B.Comp. Dissertation

Datalog-based WebAssembly vulnerability detection

By

Tran Gia Phong

Department of Computer Science

School of Computing

National University of Singapore

2022/2023

B.Comp. Dissertation

Datalog-based WebAssembly vulnerability detection

by

Tran Gia Phong

Department of Computer Science

School of Computing

National University of Singapore

2022/2023

Project No: H130220

Advisor: Prof Liang Zhenkai

Deliverables:

Report: 1 Volume

Source code: available at https://github.com/PhongTran98/fyp

Abstract

WebAssembly is a new language to be executed by browsers, designed as a compilation target for higher-level languages such as C++. WebAssembly aims to support near-native execution performance of languages other than JavaScript on the browser. By design, WebAssembly prevent certain vulnerabilities such as executable data, but it is still suffered from various kinds of memory vulnerabilities, such as use-after-free and stack overflow. In this project, we propose a solution to extract the code property graph out of a WebAssembly binary before detecting vulnerabilities based on common patterns defined as Datalog rules. We utilize two types of vulnerabilities, use-after-free, and double-free, to showcase the capability of our end-to-end vulnerability detection. In addition, we also provide detection rules for other vulnerabilities, such as uncontrolled format strings and the use of dangerous functions. Through our evaluation, we demonstrate that our solution can detect roughly 80% of the use after free and double-free test cases provided in the Juliet Test Suite.

Subject descriptors:

- D.1.6: LOGIC PROGRAMMING
- D.4.6: SECURITY AND PROTECTION

Keywords: WebAssembly, Datalog, logic programming, vulnerability detection, use-after-free, double-free, static analysis

Implementation softwares: Kali GNU/Linux 2022.4, Soufflé 2.3, Emscripten 3.31, Wasmati 1.0.0

Acknowledgement

I wish to extend my sincere appreciation to all those who have supported me in this year-long project. Firstly, I am grateful to Professor Liang Zhenkai for providing me with the opportunity to work on this project. Thanks to his insightful guidance, my project was able to stay on course and make progress in the right direction. Secondly, I would like to express my gratitude to Wang Jianing for addressing numerous concerns and helping to refine this report. Our many discussions help me greatly enhance the proposed solution for this project. Finally, I would like to acknowledge the unwavering support of my friends and family throughout this project.

List of Figures

Contents

Chapter 1

Introduction

WebAssembly is a bytecode language designed to be run in web browsers with near-native speed. Mainly used as a compilation target for other high-level programming languages such as C++ and Rust, it enables a web application to run computation-intensive processes such as scientific visualization and simulation [1]. First announced in 2015 and released in 2017 [2], its high-level goals are to be “efficient and fast”, “safe”, “open and debuggable”, as well as “part of the open web platform” [3].

By design, WebAssembly provides a few security guarantees. First of all, unlike x86, in WebAssembly, only the linear memory data can be directly modified by users; code, the implicit stack of values, local/global variables, and the call stack are directly maintained by the VM. This means that overflow in linear memory cannot directly affect variables and the stack of values. Moreover, control flow integrity is guaranteed, since the control flow graph can be statically determined before execution. This helps avoid attacks such as code injection or return-oriented programming.

Nevertheless, despite the security guarantees, WebAssembly programs still carry some risks. Many of the WebAssembly vulnerabilities are not new but rather ported from the memory-unsafe source language that has already been thoroughly studied and mitigated against [12]. One such example is stack-based buffer overflow in C++, which could be mitigated with the use of stack canaries in the native compilation. This defense mechanism, however, is not available in WebAssembly. When the source program is compiled into WebAssembly, the linear memory is partitioned to play the roles of data, heap, and stack sections of the source program. Since there is no protection from stack canaries in WebAssembly, heap, and stack data could easily be corrupted.

Research into combating against WebAssembly vulnerabilities has advanced to some extent over the last decade. For static analysis, Wasmati [11] generates a code property graph to detect common vulnerability patterns, and Wassail [4] performs binary slicing to simplify the program. For dynamic analysis, Wasabi [14] functions as a general-purpose framework for other dynamic analysis, and Fuzzm [12] statically inserts stack canaries into the linear memory before executing the binary to detect stack-based buffer overflow. Static analysis tools are much more efficient, but they always carry the possibility of producing false positives. Conversely, dynamic analysis tools are unlikely to encounter false positives but are considerably more time-consuming due to the need of executing the binaries.

In our project, we focus on static analysis of WebAssembly binaries to detect common vulnerabilities. Although the problem of statically detecting all vulnerabilities in a binary is proven to be undecidable, static analysis remains to be popular due to its efficiency. Intuitively, static analysis involves applying heuristic logical reasoning to determine whether a vulnerability exists in a program. For example, we deduce the program has a use-after-free vulnerability if the following facts are true: (1) There are two statements A and B (2) Statement A can reach statement B in the control flow graph (3) Statement A calls `free()` on a pointer (4) Statement B use the same pointer freed by statement A. To determine whether two variable references are the same, we can either compare their identifier or perform dataflow analysis, with the latter being more precise but also more complicated.

As we can see from the example, to offer more precision, heuristics are becoming more complicated, which on the flip side also hinders the development process. To combat this newfound complexity, static analysis tools are moving toward the approach of separating the logic detection rule from its implementation using domain-specific languages. One such language is Datalog, a declarative logic programming language deducing new facts (which are analogous to rows in relational databases) based on existing facts and inference rules. As a domain-specific language, Datalog is much more expressive, human-readable, and easily extendable than general-purpose languages such as C++ or Python when used to perform the same task. The possibility of integrating Datalog into program analysis has already been explored in some of the recent works, such as Java pointer and taint analysis [10], Java race condition detection [16], or LLVM IR vulnerability scanning [17].

