WASMCHECKER: Effectively Detecting WebAssembly Bugs via Static Program Analysis

Junjie Zhuang Hao Jiang Baojian Hua*
School of Software Engineering, Suzhou Institute for Advanced Research
University of Science and Technology of China
zhuangjj@mail.ustc.edu.cn jh7@mail.ustc.edu.cn bjhua@ustc.edu.cn

***Abstract*—WebAssembly is a promising binary instruction set that is rapidly being adopted in various security-critical scenarios such as blockchain, edge computing, and cloud computing. However, WebAssembly programs are still vulnerable due to its type system and linear memory design defects, leading to security issues of integer overflows and memory vulnerabilities.**

In this paper, we present WASMCHECKER, the first static approach for analyzing WebAssembly programs based on abstract interpretation. WASMCHECKER consists of three key components: a frontend module for parsing and constructing control-flow graphs, a static analysis module for iterative state analysis, and a vulnerability detection module to identify security flaws. We design and implement a software prototype and conduct extensive evaluations. Our results demonstrate that WASMCHECKER effectively achieves a recall of 97.33% on a microbenchmark and accurately identifies 46 vulnerabilities across four real-world projects with a recall of 92%. Additionally, WASMCHECKER is efficient in processing WebAssembly programs smaller than 1 MB with 13.85 seconds on average.

***Index Terms*—WebAssembly, Static Analysis, Vulnerabilities Detection**

I. INTRODUCTION

WebAssembly (Wasm) [1] is a secure, efficient and portable binary instruction set architecture and code distribution format, designed to adapt to the increasingly complex and diverse program execution scenarios in the Internet of Everything era. Wasm ensures safe program executions by incroporating a diverse set of security featureas such as strong type system, software fault isolation, security control flows, and linear memory. Given its security advantages, Wasm is being deployed in security-critical scenarios [30] including blockchain, edge computing, and cloud computing.

Despite its promising potentials as a powerful secure format for code execution, Wasm programs still contain security vulnerabilities of integer overflow and memory safety issues [29]. First, Wasm incroprates fixed-width integers for arithmetic and logic operations. Consequently, arithmetic operations that exceed their maximum size leads to undefined behaviors, causing program errors or security vulnerabilities. Second, Wasm's linear memory design can lead to memory safety issues, including buffer overflows, use-after-frees, and memory leaks, that can be exploited by adversaries to manipulate or corrupt programs. Therefore, it is essential to develop effective approaches to detect integer overflows and memory vulnerabilities in Wasm programs.

In response, researchers have started conducting extensive studies in this direction [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [46]. However, existing studies are still limited in detecting Wasm vulnerabilities. First, some studies propose to utilize secure programming languages such as Go or Rust to generate Wasm programs, thus eliminating vulnerabilities from the sources that may otherwise propagated to Wasm binaries. Unfortunately, many Wasm programs are compiled from insecure languages such as C/C++ [29], comprising integer overflows and memory vulnerabilities that are difficult to detect. Moreover, Wasm programs often consists of binary libraries and third-party modules without sources, hindering source-level vulnerability detection. Second, existing studies only cover certain categories of vulnerabilities but struggle to detect integer overflows and memory issues holistically.

In this paper, we present WASMCHECKER, the first static analysis approach based on abstract interpretation to detect integer overflows and memory vulnerabilities in Wasm programs. We first design a language model for Wasm and build abstract domains of numerical and symbolic values that are used to perform numerical and symbolic analyses, respectively. We then design a translation of Wasm programs to our language model and detect integer overflows and memory vulnerabilities leveraging a fix-point algorithm.

We implement a software prototype for WASMCHECKER and conduct extensive experiments to evaluate its effectiveness, usefulness, and performance. Our results demonstrate that WASMCHECKER effectively detected 146 out of 150 vulnerabilities on a microbenchmark, achieving a recall of 97.3% on average. Meanwhile, WASMCHECKER is useful in detecting 46 out of 50 real-world vulnerabilities, resulting in a recall of 92.0%. Finally, WASMCHECKER is performant to process 941 Wasm programs less than 1MB in 13.85 seconds on average.

To the best of our knowledge, our work represents the first step towards detecting Wasm integer overflows and memory vulnerabilities using the abstract interpretation approach. To summarize, our work makes the following contributions:

- We present a vulnerability detection approach for Wasm programs based on abstract interpretation.

* The corresponding author.

- We establish an abstract domain to track both numerical and symbolic values in Wasm programs, which are then used to effectively detect integer overflows and memory vulnerabilities in Wasm.
- We design and implement a prototype for WASMCHECKER and conduct extensive evaluations to demonstrate its effectiveness, usability, and performance.

The rest of this paper is organized as follows. Section II presents an overview of the background. Section III presents our motivation and the threat model. Section IV presents the approach and design of WASMCHECKER. Section V and VI present the design of abstract interpretation and the prototype implementation, respectively. Section VII presents the evaluations we performed. Section VIII discusses the limitations. Section IX discusses the related work, and Section X concludes.

II. BACKGROUND

To be self-contained, in this section, we present the background knowledge of Wasm and abstract interpretation.

WebAssembly. Wasm [1] is a novel binary instruction set developed by Google and Mozilla since 2015, aiming at safety, efficiency, and portability. Officially standardized by the W3C in 2019, Wasm becomes the fourth web language alongside HTML, CSS, and JavaScript. The initial release, version 1.0, establishes a formal specification, while subsequent updates, including the 2022 draft of version 2.0, further extends its functionality. Wasm guarantees secure program execution by incorporating many features, including strong typing, software fault isolation, and managed memory control. Additionally, Wasm is designed to be language-neutral, supporting various source languages, including C/C++, Rust, and JavaScript.

