Islamic University of Technology (IUT)

Department of Computer Science and Engineering (CSE)

Vulnerability Analysis of WebAssembly Binaries

Authors

Farhan Saif (180041124)

Shihab Sikder (180041132)

Adib Abrar Kabeer (180041134)

Supervisor

Dr. Md Moniruzzaman

Assistant Professor,

Department of CSE

Co-Supervisor

Imtiaj Ahmed Chowdhury

Lecturer,

Department of CSE

A thesis submitted to the Department of CSE

in partial fulfillment of the requirements for the degrees of B.Sc.

Engineering in CSE

Academic Year: 2021-22

May - 2023

Declaration of Authorship

This is to certify that the work presented in this thesis is the outcome of the analysis and experiments carried out by Farhan Saif, Shihab Sikder and Adib Abrar Kabeer under the supervision of Dr. Md Moniruzzaman, Assistant Professor of the Department of Computer Science and Engineering (CSE), Islamic University of Technology (IUT), Dhaka, Bangladesh. It is also declared that neither this thesis nor any part of it has been submitted anywhere else for any degree or diploma. Information derived from the published and unpublished work of others has been acknowledged in the text, and a list of references is given.

Authors:

Farhan Saif

Farhan Saif
Student ID - 180041124

Shihab

Shihab Sikder
Student ID - 180041132

Adib.

Adib Abrar Kabeer
Student ID - 180041134

Approved By:

Co-supervisor:

[illegible] 27.05.23

Imtiaj Ahmed Chowdhury

Lecturer

Department of Computer Science and Engineering

Islamic University of Technology (IUT)

Supervisor:

[illegible]

Dr. Md Moniruzzaman

Assistant Professor

Department of Computer Science and Engineering

Islamic University of Technology (IUT)

Abstract

The evolution of web technologies has brought forth innovative coding structures, among which JavaScript and WebAssembly stand out prominently. This paper presents Wasmosys, a state-of-the-art source code analyzer designed to generate and unify Abstract Syntax Trees (ASTs) for JavaScript and WebAssembly code. It aims to pave the way towards advanced vulnerability detection and mitigation in these modern web environments.Wasmosys tackles two major challenges: creating a seamless combination of separate ASTs and standardizing AST labels for JavaScript and WebAssembly. The system comprises four primary modules. The first two modules, written in JavaScript and C respectively, generate ASTs from JavaScript source files and WebAssembly Text (WAT) files. The third module constructs a unified AST from the generated JS and Wasm ASTs, and the fourth module, a connector written in python, links the system with a Neo4j graph database hosted in a Docker container.Despite its capabilities, tested on a limited version of WasmBech, Wasmosys currently presents certain limitations, including the use of AST over Code Property Graphs (CPG), manual AST unification, and constraints in the experimental dataset. These limitations serve as insights for future development, hinting at the prospect of an even more robust and accurate tool for JavaScript and WebAssembly code analysis.

Contents

1 Introduction

WebAssembly, or Wasm, is a byte code format that is designed to run directly on web browsers, as a compilation target for various server side languages to run on the browser such as C/C++ to rust. It was released in 2017 and its goal is to run high performance codes in browsers, which means the client would be able to load the whole server code directly and run it on their device. Despite being so young, it's gaining traction in the web development community quickly because of its lightweight memory consumption and simple execution style with the sand-boxing method. By analyzing Alexa's most visited sites, it was found that, 1 in 600 sites of top one million most visited websites uses Wasm modules [1]. With its increasing use, possible security risks in Wasm binaries are also surfacing, which will require deep analysis in order to ensure the practicality of Wasm binaries in modern web development. Static analysis, on the other hand, is a method of finding out vulnerabilities by analyzing code without running it. The use case of static analysis relies on the fact that it doesn't need to run the code while analyzing it. It gives a proper security layer over the running of the certain portion of the code. In case of analyzing the code, WebAssembly provides a text format called WebAssembly Text Format, which is used to give users a better readability. The task of static analyzer here is to find vulnerable code patterns. In case of WebAssembly, dynamic analysis of the code works as a faster method to detect potential security vulnerabilities. The Code Property Graph (CPG) is regarded as one of the most prominent discoveries in the field of static analysis and it is widely used in every case of static analysis techniques. The CPG provides a graph mining based algorithm which in turn matches a pattern based query on the graph database and it works efficiently as the code is saved in the graph database as nodes and edges with relations in-between.

2 WebAssembly

2.1 Overview

WebAssembly is designed to be the byte code format in web. It is used as a portable compilation target for programming languages, enabling its use in web client and server applications [2]. As of January 2023, WebAssembly is supported in 95.12% browsers including 4 major browser engines, i.e. Mozilla Firefox, Google Chrome, Safari and Microsoft Edge [3].

2.2 Background

JavaScript is the primary language of web browsers. Most modern web frameworks compile or transpile into JavaScript so that the applications can run on the browser. But it wasn't perfect. For loose typing, implicit coercion, callback hell and lack of modular system javascript was poorly designed and lacked proper specifications for running high performance applications. Modern day applications sometimes need on-device machine learning capabilities and performing high overload of calculations which are provided by the webgl and opencl format. These would have worked better with a new bytecode format. However, there has been ongoing research for a new bytecode format in browsers for a long time. The goal was to have a safe, fast, portable and compact language that can act as the compilation target for high level languages. Microsoft's ActiveX, NaCl (Native Client) [4], Emscripten [5] and many other technologies have been developed in order to attempt this feat but they were not good enough to replace the old and renowned JavaScript because they were not portable enough and lacked several features including high performant code structure and simplicity, sandboxing and other prominenet WebAssembly featueres. In 2017, the first version of WebAssembly was released and it proved itself to be the most complete browser byte code format in existence [6].

2.3 Compilation

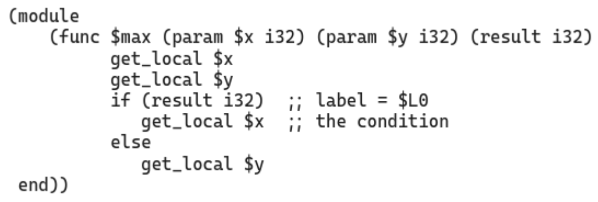
Wasm is a 32-bit machine language which is designed to be compiled primarily from C/C++ and Rust which gives a compilation target for the server side languages and creates a sandbox like environment to execute code by browser forntends which includes arbitrary client codes that are onot secured[7]. The C/C++/Rust source file is compiled into Wasm binaries directly, but an intermediate assembly-like syntax can be extracted, which is known as a *wat* (WebAssembly Text) file [8].

2.4 Data Types

Unlike other bytecode languages, Wasm supports 4 type primitives - *i32* (32-bit integer), *i64* (64-bit integer), *f32* (32-bit float) and *f64* (64-bit float).

2.5 Control Flow

WebAssembly uses structured control flow where function instructions are nested in blocks. Branches can transfer the flow to the end of these blocks only. A separate indirect call method is used for function pointers, which controls program flow using indices from a function table. This WebAssembly code snippet creates



```
(module
    (func $max (param $x i32) (param $y i32) (result i32)
          get_local $x
          get_local $y
          if (result i32)  ;; label = $L0
             get_local $x  ;; the condition
          else
             get_local $y
 end))
```

Figure 1: A simple Wasm module (in *WAT* format) that shows if-else control flow example

a function $max that takes two parameters, $x and $y, both of type i32 (32-bit integer). The if operation tests the condition (whether x >y), and if true, it returns $x, else it returns $y.

2.6 Memory

Storage in Wasm is designed as a global single array of bytes, i.e. simple linear memory. Memory address pointers are in *i32* data type, as the addresses are 32-bit. Heap and stack are both implemented onto the linear memory. Wasm allocates memory on its own and does not provide memory management.

2.7 Execution Environment

Wasm modules are executed in a host environment like web browsers or NodeJS. They provide the Wasm modules with necessary APIs, e.g. browser API. These environments provide a sandbox environment for the Wasm modules, so that unsafe activities can be prevented.

2.8 WebAssembly Architecture

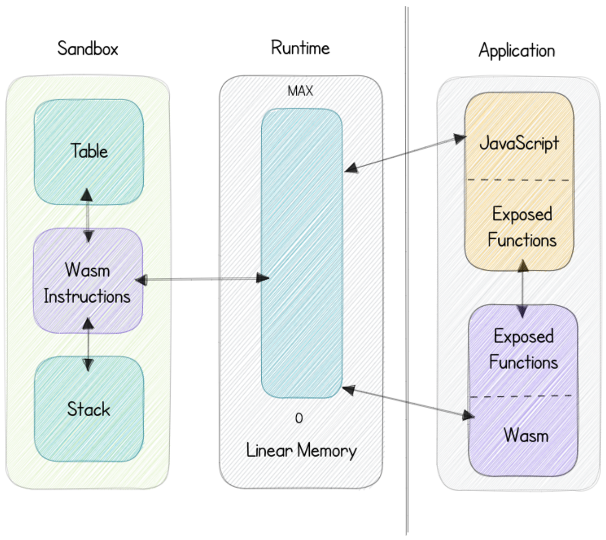


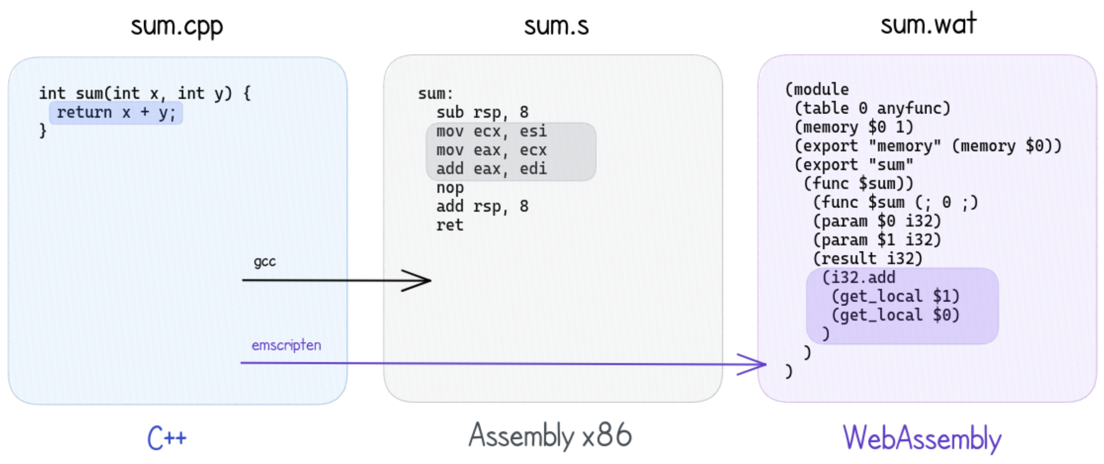
Figure 2: A High Level View of WebAssembly Execution Architecture.