Our project proposes a solution for WebAssembly vulnerability detection using Datalog. At a high level, our solution utilizes the code property graph generator Wasmati [11] to generate the abstract syntax tree (AST) and the control flow graph (CFG) from a given WebAssembly binary. Although Wasmati also generates the dataflow graph, since the tool ignores dataflow involving linear memory, the graphs are insufficient for our needs. Therefore, after extracting the graph, we perform our version of dataflow analysis to handle instructions involving linear memory by taking into account the memory locations. Finally, aided by the previous step, we infer the existence of vulnerabilities in the binary based on declarative rules defining their common patterns. Both dataflow analysis and vulnerability detection are done using Datalog. Due to the time constraints of our project, although we propose detection rules for various classes of vulnerabilities, we mainly focus on use-after-free vulnerabilities as the vulnerability is neither overly complicated nor trivial to detect. Our project is available at `https://github.com/PhongTran98/fyp`.

In summary, we make the following contributions:

- We build a logic inference-based system for detecting vulnerabilities in WebAssembly binaries via dataflow analysis. Our system addresses the limitations of existing solutions, such as linear memory dependencies.

- We propose detection rules for double free (CWE-415), use-after-free (CWE-416), uncontrolled format string (CWE-134), use of dangerous function (CWE-242), unchecked value (CWE-252), with use-after-free being the main focus.

- We test our solution on the Juliet test suite to demonstrate the effectiveness of our use-after-free vulnerability detection.

Chapter 2

Background and Related works

2.1 Datalog language

Datalog is a declarative logic programming language that is based on the concept of relation in mathematics. More specifically, it is used to query deductive databases, a type of database consisting of facts and rules. There are three main concepts in Datalog: relations, rules, and facts. Relations represent the relationship between their fields, such as a relation `edge` represents the relationship between the two endpoints. Facts are concrete instances of relations, similar to how objects are to their classes in the object-oriented programming paradigm. Rules either list the facts of a relation or define how a fact in the relation can be "generated" from existing facts (not necessarily of the same relation). The following is an example of Datalog relations, facts, and rules:

```
// rules listing facts
edge("A", "B")
edge("A", "C")
edge("B", "D")

// rules defining facts from other facts
reaches(X, Y) :- edge(X, Y)
reaches(X, Y) :- edge(Y, X)
reaches(X, Y) :- reaches(X, Z), edge(Z, Y)

// these rules define two relations:
// edge and reaches
```

The facts define a graph, and the rules define whether two vertices are connected in this graph. For example, `reaches(‘‘A’’, ‘‘D’’)` holds due to rule 3 because `reaches(‘‘A’’, ‘‘B’’)` holds due to rule 1 and `edge(‘‘B’’, ‘‘D’’)` also holds. In rough terms, Datalog generates more facts based on the initially provided facts by following the defined rules.

The growing complexity of program analysis demands a solution that could help manage the logic involved. Hence, Datalog is highly applicable to the domain due to its design to separate the logic from the implementation, as well as its ability to work well with a large database. To date, Datalog has been explored in various studies in the literature, which we will elaborate on more in the related work section.

Various Datalog engines could be used to run Datalog programs. In our project, we use Soufflé [18], which is a high-performance Datalog engine written in C++ that is optimized specifically for program analysis. Soufflé is also open-sourced and currently in active development at the time of writing.

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

Figure 2.1: An example of a WebAssembly binary, from Wasmati paper [11]

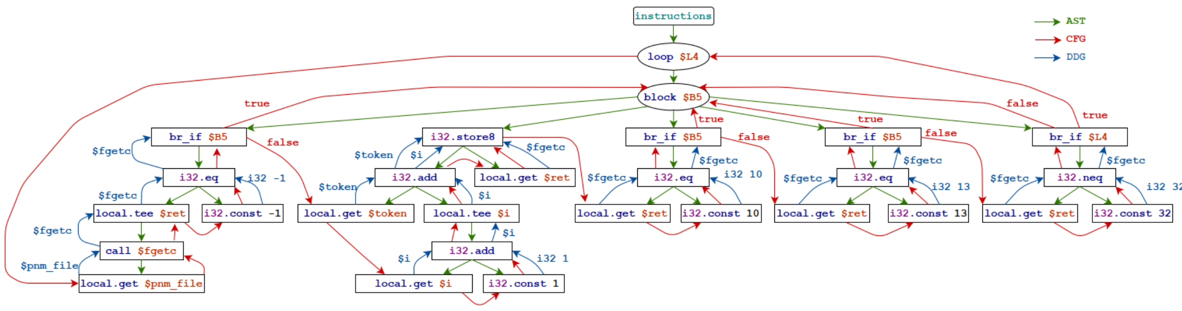


Figure 2.2: The corresponding CPG of the binary in Figure 2.1, from Wasmati paper [11]

2.2 Wasmati code property graph generator

Wasmati [11] is a static vulnerability scanner for WebAssembly consisting of a code property graph (CPG) builder and a graph database backend. In Wasmati, the CPG of WebAssembly binary is a combination of the abstract syntax tree (AST), the control flow graph (CFG), the data dependency graph (DDG), and the call graph (CG). Figure 2.1 and 2.2 illustrates an example of a WebAssembly program and its corresponding (simplified) code property graph.

Some aspects of Wasmati are relevant to our project. Firstly, Wasmati performs AST folding: for instructions directly contained in the same block, instead of putting all such instructions at the same level in the AST, Wasmati "folds" the AST so that instructions pushing a value onto the stack are children of the instruction popping and using those operands, as illustrated in figure 2.2. Our project utilizes this capability to track flows of data passing through the value stack.

Secondly, Wasmati does not track dataflow that involves linear memory operations such as `i32.load` and `i32.store`. Although the dataflow analysis done by Wasmati is still powerful, WebAssembly compilers such as Emscripten often use the WebAssembly linear memory to store local variables of C functions, while using WebAssembly local variables as temporary variables or stack pointers, similar to how processor registers are used. Therefore, we perform our version of dataflow analysis that also involve linear memory.

Finally, Wasmati supports a variety of output format that is compatible with various query languages, among which is Datalog, where the output is formatted as facts representing the nodes and edges of the CPG. In our solution, we import those facts and perform some modifications before performing analysis, which we will elaborate on in section 4.

2.3 Other WebAssembly analyzing tools

Other than Wasmati in the earlier section, other tools can be used to analyze WebAssembly binaries either statically or dynamically.