Abstract interpretation. Static program analysis [12] is a key technique to analyze software properties without executing the code. Specifically, static program analysis automates bug detections by reducing manual efforts and eliminating runtime overhead. Despite its effectiveness in uncovering critical issues prior to deployment, static analysis is inherently limited by the undecidability as established by Rice's theorem. To address this limitation, abstract interpretation [4] provides sound over-approximations of program states at each execution point. It models program behavior using an abstract domain, typically represented as a lattice, where each element denotes a specific program state. Abstract interpretation then uses transfer functions to model the state changes over abstract domains. Depending on the application, abstract interpretation uses a diverse set of abstract domains, includeing intervals, octagons, polyhedra, and congruences, to approximate program behaviors.

III. MOTIVATION

Before delving into the details of WASMCHECKER, we first provide our motivation through an overview of bugs in Wasm with illustrating examples (§ III-A), then discuss the threat model for this work (§ III-B).

```
1  static CURLUcode seturl(const char *url,
2                          …){
3      …
4      size_t urllen = strlen(url);
5      …
6      char *path = malloc(urllen * 2 + 2);
7      …
8  }
```

compile →

```
1  module{
2      …
3      call $strlen
4      local.set $urllen
5      local.get $urllen
6      i32.const 2
7      i32.mul
8      i32.const 2
9      i32.add
10     call $dlmalloc
11     …
12 }
```

Fig. 1: A sample Wasm program illustrating an integer overflow bug we adapted from CVE-2019-5435 [5]. Unrelated details have been omitted for clarity.

A. Wasm Bugs

Despite its design goal of security, Wasm still comprises vulnerabilities. To understand the root causes leading to security issues, we conducted a thorough examination of existing security flaws and identified two main categories of issues: integer overflows and memory safety issues including buffer overflows and use-after-free.

Integer overflows. An integer overflow vulnerability is caused by an integer operation that exceeds the range of an integer. Integer overflow not only affects the correctness of the operation result, but also causes serious errors such as out-of-bounds memory access, leading to security vulnerabilities. Furthermore, since the Wasm standard specification lacks the protection mechanism for integer overflow checking, vulnerabilities in high-level programs can be compiled into the Wasm and persist.

Fig. 1 illustrates an integer overflow in libcurl (CVE-2019-5435) [5]. By providing a large enough string `url`, an adversary triggers an integer overflow in the expression `urllen*2+2`, leading to an undersized buffer allocation for `path`, which further causes a heap buffer overflow vulnerability. By exploiting this vulnerability, an adversery can overwrite arbitrary memory locations and gain control of the target system.

Memory vulnerabilities. Memory vulnerabilities are common in applications written in memory-unsafe languages, significantly impacting program stability and correctness. For example, memory leaks can deplete an application's memory, while buffer overflows may overwrite critical data in memory. Since Wasm supports C/C++ languages, memory issues in these languages can propagate to Wasm.

Fig. 2 presents a double-free vulnerability from CWE-415 [6], in which the pointer `ptr` is freed twice. Since such vulnerabilities are inherently path sensitive, programmers struggle to identify and mitigate them.

Based on our observations, our goal in this work is to utilize static analysis to track both numerical and symbolic values, since these values can capture the patterns of the aforementioned errors. Specifically, we first use interval abstract values to determine the bounds of each integer variable, enabling the detection of potential integer overflows. We then utilize symbolic abstract values to analyze memory operations, enabling the detection of memory issues.

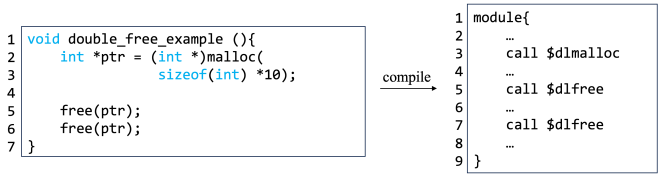


Fig. 2: A sample Wasm program illustrating a double free vulnerability we adapted from CWE-415 [6].

B. Threat Model

Wasm has a comprehensive ecosystem that includes high-level language support, compilation toolchains, binary representation, and virtual machines. This work focuses on developing a secure static detection framework directly on the binary representation. Accordingly, the threat model in this study is built on the following assumptions.

We assume the source code generating Wasm is untrusted, as developers may either intentionally or unintentionally introduce vulnerabilities. Additionally, the use of untrusted third-party libraries can introduce further risks. Even if a developer's own code is free of bugs, third-party libraries may still contain unsafe code, posing security threats.

We also assume that the compilation toolchains are untrusted. The toolchain, which includes compilers, linkers, and other utilities, are complex code bases. Due to their considerable code size and complexity, the toolchains may contain vulnerabilities or defects.

We assume the Wasm virtual machines are secure and trustworthy. Although Wasm virtual machines may comprise vulnerabilities, they do not generate or introduce vulnerabilities into Wasm code being analyzed. Meanwhile, there have been considerable studies on Wasm virtual machine security that are orthogonal and thus supplement this work.

IV. Approach

In this section, we present our approach for detecting integer overflows and memory vulnerabilities for Wasm programs. We first describe our methodology of static analysis (§ IV-A), then present an overview of the high-level workflow of WasmChecker (§ IV-B).

A. Methodology

In general, performing static analysis requires modeling the program's semantics statically. We propose combining numerical and symbolic static analysis and conducting the analysis directly over Wasm binary programs.

Static program analysis is an effective technique for detecting vulnerabilities by examining program code without executing the program. It identifies security issues early in the software development process, thereby reducing the risk of security incidents in production environments. Existing static analysis approaches, including those based on abstract interpretation (e.g., CodeHawk [7] and Clam [8]), have proven effective in detecting software vulnerabilities. However, existing studies and tools cannot directly apply to Wasm programs. First, as a new binary format, Wasm lacks the rich contextual information available in high-level programming languages. Second, Wasm differs significantly from other binary formats in aspects such as memory management, calling conventions, and exception handling. Finally, language feature differences require extensive type system conversions and adjustments when using existing static analysis techniques.

To address these issues, we propose employing interval numerical abstract domains for integer boundary analysis because they are well-suited for modeling low-level integer operations in Wasm programs. Additionally, we use symbolic abstraction for Wasm's memory model, leveraging symbolic abstract domains to analyze memory issues. Compared to symbolic execution-based static analysis, abstract interpretation is more scalable and thus avoids path explosions, which often render pure symbolic execution infeasible.