2.9 WebAssembly Syntax

```
(module
 (type $i32_i32_=>_i32
 (func (param i32 i32) (result i32)))
 (memory $0 0)
 (export "add" (func $module/add))
 (export "memory" (memory $0))
 (func $module/add
  (param $0 i32) (param $1 i32) (result i32)
   local.get $0
   local.get $1
   i32.add
 )
)
```

Figure 3: A simple Wasm module (in *WAT* format) that adds two 32-bit integers (*i32*) and returns the sum.

2.10 Code Comparison



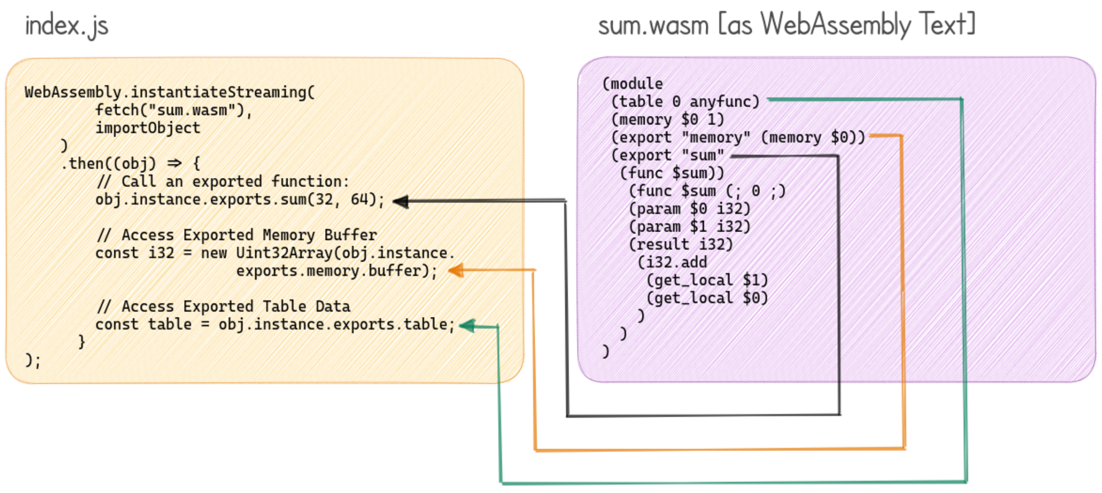
```
int sum(int x, int y) {
  return x + y;
}
```

```
sum:
  sub rsp, 8
  mov ecx, esi
  mov eax, ecx
  add eax, edi
  nop
  add rsp, 8
  ret
```

```
(module
 (table 0 anyfunc)
 (memory $0 1)
 (export "memory" (memory $0))
 (export "sum"
  (func $sum))
   (func $sum (; 0 ;)
   (param $0 i32)
   (param $1 i32)
   (result i32)
    (i32.add
     (get_local $1)
     (get_local $0)
    )
   )
 )
```

Figure 4: Code Comparison (C++ vs x86 vs WAT)

2.11 Calling Mechanism



```
WebAssembly.instantiateStreaming(
        fetch("sum.wasm"),
        importObject
    )
    .then((obj) => {
        // Call an exported function:
        obj.instance.exports.sum(32, 64);

        // Access Exported Memory Buffer
        const i32 = new Uint32Array(obj.instance.
                        exports.memory.buffer);

        // Access Exported Table Data
        const table = obj.instance.exports.table;
      }
);
```

```
(module
 (table 0 anyfunc)
 (memory $0 1)
 (export "memory" (memory $0))
 (export "sum"
  (func $sum))
   (func $sum (; 0 ;)
   (param $0 i32)
   (param $1 i32)
   (result i32)
    (i32.add
     (get_local $1)
     (get_local $0)
    )
  )
)
```

Figure 5: Mechanism of accessing Wasm module data/function from JavaScript programs

3 WebAssembly Vulnerability

3.1 Memory Corruption Vulnerabilities

Memory corruption vulnerabilities include a set of primitives that enables attackers to overwrite program memory causing unpredictable and malicious behaviour. These are the most common vulnerabilities found in memory unsafe languages. There is a possibility of these vulnerabilities translating into Wasm binaries when compiled. After decades of hardening, possible memory corruptions in x86 binaries are well defined with proper mitigation. The same cannot be said for WebAssembly. Possible memory corruption vulnerabilities in WebAssembly can be [9]:

3.1.1 Stack-based Buffer Overflow

WebAssembly dynamically allocates memory and does not provide any manual memory management. Even though Wasm VM isolates Wasm module memory access, parts of C/C++ function data are stored on unmanaged stack of the VM. So, stack-based buffer overflow vulnerabilities are prevented in internal memory but not the linear unmanaged memory of Wasm sandbox. Native platforms can prevent this type of exploitations using stack canaries. Wasm modules can increase their memory allocation with certain API calls. That means, it is possible to control the memory allocation size and insert corrupted input data into the stack to invoke stack overflow. Native platforms use guard page to prevent this type of attacks but such mitigation is absent in WebAssembly. The unmanaged stack in WebAssembly execution memory contains function-scoped data. With a proper write primitive, it is possible to overwrite function-scoped local data in the stack.

3.1.2 Heap Metadata Corruption

Wasm developers can choose their preferred memory allocator as per their intended use. Default Wasm allocators ”dlmalloc” is hardened against heap metadata corruption attacks. However, since the binary size is an important consideration in web development, developers may use lightweight allocators which might not be hardened against memory corruption. In this case, if there is no fortification, attackers can write to adjacent metadata of chunks in heap memory when these allocators allocate/de-allocate memory. Linear stack based overflow can easily overwrite heap data as heap and stack share the same linear memory in WebAssembly. Native mitigation like guard pages are absent in WebAssembly, which means there is no way to avoid overflow based vulnerabilities in the linear memory.

3.1.3 Injecting Code into Host

As mentioned before, WebAssembly modules use different API calls from its host environment in order to extend its uses to practical environment. Using functions like eval/exec, found in browser/NodeJS host environment, Wasm modules can be developed to execute code in the host environment. As a result, host environment vulnerabilities can be exploited from within Wasm modules. The arrays of possible exploitation include remote code execution, cross site scripting etc.

A detailed study on Wasm design and specifications [9] indicates that these old vulnerabilities found in the high level programming languages that can compile into Wasm, may translate into equivalent vulnerabilities in Wasm binaries. Though there are certainly more fortification in WebAssembly by default, security is not completely guaranteed.

3.2 Side Channel Attacks

Side channel attacks target the execution environment of the system by exploiting indirect effects of a system or its hardware. An example of this is Spectre [7], which affects modern microprocessors. Recent processors use branch predictions to increase performance and throughput. On these processors, execution resulting from a branch misprediction may leave side effects that can reveal private data. JavaScript Just-In-Time (JIT) compilers and transpilers are directly affected by this vulnerability.

Wasm isolates untrusted modules using run time as well as compile time checks. This includes heap memory access and indirect function call checks. The validity of return addresses are also ensured using a safe stack. However, these fortifications can be bypassed using the following techniques [7]:

3.2.1 Spectre-PHT (Pattern History Table)

The pattern history table (PHT) can be exploited to confuse the conditional branch predictor and make it mispredict a path. A wrong path execution like this can be exploited to bypass control flow and memory isolation.

3.2.2 Spectre-BTB (Branch Target Buffer)

BTB helps in predicting the target address of indirect jump instructions in programs. Similar to PHT, BTB's entries can be changed maliciously to change control flow to a specific target.

3.2.3 Spectre-RSB (Return Stack Buffer)

RSB stores the return addresses of executed call instructions and helps in predicting the return points from executed functions. By overflowing RSB using a series of call and ret instructions, control flow of the program can be hijacked.

3.2.4 Sandbox Breakout

Wasm is widely used in FaaS (Function as a Service) platforms, where Spectre-PHT or BTB can be used to access data/control outside the sandbox. This is known as Sandbox breakout.

3.2.5 Sandbox Poisoning

After using Spectre attack to misdirect the control flow, sandbox data can be leaked by accessing data from sandbox cache or similar state stores.