Similar to Wasmati, Wassail [4] is another tool that generates the control flow graph of the binary and performs data analysis by propagating over the graph. However, Wassail is more focused on analyzing the data flow inside the binaries, including performing binary slicing on WebAssembly while preserving the stack state for each statement. The Wasmati's author also compare the performance between Wasmati and Wasail and noted that while the output data structure between the two is comparable, Wasmati's performance speed is much better.

Wasabi [14] is a framework that greatly shortens the process of performing dynamic analysis on WebAssembly binaries. It instruments each instruction with a hook based on the instruction type, and users (who use Wasabi to perform dynamic analysis) can implement the hooks that would be called during the execution. Using Wasabi, we could profile the executed instructions, calculate coverage, or perform taint analysis.

Fuzzm [12] is a binary-only fuzzer that detects heap and stack-based buffer overflow in a WebAssembly binary. The paper introduced an instrumentation technique to harden WebAssembly binaries with canaries to detect overflow, as well as a WebAssembly binary grey-box fuzzer to detect those vulnerabilities based on the AFL fuzzer [5]. Another approach to perform binary-only grey-box fuzzing on WebAssembly is introduced in WAFL [13]. Unlike Fuzzm which uses wasmtime and AFL, WAFL uses WAVM [6] as the runtime and AFL++ [7] as the fuzzer.

2.4 Datalog Program Analysis

In this section, we briefly mention some program analysis solutions using Datalog in the literature. Although none of these is for WebAssembly, we study them regardless since many of the techniques used are also applicable to WebAssembly.

DOOP [10] focused on pointer analyses in Java programs using Datalog, which out-performed the then state-of-the-art pointer analyses framework Paddle [15] which is based on Binary Decision Diagrams. On the other hand, Chord [16] proposed a new algorithm to detect race conditions in Java programs implemented in Datalog.

However, the closest to our project is VANDALIR [17], which uses the Soufflé Datalog engine to detect vulnerabilities in LLVM IR. It is similar in the sense that unlike the high-level programming language Java, both WebAssembly and LLVM IR are low-level and have similar characteristics such as how they both store local variables in the source language (recall that WebAssembly is mainly

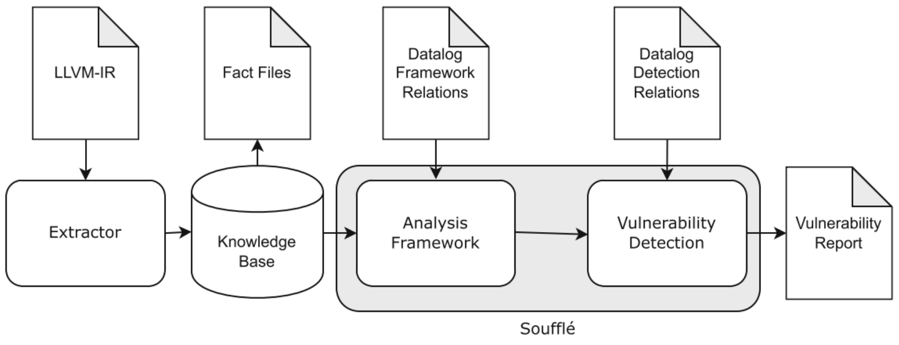


Figure 2.3: Overview of the VANDALIR pipeline, from VANDALIR paper [17]

used as the compilation target for other languages) in linear memory and how they both use temporary one-time local variables to expand statements in the source language. Figure 2.3 from the original paper provide a high-level overview of the VANDALIR. The program first runs a Python script to generate Datalog facts from the LLVM IR binaries. It then uses Datalog rules to perform analysis on the binaries, such as pointer and alias analysis, as well as rules for vulnerability detection such as buffer overflow and format string vulnerabilities.

Chapter 3

Our Approach

3.1 Overview

Our solution is written completely as Datalog rules. Figure 3.1 shows the high-level view of the solution, which has three main components: Datalog rules for input processing, program analysis, and vulnerability detection. Figure 3.2 shows the hierarchy of sets of rules (each in a separate file) in our solution, with each set only using rules defined on rule sets on the same layer or layers to the left.

The input processing component takes in the code property graph of a given WebAssembly binary generated by Wasmati and represented as Datalog facts. These facts consist of nodes and edges of the code property graph. Since the Wasmati-generated graph contains elements irrelevant to our analysis, we simplified the graph to suit our needs. We also develop various auxiliary rules (e.g. rules for instructions) to make later analysis more convenient.

The program analysis component consists of data dependency analysis rules and alias analysis rules. The data dependency analysis rules represent how an instruction's result affects other instructions' execution and are further divided into four subcategories: dependency through the implicit stack, the local/global variables, the linear memory, and through the execution of the instruction, which

