property graph to address indirect data dependencies statically. There are other challenges that we are not yet total aware of and need further investigation, specially the techniques explained in Section 4.2 on post-dominator trees.

5.3 Detecting Multiple Types of Vulnerabilities

Even thought many vulnerabilities are mitigated at runtime by the Wasm interpreter, Wasmati may capture many errors that can lead later to failures. Using our tool, users should be able to detect multiple types of vulnerabilities in Wasm code. Design errors, integer overflows, integer type issues, division by zero, buffer overflows, among others can be caught early in the development workflow and enhance security and efficiency. This means that we should be able to represent such vulnerabilities in terms of queries to be performed to the CPG of the target program.

In our work, we will first focus on the detection of certain well known vulnerabilities from C/C++ that can be imported to WebAssembly and it context. (Note that not all vulnerabilities present in C/C++ code become a vulnerability when compiled down to WebAssembly.) The most common vulnerability is buffer overflow. Despite some safety mechanism present in the specification of the language, it is still possible to corrupt data in the linear memory. Other well

known vulnerability is integer overflow. All number in JavaScript are represented as a 64 bits double precision, and due to the interoperability between JavaScript and WebAssembly can lead to integer overflows. Taint-style vulnerabilities are also a big concern, specially when we talk about the Web. Finding sources and sinks and lack of sanitization of data can lead to prohibit further attacks like cross-site scripting and code injection.

5.4 Integration with JavaScript

WebAassembly and JavaScript work interchangeably, where each one calls functions on each other. The compiler Emscripten, for instance, emits a combination of both languages and not only WebAssembly. Analysing data and control flow in such environment becomes challenging as there are two domains where the data and control can be in each instant of the execution. The target function can either be in WebAssembly or in JavaScript and belong also to different modules. The construction of a code property graph takes in consideration the granularity of given code, in case of WebAssembly, the largest is module granularity. However, after created, it is possible to merge two code property graphs if they are in the same domain and semantics by just adding edges to link them. It becomes fairly easy to glue two WebAssembly modules.

Problems arise when the target function is in JavaScript. Gluing a CPG generated from JavaScript code with one generated from WebAssembly is not trivial as both graphs are intrinsically different, not just in the domain, but also in the semantics of each node and property on it. The first approach to handle this problem is to treat the function as a black box. We know by advance which parameters the function takes and what type it returns as the function signature must be specified in the WebAssembly mode when it is imported. For future work, would be desirable to engineer a solution to integrate a code property graph of WebAssembly's code with a code property graph of JavaScript's code.

5.5 Implementation

We propose to implement Wasmati by taking full advantage of the WebAssembly Binary Kit (WABT for short), developed by the official WebAssembly team. It has robust tools to parse and compile WebAssembly whether it is in binary format (wasm) or in textual format (wat). We will try to reuse that functionality to produce the AST, CFG, and PDG, and merge them into a joint data structure which is the expected result of a CPG. After that process, a set of queries will be performed over the code property graph in order to find vulnerabilities. In practice, the queries will be translated to multiple traverses of the graph in the pursue of certain properties, that are identified as vulnerabilities.

Given that WABT is developed in C++, for simplicity, Wasmati will also be developed in C++. This brings many opportunities to employ this tool. Nowadays, almost every platform supports C++ and can even be compiled to the Web using WebAssembly and run in the browser. Given the tool's compatibility with many platforms, it can be deployed in various models of execution. It can

be deployed as a stand alone program, or integrated with some package/library or even be deployed as a service where everyone can access it and run. As it can be compiled down to WebAssembly, it is possible to create a plugin that can be installed in the browser and verify code on-the-fly while browsing, before any code execution, preventing possible faults/errors.

6 Evaluation

In our evaluation, our main goal is to assess the precision of our tool. Our approach starts by verifying real world applications with known vulnerabilities and get them as output and also find other public repositories (that can compile to WebAssembly) and output vulnerabilities that are possibly unknown. We intend to crawl providers of code repositories such as GitHub, GitLab, among others and download, compile and analyze. We intend to evaluate the Wasmati prototype, we focus in three different dimensions:

Vulnerability coverage: 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. Despite both being problematic, having false negatives is much more dangerous because they lead to a false sense of security. So we aim to a sound tool, where there are no false negatives. High percentage of false positives is also problematic because, eventually, leads to 100 percent of false nevatives which is what we get when people stop using the tool.

Performance of the tool: For performance, given the size of the input, we will measure the time it takes to process and identify vulnerabilities by testing the system against a large and representative collection of programs.

Involvement of the developer: Since the tool is to be used in the workflow of the developer, we must assess how it integrates with it. It can also be useful to assess how difficult is to apply the tool to wasm code in the wild, including dynamically loaded wasm code.