B. Overview

We present in Fig. 3 an overview of WasmChecker's workflow, comprising three key components: (1) a front-end, (2) a static analyzer, and (3) vulnerability detectors.

Front-end. The front-end takes as input the Wasm binary program to perform program parsing and preparation. This component comprises three functionalities: extracting Wasm program data, generating CFGs, and initializing the static analyzer. First, the front-end extracts essential information, such as variables, functions, and memory layout, from the Wasm binary program to aid in static analysis. Second, the front-end generates a CFG for each function in the target Wasm program to facilitate intra-procedural analysis. Finally, the front-end integrates the extracted program data with the CFGs and passes them to the static analyzer for subsequent analysis.

Static analyzer. The static analyzer performs comprehensive static analysis on the Wasm program and is a core component of WasmChecker. In designing this component, we must address a technical challenge of loop processing. Specifically, loops are inherent in the CFG, making sequential node traversal unsuitable for CFGs with loops. To address this challenge, we employs a weak topological ordering algorithm [11] to decompose the CFG into strongly connected components, ensuring correct program state updates during loop analysis and convergence of the fixed-point algorithm [12]. Next, WasmChecker iteratively analyzes each basic block with the weak topological ordering using a fixed-point algorithm. For each basic block, the visitor traverses each instruction in that block to update the abstract state accordingly. Once the CFG is fully analyzed, the fixed-point algorithm generates final abstract states and passes them to the vulnerability detector for subsequent analysis.

Vulnerability detector. The vulnerability detector verifies and screens abstract states obtained from the static analyzer to identify potential vulnerabilities, leveraging SMT solvers [13]. First, the constraint generator generates a set of constraints based on the abstract states provided by the static analyzer, to capture potential security issues that may arise along

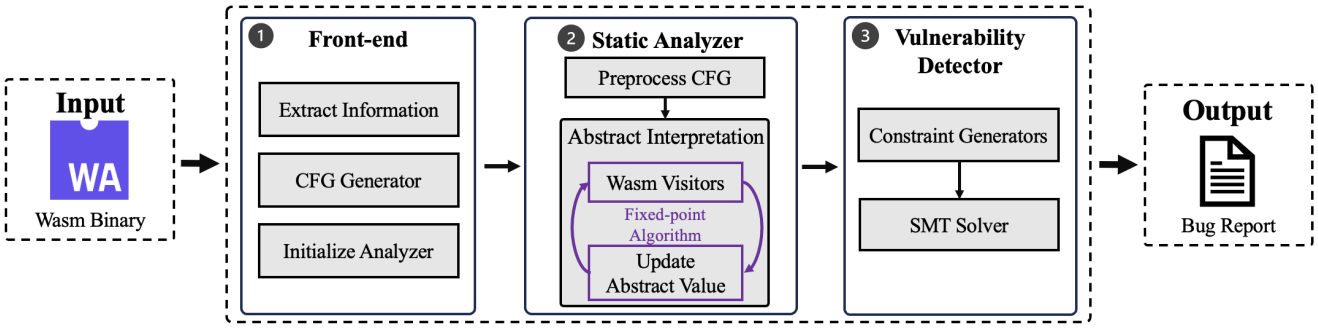


Fig. 3: An overview of WASMCHECKER's workflow.

Value Types	t	::=	$i32 \mid i64 \mid \ldots$
Function Types	ft	::=	$t^* \rightarrow t^*$
Binary op.	b	::=	$\textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{div\_sx} \mid \textbf{rem\_sx} \mid \ldots$
Unary op.	u	::=	$\textbf{abs} \mid \textbf{neg} \mid \ldots$
Load/Store	l	::=	$t.\textbf{load offset}\ i \mid t.\textbf{store offset}\ i$
Local op.	lc	::=	$\textbf{local}.(\textbf{set} \mid \textbf{get} \mid \textbf{tee})\ x$
Global op.	g	::=	$\textbf{global}.(\textbf{set} \mid \textbf{get})\ x$
Call	c	::=	$\textbf{call}\ func_{id} \mid \ldots$
	sx	::=	$s \mid u$
Instructions	e	::=	$t.b \mid t.u \mid l \mid lc \mid g \mid c \mid \textbf{select} \mid \textbf{drop} \mid$ $\textbf{block}\ ft\ e^*\ \textbf{end} \mid \textbf{loop}\ ft\ e^*\ \textbf{end} \mid$ $\textbf{if}\ ft\ e^*\ \textbf{else}\ e^*\ \textbf{end} \mid \textbf{br}\ i \mid \textbf{br\_if}\ i \mid$ $\textbf{return} \mid \textbf{unreachable} \mid \textbf{nop} \mid$ $t.\textbf{const}\ c \mid \ldots$
Functions	f	::=	$ex^*\ \textbf{func}\ ft\ \textbf{local}\ t^*\ e^* \mid ex^*\ \textbf{func}\ ft$
Modules	m	::=	$\textbf{module}\ f^*$

Fig. 4: Core syntax of Wasm language model.

specific execution paths. Next, the SMT solver processes and discharges these constraints, to find specific values that satisfy the constraints. Our use of SMT solvers effectively reduce false positives from static analysis.

Finally, the vulnerability detector generates as output a report that details the types of potential vulnerabilities and their locations, to aid in end users to diagnose and rectify vulnerabilities.

V. ABSTRACT INTERPRETATION

In this section, we present the abstract interpretation used to detect Wasm vulnerabilities. We begin by introducing a simplified language model for Wasm (§ V-A) and the design of abstract domains (§ V-B). We then introduce how to perform symbolic and numerical analysis (§ V-C to § V-E) and describe the fixed-point algorithm (§ V-F).

A. Language Model

We present a simplified language model capturing the core syntax of Wasm. The primary objective of this model is to define abstract domains and abstract values for abstract interpretation and to support numerical and symbolic analysis. The Wasm core language model is represented using abstract syntax, as given by the context-free grammar in Fig. 4. For clarity, irrelevant instructions are omitted.