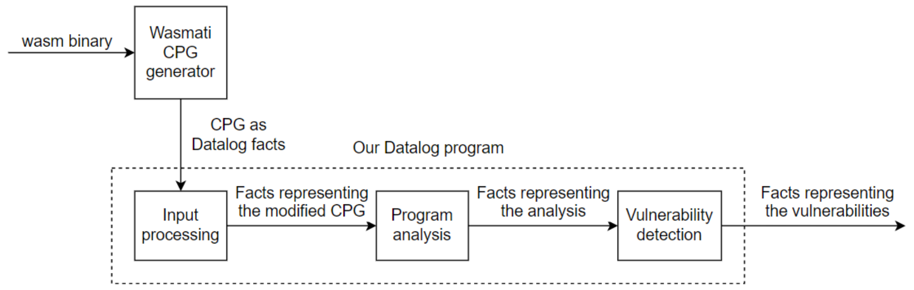


Figure 3.1: Overview of our solution

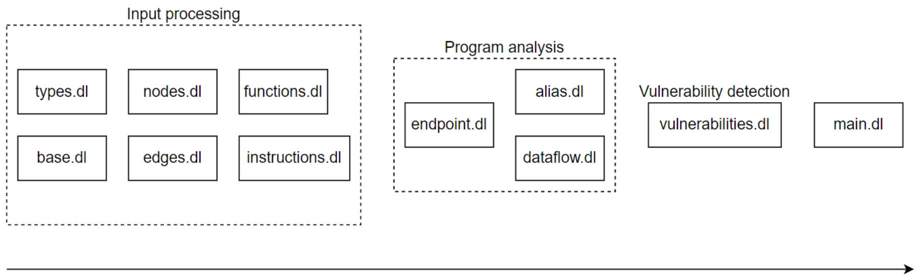


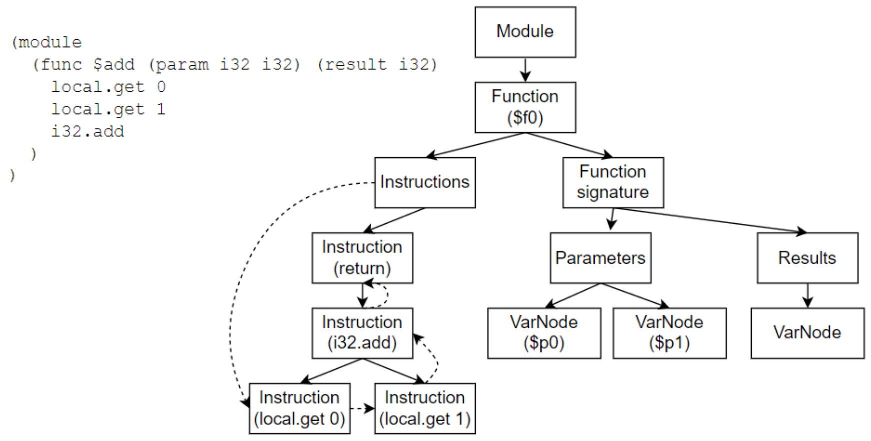
Figure 3.2: A more detailed structure of the solution

we will elaborate in a later section. The alias analysis rules use the previously done data dependency analysis to determine whether different operands and/or results have the same value.

The vulnerability detection rules define the pattern of common vulnerabilities such as use-after-free using the analysis done by the previous component. After the Datalog program execution, the existence of facts in any of these rules implies the given WebAssembly binary contains vulnerabilities.

3.2 Graph representation

Given a WebAssembly binary, Wasmati generates a corresponding code property graph (CPG) consisting of an abstract syntax tree (AST), a control flow graph (CFG), a data dependency graph (DDG), and a call graph (CG). Figure



```
(module
  (func $add (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add
  )
)
```

Figure 3.3: Wasmati example output

3.3 demonstrate the corresponding CPG generated by Wasmati for a simple WebAssembly binary. The solid lines represent the AST edges of the CPG, while the dashed lines represent the CFG edges. We exclude the DDG and the CG edges.

When output with a Datalog-compatible format, each of these nodes and edges are represented as a fact which we then import into our solution. For nodes, each fact contains the unique ID of the node, the node type (e.g. the "Function" type or "Instruction" type), and information specific to each type of node, such as the function signature of "Function" nodes, or the type of instruction of "Instruction" nodes. For edges, each fact contains the unique ID of the nodes on the two ends, the type of edge (e.g. the "AST" type or "CFG" type), and information specific to each type of edge. Since facts of the same

Our modification on the CPG is as follows:

- We only keep the AST and the CFG, while discarding the DDG and the CG. This is because the Wasmati doesn't track dependency involving linear memory as we discussed in the earlier section. We decide to perform our dataflow analysis from scratch to handle compatibility issues when we

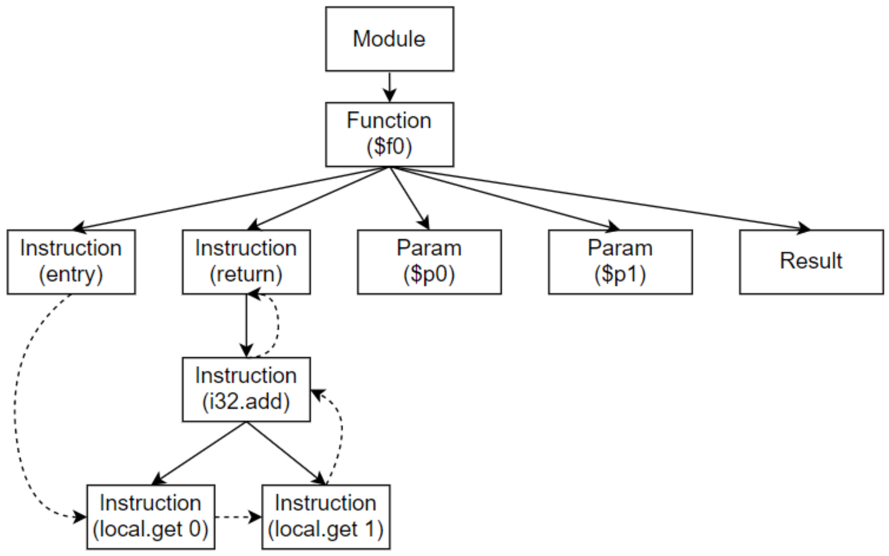


Figure 3.4: The modified CPG

introduce dataflow analysis involving linear memory.

- We remove the "Instructions", "Function signature", "Parameters", "Result" nodes and the edges they are connected to. Since we are dealing with Datalog rules rather than visualization, we aim to minimize the graph while retaining all information on the original binary. This help makes the logic simpler.

- We allow configuration to state which function is within scope, as most of the time we do not need to analyze functions compiled from the standard library. We also remove all "Instruction" nodes of functions not within the scope.

- For the nodes with their parents deleted, we connect them to their closest ancestor, which is the "Function" node.

- To distinguish different kinds of "VarNode", which is necessary since we remove their parents, we change their type into either “Parameter”, “Result”, or “Local” depending on their deleted parents.

- Since the “Instructions” node is removed, we add an auxiliary “Instruction” node to represent the entry of the functions.

Figure 3.4 shows the modified version of the CPG shown in figure 3.3. Rules defining the modification are in `base.dl`.