7 Scheduling of Future Work

Future work is scheduled as follows:

- January - March: Detailed design and implementation of the proposed architecture, including preliminary tests.
- April - May: Perform the complete experimental evaluation of the results.
- June: Write a paper describing the project.
- July - August: Finish the writing of the dissertation.
- September: Deliver the MSc dissertation.

8 Conclusions

WebAssembly is a new technology that allows web developers to run native C/C++ on a webpage 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 buffer and integer overflows. These vulnerabilities can be leveraged by an adversary for performing exploits in the Web, such as cross-site scripting and SQL injection. We explained our approach using code property graph, and how it fits within the current research on security analysis for WebAssembly. Finally we presented a detailed plan in how to achieve the goals, to structure the different sub-tasks involved in the project, and the evaluation methods to evaluate the resulting tool.

References

1. Auler, R., Borin, E., Halleux, P., Moskal, M., Tillmann, N.: Addressing javascript jit engines performance quirks: A crowdsourced adaptive compiler. In: International Conference on Compiler Construction. Volume 8409., Springer (04 2014) 218–237
2. Selakovic, M., Pradel, M.: Performance issues and optimizations in javascript: an empirical study. In: Proceedings of the 38th International Conference on Software Engineering, ACM (2016) 61–72
3. Systems, A.: Adobe Flash Player. https://get.adobe.com/flashplayer/about/ Accessed: 2019-12-28.
4. Oracle: Java Applets. https://www.oracle.com/technetwork/java/applets-137637.html Accessed: 2019-12-28.
5. Microsoft: Introduction to ActiveX Controls. https://msdn.microsoft.com/en-us/library/aa751972(VS.85).aspx Accessed: 2019-12-28.
6. Metz, C.: Google Native Client: The web of the future - or the past? https://www.theregister.co.uk/2011/09/12/google_native_client_from_all_sides/ Accessed: 2019-12-28.
7. CVE: Adobe Flash Player. https://www.cvedetails.com/vulnerability-list/vendor_id-53/product_id-6761/Adobe-Flash-Player.html Accessed: 2019-11-27.
8. Wressnegger, C., Yamaguchi, F., Arp, D., Rieck, K.: Comprehensive analysis and detection of flash-based malware. In: Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721. DIMVA 2016, Berlin, Heidelberg, Springer-Verlag (2016) 101–121
9. Chromium Blog: Goodbye PNaCl, Hello WebAssembly! https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html (05 2017) Accessed: 2019-11-27.
10. Microsoft: A break from the past, part 2: Saying goodbye to ActiveX, VBScript, attachEvent.... https://blogs.windows.com/msedgedev/2015/05/06/a-break-from-the-past-part-2-saying-goodbye-to-activex-vbscript-attachevent/ (05 2015) Accessed: 2019-11-27.
11. Adobe Corporate Communications: Flash & The Future of Interactive Content. https://theblog.adobe.com/adobe-flash-update/ (07 2017) Accessed: 2019-11-27.
12. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with webassembly. SIGPLAN Not. **52**(6) (June 2017) 185–200

13. Jangda, A., Powers, B., Guha, A., Berger, E.: Mind the gap: Analyzing the performance of webassembly vs. native code. CoRR **abs/1901.09056** (2019)
14. Deveria, A.: Can I use WebAssembly? https://caniuse.com/#feat=wasm Accessed: 2019-12-17.
15. Bridgewater, R.: Node v12.3.0. https://nodejs.org/en/blog/release/v12.3.0/ (05 2019) Accessed: 2019-12-17.
16. Wasmer: An open-source runtime for executing WebAssembly on the Server. https://wasmer.io/ Accessed: 2019-12-17.
17. Clark, L.: Standardizing WASI: A system interface to run WebAssembly outside the web. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/ (03 2019) Accessed: 2019-12-17.
18. Hall, A., Ramachandran, U.: An execution model for serverless functions at the edge. In: Proceedings of the International Conference on Internet of Things Design and Implementation. IoTDI '19, New York, NY, USA, ACM (2019) 225–236
19. Hickey, P.: Announcing Lucet: Fastly's native WebAssembly compiler and runtime. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime (03 2019) Accessed: 2019-12-17.
20. McFadden, B., Lukasiewicz, T., Dileo, J., Engler, J.: Security chasms of wasm. BlackHat US-18 (August 2018)
21. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. Proceedings - IEEE Symposium on Security and Privacy (05 2014)
22. Mozilla: An extraordinarily optimizable, low-level subset of JavaScript. http://asmjs.org/ Accessed: 2019-12-02.
23. IEEE: IEEE standard for floating-point arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) (July 2019) 1–84
24. Pyodide: The Python scientific stack, compiled to WebAssembly. https://github.com/iodide-project/pyodide Accessed: 2019-12-02.
25. Microsoft: Blazor. https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor Accessed: 2019-12-01.
26. Steve Akinyemi: awesome-wasm-langs. https://github.com/appcypher/awesome-wasm-langs Accessed: 2019-12-02.
27. Zakai, A.: Why WebAssembly is Faster Than asm.js? https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/ (03 2017) Accessed: 2019-12-17.
28. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, IEEE Computer Society (2004) 75
29. Alliance, B.: New Bytecode Alliance Brings the Security, Ubiquity, and Interoperability of the Web to the World of Pervasive Computing. https://bytecodealliance.org/press/formation (11 2019) Accessed: 2019-12-17.
30. Group, W.C.: WebAssembly: Text Format. https://webassembly.github.io/spec/core/text/index.html# Accessed: 2019-12-28.
31. LLVM: Clang: a C language family frontend for llvm. https://clang.llvm.org/
32. Emscripten: An Open Source LLVM to JavaScript and WebAssembly compiler. https://emscripten.org Accessed: 2019-12-01.
33. Alliance, B.: Wasmtime: a WebAssembly Runtime. https://wasmtime.dev/ Accessed: 2019-12-17.
34. Group, W.C.: WebAssembly: Security. https://webassembly.org/docs/security/#users Accessed: 2019-12-28.