A Wasm program comprises a module m with multiple functions $f*$. Each function f comprises a list of instructions e. Specifically, Wasm's control flow instructions are inherently structured, including operations $block$, end, if, $else$, br, and $loop$. Certain structural instructions (e.g., $block$, and $loop$, etc.) are paired with end to create logically independent code blocks with distinct execution stacks. Moreover, the $unreachable$ instruction reprents a code path that, if incorrectly executed, will trigger an exception.

Wasm provides a comprehensive set of binary operator b (e.g., add for addition, and mul for multiplication, etc.) and unary operator u (e.g., abs for absolute value, and neg for negation, etc.). Memory load and store operations l resemble standard memory access operations, using the stack top value t as the base address and an offset i, forming an effective address $t + i$. Variable operations lc/g load or store local or global variables, repectively. A function call c invokes the target function with an index id that is an immutable constant.

B. Abstract Values

In static program analysis [12], abstract values are abstract representations of program variables or instructions. In this paper, we classify abstract values into two main categories of numerical abstract values and symbolic abstract ones, based on Wasm's core semantics. Numerical abstract values represent potential numerical ranges for a variable in the program, while symbolic abstract values capture abstract memory addresses.

Numerical abstract values. Numerical abstract values represent numerical variables in a program by abstracting concrete numerical ranges as intervals or more complex geometric forms, such as octagons or polyhedra. These abstractions enable program analysis to approximate the potential values that a variable may take during execution. WASMCHECKER uses intervals [12] to perform program analysis. An interval maps a variable's concrete value v to an interval $[l, u]$ with the lower bound l and the upper bound u.

Symbolic abstract values. Many Wasm instructions, including memory operations and function calls, cannot be easily represented using integer intervals, because they take arbitrarily large ranges. To address this issue, we use symbolic abstract values to track variables in these operations. However, a key challenge is that excessive tracking could consume significant

TABLE I: Symbolic abstract values.

Vulnerabilities	Abstract Values
Double-Free Use-After-Free	Var = \$variable $Alloc = false$ $Free\_counts = 0$ $Counts = 0$ $Pre\_Var = NULL$
Stack-Based Buffer Overflow	$Stack\_base = Int$ $Stack\_top = Int$ $Overflow = false$ BO_Var_num = \$variable

$$\hat{op}([l_1,u_1],[l_2,u_2]) = \left[\min_{x\in[l_1,u_1],y\in[l_2,u_2]} x\ op\ y,\quad \max_{x\in[l_1,u_1],y\in[l_2,u_2]} x\ op\ y\right]$$

$$\text{Negate}(x) = [-u,-l],\quad \text{where } x\in[l,u]$$

$$\text{Abs}(x) = \begin{cases}[l,u], & \text{if } l\geq 0\\ [|u|,|l|], & \text{if } u\leq 0\\ [0,\max(|l|,|u|)], & \text{if } l<0<u\end{cases}$$

Fig. 5: Wasm numerical abstraction operations.

computation resources and thus leads to memory overload or crashes. To address this challenge, WASMCHECKER does not abstract every instruction. Instead, WASMCHECKER utilizes a selective heuristic that determines instructions being abstracted based on the types of vulnerabilities involved.

For Wasm memory safety, we focuse on three categories of vulnerabilities: double frees, use-after-frees, and stack-based buffer overflows. Double free and use-after-free vulnerabilities arise from call instructions that improperly allocate or deallocate memory. Meanwhile, stack-based buffer overflow vulnerabilities are caused by the local, store, and load instructions that handle variable retrieval and storage in linear memory. Therefore, we use symbolic values to abstract the call, local, load, and store instructions.

Based on these observations, we classfity symbolic abstract values into two categories as shown in Table I. The first category detects double free and use-after-free vulnerabilities, comprising five fields with corresponding initial values: Var, $Alloc$, $Free\_counts$, $Counts$, and Pre_Var. The second category detects stack-based buffer overflow vulnerabilities, comprising four fields with corresponding initial values: $Stack\_base$, $Stack\_top$, $Overflow$, and BO_Var_num.

C. Abstract Domains

In abstract interpretation, abstract domains are defined using complete lattices [12]. A lattice is a partially ordered set $(S, \sqsubseteq)$, consisting of a set S and a partial order $\sqsubseteq$. For any two elements $x, y \in S$, the lattice defines two operations: the least upper bound $x \sqcup y$, representing the smallest common upper bound of x and y, and the greatest lower bound $x \sqcap y$, representing their largest common lower bound. A complete lattice extends this definition by ensuring that for any subset $X \subseteq S$, both the least upper bound $\sqcup X$ and the greatest lower bound $\sqcap X$ exist. This property of complete lattices enables fixed-point computation in static analysis, allowing effective analysis of program behavior.

We use both a numerical abstract domain and a symbolic abstract one. Specifically, we use interval lattices [12] to define the numerical abstract domain and use map lattices [12] to define the symbolic abstract domain.

Based on these abstract domains, we define transfer functions to model each instruction. The input and output of transfer functions represent the abstract states before and after the instruction, respectively. Transfer functions process each instruction in a syntax-directed manner according to the language model (§ V-A). For example, to analyze an assignment operation, the function first calculates the abstract value for the right-hand side and then assigns that value to the left-hand side. We highlight some details for the analysis in the following.

D. Numerical Analysis

Following Møller et al. [12], we present in Fig. 5 the abstract operator $\hat{op}$ for Wasm binary arithmetic instructions, including addition (`add`), subtraction (`sub`), multiplication (`mul`), and division (`div_sx`). These operations take as inputs two intervals $[l_1, u_1]$ and $[l_2, u_2]$ and generate as output a result interval bounded by the minimum and maximum of the input ranges. Specifically, when $\hat{op}$ is a division and the divisor interval contains 0, the result interval is the top element $\top$. Moreover, the negation operation $\text{Negate}(x)$ negates both the lower and upper bounds l and h, variable x, while the absolute operation $\text{Abs}(x)$ computes the absolute value for x.