Since Wasmati only uses two rules to represent the CPG (the node rule and the edge rule), both of these rules contain many parameters (the node rule has 18 parameters, while the edge rule has 9 parameters) to distinguish different types of nodes and edges. However, this makes the rules unwieldy to use directly in later analysis. Therefore, we also define more specific rules for each type of edge and node which only contain a subset of parameters of the original rules.

3.3 Program analysis

3.3.1 Endpoints

To understand the flow of data in a WebAssembly program, we need to understand the relationship between the operands used and the results generated by every instruction. First of all, we need to define how we refer to them. One way is to use the function ID and a name for the memory location (such as local/-global variable or memory index), which is the approach VANDALIR used in their analysis. However, the same variable in different locations in the function may not be related to each other in any way, either due to being on different CFG paths or being on the same CFG path but the value of the variable is reset in between. Moreover, the results of some instructions such as `i32.add` have no obvious way of naming. In our project, we use the instruction ID (the node ID representing the “Instruction” node) instead of the function ID. Instead of the name, we use a boolean and an index to indicate whether we are referring to the instruction’s operand or result as well as which among them we are dealing with.

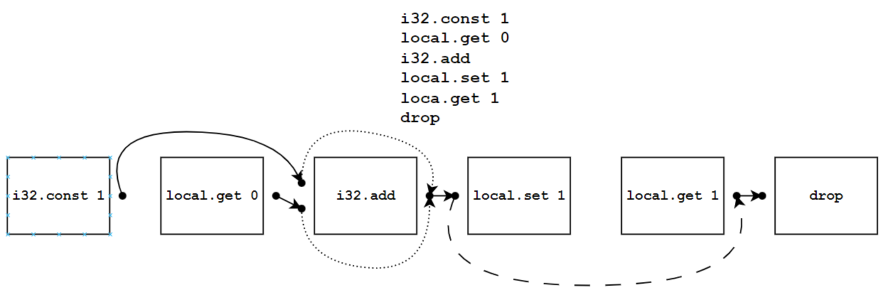


Figure 3.5: An example of direct flows between endpoints

To make the explanation more convenient, we call the 3-value tuples above as "endpoints". Each instruction has several "in-endpoints" and "out-endpoints" representing the operands it takes from and the results it puts onto the implicit stack. The value of an endpoint is the value of the operand or the result it is representing. Thanks to the design of WebAssembly, these "instruction signature" can be statically determined, including block, call and indirect_call. For example, the local.get 0 in figure 3.5 instruction has 0 in-endpoint and 1 out-endpoint, while the i32.add instruction has 2 in-endpoints and 1 out-endpoint. The dataflow analysis problem now becomes the problem of determining the relationship between these endpoints.

3.3.2 Dataflow analysis

Currently, our dataflow analysis is limited to intra-procedural only. We say there is a "direct flow" from one endpoint A to another endpoint B if the value of A is "passed" to B without being modified by any other instruction except the ones associated with A and B. For example in figure 3.5, the endpoint representing the result of i32.const 1, which pushes a constant onto the implicit stack, directly flows to the endpoint representing the operand of i32.add, which popped the earlier value from the stack. We classify the direct flow from a source endpoint to a destination endpoint into four categories:

- Stack direct flow: the value leaves the source endpoint to be pushed onto the implicit, then popped into the destination endpoint. In figure3.5, the stack direct flows are represented by solid lines.

- Variable direct flow: the value is stored in a variable using local.set or global.set, and later retrieved by local.get or global.get. In figure 3.5, the variable direct flows are represented by dashed lines.

- Memory direct flow: the value is stored in the memory and later loaded from the memory, similar to the variable direct flow. The direct flow is not shown in the figure.

- Internal direct flow: the source and destination endpoints is an operand and a result of the same instruction, respectively. In figure 3.5, the internal direct flows are represented by dotted lines.

Among the four categories, only internal direct flow modifies the data value. Therefore, for alias analysis, we would only use the first three types of direct flow. However, although not implemented yet, all of these categories are useful to perform taint analysis on the WebAssembly binary.

Stack direct flow is done by Wasmati, in the way they fold the AST of the WebAssembly binary. Thanks to the WebAssembly design, we could statically determine which instruction a given instruction pops the value from. However, due to branching, an instruction can obtain its operand from a different source depending on the execution path.

For internal direct flow, the rule is simply to connect all in-endpoint to all out-endpoint of the same instruction, which also includes call instructions. Since we are performing intra-procedural analysis only, we over-approximate and assume that the result is affected by all parameters of the function call.

Variable direct flow can be initially defined by the following rule:

- The source endpoint is an operand of a set instruction.

- The destination endpoint is a result of a get instruction.

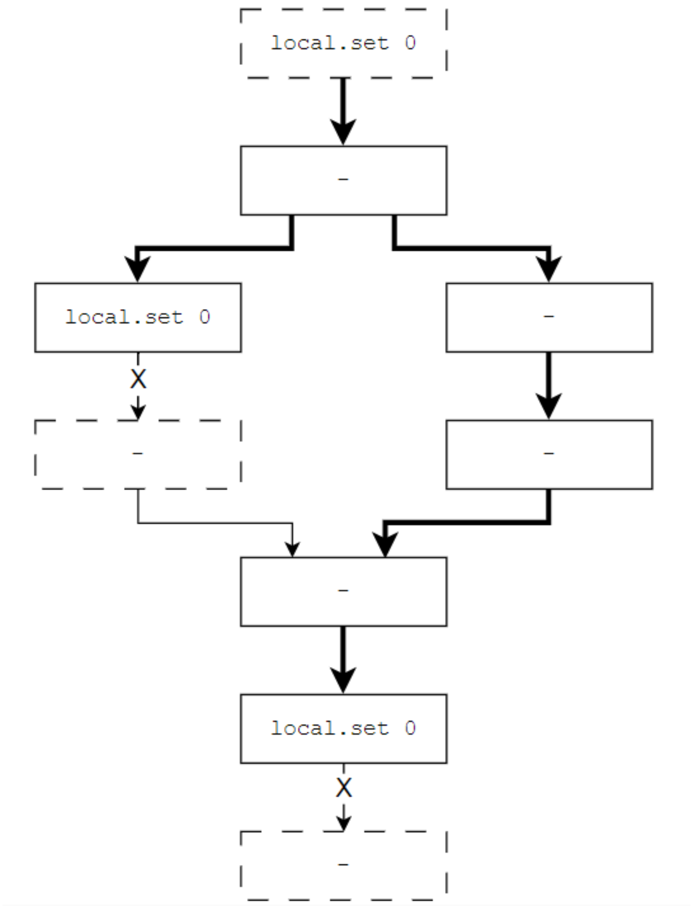


Figure 3.6: An example of the scope of an endpoint

- There is a path in the CFG from the first to the second instruction.
- The two instructions are dealing with the same variable.