35. Deckard, J.: Buffer overflow attacks: detect, exploit, prevent. Elsevier (2005)
36. Vallentin, M.: On the evolution of buffer overflows. Munich, May (2007)
37. Group, W.C.: WebAssembly: Memory Safety. https://webassembly.org/docs/security/#memory-safety Accessed: 2019-12-28.
38. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX Security Symposium. Volume 98., San Antonio, TX (1998) 63–78
39. Microsoft: Data Execution Prevention. https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention Accessed: 2019-12-28.
40. Spengler, B.: Pax: The guaranteed end of arbitrary code execution. G-Con2: Mexico City, Mexico (2003)
41. OWASP: OWASP: Code Injection. https://www.owasp.org/index.php/Code_Injection Accessed: 2019-12-28.
42. CWE: CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). http://cwe.mitre.org/data/definitions/89.html Accessed: 2019-12-28.
43. CWE: CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'). https://cwe.mitre.org/data/definitions/79.html Accessed: 2019-12-28.
44. CWE: CWE-352: Cross-Site Request Forgery (CSRF). https://cwe.mitre.org/data/definitions/352.html Accessed: 2019-12-28.
45. CWE: CWE-502: Deserialization of Untrusted Data. https://cwe.mitre.org/data/definitions/502.html Accessed: 2019-12-28.
46. CWE: CWE-98: Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion'). https://cwe.mitre.org/data/definitions/98.html Accessed: 2019-12-28.
47. CWE: CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'). https://cwe.mitre.org/data/definitions/22.html Accessed: 2019-12-28.
48. CVE: CVE-2017-5116. https://nvd.nist.gov/vuln/detail/CVE-2017-5116 (10 2017)
49. Bugs, C.: Issue 1819: Chrome: Use-after-free in wasmmemoryobject::grow. https://bugs.chromium.org/p/project-zero/issues/detail?id=1819 (04 2019)
50. Bugs, C.: Issue 766253: Chrome os exploit: Webasm, site isolation, crosh, crash reporter, cryptohomed. https://bugs.chromium.org/p/chromium/issues/detail?id=766253 (09 2017)
51. Bugs, C.: Issue 1522: Webkit: Webassembly parsing does not correctly check section order. https://bugs.chromium.org/p/project-zero/issues/detail?id=1522 (01 2018)
52. Bugs, C.: Issue 1793: Chrome: Integer overflow in newfixeddoublearray. https://bugs.chromium.org/p/project-zero/issues/detail?id=1793 (03 2019)
53. Griffith, A.: GCC: the complete reference. McGraw-Hill, Inc. (2002)
54. International, E.: ECMA-262 - ECMAScript® 2020 Language Specification. https://tc39.es/ecma262/#sec-ecmascript-language-types-number-type (December 2020) Accessed: 2019-12-16.
55. CVE: CVE-2018-6551. https://nvd.nist.gov/vuln/detail/CVE-2018-6551 (02 2018)
56. Musch, M., Wressnegger, C., Johns, M., Rieck, K.: New kid on the web: A study on the prevalence of webassembly in the wild. In: International Conference on

