Is This the Same Code? A Comprehensive Study of Decompilation Techniques for WebAssembly Binaries

Wei-Cheng Wu[1⋆], Yutian Yan[2⋆], Hallgrimur David Egilsson[2], David Park[2], Steven Chan[2], Christophe Hauser[1], and Weihang Wang[2]

[1] Dartmouth College, Hanover NH 03755, USA
{wei-cheng.wu.gr,christophe.hauser}@dartmouth.edu
[2] University of Southern California, Los Angeles CA 90007, USA
{yutianya,egilsson,dpark946,sychan,weihangw}@usc.edu

Abstract. WebAssembly (abbreviated WASM) is a low-level bytecode language designed for client-side execution in web browsers. As WASM continues to gain widespread adoption and its security concerns, the need for decompilation techniques that recover high-level source code from WASM binaries has grown. However, little research has been done to assess the quality of decompiled code from WASM. This paper aims to fill this gap by conducting a comprehensive comparative analysis between decompiled C code from WASM binaries and state-of-the-art native binary decompilers. To achieve this goal, we presented a novel framework for empirically evaluating C-based decompilers from various aspects, thus assessing the proficiency of WASM decompilers in generating readable and correct code when compared to native binary decompilers. Specifically, we evaluated the decompiled code's *correctness*, *readability*, and *structural similarity* with the original code from current WASM decompilers. We validated the proposed metrics' practicality in decompiler assessment and provided insightful observations regarding the characteristics and constraints of existing decompiled code. By encouraging improvements in these tools, we seek to enhance their use in critical tasks such as auditing and sandboxing third-party libraries. This, in turn, contributes to bolstering the security and reliability of software systems that rely on WASM and native binaries.

Keywords: WebAssembly· Decompiler· Reverse Engineering.

1 Introduction

WebAssembly (WASM) is a portable, low-level language designed for near-native execution on the web [40]. Since it was first introduced by Haas et al. in 2017 [16] and initially developed for web browsers [43], its application has extended to diverse areas, including Internet of Things [15,24], mobile devices [37], smart

⋆ Both authors contributed equally to this research.

contracts [28], and its dedicated runtime environments [1,14]. Notably, WASM is commonly used as a compilation target for popular high-level languages like C, C++, and Rust [18].

Given the growing adoption of WASM, inspecting third-party binaries for potential security vulnerabilities has become imperative. However, the low-level nature of WASM bytecode makes it challenging to audit compared to high-level code, such as C. Additionally, 28.8% of WASM binaries are minified [18], stripping away variable/function names and making manual inspection cumbersome. To address these challenges, security experts can leverage decompilers to analyze high-level code instead of grappling with thousands of lines of minified low-level WASM code.

Nevertheless, WASM decompilers have received less attention than decompilers designed for native binaries. Over the years, significant progress has been made in developing powerful native binary decompilers that can accurately generate decompiled code for C and C++ programs. Recent studies have also focused on enhancing the readability of decompiled code [8,41].

To this end, we perform a comprehensive study to assess the effectiveness of state-of-the-art WASM decompilers.

We will be approaching from two directions, as there are two different types of decompilers for WASM: decompilers tailored for readability, and decompilers focus on correctness, i.e., the decompiled code adheres to the behavior of the original WASM program. Their performance is compared with off-the-shelf native binary decompilers [21,35]. To evaluate these decompilers, we utilize various widely-used complexity metrics for source code and adopt methodologies presented in previous studies [25,41,46]. Our study focuses on the following three aspects:

- **Correctness** of the decompiled code (Section 3.1);
- **Readability** of the decompiled code (Section 3.2);
- **Structural similarity** between the decompiled code and the original code (Section 3.3).

With our research, we aim to draw attention to the capabilities of WASM decompilers and the performance of native binary decompilers. By encouraging improvements in these tools, we seek to enhance their use in essential tasks such as auditing and sandboxing third-party libraries [32]. This, in turn, contributes to bolstering the security and reliability of software systems that rely on WASM and native binaries.