However, this would result in false positives when the variable is modified in between. To combat this, we define an additional relation to representing the "scope" of an endpoint, i.e. which instructions the value of the variable can reach unmodified. This auxiliary relation is defined similarly to how breadth-first-search works: the "scope" will propagate until it reaches an instruction overwriting the value of the variable. Figure 3.6 shows an example of the scope of an endpoint. In the figure, each box represents an instruction in the CFG. Boxes with dashed borders indicate the endpoint (of the top most local.set 0 instruction) cannot reach the instruction. Now we can introduce another rule to the initial rules for variable direct flow:

- The source endpoint can reach the second instruction.

Memory direct flow is similar to variable direct flow where we also need to consider the scope of the endpoint. We can define the rule as follow:

- The source endpoint is the second operand of a store instruction (the first operand is the memory location, while the second is content to be stored).
- The source endpoint can reach the second instruction.
- The destination endpoint is a result of a load instruction.
- There is a path in the CFG from the first to the second instruction.
- The first operand of the two instructions is the same.
- The linear memory offset of the two instructions is equal.

3.3.3 Alias analysis

Alias analysis determines whether two endpoints have equal value. This could be performed using the first three types of direct flow (stack, variable, and memory direct flow), which preserve the value of the endpoint.

Initially, we could view the "equal" relation as an equivalent relation, which means it is a reflexive, transitive, and symmetric binary relation. For the base case, we define any two endpoints in the three types of direct flow being equal. We then define more rules to introduce the three properties of an equivalence relation to obtain the equivalence closure of the base case. To account for constant instructions such as i32.const, we also include pairs of endpoints with the same constant value into the equal relation. Since the number of facts would be in the order of quadratic the number of elements of an equivalent

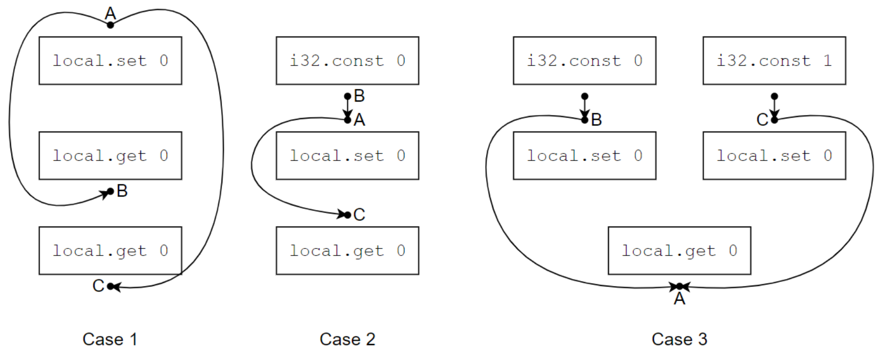


Figure 3.7: Transitivity in the equal relation

relation, Souffle also implements a custom data structure to reduce the overhead of equivalence relations [8].

However, viewing the relation as transitive will introduce false positives that could result in even much more false positives. Case 3 in figure 3.7 demonstrates such a case, where both B and C are equal to A, but B is equal to the constant 0 while C is equal to the constant 1. By transitivity, endpoints that are equal to 0 or 1 will also be treated as equal. To restrict the false positives, we add an additional check: suppose the pairs (A, B) and (A, C) are in the equal relation, (B, C) will be in the relation if A can reach both B and C (case 1 in figure 3.7) or of B can reach A while A can reach C (case 2 in figure 3.7). As a result, our alias analysis relation is only half-equivalence: although both reflexive and symmetric, the relation is not completely transitive.

3.4 Vulnerability detection

Our solution currently supports the detection of several common vulnerabilities.

Double free (CWE-415), Use after free (CWE-416). The double free vulnerability refers to the case when the program called free() twice on the same pointer, which is a special case of the use after free vulnerability, which attempts to use an already freed pointer. Using an already freed pointer that leads to

```
// instruction id1, id2 in func,
// instruction id1 calls free,
// instruction id2 uses the
// freed pointer
uaf(func, id1, id2) :-
    Instruction(id1),
    Instruction(id2),
    parentFunction(id1, func),
    parentFunction(id2, func),

    Endpoint([id1, 0, 0]),
    CallFree(id1),

    Endpoint([id2, 0, x]),
    equal([id1, 0, 0],
        [id2, 0, x]),
    reachCfg(id1, id2).
```

```
// instruction id in func,
// instruction id call format,
// string function using
// non-constant string
fmtStr(func, id) :-
    Instruction(id),
    parentFunction(id, func),

    Endpoint([id, 0, x]),
    CallPrintf(id, x),
    !value([id, 0, x]).
```

Figure 3.8: Rules for use-after-free and format string vulnerabilities

undefined behavior could be exploited by attackers. Figure 3.8 show a simplified version of our detection rule:

- Clause 1, 2, 3, 4: id1 and id2 refer to instructions in a function called func.
- Clause 5: instruction id1 have an in-endpoint order 0 (i.e. first in-endpoint).
- Clause 6: instruction id1 is a call instruction to free.
- Clause 7: instruction id2 have an in-endpoint order x (i.e. the (x + 1)-th in-endpoint).
- Clause 8: the two endpoints are equal (via alias analysis)