3.2.6 Host Poisoning

The host runtime can also be exploited in a similar way as sandbox, and host system data can be accessed maliciously.

4 Proposed Mitigation

While finding vulnerabilities in WebAssembly, studies have proposed [9] possible mitigation for different types of attacks. For memory corruptions, the following mitigation can be directly incorporated into WebAssembly standard:

4.1 Multi Memory Implementation

Rather than having a single linear memory, multi memory system in WebAssembly [10] can enable the system to have separate data spaces for heap, stack and constant data. As a result, indirect overflows and pointer forging can be prevented for the most part. This is a proposed specification for WebAssembly and may be implemented into the language in the future.

4.2 MS-Wasm Proposal

The MS-Wasm Proposal [11] suggests the addition of memory segments of specific size and lifetime in the WebAssembly language. Implementing this may prove to be hard for hosts but it provides high memory safety.

4.3 Address Space Layout Randomization

Using the presently available linear memory with randomized address layout, an additional layer of security can provided against memory based exploits. This causes obfuscation of memory locations of contiguous data segments in the program.

4.4 Safe Unlinking

Safe unlinking may prevent metadata corruption as it disables exploits from writing into arbitrary chunks in memory.

4.5 Guard page

Guard Page protection mechanism triggers page fault when the stack grows into restricted guarded pages. If such page fault occurs during any exploitation, the program will simply crash and invalid data access can be prevented.

To address side channel attacks like Spectre, Swivel [7] has been developed to mitigate exploitation:

4.6 Swivel-SFI

Swivel-SFI mitigates sandbox breakout, sandbox poisoning and host poisoning through a series of fortifications. These fortifications include the use of a separate stack to protect return addresses, Branch-To-Branch flushing to prevent polluting Branch-To-Branch entries and elimination of Code-Based-Partioning poisoning.

4.7 Swivel-CET

Swivel-CET mitigates sandbox breakout and poisoning with shadow stack, forward-edge Control Flow Integration and conditional BTB flushing. It also includes register interlocking and leak prevention during execution to provide poisoning detection and fortification. This, even if the vulnerabilities are exploited, the defenses can protect data from getting leaked.

5 Mitigation Challenges

The biggest challenge in the face of WebAssembly security updates is browser support. There is a large number of browsers in the market and all of them need to support Wasm in order for it to become the staple byte code for web. Thus, WebAssembly specification update means that the maintainer of all these browsers will need to update their VMs accordingly to match the new security standard. This method has no alternative and as a result it will take some time for Wasm to mature into what it aims to become.

6 WebAssembly Binary Analysis

WebAssembly security requires continuous analysis of Wasm modules and programs to develop further. Native application domain already has a huge number of tools to have their binaries analyzed for security concerns. Since WebAssembly is a new language, there is a scarcity of tools in this field. However, many high quality tools have been developed already. Some of these tools are:

6.1 Wasabi

Wasabi [8] is a framework for dynamic analysis of WebAssembly binaries. This open source framework instruments Wasm binary while preserving program behaviour and affecting performance and size slightly. A security researcher can use Wasabi to write general-purpose custom dynamic analyses, e.g. instruction count, call graph extraction and analysis, memory tracing and taint analysis.

6.2 WasmBench

WasmBench [12] is the largest open-source Wasm binary database. It gathered real WebAssembly programs from existing websites using web crawling, GitHub repositories and manual module extraction. For any binary analysis toolset, a dataset of real binaries is required for experimentation and validation. WasmBench can fulfill that role for future binary analysis tools targeting Wasm.

Types	**Count**
Games	25
Text Processing	11
Visualization	11
Media Processing	9
Demo	7
Wasm Test	5
Chat	3
Online Gambling	2
Barcode & QR code scanning	2
Room Planning & Furniture	2
Blogging	2
Crypto-currency wallet	2
Regular Expressions	1
Hashing	1
PDF Viewer	1

Table 1: WebAssembly binary type count in 100 randomly selected samples from WasmBench

6.3 Fuzzm

Fuzzm [13] is a greybox fuzzer for WebAssembly binaries. It integrates an AFL style fuzzer to test inputs on Wasm binaries. The main goal of this fuzzer is to identify suspicious program behaviour, which is not present in WebAssembly toolchain by default. Fuzzm provides hardening against stack and heap based attacks with the use of canaries. As it is the only fuzzer available for WebAssembly at present, Fuzzm is pioneering in the development of binary fuzzer targeting Wasm.

6.4 Wassail

Wassail [14] (WebAssembly static analyzer and inspection library) is a toolkit to perform lightweight and heavyweight static analysis of WebAssembly modules. The tool uses a practical implementation of the program slicing concept.

Program slicing is a program decomposition technique based on a specific program point called the slicing criterion, which identifies a subprogram of the code relevant to the slicing criterion, here slicing criterion is a specific point in the program the analyzer is interested in It has numerous applications in debugging, program comprehension, software maintenance, and vulnerability detection.

Program slicing approaches can be divided along multiple dimensions. Static approaches compute a slice that preserves the behavior for all possible program inputs, while dynamic approaches consider only a subset of the input.

The program slice with respect to a slicing criterion (s, v) is then computed as follows: Start with the set just containing the statement 's'. Then, iterative add to the set any statement that directly or indirectly affects the value of a variable at a statement in the set. This iterative process continues until no more statements

can be added, at which point you have your program slice. This will include all the statements that could potentially influence the value of ’v’ at the statement ’s’, directly or indirectly. This is a static program slice because it considers all possible executions of the program. It is also possible to define a dynamic program slice, which only considers a specific execution of the program.
The first two phases of this three-phase algorithm computes a closure of a Wasm binary. To produce an executable slice, the algorithm’s third phase implements a stack-preserving approach to produce a valid execute Wasm program from the closure slice.

6.5 Wasmati

Wasmati [15] is a tool for analyzing and debugging WebAssembly (Wasm) code. It includes a set of features for disassembling Wasm code, visualizing control flow graphs, and identifying potential vulnerabilities.

Wasmati is implemented to understand the potential security issues which in turn can be used to analyze both the compiled Wasm code and the source code which was used to generate it.

Some features of Wasmati include:

- **Disassembly**

  Disassembly of WebAssembly code into the human readable format which will make it easier to understand the way of the code to work and it will also identify the potential security vulnerabilies.

- **Visualization** Wasmati has the feature of visualizing the source code with the code property graph database. Code Property Graph consists of 3 sub-graphs.

  - **Abstract Syntax Tree** A WebAssembly module is parsed and the tree containing high level abstraction of the code is made from that

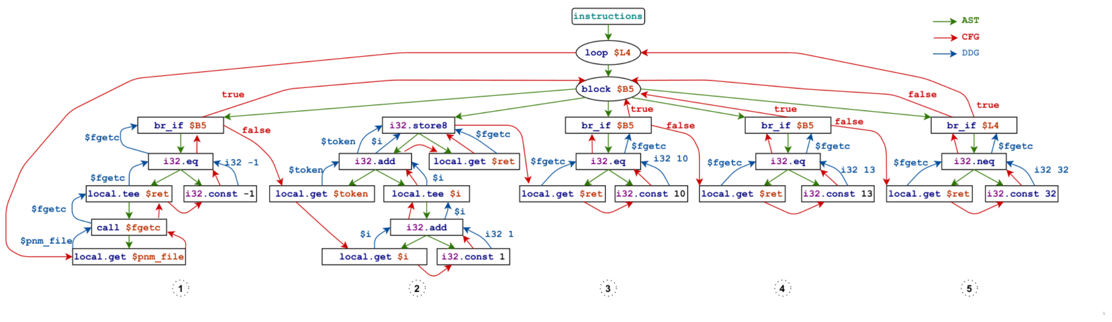


Figure 6: Wasmati Code Property Graph

parsing.

- **Control Flow Graph** This graph depicts the control flow of the program.
- **Data Dependency Graph:** Dependency of the data is depicted by this graph.

- **Vulnerability analysis**

  Wasmati is the static analysis tool that is used for different types of common vulnerabilities in the WebAssembly code, the examples can include buffer overflows, user-after-free, indirect-object relocation error.

7 Problem Description

Existing literature and toolset -

- Supports Wasm binaries only
- Does not support Wasm execution environment
- The exposed Javascript function inside the WebAssembly code cannot be analyzed.

8 Compiler Data Structures

8.1 Abstract Syntax Tree

An abstract syntax tree is a structured representation of syntactic structure of a code. The intermediate representation is defined by these structures, which are produced while compilation or when the program is parsed. This is typically used in programming languages related tools for analysis and transformation. [16]

The ast removes the details of the syntax or textual representations and gets the skeleton of the code by making an abstraction over the syntax. The logical structure and connection between various elements of the code is produced such as statements, nested expressions, identifiers, variables and different control flow statements.

8.1.1 Lexical Analysis

The first step while making an ast is the scanning or the lexical analysis which produces a stream of tokens. We know the source code is dividied into tokens, such as keywords, identifiers, operators, literals and different types of tokens. These are the basic building blocks while building the parse tree.

8.1.2 Parsing

The tokens which we get as the result is then parsed by a parser with respect to the rules of the programming language. It validates the syntax and produces a parse tree. Parse tree is an intermediate representation which captures the structure of the code which is based on those production rules. Parse tree is a detailed reflection of the code.

8.1.3 AST Construction

We can construct the AST from parse tree by de-sugaring another level of textual details with more abstract and concise detail. This involves discarding nonessential nodes that do not contribute to the rules of the program, such as parentheses or punctuation marks. In addition to that, some structural nodes might be combined to represent higher level constructs.