In summary, this paper makes the following contributions:

- **First attempt to evaluate WebAssembly decompilers:** As far as we know, we are the first ones to investigate the correctness, readability, and structural similarity of decompilers for WebAssembly. We have created quantifiable and comprehensive metrics, which can be used as useful tools to evaluate the quality of decompiled code. The decompilers are tested on popular benchmarks, synthesized programs, and real-life scenarios to assess their adaptability to various inputs.

– **Inconsistencies of decompiling WASM vs. native binaries:** Our investigation delved into the underlying reasons for inconsistencies arising when decompiling WASM vs. native binaries. These observations highlighted several critical issues, including aggressive compiler optimization, WASM language features, and platform-specific concerns.
– **First analysis framework for decompiled C code:** We propose the first comprehensive analysis framework to empirically measure the quality of decompiled C code. Our benchmark and analysis framework are publicly available [3], which can be used for future studies and further advancements in decompilers and WASM analysis.

The rest of the paper is structured as follows: In Section 2, we present examples of the decompiled code from current WASM decompilers. In Section 3 and Section 4, we describe the metrics and mechanisms we used to evaluate the decompilers. The results of our evaluation are presented in Section 5. In Section 6, we discuss the limitations of our study and future work. Finally, we discuss related work in Section 7 and conclude the paper in Section 8.

2 Motivation

The main goal of this paper is to assess the effectiveness of current decompilers for the WebAssembly (WASM) language and determine their limitations, particularly in their ability to reconstruct source code from WASM precisely. We present a motivating example to demonstrate how code readability is enhanced by introducing a decompiler. The example is taken from the paper [23] and showcases a stack overflow vulnerability in both C and WASM.

The C source code in Listing 1.1 contains the `vulnerable` function, which utilizes the unsafe `strcpy` function. This is dangerous because `strcpy` lacks an input size check and can trigger the buffer overflow if `bar` is larger than `buf`, which is exactly what Listing 1.1 does.

The C source code is compiled into WASM code using Emscripten [10]. WASM code is binary and not human-readable, so it is converted into WAT (WebAssembly Text) format. The WAT code is shown in Listing 1.3. The `$f1` function (corresponding to the `vulnerable` function in the source code, function name striped) calls the `$f2` (`stpcpy` as aforesaid) function. `$f2` implements the similar functionality as the `strcpy` function in the C standard library[4]. The `$f2` function has 101 LoC (Lines of Code). Since this function is implemented in WASM and does not depend on external functions, one cannot confirm its vulnerability nature without fully understanding it and realizing its similarity to the `stpcpy` in the C standard library.

[3] `https://github.com/spencerwuwu/WASM-decomp_eval`
[4] The only difference between `stpcpy` and `strcpy` is the return value. `stpcpy` returns a pointer to the terminating `\0` character of the target string while `strcpy` returns a pointer to the beginning of the string. Both functions are vulnerable to stack overflow.

```c
void vulnerable(char *bar) {
  char buf[8];
  strcpy(buf, bar); // no bounds
       checking
}

int main() {
  vulnerable("aaaaaaaaaaaaaaaaaa\04");
}
```

Listing 1.1: C program performing stack overflow (simplified)

```c
function f1(a:int) {
  var b:int = g_a - 16;
  g_a = b;
  f2(b + 8, a);
  g_a = b + 16;
}

function f2(a:int, b:int) {
  var c:int;
  if ((a ^ b) & 3) goto B_b;
  // 27 lines omitted
}
```