E. Symbolic Analysis

We detect double-free, use-after-free, and stack-based buffer overflows vulnerabilities using symbolic analysis.

To detect double-free and use-after-free vulnerabilities, we focus on memory allocation and deallocation instructions because they lead to the these vulnerabilities. Specifically, for memory allocation instruction, we create a symbolic abstract value AS_{F1}, and set Alloc to true, before assigning Var to the variable a. Moreover, if a is assigned to a new variable b, we copy AS_{F1} to a new abstract value AS_{F2}, and set Var to b, before assigning Pre_Var to a. For deallocation instructions, we first identify the corresponding abstract value AS_F and increment $Free\_counts$ by one. If a load operation addresses a variable with a non-zero $Free\_counts$, we increment $Counts$ in AS_F. Additionally, if AS_{F0} has a predecessor variable Pre_Var, its corresponding abstract value AS_{F1} is also updated.

To detect stack-based buffer overflows, we maintain the stack space location during CFG traversal. If the program attempts to access memory beyond the allocated stack size, signifying a buffer overflow, AS_B is updated with Overflow $=$ true and the affected variable is recorded.

TABLE II: Safety constraints.

Vulnerabilities	Constraints
Integer overflow	$(Type\_MIN \leq Var\_lower) \wedge (Var\_upper \leq Type\_MAX)$
Double free	$Alloc == true \wedge Free\_counts \leq 1$
Use after free	$Alloc == true \wedge Free\_counts \leq 1 \wedge counts == 0$
Stack-based buffer overflow	$Overflow == false$

F. Fixed-point Algorithm

We use a fixed-point algorithm to iteratively analyze the CFG of a Wasm program. A straightforward strategy to traverse the CFG is to follow the topological order of the graph, enabling the calculation of abstract states of current block from its predecessors. However, a key challenge is that loops prevent the construction of topological order. To address this challenge, we employs an approach of weak topological ordering [11]. Our approach separates cyclic nodes from non-cyclic nodes in the CFG, minimizing unnecessary recalculations and speeding up convergence.

VI. Implementation

We implement a prototype system for WasmChecker, comprising three core components: a front-end, a static analyzer, and a vulnerability detector. We highlight some implementation details in the following.

We implement the front-end by primarily leveraging two tools: 1) WABT [15] to extract program information from the Wasm binary; and 2) WasmA [16] to generate the program's CFG.

We implement the static analyzer of WasmChecker in C, with the numerical abstract domain built using the Apron library [17]. The Apron library provides comprehensive support for various numerical abstract domains, allowing the analyzer to efficiently manage numerical variable abstractions. However, the Apron library does not support symbolic abstract domains, so we implement a customized symbolic domain to handle symbolic abstract values. Additionally, we incorporate into the analyzer a fixed-point algorithm to iteratively compute abstract program states.

We implement the vulnerability detector in Python, utilizing the Z3 library's Python interface [13] for SMT discharging. The vulnerability detector first converts the abstract values derived from the program into security constraints according to the rules in Table II, then feeds them into the Z3 solver.

VII. Evaluation

In this section, we present the experiments we conducted to evaluate WasmChecker. Our evaluation is guided by the following research questions.

RQ1: Effectiveness. As WasmChecker is proposed to detect vulnerabilities in Wasm programs, is it effective in accomplishing this goal?

RQ2: Usefulness. Is WasmChecker useful in detecting security vulnerabilities in practical, large-scale, and real-worldWasm applications?

TABLE III: Macrobenchmarks of 4 real-world applications.

Application	Domain	Stars(k)	Vulnerabilities
Libpng	PNG image processing	1.2	14
Libcurl	Url transfer library	35.2	14
Libexpat	XML parser	1.0	13
Flac	Audio codec	1.6	9

RQ3: Performance. What is WasmChecker's performance to analyze Wasm programs of varying sizes?

We perform the experiments and measurements on a server with one 4 physical Intel i7 core (8 hyperthread) CPU and 128 GB of RAM running Ubuntu 20.04.

A. Datasets

To conduct the evaluation, we create two datasets: a microbenchmark and a macrobenchmark derived from real-world applications.

Microbenchmarks. Evaluating the effectiveness of detection techniques (WasmChecker in this work) requires a dataset of Wasm programs with groundtruth of known vulnerabilities. However, to the best of our knowledge, such a dataset does not currently exist. Therefore, we manually create a dataset called WBench to assess detection capabilities. WBench Currently consists of 300 Wasm programs selected from the Juliet Test Suite [18], covering 12 categories of integer overflow vulnerabilities (both signed and unsigned 32-bit and 64-bit addition, subtraction, and multiplication) related to CWE-190 [19], and 3 categories from CWE-121 [20], CWE-415 [6], and CWE-416 [21]. Each category contains 10 vulnerable test cases and 10 corresponding non-vulnerable test cases. We will continue to maintain and expand this benchmark by adding more programs.

Real-world applications. For better benchmark representation, we select the following real-world applications based on three criteria: 1) they are widely used and open-source; 2) they can be smoothly ported to Wasm; and 3) they contain known security vulnerabilities or are written in unsafe high-level languages like C/C++.

Based on these selection criteria, we select four real-world applications from different domains as shown in Table III: 1) Libpng [22], the official library for handling PNG files, which provides a range of functions that enable developers to read, write, manipulate, and convert PNG files within their applications; 2) libcurl [23], a widely used URL transfer library that supporting multiple protocols, including HTTP, HTTPS, FTP; 3) libexpat [24], an open-source library for parsing XML files, offering an efficient and flexible interface for reading and processing XML data; and 4) FLAC [25], an open-source free lossless audio codec tool that ensures no quality is lost during audio compression.

Although these four projects align with our criteria for selecting real-world applications, a challenge remains in using them to evaluate WasmChecker: the collected CVEs are all fixed vulnerabilities that no longer exist in newer versions. To address this issue, we apply a vulnerability injection technique

TABLE IV: Experimental results for the microbenchmark.