- Clause 9: instruction id1 can reach instruction id2 in the CFG,

Uncontrolled format string (CWE-164). Format string is a common feature of many programming languages that allows formatting a string with runtime value. In memory-unsafe language, improper use of format string may make the format string use unexpected runtime values, which could lead to leaking memory or even overwriting memory. In source language C and C++, format string vulnerability often comes in the form of a call to a function in the printf-family with the format string parameter, which is often the first or second, could be externally tampered with. Regardless, using a variable rather than a constant string as the format string is not a good practice. Therefore, we detect the vulnerability by considering the name of the function call and whether the format string parameter is a constant string. When compiled into WebAssembly, pointers to a constant string are replaced with constants number calculated in compiled time, pointing to the location of the string in the linear memory that is loaded from the beginning. Using this heuristic, we can define the pattern of format string vulnerabilities in WebAssembly as in figure 3.8:

- Clause 1, 2: id refers to an instruction in a function called func.
- Clause 4: instruction id is a format string function with the (x + 1)-th parameter define the format string.
- Clause 3, 5: instruction id has an in-endpoint order x, and the endpoint doesn't equal to any constant (via alias analysis)

The C or C++ local variables are stored in the linear memory of WebAssembly, of which the exact location can only be determined during runtime. Hence if we cannot determine the value of the function parameter, the parameter is likely from a variable.

Use of Inherently Dangerous Function (CWE-242) Some C/C++ functions are inherently unsafe to use regardless of how they are being used. These functions often handle transferring strings into an array of characters from either another array or from user input without bound checking, leading

```
// instruction id in func,
// instruction id call
// dangerous function
dangerFunc(func, id) :-
    Instruction(id),
    parentFunction(id, func),

    Call(id, funcId),
    unsafeFunc(funcId).
```

```
// instruction id in func,
// instruction id has unchecked
// return value
uncheckRet(func, id) :-
    Instruction(id),
    Instruction(id2),
    parentFunction(id, func),
    parentFunction(id2, func),

    Call(id, funcId),
    funcRetNeedCheck(funcId),
    Drop(id2),
    CfgEdge(id, id2).
```

Figure 3.9: Rules for dangerous functions usage and unchecked return values

to buffer overflow when maliciously tampered with. Some examples are get(), strcat() and strcpy(). Since retrieving the library function symbol from just the binary (with no debugging symbol) is currently outside of our scope, we simply compare the name of the function being called against a blacklist of dangerous functions. A simplified version of the rules is shown in figure 3.9.

Unchecked return value (CWE-252) Library function calls often return values that could indicate whether the operation is successful, such as returning negative numbers as the error code. Leaving these errors unchecked could lead to unexpected behavior of the program that might be exploited. In WebAssembly, the unused return value of function calls is dropped from the stack (i.e. is used by the drop instruction). By checking whether the out-endpoints of certain function calls are equal to any in-endpoints of drop instruction, we could determine whether the return value is used or not. However, one false negative case is for the malloc function in *C* or other similar functions. Checking the return of malloc means checking whether the pointer is valid or null (which means the allocation failed). However, the pointer returned by malloc is almost always used regardless of whether it is checked, hence the heuristic using drop would not work. Figure 3.9 shows a simplified version of the rule:

- Clause 1, 2, 3, 4: id and id2 refer to instructions in a function called func.
- Clause 5, 6: instruction id is a call to a function of which the return values need to be checked.
- Clause 7: instruction id2 is drop.
- Clause 8: instruction id is followed by instruction id2 in the CFG.

Chapter 4

Evaluation

4.1 Dataset and workflow

To determine the effectiveness of our solution, we evaluate it on Juliet Test Suite C/C++ version 1.3 [9], which contains more than 64,000 handcrafted test cases categorized by CWE entries. Each test case consists of a simple vulnerable implementation and its patched version. In our detection, we aim to maximize the positive results in these vulnerable implementations and the negative results in their patched version.

Due to our focus on use-after-free vulnerabilities, we only test our solution on test cases belonging to the two categories CWE-415 (Double Free) and CWE-416 (Use After Free). There are more than 900 test cases for CWE-415 and 400 cases for CWE-416, and we test our solution on each case separately. For each case, we compile the C/C++ source files (potentially more than one) into a single WebAssembly binary using the Emscripten compiler. We then use Wasmati to extract the code property graph from the binary. Finally, we run our Datalog program with the input from Wasmati using the Soufflé engine.

```
// FN due to indirectly
// calling free()
void foo() {
    ...
    bar(ptr);
    free(ptr);
}
void bar(void* input) {
    free(input);
}
```

```
// FN due to pointer
// dereference
...
free(ptr);
...
foo(&ptr);
...
```

Figure 4.1: False negative cases

4.2 Result & analysis on false detections

4.2.1 Evaluation result

The evaluation result is measured using sensitivity (true positive rate/TPR) and specificity (false positive rate/FPR). The following table summarizes the result:

Test case	TP	FN	FP	TN	TPR	FPR
CWE-415	774	186	344	616	80.62%	30.77%
CWE-416	324	90	13	401	78.26%	3.86%

Some test cases are filtered out due to compilation errors. More specifically, Wasmati doesn't support WebAssembly SIMD instructions, which are used in binaries compiled from C/C++ programs that call `rand()` among the test cases.

4.2.2 False negatives analysis

For both cases, roughly 80% of the vulnerable cases are detected. The remaining 20% false negative rate is mainly due to our solution currently not supporting inter-procedural analysis yet. More specifically, we can detect the vulnerability

```
for (...) {                       loop:
    int* ptr =                        ...
        (int*)malloc(1024)            call $malloc
    ...                               local.set 0
    free(ptr)                         ...
}                                     local.get 0
                                      call $free
                                      ...
```

Figure 4.2: False positive case in C and after compiled to WebAssembly (simplified)

if there are two instructions, with the first using free() on a pointer and the second using the same pointer as input. However, we wouldn't be able to detect the use-after-free if the first instruction is a custom function call on the pointer, which then calls free() on the pointer (see figure 4.1). A less frequent cause of false negatives is the address-of operator (&). Our solution currently establishes no connection between a variable and the dereference of its address (e.g. between var and *(&var) in C/C++). Therefore, when the address of a free pointer is used, our solution won't detect it as use-after-free (see figure 4.1).