Listing 1.2: Decompiled code (simplified) by wasm-decompile

```
(func $f1 (type 14) (param i32) ;;
     function vulnerable
    (local i32)
    global.get 0   ;; stack header
    i32.const 16
    i32.sub
    local.tee 1
    global.set 0
    local.get 1
    i32.const 8
    i32.add        ;; parameter buf
    local.get 0    ;; parameter bar
    call $f2 ;; call function f2
         (stpcpy)
    ;; 4 lines omitted
    )
(func $f2 (type 21) (param i32 i32)
    (local i32)
    block
        block
            local.get 0
            ;; 93 lines omitted
            br_if 0 (;@2;)
        end
    end)
```

Listing 1.3: Wasm program in WAT (simplified)

```c
int sum(int n) {
  int sum = 0;
  for (int i = 1; i <= n; i++)
    sum += i;
  return sum;
}
```

Listing 1.4: C program performing summation

```c
int f2(int n) {
   if (n <= 0) {return 0; }
return n - 1 * n - 2 >> 1 + n << 1 -
     1}
```

Listing 1.5: Decompiled code (simplified) by wasm-decompile

We use *wasm-decompile* [12] to convert the WASM program back into C-like code (Listing 1.2). The decompiled code has improved readability with 31 LOC (C-like) compared to 101 LOC of WAT. The decreased LoC indicates the improved readability of the program. However, we also need to point out that compared to the original program in Listing 1.1, the decompiled program has increased LoC, and the parameter data type of function `f1` is different from `vulnerable` (we would explain this in Section 5), which shows the improvement comparing with the WASM binary and limitations regarding the original code.

Besides readability, WASM decompilers may generate incorrect results, which breaks the correctness of the decompiled code. To demonstrate this, we tested a simple C program that contains a `sum` function (see Listing 1.4). We compiled the C program into WASM and retained the exported function symbols. We then used *wasm-decompile* to convert the WASM program into C-like code (Listing 1.5) and found that the decompiled code was incorrect. Specifically, line 3 in the decompiled code, which calculates the sum from 1 to n, returns $2n - 2$ instead of the correct result, $n(n + 1)/2$. This could potentially mislead reverse

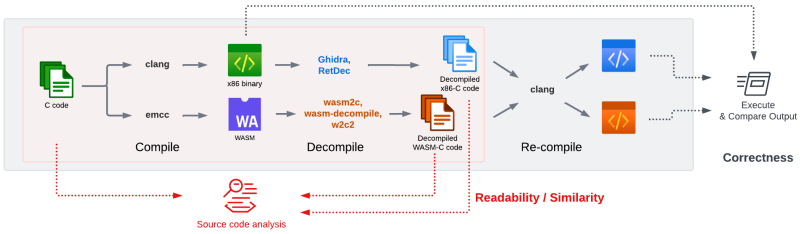


Fig. 1: Overview of methodology workflow

engineers working with the WASM code, leading them to manually check the WASM code instead of the decompiled one to ensure its functionality.

The observed differences between the original and decompiled code raised the following intuitive questions that we aim to address in this paper:

- **Q1:** To what extent can we trust the decompiled code to present the underlying program's functionality accurately? In other words, how do we examine the **correctness** of the decompiled code? Ensuring that the decompiled code faithfully reproduces the original program's functionality is crucial for reliable reverse engineering and code analysis.
- **Q2:** Some examples of decompiled code (e.g., Listing 1.2) could still be too long to understand for developers. Is there an automated and objective way to measure the **readability** of the decompiled code? Quantifying readability would provide valuable insights into the code's understandability, maintainability, and ease of debugging.
- **Q3:** Sometimes, even if a piece of code is initially difficult to understand, it may still be comprehensible if we can observe a similar structure in the original source code. Can we quantify the **structural similarity** between the decompiled code and the original source code in addition to assessing readability? Measuring structural similarity can help determine how closely the decompiled code resembles the original, aiding in code comprehension and validation.

Addressing these questions is crucial for advancing the capabilities of decompilers and enhancing the accuracy and usability of decompiled code in various application domains.