Test	Vulnerability Type	WasmChecker				Wasmati			
		TP	FP	TN	FN	TP	FP	TN	FN
1	unsigned_int_add	10	0	10	0	0	0	10	10
2	signed_int_add	10	0	10	0	0	0	10	10
3	unsigned_int_sub	10	0	10	0	0	0	10	10
4	signed_int_sub	10	0	10	0	0	0	10	10
5	unsigned_int_mul	10	0	10	0	0	0	10	10
6	signed_int_mul	10	0	10	0	0	0	10	10
7	unsigned_int64_add	10	0	10	0	0	0	10	10
8	signed_int64_add	10	0	10	0	0	0	10	10
9	unsigned_int64_sub	10	0	10	0	0	0	10	10
10	signed_int64_sub	10	0	10	0	0	0	10	10
11	unsigned_int64_mul	10	0	10	0	0	0	10	10
12	signed_int64_mul	10	0	10	0	0	0	10	10
13	double free	10	0	10	0	6	0	10	4
14	use after free	10	0	10	0	4	0	10	6
15	stack-based BO	6	0	10	4	2	0	10	8

that modifies the original sources to inject the already-fixed vulnerabilities. The fourth column of Table III shows the distribution of 50 injected vulnerabilities in each project, respectively.

B. Effectiveness

To answer RQ1 by demonstrating the effectiveness of WASMCHECKER, we conduct experiments to detect vulnerabilities. We first compile all test cases from WBench into Wasm using the Emscripten compiler, then feed generated Wasm binaries to WASMCHECKER.

We present experimental results in Table IV. WASMCHECKER detected 146 out of 150 vulnerable test cases, achieving a recall of 97.33%. Meanwhile, WASMCHECKER produces no false positives. These results demonstrate WASMCHECKER is effective in detecting integer overflow and memory safety vulnerabilities. To investigate why WASMCHECKER missed 4 stack-based buffer overflow vulnerabilities, we conduct a manual inspection of the relevant source code. This inspection revealed a root cause of insufficient stack address checking. Specicifally, WASMCHECKER only checks if stack accesses are with the allocated space, but does not verify if these accesses remain within the same stack frame.

Futhermore, to gain an understanding of WASMCHECKER detection capabilitiy, we compare WASMCHECKER with Wasmati, a state-of-the-art detection tool for Wasm. As shown in Table IV, Wasmati did not detect any integer overflows, because it lacks specific mechanisms for detection such vulnerabilities. Meanwhile, Wasmati detected 6 double-frees, 3 use-after-free, and 2 stack-based buffer overflows. This result shows that WASMCHECKER outperforms Wasmati regarding bug detection.

C. Usefulness

To answer RQ2 and demonstrate the usefulness of WASMCHECKER, we apply WASMCHECKER to real-world Wasm programs.

We present in Table V the experimental results on real-world projects. For all projects, WASMCHECKER achieves high recalls of 92.8%, 85.7%, 92.3%, and 100%, respectively, with an average of 92.0%. Moreover, WASMCHECKER does

TABLE V: Experiemental results on real-world programs.

Program	Size	Functions	Analysis Time(sec)	No. Bugs (Detected/All, Recall)	Wasmati (Detected/All, Recall)
libpng	672KB	906	97	13/14, 92.8%	5/14, 35.7%
libcurl	1.84MB	4,578	323	12/14, 85.7%	6/14, 42.8%
libexpat	101KB	345	58	12/13, 92.3%	2/13, 15.3%
flac	755KB	1,150	105	9/9, 100%	5/9, 55.5%

generate flase negatives (1 for libpng, 2 for libcurl, and 1 for libexpat). To investitgate root causes leading to these FPs, we conduct a manual analysis of the relevant source code that reveals a key reason is the inter-procedural usage of pointers. Since WASMCHECKER only performs intra-procedural analysis, it lacks the inter-procedural information that is essential to analyze such pointer usage, hindering the detection of such vulnerabilities.

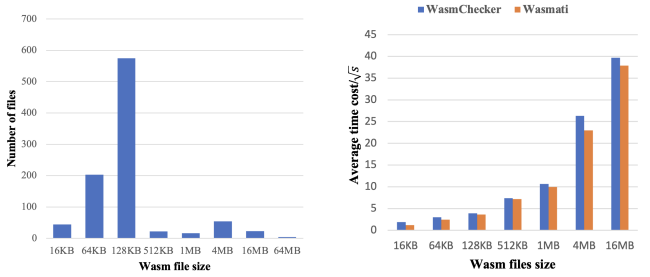
To compare WASMCHECKER with Wasmati, we apply Wasmati to these real-world projects. Wasmati achieves recalls of 35.7%, 42.8%, 15.3%, and 55.5% for each project, respectively. A manual analysis of the source code revealed that the FPs are caused by Wasmati's lack of integer overflow detection capabilities. Furthermore, Wasmati does not perform interprocedural analysis, missing vulnerabilities arising from interprocedural contexts.

D. Performance

To answer RQ3 by investigating the performance, we apply WASMCHECKER to a dataset provided by Zheng et al. [26], consisting of 941 Wasm binary programs from various domains, including gaming, media processing, and databases. The file sizes for this dataset range from 16K to 64M, and their distributions are shown in Fig. 6a, where the x-axis represents file size intervals and the y-axis indicates the number of files in each interval. Moreover, as some files being analyzed are large, we set a timeout of 30 minutes (1,800 seconds) per file, because we observe that most files can be processed within this timeout.

We present in Fig. 6b the average analysis time, where the x-axis represents the files in increasing order of sizes, and the y-axis represent the corresponding analysis time. We exclude from the dataset all files larger than 16MB, because they require more than 1,800 seconds and thus trigger the timeout. Additionally, we also exclude files that are less than 16MB but trigger timeout. The average analysis time is 13.85 seconds for files smaller than 1MB, but increases significantly with file sizes.