8.1.4 Node Types

AST represents different elements of the code including assignment operations, binary operations, function calls, loops, and different types of operations. These are typically organized in a hierarchical structure, where child nodes are represented by sub-expressions or sub-statements.

8.1.5 Relationships and Attributes

The ast represents the relationships to indicate the flow and dependencies between different part of the code is dictated by the nodes. For example, control flow constructs like if-else statements or loop child node, which represent structures like branches. Also, nodes can have attributes that are stord information relevant to their related language constructs such as the name of a variable or the operator in the expression.

The representation of the code from the AST makes it simple to manipulate the structure of the program. ASTs are used in different kinds of programming tools, which are compilers, interpreters and different kinds of static analyzers, optimizations and code refactoring, also code generation. [16]

8.2 Control Flow Graph

A control flow graph is a graphical representation of the execution path of the program which provides a visual representation of the program states and structure of its control that are organized and their interaction. A CFG is usually used in analysis of the software, program optimization and design of compiler related tools and techniques.

A usual CFG is made of nodes and edges. Every node describes a basic block which is a list of instructions which doesn't have any jump statements or branches in the middle. These blocks are mainly made by dividing the code into pieces where control flow goes and gets out only through the first and last instructions. These nodes are ideally represented by the rectangles or the circles and they have the instructions or statements of the program.

Control flow of the basic blocks are represented by the edges of the CFG. There are three types of edges:

8.2.1 Sequential edges

The edges which are sequential indicates the normal control flow and then to the next. A block A is followed by block B then there will be a sequential edge from A to B.

8.2.2 Conditional edges

Conditional branches such as if-else or loop execution path is represented by these programs. There is a labelling process and these labels determine which path is taken. As an example there is a condition that can be $x > 6$, a conditional edge will be constructed from the if statement block, which will be then corresponding to the true branch, and there will be another edge which is conditional to the basic block from the conditional edge from the false branch.

8.2.3 Jump edges

Jump edges or unconditional branches in the program is represented by these edges, statements can include go-to or break or continue statements. These edges allow control flow to skip one or multiple blocks and then transfer the control to a specific target based basic block.

The concept of CFG gives a proper overview of the execution code's control flow which allows the users to reason about the code behaviour in addition to performing various code analyses which can also be used to find unreachable codes, identifying loops and their boundaries, determining program execution path and analyzing code complexity and many more things.

Static CFG's are the usual types of CFGs which are made based on the source code or the intermediate code of the program the dynamic ones are a bit different. The dynamic CFG's are produced by analyzing the runtime of the code adding analysis of performance capturing the execution paths which are taken during the code execution.

The control flow graphs are the type of tool which can be used to reason about the execution path of a program and thus helping the end users and makes the compiler able to perform various tasks which are used for optimizations and debugging purposes.

8.3 Program Dependency Graph

Program Dependency Graph (PDG)[17] is a graphical representation which depicts the dependencies between the statements that we get from the actual code representation. We can get a view of data flows with the statements that actually interact within a program. The usual use cases include program analysis, optimization and understanding the behaviour of the software.

The PDG consists of nodes and edges. Each node in the PDG represents a program statement, such as assignments, function calls, or control flow statements. The nodes are usually labeled with the statement or expression they represent. Edges in the PDG represent the dependencies between statements. There are different types of dependencies that can be captured in a PDG:
PDG have edges which draws the dependencies between various statements. There are multiple types of dependencies that are depicted in a PDG:

8.3.1 Data dependencies

Data dependencies are the type of dependencies which gets the control flow of data between various statements. As an example an assignment operation can be regarded as a proper example for data dependency, such as if there is an assignment expression and the next one is using that certain value then there is a data dependency operation happening between the two statements. Data dependencies can be further differentiated as read-after-write, write-after-read and write-after-write, based on the order, the sequence of which the data will be accessed.

8.3.2 Control dependencies

The dependencies which capture the control flow between the statements are called the control dependencies. These type of dependencies depicts the order of the execution, which are in the control flow constructs like conditionals, loops etc. An example of control dependency might be if a while loop executes based on the condition then the statements inside that loop are based on that certain condition which gets perfectly depicted on the control flow graph of the certain execution path.

8.3.3 Anti-dependencies

If the execution of a statement is dependent on the future update of a variable then these type of dependency is called anti-dependency. For example, a statement using a variable which will be modified in a later statement then we can say there can be an anti-dependency between the statements.

8.3.4 Output dependencies

The type of dependencies where multiple statements write to the same variable are the type of dependencies which are called output dependencies. For example, if a statement writes to a variable and then another statement reads or writes the same variable, it will turn into an output dependency between them.

The PDG provides a detailed view of the dependencies which are inside the program execution by which it makes the program optimization and analysis possible for the end users. It also helps to find potential performance bottlenecks, race conditions and ways for parallelization of execution of the program. Through the analysis of the PDG, end users can identiffy dependencies which may make the program able to impact the performance and correctness with enabling the informed decisions for optimization and refactoring.

PDG are used to analyze programs, such an application can be data-flow analysis where program dependency graphs are used to find the definitions of variables, which in turn forms the use-def chains. It identifies the definitions, identifies the uses, identifies the uses and then creates the chains. These can be studied to find the program paths, finding loops and detecting various unreachable codes.

The program dependency graph is a formidable tool which can be used as a representation that will capture the dependencies between different program statements. The PDG provides a higher level representation of the above mentioned dependencies which enables the end user to reason about program behaviour, thus facilitating program understanding and analyzing codes in a detailed manner, in turn which will give the users insights into the data flows and the statements that interact within a program thus improving the performance, correctness and maintainablitiy.

8.4 Code Property Graph

The code Property Graph (CPG)[**?**] is the kind of representation which gets the different aspects of the source code, program skeleton and can reason between their relationships. This kind of property graph provides a detailed view of the programs different static and dynamic properties and behaviour in turn which enables program analysis of the advanced kind also the visualization and the overall reasoning about the program.

Inside the graph database the CPG is made of different nodes and edges. The nodes which are in the CPG are related to different types of code entities such as classes, methods and different types of variables, statements and expressions. Every node is labeled with different types of information about the program structure such as its name, type or different values. The edges there represent the CPG which depicts the relationships and dependencies between the program entities.

The CPG captures different types of relationships and dependencies:

8.4.1 Control Flow

The control flow edges of the Code Property Graph represent the execution path of the control flow between the statements of the program. They depict the order of execution, and the possible paths that can go through the program. Thus they make understanding of the code from different types of control behaviour, they can identify loops and they can detect the codes that are unreachable.

8.4.2 Data Flow

The edges which are called data flow are the flow of data from various parts of the program, from different named entities. They depict the way values get passed from one named entity to the other one. Data dependencies, reaching definitions or taint analysis - different types of anlayses can be done from these kind of data-flow edges.

8.4.3 Type Hierarchy

The representation of inheritance or sub-type kind of relationships between different types of classes and interfaces. The depiction of class hierarchy and polymorphism within the code is very indetailed in these cases. Type hierarchy edges enable type-based analysis, method dispatch and different types of understanding of various kinds of object-oriented features.

8.4.4 Method Call

The invocation of methods or functions within the program is represented by the method call edges. The calling relationships are depicted by these kind of edges. This edges are important for getting the reasoning out of program behavior, there are different types of inter procedural analysis which can be done by this and there are different types of function calls which can be detected through the method call edges.

8.4.5 Variable Access

Variable access edges can represent the access to different kinds of variables within the program. They depict read and write access to different types of variables, which in turn enables the analysis of different types of variable usage, the types of liveness analysis and there are different types of detection between shared variables which can be reasoned through the variable access edges.

The Code Property Graph (CPG) is considered as a powerful representation which is captured by the various aspects of the source code, code structure and the types of relationship between these codes. The code property graph structuer provides a detailed view of the static propertyies of a program and the dynamic behaviour which in turn enables advanced tyeps of program analysis, visualization and different types of understanding. By representing the code as a graph, the CPG allows for a wide range of program analysis techniques. It enables advanced static analyses, such as control flow analysis, data flow analysis, points-to analysis, or program slicing. It also facilitates dynamic analyses by capturing runtime behavior, such as profiling, runtime checks, or security analysis. The CPG will get an addition of properties and different kind of metadata which will be able to enrich the representation. It can include properties about various code evaluation metrics and smells with comments and source code locations. These properties will in turn enhance the kind of capabilities which will provide a better way of reasoning about the source code.

8.5 Code Property Graph in Vulnerability Detection

The CPG can be used to find different types of vulnerabilities by leveraging the contextually rich representation of code and the relationship that are stored in a graph db. The graph database will give the end users the ability to efficiently query and traversal of different types of CPG, which in turn will enable vulnerability detection techniques which will exploit different types of graph structures including different types of properties.

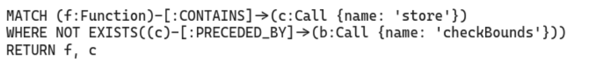
Here is a step-by-step explanation of how CPGs can be used to detect vulnerabilities from a graph database:

8.5.1 Building the CPG

The CPG can be built using 3 types of different graphs which are built from the source code. The grpahs include the Abstract Syntax Tree, Control Flow Graph and the Program Dependency Graph. These graphs can extract the different entities such as control flow, data flow, method calls and different types of variable acceses, which are in turn depicted as edges which are in the graph.