3 Methodology

To answer **Q1-Q3** from Section 2, we introduced a series of techniques to comprehensively evaluate existing WASM decompilers in terms of the decompiled output's correctness, readability, and structural similarity to the original code.

To establish a baseline for comparison, we used the decompiled results obtained from native binary decompilers. We reasoned that these tools have undergone rigorous development and real-world utilization over the years, making them reliable reference points.

To conduct our evaluation, we carefully selected a consistent set of C programs and compiled them separately into both native binaries and WASM files. Subsequently, we applied the chosen decompilers to reverse-engineer the executable files and obtain the decompiled C code from both sides. We then conducted a function-level comparison between the decompiled outputs from native binaries and WASM.

For correctness, we further re-compiled the decompiled code into executable files and executed them to test whether they preserved the same functionality as the original C code. This step allows us to verify the accuracy of the decompilers in faithfully reproducing the intended behavior of the original programs.

Regarding readability and structural similarity, we applied various metrics commonly used in software engineering to evaluate the decompiled code.

An overview workflow of our methodology can be found in Figure 1.

3.1 Correctness

The accuracy of decompilation tools plays a crucial role in software development, security, and reverse engineering. A perfectly accurate decompiled program should be capable of being re-compiled back into executable files and exhibit the same functionality as the original binary.

However, most state-of-the-art decompilers prioritize enhancing the readability of the decompiled code, which often results in C pseudo-code that may not strictly adhere to correct syntax. Additionally, these tools typically focus on analyzing the semantics per function and may struggle with reasoning about data structures or global variable access, which are commonly used in C programs.

In short, while readability is a crucial aspect of decompilation, accessing the correctness of decompiled code presents two main challenges:

C1: Decompilers often produce C-like code that is not directly re-compilable, emphasizing readability over perfect accuracy in reproducing the original binary.
C2: Decompilers may face difficulties effectively handling global variables and memory pointers, which can lead to discrepancies in the decompiled code.

On the other hand, WASM decompilers inherently incorporate features that facilitate code re-compilability (effectively addressing **C1**). However, challenges persist in generating re-executable programs due to the inherent nature of WASM and its primary use as a library and function within web browsers. Two properties of WASM introduce divergences in the decompiled code:

- **Differences in memory management and representation:**
  WASM employs a stack-based, linear memory model, which contrasts with the memory management and representation used in traditional native binaries. This property poses **C2** as a challenge for WASM decompilers.
- **Usage of embedded environment-specific functions:**
  During decompilation, certain operations, such as standard C library functions that access system resources like memory, files, networks, and devices, are

transformed into system-call-like functions that exclusively exist within the JavaScript environment of the web browser.

The second property of WASM leads to the third challenge for evaluating the correctness of decompiled WASM:

C3: While decompiled WASM programs are represented as C code, they commonly cannot be executed directly in a native environment due to the absence of necessary runtime libraries. Additionally, these decompiled programs faithfully reproduce operations as if they were intended to be used as library modules within web browsers. As a result, they lack entry points for direct execution outside the WASM environment, making it infeasible to execute them in a non-WASM context.

To address the three challenges (**C1**, **C2**, and **C3**) in evaluating the correctness of decompiled WASM code, we leveraged the synthesized code generated by DecFuzzer [25] and implemented a sandbox environment for the execution of *wasm2c*'s decompiled code.

To the best of our knowledge, DecFuzzer is the only publicly available work that empirically assesses the correctness of native binary decompilers. It generated code containing only local-variable arithmetic computations into one single "core function", ensuring full decompilability for all existing decompilers (**C2**). Subsequently, they syntactically restructured the decompiled code to make it re-compilable (**C1**). The process concluded with the re-compilation of the code, and they compared the output of the re-compiled binary with the original binary's execution results to verify correctness.