We compare WASMCHECKER with Wasmati in terms of performance, and observe Wasmati is more efficient than WASMCHECKER, although the divergence is neglectable. WASMCHECKER's inefficiency is largely due to its leverage of SMT solvers to discharge contraints which are time-consuming. Consequently, one important future work is to refine the design of abstract domains in WASMCHECKER to improve performance.



(a) Binary file size distribution of the 941 Wasm programs.

(b) Average running time for Wasm file of diverse sizes.

Fig. 6: Performance results for files of diverse sizes.

VIII. Discussion

Interprocedural analysis. WasmChecker in its current implementation only supports intraprocedural analysis, lacking the interprocedural capability to accurately track information across function call/returns. In our future work, we aim to address this issue by introducing interprocedural analysis [27] [28] to perform more precise and accurate analysis.

Source languages and compilers. WasmChecker targets Wasm programs and thus is neutral to specific source languages or compilers. However, the current test set comprises only Wasm programs compiled from C/C++ using the Emscripten compiler, which may introduce potential bias. Existing studies demonstrate that different compilers significantly affect the instructions and memory layouts of the generated Wasm binaries [29], thus addressing these variations introduced by different compilers is an essential direction for future exploration. Furthermore, as the Wasm ecosystem increasingly incroprates more languages like Rust and Go, developing effective vulnerability analysis techniques for the Wasm programs generated from them is essential.

Other vulnerabilities. Although WasmChecker effectively detects integer overflows and memory safety issues, Wasm programs may also contain other categories of vulnerabilities, such as indirect call redirection [29] and heap metadata corruption [29]. Expanding the definition of WasmChecker's abstract domains to cover a broader range of vulnerabilities is an important research direction to explore.

IX. Related Work

Empirical security studies. There have been several empirical studies to invesitigate Wasm and its security vulnerabilities. Lehmann et al. [29] studied the security of Wasm binaries, while Musch et al. [30] analyzed the prevalence of Wasm in the wild. Romano et al. [31] studied compiler errors and their remediation, and Wang et al. [32] examined the root causes of runtime errors in Wasm.

However, these studies do not address the problem of vulnerability detection as this work does.

Static analysis. Extensive studies have been conducted on static analysis for Wasm. Stiévenart et al. [33] developed an approach that generates summaries for each Wasm function based on control flow and call graphs. These summaries capture the information flow between function parameters, return values, and global variables, which are approximated for the entire program using a fixed-point algorithm. Brito et al. [34] proposed a method that constructs code property graphs (CPGs) for Wasm programs and employs a query specification language to traverse these graphs, to detect ten common memory safety vulnerabilities through four different query patterns. Romano et al. [35] focused on analyzing cryptojacking attacks by unifying programs that contain both Wasm and JavaScript into a single Wasm program for semantic analysis. Naseem et al. [36] transformed Wasm programs into grayscale images and used a pre-trained convolutional neural network classifier to identify cryptojacking vulnerabilities by analyzing these images. Johnson et al. [37] focused on control flow security, linear memory isolation, stack frame integrity, and stack isolation by decompiling Wasm programs to x86-64 code and applying abstract interpretation analysis.

However, our work differs from these studies in that we introduce a static analysis method based on abstract interpretation that analyzes Wasm binaries directly.

Dynamic analysis. Many existing studies on Wasm security use dynamic analysis, including fuzz testings [38] [39] [40] [41] [42], cryptojacking attacks detections [43] [44] [45], and runtime taint tracking [46] [47].

However, a major limitation of dynamic analyses is that they often sacrifice soundness for pratical usability, missing key vulnerabilities that are not exercised during execution. In constrast, this work use a static analysis approach that is sound, supplementing the dynamic analysis approach.

X. Conclusion

This paper presents a static analysis approach for detecting Wasm programs based on abstract interpretation. We first design a language model for Wasm and build abstract domains of numerical and symbolic values that are used to perform numerical and symbolic analyses, respectively. We then design a translation of Wasm programs to our language model and detect integer overflow and memory security vulnerabilities leveraging a fix-point algorithm. We develop a prototype system WasmChecker and conduct extensive experiments to evaluate it. Our results demonstrate its WasmChecker is effective in detecting vulnerabilities with high recalls and is useful to real-world and large Wasm projects with acceptable performance. Overall, this work represents a first step towards applying abstract interpretation approach to detect Wasm vulnerabilities, making Wasm not only an efficient but also a more secure programming language.

References

[1] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., ... & Bastien, J. F. (2017). Bringing the web up to speed with WebAssembly. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York. pp. 185-200.