Detection of Intrusions and Malware, and Vulnerability Assessment, Springer (06 2019) 23–42
57. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. CoRR **abs/1801.01203** (2018)
58. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown. arXiv preprint arXiv:1801.01207 (2018)
59. Luke Wagner, M.S.B.: Mitigations landing for new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/ Accessed: 2019-12-10.
60. Ernst, M.D.: Static and dynamic analysis: Synergy and duality. In: WODA 2003: ICSE Workshop on Dynamic Analysis, New Mexico State University Portland, OR (2003) 24–27
61. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: whitebox fuzzing for security testing. Communications of the ACM **55**(3) (2012) 40–44
62. Ganesh, V., Leek, T., Rinard, M.: Taint-based directed whitebox fuzzing. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society (2009) 474–484
63. Newsome, J., Song, D.: Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In: Proceedings of the 12th Network and Distributed Systems Security Symposium, Citeseer (2005)
64. Bekrar, S., Bekrar, C., Groz, R., Mounier, L.: A taint based approach for smart fuzzing. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, IEEE (2012) 818–825
65. Caselden, D., Bazhanyuk, A., Payer, M., Szekeres, L., McCamant, S., Song, D.: Transformation-aware exploit generation using a hi-cfg. Technical report, California Univ Berkeley Dept of Electrical Engineering and Computer Science (2013)
66. Cha, S.K., Woo, M., Brumley, D.: Program-adaptive mutational fuzzing. In: 2015 IEEE Symposium on Security and Privacy, IEEE (2015) 725–741
67. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS. Volume 16. (2016) 1–16
68. DeKok, A.: PScan: A limited problem scanner for C source files. http://deployingradius.com/pscan/ Accessed: 2019-12-17.
69. Larus, J.R., Ball, T., Das, M., DeLine, R., Fahndrich, M., Pincus, J., Rajamani, S.K., Venkatapathy, R.: Righting software. IEEE software **21**(3) (2004) 92–100
70. Crockford, D.: JSLint, The JavaScript Code Quality Tool. http://JSLint.com/ Accessed: 2019-12-17.
71. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE software **19**(1) (2002) 42–51
72. Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing web application code by static analysis and runtime protection. In: Proceedings of the 13th international conference on World Wide Web, ACM (2004) 40–52
73. Dahse, J., Holz, T.: Simulation of built-in php features for precise static code analysis. In: NDSS. Volume 14., Citeseer (2014) 23–26
74. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: USENIX Security Symposium. Volume 15. (2006) 179–192
75. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities. In: 2006 IEEE Symposium on Security and Privacy (S&P'06), IEEE (2006) 6–pp

76. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: Composing static and dynamic analysis to validate sanitization in web applications. In: 2008 IEEE Symposium on Security and Privacy (sp 2008), IEEE (2008) 387–401
77. Gascon, H., Yamaguchi, F., Arp, D., Rieck, K.: Structural detection of android malware using embedded call graphs. In: Proceedings of the 2013 ACM workshop on Artificial intelligence and security, ACM (2013) 45–54
78. Sparks, S., Embleton, S., Cunningham, R., Zou, C.: Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), IEEE (2007) 477–486
79. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS) **9**(3) (1987) 319–349
80. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, technologies, and tools (2006)
81. ShiftLeft: Analyze Your Code in Less than 10 Minutes. https://www.youtube.com/watch?v=VaxSi1yC9mo Accessed: 2019-12-17.
82. ShiftLeft: OWASP SAST Benchmark. https://www.shiftleft.io/resources/whitepapers/OWASP-SAS-Benchmark-Whitepaper.pdf Accessed: 2019-12-17.
83. Yamaguchi, F., Maier, A., Gascon, H., Rieck, K.: Automatic inference of search patterns for taint-style vulnerabilities. In: 2015 IEEE Symposium on Security and Privacy, IEEE (2015) 797–812
84. Horwitz, S., Reps, T.: Binkley, inter procedural slicing using dependence graphs. In: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation Atlanta, GA. (1988) 25–46
85. Cooper, K.D., Harvey, T.J., Kennedy, K.: A simple, fast dominance algorithm. Software Practice & Experience **4**(1-10) (2001) 1–8
86. Backes, M., Rieck, K., Skoruppa, M., Stock, B., Yamaguchi, F.: Efficient and flexible discovery of php application vulnerabilities. In: 2017 IEEE european symposium on security and privacy (EuroS&P), IEEE (2017) 334–349
87. Alan, M.: Turing. on computable numbers, with an application to the entscheidungsproblem. Proceedings of the London mathematical society **42**(2) (1936) 230–265
88. Group, W.C.: The WebAssembly Binary Toolkit. https://github.com/WebAssembly/wabt Accessed: 2019-12-17.
89. FireEye: IDA Pro loader and processor modules for WebAssembly . https://github.com/fireeye/idawasm Accessed: 2019-12-17.
90. Lehmann, D., Pradel, M.: Wasabi. Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '19 (2019)