As the synthesized programs generated by DecFuzzer do not utilize any library functions or global variables, they are suitable for porting to WASM decompilers too. We modified them to tackle further **C2** and **C3** for WASM. Specifically, we first transformed the synthesized programs into functionally standalone modules. This step ensures the compiled WASM is self-contained and does not depend on external runtime libraries and global variables (**C2**). Next, we implemented a sandbox environment to load and execute the decompiled WASM code (**C3**). This essentially makes it possible to execute the decompiled WASM code in the native binary environment so that we can compare it with the original binary's execution results to verify correctness.

We present the detailed implementation in Section 4.1.

3.2 Readability

In our study, we introduced several metrics to quantify the readability and complexity of the decompiled programs automatically. To assess the complexity of the code, we selected several common algorithms commonly used in software engineering. This includes *Lines of code*, *Max nesting depth*, *Cyclomatic complexity*, and *Halstead complexity measures*. Generally, the higher the value of these metrics, the more complex and harder to read the code is. We briefly introduce these metrics below:

- **Lines of code.** Lines of code measures the total number of lines in the code, indicating the code's size and complexity. In our study, we counted only the physical lines in the text of the program's source code, excluding comments and blanks [34]. Generally, a higher number of lines indicates a more complex and potentially harder-to-read code.
- **Max nesting depth.** This metric calculates the maximum depth of nested structures, such as loops (e.g., `for`, `while`) and conditionals (e.g., `if`), within the code. A higher value implies deeper nesting, indicating increased complexity and reduced readability [29],.
- **Cyclomatic complexity.** Cyclomatic complexity measures a program's control-flow complexity [27]. It quantifies the number of linearly independent paths through the code, representing the number of decision points and possible execution paths. Higher cyclomatic complexity values suggest more intricate code structures, making the code more difficult to comprehend. For implementation, as we are comparing function-to-function instead of the whole program, we simply count the number of decision points in the function (such as an `if` statement or `for` statement).
- **Halstead complexity measures.** The Halstead complexity measures a program's data-flow complexity [44]. The original algorithm includes various metrics such as program vocabulary, program length, volume, difficulty, and effort. These measures assess the overall complexity of the code based on the number of distinct operators and operands used and their frequency of occurrence. In our study, we only calculated program effort for decompiled code.

In addition to the above algorithms, we carefully selected a subset of metrics from previous decompiler works [41,46] that are suitable for our function-to-function comparison. Specifically, these works aimed to reduce the reliance on `goto` statements in the decompiled code, recognizing their potential complexity in comprehension. Other properties, such as the number of cast and dead assignments, are also evaluated for the presented decompilers.

The full list of metrics used in our research is presented in Table 1. We prioritize extensibility and scalability in our metric selection, and to achieve this, we do not include user studies in the scope of our work. By incorporating these metrics, we aim to comprehensively evaluate the readability and complexity of the decompiled programs, providing valuable insights for future decompiler development and analysis.

3.3 Structural Similarity

The Abstract Syntax Tree (AST) is widely used for expressing and analyzing program structures. In this paper, we propose utilizing the AST to represent the structure of a program and then compare the AST of the original code with that of the decompiled code to determine their structural similarity. During the comparison of ASTs, we only consider the "type" of each node and not their specific values. For instance, in Figure 2, we solely compare the type of each

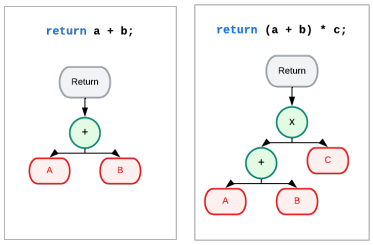


Fig. 2: An example of AST comparison

Metric	Range
Line of Code	$[0, \infty)$
Max Nesting Depth	$[0, \infty)$
Cyclomatic Complexity	$[1, \infty)$
Halstead Complexity Measure	$[0, \infty)$
# of goto, # of variables, # of cast, # of dead assignments	$[0, \infty)$
AST Node Quantity Compare	$[0, 1]$
AST Tree Edit Distance	$[0, 1]$

Table 1: List of readability and structural similarity metrics

node (represented by colors) and not the actual content of each node (e.g., "+", "*", "a", and "b"). The main reason for this design is that we focus only on the structural differences with this metric and leave the numerical evaluation to correctness section.