8.5.2 Defining vulnerability patterns

There are different types of vulnerabilities which are in turn being built from different types of vulnerability patterns. These patterns represents different vulnerability signatures, different characteristics which indicates the presence of a vulnerability. For example, a certain method call can be specified by a pattern which has a kind of sequence that is indicated that a buffer overflow, a pattern can be generated for this kind of specific data flow, which eventually leads to buffer overflow.



```
MATCH (f:Function)-[:CONTAINS]→(c:Call {name: 'store'})
WHERE NOT EXISTS((c)-[:PRECEDED_BY]→(b:Call {name: 'checkBounds'}))
RETURN f, c
```

Figure 7: An example cypher query for finding buffer overflow in WebAssembly

Here, the query looks for the store function. This function is used to write data into the WebAssembly linear memory, those which don't have the checkbounds functions. It finds the out of bounds correctly.

8.5.3 Querying the graph database

The process which detects the vulnerability involves the querying of graph database which is used by the defined vulnerability patterns. There are different types of queries which can be formulated to traverse the graph and that can be used to identify the instances where the pattern can be matched with the different structures or properties of the certain code. Thus, the queries can be expressed which will be regarded as a graph query language and then the structure can be stored as a graph inside the graph traversal API, which can be regraded as a property graph.

8.5.4 Pattern matching and vulnerability Identification

The CPG is traversed by the queries which in turn look for nodes and edges that can be matched with the defined vulnerability patterns. When a match is found, it is indicated that the presence of a potential vulnerability, which will be identified. The specific nodes and edges involved with the match can provide information about the vulnerable code section which can be regarded as location of the code, or some relevant variables or a different data source.

8.5.5 Contextual analysis

The vulnerabilities which were identified needs to be analyzed in a more detailed manner, with the context of the codebase. User input, data sanitization, variable scopes, security configuration can be used to determine the actual presence and the severity of the vulnerabilities which can be found. This type of analysis can be used to filter out the false positives and can be used as a providing factor for a more accurate assessment which can be used to analyze the vulnerability.

8.5.6 Reporting and remediation

The final part is the reporting and remediation of the vulnerability to the developers or the security audit team for more investigation. The CPG can be used as a major factor in terms of vulnerability, its impact and the recommended mitigation strategies which, by leveraging them, the graph representation, the report can give a clear visualization for the vulnerable code sections and their relationships which by aiding developers makes them understand and fix the issue.
The use of a graph database for storing the CPG enables efficient querying and

traversal of the code, making vulnerability detection scalable and performance higher. Additionally, the graph structure allows for more sophisticated analyses, such as graph-based algorithms or path finding, to uncover complex vulnerabilities that may involve multiple code paths and interactions.
Overall, leveraging the power of graph databases and CPGs enables effective vulnerability detection by providing a comprehensive representation of code and its relationships. It enhances the accuracy and efficiency of the detection process, enabling developers and security professionals to identify and address vulnerabilities in a timely manner.

8.6 Abstract Syntax Tree in Vulnerability Detection

The AST is a tree like structure, which represents the syntactic structure of the source code. It gets the hierarchical relationships between program elements, such as different types of expressions, statements and different types of declaration. In case of vulnerability detection the AST is widely used as it provides a structured representation of the structure which can be leveraged later for different kinds of analysis and pattern matching.

Here's an explanation of how the AST is used to detect vulnerabilities from a graph database:

8.6.1 Building the AST

By parsing the source code, the AST is first constructed. The AST depicts the syntactic structure of the code and records the connections between various elements. Each node in the AST represents a particular syntactic element, such as an if statement, a function call, or a variable declaration. The parent-child connections between the nodes represent the hierarchy and nesting of the code

8.6.2 Defining vulnerability patterns

Next, vulnerability patterns or rules are defined. These patterns specify the syntactic structures or combinations of elements that indicate the presence of a vulnerability. For example, a pattern might define a specific sequence of method calls that is indicative of a cross-site scripting (XSS) vulnerability.

8.6.3 Querying the graph database

The AST can be transformed into graph representation and then it can also be stored in graph database for querying effciently. The vulnerability detection process will also involve formulating queries for the graph query language so that it can traverse the AST graph which is stored in database. These queries are designed such way so that they can match the defined vulnerability patterns.

8.6.4 Pattern matching and vulnerability identification

The queries traverse the AST graph, looking for nodes and edges that match the defined vulnerability patterns. When a match is found, it indicates the potential presence of a vulnerability. The specific nodes or edges involved in the match provide information about the vulnerable code section, such as the location, relevant variables, or data sources.

8.6.5 Contextual analysis

Identified potential vulnerabilities can be used for further contextual analysis and this performance can be considered as factors such as different kinds of user input, the data sanitization, different kinds of variable scopes or security configurations to determine the actual presence and the severity of the vulnerabilities. The contextual analysis can be used to help filter out the false positives providing a more accurate asesment of the vulnerability.

Vulnerability detection is scalable and performant thanks to the use of a graph database to store the AST and enable efficient querying and traversal of the code. The graph structure makes it possible for complex analysis methods to identify vulnerabilities that may involve numerous code paths and interactions, such as graph-based algorithms or path finding.

In general, the AST acts as an essential intermediate representation for finding vulnerabilities. Developers and security experts can efficiently analyze the structure and relationships of the code to find vulnerabilities by converting the AST into a graph and using a graph database. The AST and graph database combination improves the detection process' accuracy and effectiveness, making it easier to spot potential security flaws and fix them.

8.7 Difficulties in CPG Generation Compared to AST

8.7.1 Abstraction Level

The AST contains the syntactic structure of the source code with a very high level of abstraction compared to others. It captures the hierarchical relationships between the program elements such as different kinds of statements, expressions and declarations and their organization within the code properties. The AST then fouces on the syntax and the static strucutre of the code, which abstracts many details related to the dynamic behavior, the dataflow and different types of control flow. This higher level of abstraction then makes the AST generation process very simple and more straightforward.

Parsing and Language-specific Tools: The process of generating an AST is well-supported by parsing techniques and language-specific tools. Many programming languages provide built-in or third-party libraries for parsing the source code and generating an AST representation. These tools handle complex aspects such as lexical analysis, parsing, and building the tree structure based on the grammar rules of the language. Developers can leverage these tools to generate an AST without having to build the entire infrastructure from scratch.

Standardized Structure: The AST has a standardized structure that is common across different programming languages. While the details may vary depending on the language, the fundamental concepts and relationships captured by the AST (such as parent-child relationships, expressions, and statements) remain consistent. This standardization simplifies the process of generating and manipulating the AST, as developers can rely on established conventions and patterns.

In contrast, generating a Code Property Graph (CPG) involves capturing a more

comprehensive representation of the code, including dynamic behavior, data flow, and control flow. The CPG's main goal is to get both the static properties and the dynamic interactoins within the source code. This will need additional analysis and the data processing which is beyond the syntactic analysis which is performed for the Abstract Syntax Tree.
Below are the steps to build up a CPG:

8.7.2 Parsing and AST Generation

The first step is to parse the source code and based on that making of a parsing tree which will in turn become an abstract syntax tree. This involves the procedure of lexical analysis, parsing and the building of the tree structure which is based on the the grammar of the language, which is similar to the AST generation process.

8.7.3 Data Flow and Control Flow Analysis

When the AST will be generated, there will additional analysis which will be performed to capture the data flow which will be within the code. This analysis involves traversing the AST, tracking variable assignments, data dependencies, method invocations, and control structures (such as loops and conditionals). This information is then used to establish the data flow and control flow edges in the CPG.

8.7.4 Intermediate Representation

Generating a CPG often requires transforming the AST and other analysis results into an intermediate representation suitable for graph-based representation. This involves mapping the AST nodes, control flow information, and data flow information to graph nodes and edges, incorporating additional properties and metadata as needed.

8.7.5 Building the Graph

Finally, the transformed information is used to construct the graph representation of the CPG. This involves creating nodes to represent program entities (e.g., classes, methods, variables) and connecting them with edges to represent the various relationships (e.g., control flow, data flow, method calls). This step requires building the graph structure and populating it with the relevant information captured during the analysis process.
The generation of a CPG is more involved than generating an AST because it encompasses a broader scope, capturing a richer set of properties and relationships within the codebase. The additional analysis steps and the need for an intermediate representation add complexity to the process. However, the CPG provides a more comprehensive representation of the code, enabling advanced program analysis and detection of complex vulnerabilities that go beyond the capabilities of the AST.

9 Proposed Solution

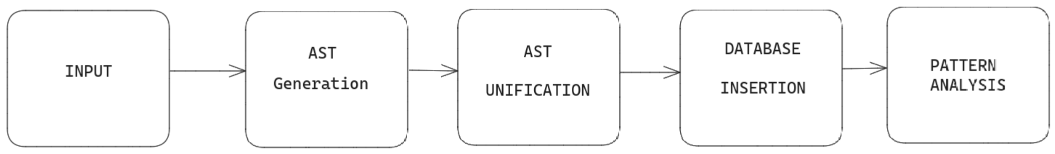


Figure 8: Architecture of the proposed solution

The proposed solution is designed to provide a comprehensive analysis of JavaScript and WebAssembly (Wasm) code in a seamless, integrated manner. The overall workflow consists of five significant steps:

1. **Input**: The first step involves receiving the JavaScript and WebAssembly source codes as input. This JavaScript code calls the Wasm code, integrating these two distinct languages. This type of coding paradigm is becom-

ing increasingly prevalent as developers leverage the benefits of Wasm in JavaScript environments for performance-critical components.