[2] Stiévenart, Q., De Roover, C., & Ghafari, M. (2022). Security risks of porting c programs to WebAssembly. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. New York. pp. 1713-1722.
[3] Rice, H. G. (1953), Classes of recursively enumerable sets and their decision problems, Transactions of the American Mathematical Society, 74 (2): 358–366.
[4] Cousot, P. (1996). Abstract interpretation. ACM Computing Surveys (CSUR), 28(2): 324-328.
[5] Microsoft. (2019) CVE-2019-5435. https://www.cve.org/CVERecord?id=CVE-2019-5435.
[6] MITRE. (2023) CWE-415: Double Free. https://cwe.mitre.org/data/definitions/415.html.
[7] Anton, J., Bush, E., Goldberg, A., Havelund, K., Smith, D., & Venet, A. (2006). Towards the industrial scale development of custom static analyzers. Proceedings of the Static Analysis Summit, 500-262: 36-40.
[8] Gurfinkel, A., & Navas, J. A. (2021). Abstract interpretation of LLVM with a region-based memory model. In International Workshop on Numerical Software Verification, 122-144.
[9] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., & Sundaresan, V. (2010). Soot: A Java bytecode optimization framework. In CASCON First Decade High Impact Papers. 214-224.
[10] Liang, H., Wang, L., Wu, D., & Xu, J. (2016). MLSA: a static bugs analysis tool based on LLVM IR. In 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). 407-412.
[11] Bourdoncle, F. (2005). Efficient chaotic iteration strategies with widenings. In Formal Methods in Programming and Their Applications: International Conference Academgorodok, Novosibirsk, Russia June 28–July 2, 1993 Proceedings. Berlin. pp. 128-141.
[12] Møller, A., & Schwartzbach, M. I. (2024). Static program analysis. https://cs.au.dk/ amoeller/spa/spa.pdf.
[13] De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In International conference on Tools and Algorithms for the Construction and Analysis of Systems. Berlin. pp. 337-340.
[14] Cousot, P., & Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. New York. pp. 238-252.
[15] Zakai A, Dawborn T, Shawabkeh M, et al. (2024). WABT: The WebAssembly Binary Toolkit. https://github.com/WebAssembly/wabt.
[16] Darmstadt T. (2024). Wasma: A static analysis framework for webassembly. https://github.com/stg-tud/wasma.
[17] Jeannet, B., & Miné, A. (2009). Apron: A library of numerical abstract domains for static analysis. In International Conference on Computer Aided Verification. Berlin. pp. 661-667.
[18] Center N. (2024). Juliet c/c++ 1.3. https://samate.nist.gov/SARD/test-suites/112.
[19] CWE. (2024). Cwe-190: Integer overflow or wraparound. https://cwe.mitre.org/data/definitions/190.html.
[20] CWE. (2024). CWE-121: Stack-based Buffer Overflow. https://cwe.mitre.org/data/definitions/121.html.
[21] CWE. (2024). CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html.
[22] Truta C. (2024). libpng. https://github.com/pnggroup/libpng.
[23] Haxx. (2024). libcurl. https://github.com/curl/curl.
[24] Clark J. (2024). libexpat. https://github.com/libexpat/libexpat.
[25] de Castro Lopo E. (2024). Free Lossless Audio Codec (FLAC). https://github.com/xiph/flac.
[26] Zheng, W., & Hua, B. (2024). WASMDYPA: Effectively Detecting WebAssembly Bugs via Dynamic Program Analysis. In 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). Piscataway, NJ. pp. 296-307.
[27] Albarghouthi, A., Kumar, R., Nori, A. V., & Rajamani, S. K. (2012). Parallelizing top-down interprocedural analyses. ACM SIGPLAN Notices, 47(6): 217-228.
[28] Cheng, B. C., & Hwu, W. M. W. (2000). Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation. New York. pp. 57-69.
[29] Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything old is new again: Binary security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20). Berkeley, CA. pp. 217-234.
[30] Musch, M., Wressnegger, C., Johns, M., & Rieck, K. (2019). New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference. Gothenburg. Berlin. pp. 23-42.
[31] Romano, A., Liu, X., Kwon, Y., & Wang, W. (2021, November). An empirical study of bugs in webassembly compilers. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering. Piscataway, NJ. pp. 42-54.
[32] Wang, Y., Zhou, Z., Ren, Z., Liu, D., & Jiang, H. (2023). A comprehensive study of webassembly runtime bugs. In 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). Piscataway, NJ. pp. 355-366.
[33] Stiévenart, Q., & De Roover, C. (2020). Compositional information flow analysis for WebAssembly programs. In 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM). Piscataway, NJ. pp. 13-24.
[34] Brito, T., Lopes, P., Santos, N., & Santos, J. F. (2022). Wasmati: An efficient static vulnerability scanner for

WebAssembly. Computers & Security, 118, 102745.
[35] Romano, A., Zheng, Y., & Wang, W. (2020). Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. Piscataway, NJ. pp. 1129-1140.
[36] Naseem, F. N., Aris, A., Babun, L., Tekiner, E., & Uluagac, A. S. (2021). MINOS: A Lightweight Real-Time Cryptojacking Detection System. In NDSS.
[37] Johnson, E., Thien, D., Alhessi, Y., Narayan, S., Brown, F., Lerner, S., ... & Stefan, D. (2021). SFI safety for native-compiled Wasm. In Network and Distributed Systems Security (NDSS) Symposium.
[38] Haßler, K., & Maier, D. (2021). Wafl: Binary-only WebAssembly fuzzing with fast snapshots. In Reversing and Offensive-oriented Trends Symposium. Vienna. pp. 23-30.
[39] Lehmann, D., Torp, M. T., & Pradel, M. (2021). Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly. arXiv preprint arXiv:2110.15433.
[40] Chen, W., Sun, Z., Wang, H., Luo, X., Cai, H., & Wu, L. (2022). Wasai: uncovering vulnerabilities in wasm smart contracts. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. New York. pp. 703-715.
[41] Lin M, & Zhang C. (2020). Fuzzing scheme for WebAssembly virtual machine. Network Security Technology & Application, 2(6): 15-18.
[42] Jiang, B., Li, Z., Huang, Y., Zhang, Z., & Chan, W. (2022). Wasmfuzzer: A fuzzer for webassembly virtual machines. In 34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022. Pittsburgh, Pennsylvania. pp. 537-542.
[43] Bian, W., Meng, W., & Zhang, M. (2020). Minethrottle: Defending against wasm in-browser cryptojacking. In Proceedings of The Web Conference 2020. New York. pp. 3112-3118.
[44] Kelton, C., Balasubramanian, A., Raghavendra, R., & Srivatsa, M. (2020). Browser-based deep behavioral detection of web cryptomining with coinspy. In Workshop on measurements, attacks, and defenses for the web (MADWeb). Reston, VA. pp. 1-12.
[45] Konoth, R. K., Vineti, E., Moonsamy, V., Lindorfer, M., Kruegel, C., Bos, H., & Vigna, G. (2018). Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. New York. pp. 1714-1730.
[46] Szanto, A., Tamm, T., & Pagnoni, A. (2018). Taint tracking for webassembly. arXiv preprint arXiv:1807.08349.
[47] Fu, W., Lin, R., & Inge, D. (2018). Taintassembly: Taint-based information flow control tracking for webassembly. arXiv preprint arXiv:1802.01050.