One way we can perform inter-procedural analysis for double free/use after free is to borrow the idea of taint analysis: a function "indirectly frees" its n-th parameter if there is a value-preserving flow (i.e. a path consists of direct flows other than internal direct flow) from the n-th parameter to a free function call. Additionally, if function f indirectly frees its n-th parameter, function g calls f and there is a value-preserving flow from the m-th parameter of g to the n-th parameter of call f, then g also indirectly free its m-th parameter.

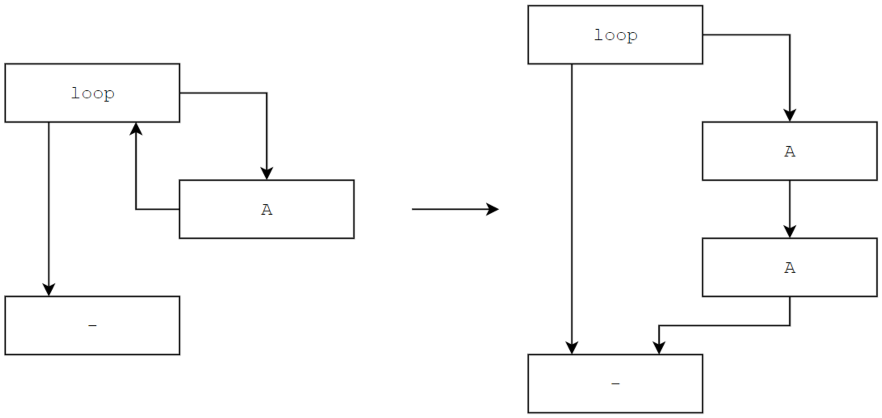


Figure 4.3: Transforming loops

4.2.3 False positive analysis

The 30% false positive rate of the test case in CWE-415 is mainly due to vulnerable cases that are use-after-free but not double-free. Currently, we over-approximate every double-free as use-after-free since the former is a special case of the latter. However, this leads to classifying returning a free pointer as double-free, regardless of whether the returned value will be consumed by a free() call or not. Another cause of false positives is allocating and freeing a pointer in a loop. For example, in figure 4.2, the input of instruction call $free is the same as the input of instruction local.set 0 since they are the same pointer when they are in a same loop. However, we classify the case as use-after-free when the first instruction and the second one use the same value, and the first can reach the second in the CFG. Therefore, we also classify the example case in figure 4.2 as use-after-free as the call $free instruction can loop back to the local.set 0 instruction, resulting in a false positive.

To reduce the false positive rate due to over-approximating double free into use-after-free, we need the inter-procedural analysis from the earlier section to see whether a function call indirectly frees any of its parameters. For the second case, the root cause is due to not differentiating between the same endpoint in different loops. To overcome this, we can duplicate the block content and remove the CFG edge back to the start of the loop (see figure 4.3). In this case, true positives due to loop (e.g. block A only contains a call to free) will be detected, while false positives due to loop (e.g. block A contains the same content as in figure 4.2) will not be detected.

Chapter 5

Conclusion

WebAssembly is a relatively new language with a focus on bringing computation-intensive programs up to near-native speed in the browser environment. While WebAssembly guarantees security to a certain degree, some vulnerabilities from source languages can still be ported over WebAssembly. In this project, we propose a solution to detect such vulnerabilities in WebAssembly binaries using the Datalog logic programming language. The choice of Datalog helps us better focus on the theoretical aspect of WebAssembly vulnerability detection. Our solution can detect approximately 80% of the use after free and double-free test cases provided in the Juliet Test Suite.

There are still future works to be done. We could improve on double-free and use-after-free detection as elaborated in the earlier analysis on the false detection causes. We can also expand our scope to other vulnerabilities such as buffer overflow. Finally, we can better design our solution to support other forms of input other than Wasmati.

Bibliography

[1] `https://webassembly.org/docs/use-cases/`. Accessed: 2022-10-21.

[2] `https://en.wikipedia.org/wiki/WebAssembly`. Accessed: 2022-10-21.

[3] `https://github.com/WebAssembly/design`. Accessed: 2022-10-21.

[4] `https://github.com/acieroid/wassail`. Accessed: 2022-10-21.

[5] `https://github.com/google/AFL`. Accessed: 2022-10-21.

[6] `https://github.com/WAVM/WAVM`. Accessed: 2022-10-21.

[7] `https://aflplus.plus/`. Accessed: 2022-10-21.

[8] `https://souffle-lang.github.io/eqrel-post.html/`. Accessed: 2023-03-19.

[9] `https://samate.nist.gov/SARD/test-suites/112`. Accessed: 2023-03-26.

[10] Martin Bravenboer and Yannis Smaragdakis. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 243–262, New York, NY, USA, 2009. Association for Computing Machinery.

[11] Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. Wasmati: An Efficient Static Vulnerability Scanner for WebAssembly. *Comput. Secur.*, 118(C), jul 2022.

[12] Mechael Pradel Daniel Lehmann, Martin Torp. Fuzzm: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly. 2021.

[13] Keno Haßler and Dominik Maier. WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots. In *Reversing and Offensive-Oriented Trends Symposium*, ROOTS'21, page 23–30, New York, NY, USA, 2022. Association for Computing Machinery.

[14] Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing WebAssembly. `https://arxiv.org/abs/1808.10652`, 2018.

[15] Ondřej Lhoták and Laurie Hendren. Evaluating the Benefits of Context-Sensitive Points-to Analysis Using a BDD-Based Implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), oct 2008.

[16] Mayur Naik, Alex Aiken, and John Whaley. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 308–319, New York, NY, USA, 2006. Association for Computing Machinery.

[17] Joschua Schilling and Tilo Müller. VANDALIR: Vulnerability analyses based on datalog and LLVM-IR. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 19th International Conference, DIMVA 2022, Cagliari, Italy, June 29 –July 1, 2022, Proceedings*, page 96–115, Berlin, Heidelberg, 2022. Springer-Verlag.

[18] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On Fast Large-Scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 196–206, New York, NY, USA, 2016. Association for Computing Machinery.