2. **AST Generation:** In the next step, Wasmosys separately generates the Abstract Syntax Trees (AST) for the JavaScript and WebAssembly source code. The JavaScript AST Generator uses the Babel library, while the WebAssembly AST Generator uses a custom-built lexer and recursive descent parsing algorithm. Each generator exports the resulting AST to a JSON file

3. **AST Unification**: After generating the separate ASTs, the next step involves manually connecting the two ASTs to create one unified tree. The AST Reconstruction phase accomplishes this by identifying key nodes in both trees and then carefully linking these nodes to form a single, suitable AST. This is done manually and during the merge the connecting nodes of javascript AST and WebAssembly AST are the points of respective function calls. The unification is done manually label by label.

4. **Database Insertion**: Once the unified AST is created, a Python script inserts it into a Neo4j graph database. This graph database is hosted in a Docker container. The Neo4j Connector, written in Python, plays a crucial role here by bridging Wasmosys and the Neo4j database.

5. **Pattern Analysis**: Pattern analysis can be conducted with the unified AST in the database. This is where Cypher queries are vital. Using Cypher queries, the Wasmosys system can probe into the suitable AST to identify potential patterns, vulnerabilities, or other insights within the JavaScript and WebAssembly source code. This powerful querying ability makes Wasmosys indispensable for comprehensive source code analysis.

This is the essential workflow of Wasmosys. By executing this workflow, Wasmosys enables developers and analysts to understand better and inspect the combined behavior of JavaScript and WebAssembly code, further advancing state of the art in source code analysis.

10 Implementation

10.1 Wasmosys

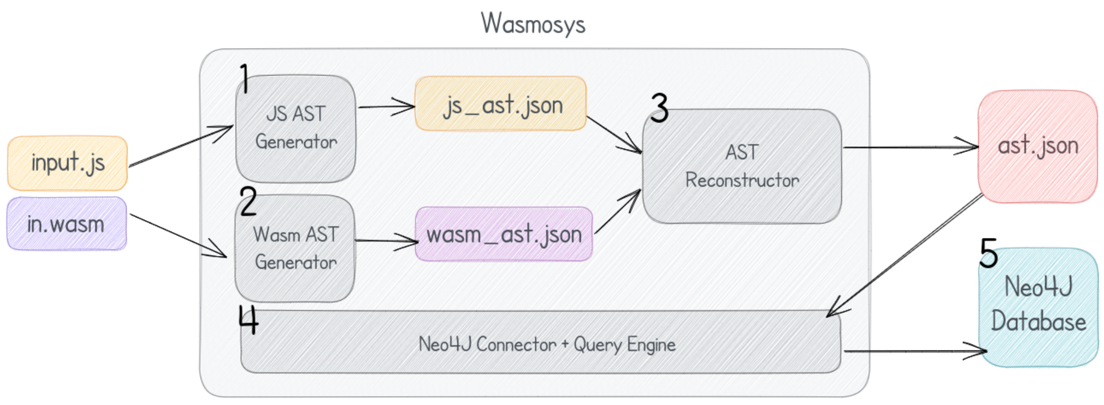


Figure 9: Wasmosys - Modules

Wasmosys is the implemented tool that aims to generate a unified Abstract Syntax Tree (AST) from the source code of the Javascript and WebAssembly bundle. It takes source code as input and utilizes a series of specialized modules to break down the JavaScript and WebAssembly parts separately, transform them into their respective ASTs, and then unify them into a singular AST. This unified AST is generated to enable developers or security analysts to understand and inspect the combined behavior of JavaScript and WebAssembly code more intuitively.

Moreover, Wasmosys also provides a standard querying interface that leverages the Cypher Query Language, a robust, declarative graph query language widely used for querying graph databases. This Cypher querying interface gives users a powerful tool to probe into the unified AST. It allows them to conduct deep inspections and facilitate many use cases ranging from security vulnerability detection to performance optimization and refactoring analysis.
The Wasmosys system is a comprehensive source code analysis tool that is modularly designed. It comprises five main interconnected modules that work collectively to deliver a unified JavaScript and WebAssembly AST. Each module has

its unique functionalities, as detailed below:

1. **JavaScript AST Generator**:

   The JavaScript AST Generator module plays a crucial role as the first component in the Wasmosys system. Its primary function is to take a JavaScript source file as input and utilize the Babel library to automatically generate the Abstract Syntax Tree (AST). This AST will represent the structural representation of the source code which will contain the syntactic elements and their hierarchical relationships. By leveraging the Babel library, the Javascript AST generatror will transform the Javascript source code into a strucutre which will look like a tree that will correctly reflect the syntax and organization of the code. The babel compiler employs a parser analyzing the source code and the constructs which are based on the language's rules. Once the AST is generated it is exported as a JSON file. This JSON format will allow for easy serialization and portability of the AST, facilitating the further processing and the analysis by other components in the Wasmosys system. The Javascript AST which will be represented as a json file, it will be able to provide a comprehensive and structured representation of the code. It captures essential information such as function definitions, variable declarations, control flow structures, expressions, and more. This level of detail enables subsequent components in the Wasmosys system to perform in-depth analysis, vulnerability detection, and security assessment. Exporting the JavaScript AST as a JSON file enhances interoperability and enables seamless integration with other modules in the system. The JSON format is widely supported and easily consumable by various programming languages and tools, making it a convenient choice for exchanging and processing the AST data. By automatically generating the JavaScript AST through the JavaScript AST Generator module, Wasmosys sets the foundation for subsequent stages of analysis and security assessment. The availability of the AST in a JSON format empowers the system to perform comprehensive static analysis, enabling the detection of potential vulnerabilities, identifying security risks, and aiding in code optimization and improvement efforts.

2. **WebAssembly AST Generator**:

   The second module, WebAssembly AST Generator module serves as the second crucial component within the Wasmosys system. Its purpose is to process WebAssembly Text (WAT) files, which contain human-readable representations of WebAssembly code, and generate the corresponding Abstract Syntax Tree (AST). This module takes inspiration from Wasmati, a well-known tool for WebAssembly analysis. Implemented in the C programming language, the WebAssembly AST Generator module employs a custom-built lexer specifically designed to tokenize the WebAssembly source code. The lexer breaks down the WAT file into a series of discrete tokens, which represent the fundamental building blocks of the WebAssembly syntax. Using a recursive descent parsing algorithm, the module then utilizes these tokens to construct the Abstract Syntax Tree (AST) of the WebAssembly code. The recursive descent parsing technique involves recursive functions that correspond to grammar productions in the WebAssembly language. These functions consume the tokens and build the AST nodes based on the syntactic rules and structure defined by the WebAssembly specification. The AST nodes generated by the WebAssembly AST Generator module are labeled according to commonly used compiler token names, providing a standardized representation of the WebAssembly code structure. This labeling enables subsequent analysis and processing stages to leverage well-known naming conventions and facilitates interoperability with other tools and systems within the Wasmosys framework. Similar to the JavaScript AST Generator module, the WebAssembly AST Generator module exports the generated AST as a JSON file. This JSON format allows for easy serialization, storage, and exchange of the AST data.

   The JSON representation can make sure that the compatibility issue and the seamless integration with other modules within the Wasmosys system work properly. Exportation for the WebAssembly AST as a JSON file enables further analysis and processing by subsequent componenets in the system. The generated AST will be served as the foundation for the static analysis, vulnerability detection and the assesment of security for WebAssembly code. The JSON formats flexibility and widespread support will make it a suitable choice for sharing and processing the AST data across different programming languages and tools. It will play a vital role by converting the wasm text files to abstract syntax trees. Leveraging a custom lexer and a recursive descent parsing algorithm, it will be able to construct the AST nodes from the WAT file's tokens. The result will indicate an AST which will be labeled with other common compiler constructs, and then will be exported as a json file .

3. **AST Reconstructor**:

   The AST reconstruction phase is the manual phase of the whole module. In this phase what we try to achieve is to make a unified AST from both of javascript and WebAssembly runtime. The problem arises when two different AST's have different labelling for abstract syntax tokens. This is done with careful consideration. The procedure is to find the relevant nodes which are representing the calls to WebAssembly functions from javscript or calls to javascript functions from WebAssmbly modules. This linking produces an AST which is used as a comprehensive representation of the entire application. This phase plays a pivotal role in performing comprehensive analysis and assessment, taking into account the interplay between the high-level adn low-level code components.

4. **Neo4J Connector**:
   Neo4j Connector is the fourth module, it is used as the connector module between the Wasmosys system and the Neo4j database. The connector driver is written in python and it connects to the Neo4j graph database, that can

run locally or in cloud. The connector is used as the intermediary to send the unified AST to the graph database, where the graph is stored for future querying and the analysis of the graph data.

The Neo4J Connector module leverages the Python Neo4J driver or an appropriate library to establish a connection to the Neo4J graph database. This connection allows the connector program to interact with the database and perform operations such as data insertion, retrieval, and manipulation. Once connected to the Neo4J graph database, the connector program is responsible for transferring the unified AST into the database. It maps the nodes and relationships within the AST to the appropriate entities and edges in the graph database schema. The connector program carefully traverses the unified AST and translates its nodes into Neo4J nodes, representing program entities such as functions, variables, and control flow structures. It establishes relationships between these nodes to capture the connections and dependencies within the code. The properties and metadata associated with each AST node are stored as attributes of the corresponding Neo4J nodes, enabling the preservation of critical information during the transfer process. By storing the unified AST in the Neo4J graph database, the Wasmosys system gains the capability to perform efficient and powerful querying and analysis on the codebase. The graph database's graph-based structure allows for flexible exploration of relationships and dependencies within the code, enabling advanced vulnerability detection, security assessment, and optimization efforts.

The neo4j graph database can run locally and the connector module stores the unified AST efficiently in that database. This storage mechanism serves as a central repository of the source code which are represented as nodes. This connector module acts as one of the most important middle-end between the Wasmosys system and the graph database. The connector is written in python which establishes a connection to the local neo4j graph database and it makes the transfer of the unified AST into the database.

This makes the efficient storage, querying and the analysis of the source code within the graph databsase, which in turn, provides the foundation for advanced program analysis and the security assessment.

5. **Graph Database**:

   The fifth module is the databases, which is hosted in a docker container for the ease of deployment and the database is highly scalable, so that the large source codes can be analyzed faster. The Neo4j connector module leverages the python Neo4j driver to establish connection. This allows the connector program to interact with the module which is responsible for transferring the unified AST into database. It maps the nodes and the relationships within the AST to the proper entities and edges in the graph database. The mapping of AST nodes are done here and it translates its nodes to Neo4j nodes. Thus representing the program entities such as variables, functions and the control flow statements. This module produces the relationships between them and creates a repository which is a graph database, querying on the database based on patterns will find the vulnerability. The graph structure enables seamless exploration of relationships and dependencies within the code which enables advanced vulnerability detection, security assessment and optimization efforts. The Neo4j connector module ensures that the unified AST is stored in an efficiently within the graph database.

The above mentioned modules are designed so that they can work together so that they can produce a unified javascript and WebAssembly Abstract syntax tree. Thus, the combined AST is then stored in a graph databsase, and it enables the user to analyze the integradted code structure using the CQL, and this modular design of the Wasmosys ensures the extensibility and the maintainability, and thus it makes into a robust tool which can be used for source code analysis.

11 Evaluation

To find the functionality and the efficiency of Wasmosys, a series of experiments were done which involved javascript and WebAssembly specimens with the known vunlerabilities. This section is here to discuss the experimental procedure, the accuracy of the results, encountered errors and the overall performance of the Wasmosys system.

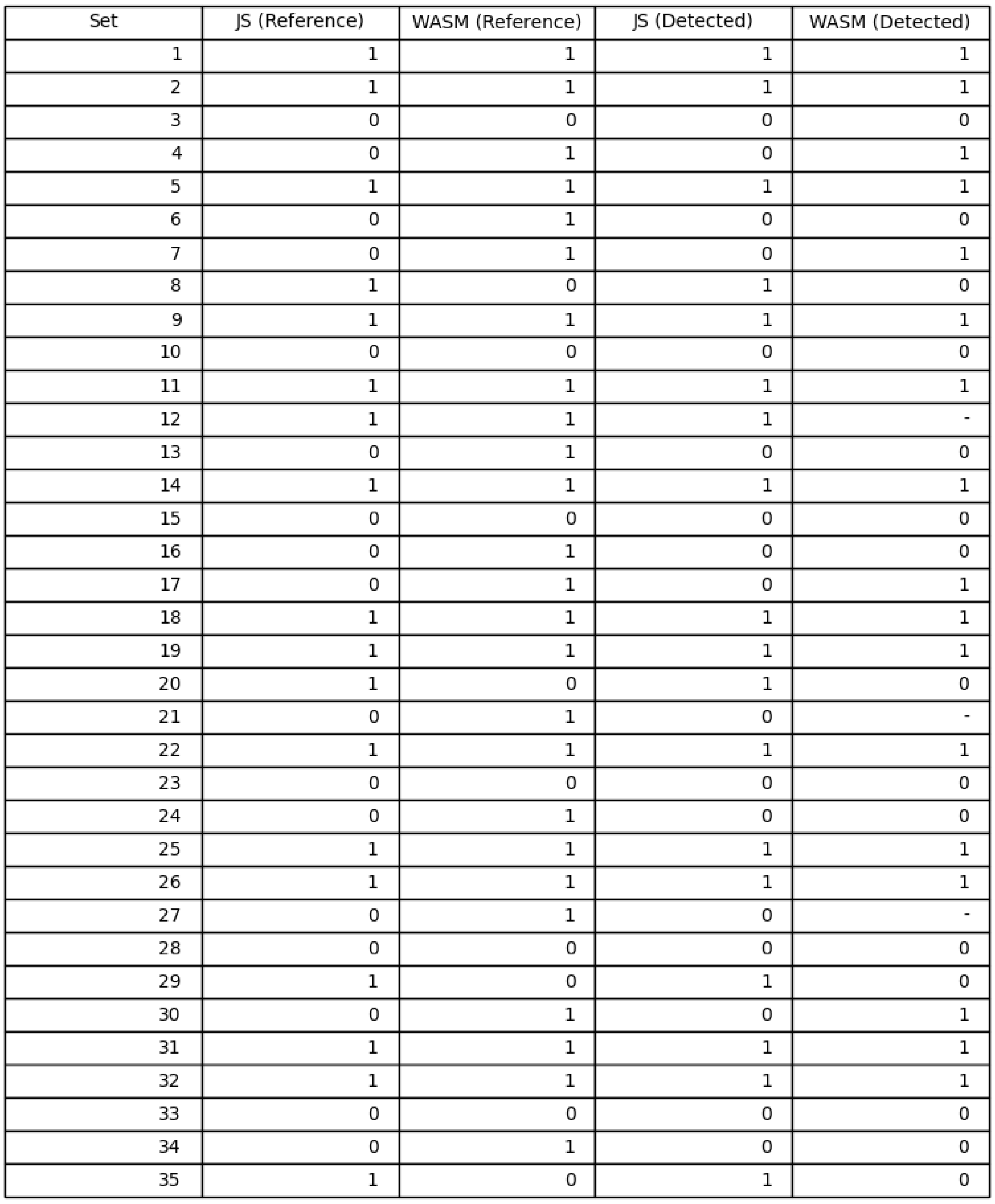
11.1 Evaluation Procedure

The experiment was began with selecting 35 WebAssembly specimens from the WasmBench dataset. These WebAssembly files were then converted into WebAssembly Text (WAT) format. Simultaneously, 35 corresponding JavaScript files were generated, each with known vulnerabilities and a single WebAssembly module call within.

These sets of JavaScript and WebAssembly files were fed into Wasmosys, where the system generated the ASTs for both, unified them, and inserted them into the Neo4j graph database. Once the unified ASTs were in the database, Cypher queries were used to detect vulnerabilities in the source code. In parallel, the time taken by Wasmosys to generate WebAssembly ASTs was recorded and compared with the time taken by Wassail, another popular AST generator, for performance benchmarking.

11.2 Accuracy

The accuracy of Wasmosys was measured in terms of its capability to detect known vulnerabilities in the given sets of JavaScript and WebAssembly files. The experiment results showed that Wasmosys could detect all the JavaScript vulnerabilities. However, some of the WebAssembly vulnerabilities were missed by the system. Nonetheless, the system produced no false positives or negatives, demonstrating high accuracy in detected vulnerabilities.

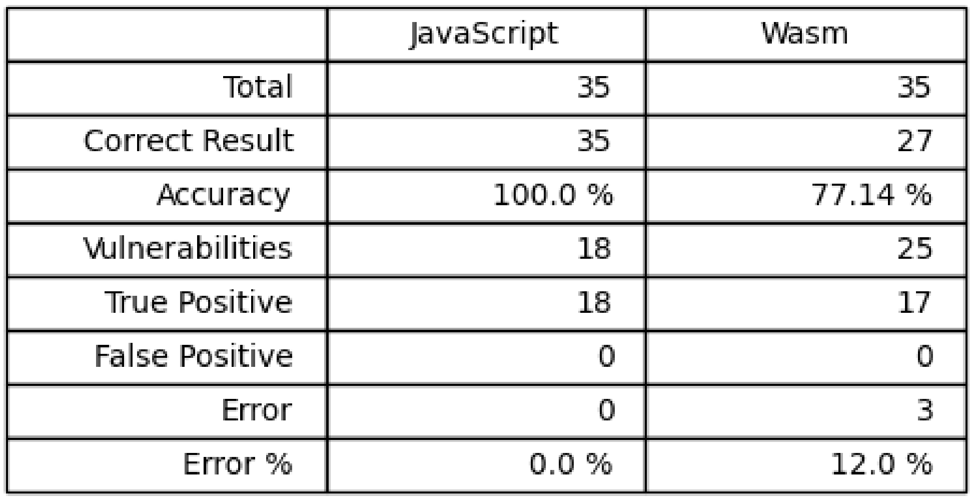


Set	JS (Reference)	WASM (Reference)	JS (Detected)	WASM (Detected)
1	1	1	1	1
2	1	1	1	1
3	0	0	0	0
4	0	1	0	1
5	1	1	1	1
6	0	1	0	0
7	0	1	0	1
8	1	0	1	0
9	1	1	1	1
10	0	0	0	0
11	1	1	1	1
12	1	1	1	-
13	0	1	0	0
14	1	1	1	1
15	0	0	0	0
16	0	1	0	0
17	0	1	0	1
18	1	1	1	1
19	1	1	1	1
20	1	0	1	0
21	0	1	0	-
22	1	1	1	1
23	0	0	0	0
24	0	1	0	0
25	1	1	1	1
26	1	1	1	1
27	0	1	0	-
28	0	0	0	0
29	1	0	1	0
30	0	1	0	1
31	1	1	1	1
32	1	1	1	1
33	0	0	0	0
34	0	1	0	0
35	1	0	1	0

Figure 10: Wasmosys - 35 Data Points

11.3 Errors

Throughout the experimental process, the system encountered three types of errors. These errors originated from WebAssembly files containing calls to Exported Tables and Exported Memory Buffers. This indicates that Wasmosys currently



	JavaScript	Wasm
Total	35	35
Correct Result	35	27
Accuracy	100.0 %	77.14 %
Vulnerabilities	18	25
True Positive	18	17
False Positive	0	0
Error	0	3
Error %	0.0 %	12.0 %

Figure 11: Wasmosys - Accuracy Data

has limitations handling specific Wasm features, and further work is needed to improve this aspect of the system.

11.4 Performance Evaluation

To measure Wasmosys's performance, the time taken to generate WebAssembly ASTs was recorded and compared with the time taken by another AST generator, Wassail. It is important to note that the JavaScript AST generation time was not considered in this comparison, as all the JavaScript files in this experiment were similar. The detailed performance data has yet to be mentioned here. Still, it was found that the time performance of Wasmosys was comparable to that of Wassail, demonstrating its viability as an efficient tool for generating ASTs from source code.

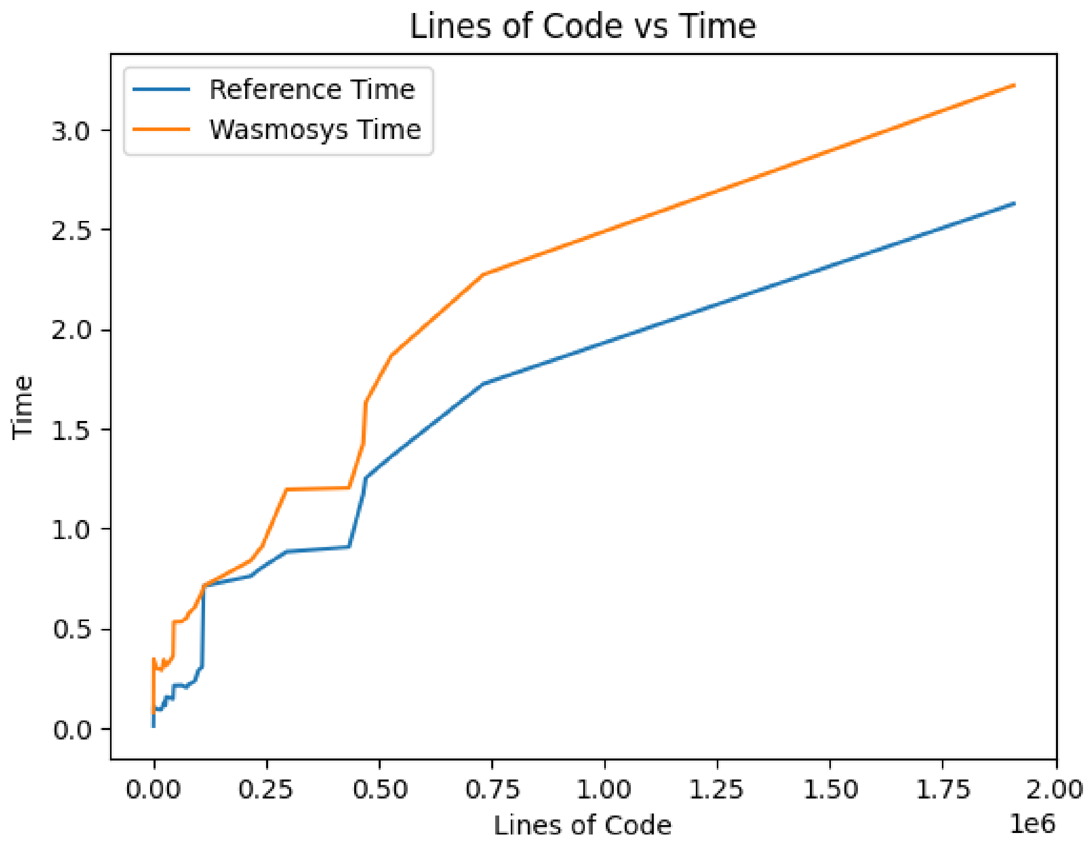


Figure 12: Wasmosys - Performance Analysis

12 Limitations

While Wasmosys stands as an innovative tool for generating unified ASTs for JavaScript and WebAssembly code, the current iteration of the system presents certain limitations, which were revealed during the experimental process. These limitations serve as learning points and areas for future enhancement and refinement of the system.

1. **Use of AST over CPG**:

   The original plan for Wasmosys involved using Code Property Graphs (CPG) due to their effectiveness in representing both a program's control and data flow. However, Wasmosys utilizes Abstract Syntax Trees (AST) in the current implementation. There were two primary reasons for this decision.

Firstly, the construction of CPG involves combining four separate graphs, which proved complex and challenging for performance. Secondly, merging the different JavaScript and WebAssembly CPGs was found to be difficult due to a lack of research in this area.

2. **Manual AST Unification**:

   The second limitation of Wasmosys is the manual unification of JavaScript and WebAssembly ASTs. The original plan called for an automatic fusion of these two ASTs using Cypher queries. However, due to the differences in AST labels produced by the two AST generators and the inability to find suitable paths automatically with Cypher queries, the process had to be done manually. This manual unification could introduce inconsistencies and inefficiencies, hindering the system's scalability.

3. **Experimental Size and Dataset**

   The third limitation lies in the size and nature of the dataset used for testing Wasmosys. The original plan was to use the WasmBench dataset for testing the capabilities of Wasmosys. However, due to a lack of dataset incorporating both JavaScript and WebAssembly code and time constraints for creating a large, suitable dataset, the team resorted to using 35 customized JavaScript-WebAssembly programs for testing. This smaller, custom dataset partially represented the wide range of potential scenarios Wasmosys might encounter in real-world applications.

13 Conclusion

To conclude, Wasmosys is an AST based static analysis tool that covers the whole WebAssembly execution path when used with function calls. While it has limitations, it creates a new method to cover JavaScript's exposed functions, and it's called the Wasm file, together with the help of AST or Abstract Syntax Tree.By leveraging the Abstract Syntax Tree (AST), Wasmosys has introduced a novel method to analyze JavaScript's exposed functions within the context of Wasm files. This approach allows for a more thorough examination of the code and enables the detection of vulnerabilities and potential security risks that might have otherwise been overlooked.Throughout this thesis book, the limitations of Wasmosys have been acknowledged. These limitations include the dependence on function calls and the inherent challenges in analyzing more complex control flow and data flow within the WebAssembly execution path. However, the research conducted demonstrates that despite these limitations, Wasmosys still provides valuable insights and contributes to the enhancement of security measures in the context of Wasm.The utilization of AST in conjunction with the Wasm file presents a promising avenue for further research and development in the realm of WebAssembly security analysis. By expanding the capabilities of Wasmosys to address its limitations, future iterations of the tool could provide even more comprehensive and effective detection of vulnerabilities and security threats in WebAssembly-based applications. Overall, this research work has shed light on the potential and significance of Wasmosys as an AST-based static analysis tool for WebAssembly. It has opened up new possibilities for improving security measures in Wasm files by harnessing the power of the Abstract Syntax Tree. With continued exploration and refinement, Wasmosys holds promise for enhancing the security posture of WebAssembly applications and contributing to the broader field of software security.

References

[1] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "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, 2019, pp. 23–42.

[2] "Webassembly," 2022. [Online]. Available: https://webassembly.org/

[3] "Up-to-date browser support tables - wasm," 2023. [Online]. Available: https://caniuse.com/?search=wasm

[4] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 79–93.

[5] A. Zakai, "Emscripten: an llvm-to-javascript compiler," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2011, pp. 301–312.

[6] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.

[7] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen *et al.*, "Swivel: Hardening {WebAssembly} against spectre," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1433–1450.

[8] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 1045–1058.

[9] D. Lehmann, J. Kinder, and M. Pradel, “Everything old is new again: Binary security of {WebAssembly},” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 217–234.

[10] “Webassembly core specification.” [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[11] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan, “Position paper: Progressive memory safety for webassembly,” in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2019, pp. 1–8.

[12] A. Hilbig, D. Lehmann, and M. Pradel, “An empirical study of real-world webassembly binaries: Security, languages, use cases,” in *Proceedings of the Web Conference 2021*, 2021, pp. 2696–2708.

[13] D. Lehmann, M. T. Torp, and M. Pradel, “Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly,” *arXiv preprint arXiv:2110.15433*, 2021.

[14] Q. Stiévenart and C. De Roover, “Wassail: a webassembly static analysis library,” in *Fifth International Workshop on Programming Technology for the Future Web*, 2021.

[15] T. Brito, P. Lopes, N. Santos, and J. F. Santos, “Wasmati: An efficient static vulnerability scanner for webassembly,” *Computers & Security*, vol. 118, p. 102745, 2022.

[16] G. Fischer, J. Lusiardi, and J. Wolff von Gudenberg, “Abstract syntax trees - and their role in model driven software development,” in *International Conference on Software Engineering Advances (ICSEA 2007)*, 2007, pp. 38–38.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, p. 319–349, jul 1987. [Online]. Available: https://doi.org/10.1145/24039.24041