We employed two comparison algorithms to measure the similarity: *Node quantity compare* (NQC) and *Tree edit distance* (TED).

Node quantity compare. In NQC, we count the number of nodes with the same type between two ASTs. The formula to calculate NQC_{score} is shown in Equation 1. The score range falls between $[0, 1]$, where a higher score indicates a higher similarity between the two ASTs regarding shared nodes.

$$\sum \frac{common\ \#\ of\ each\ type\ of\ nodes}{max\{total\ nodes\ in\ one\ tree\}} \tag{1}$$

Take the two ASTs from Figure 2 for example. Among the trees, one *Return* node, one green node (operation node), and two red nodes (variable nodes) are common. The total nodes in Trees A and B are 4 and 6, respectively. Therefore, the score of NQC between these two trees is $NQC_{score} = \frac{1+1+2}{max\{4,6\}} = 0.67$. This score indicates a relatively high level of structural similarity between the two trees in terms of common nodes.

Tree edit distance. The *Tree edit distance* (TED) algorithm calculates the number of steps required to transform one tree into another. Traditionally, the TED algorithm counts the costs of three operations: `insert`, `delete`, and `replace`. In our current implementation, we adopt a simplified approach. We calculate the differences in depths for each node, which means we only consider the `insert` and `delete` operations. We do not compare the content of each node during this process. By focusing solely on the differences in depths and considering only `insert` and `delete` operations, our implementation provides a straightforward and efficient way to measure the similarity between two trees.

The recursive formula for computing TED_{score} is shown in Equation 2. Let $F_1 = T_1[i..j]$ be the post-order sub-forest of T_1 and let r_1 denote its rightmost root. Also, let R_1 be the rightmost tree of F_1 (the one rooted at r_1). The same notation holds for T_2:

$$
\begin{aligned}
&TED(\emptyset, \emptyset) = 0 \\
&TED(F_1, \emptyset) = TED(F_1 - r_1, \emptyset) + cost_{del} \\
&TED(\emptyset, F_2) = TED(\emptyset, F_2 - r_2) + cost_{ins} \\
&TED(F_1, F_2) = \\
&\qquad min \begin{cases} TED(F_1 - r_1, F_2) + cost_{del} \\ TED(F_1, F_2 - r_2) + cost_{ins} \\ TED(F_1 - r_1, F_2 - r_1) + TED(R_1 - r_1, R_2 - r_2) \end{cases} \\
&where\ cost_{del} = cost_{ins} = 1
\end{aligned} \tag{2}
$$

We also normalize and reverse the distance to TED_{score} into values between $[0, 1]$. $TED(T_1, T_2)$ represents the TED distance between trees T_1 and T_2, while $|T_1|$ and $|T_2|$ denote the total number of nodes in each respective tree. The larger the TED_{score} we obtain, the more structurally closer the two trees are.

$$TED_{score} = 1 - \frac{TED(T_1, T_2)}{|T_1| + |T_2|} \tag{3}$$

For the example shown in Figure 2, with Equation 2, The TED distance between the two trees is 0.199, resulting in a TED_{score} of 0.801 (1 - 0.199). This score indicates a relatively high structural similarity between the two trees.

3.4 Compilers & Decompilers

In our C-to-WASM compilation process, we opted to use Emscripten (*emcc*) [10], a comprehensive compiler toolchain for WASM built on LLVM [22]. As emcc utilizes LLVM as its foundation, we chose to employ *clang* to generate the native binary to ensure consistency in both compilation and re-compilation. Additionally, we aimed to set the compilation options to be identical to those used within the emcc implementation.

For the WASM decompilation process, we utilized three widely used WASM decompilers: *wasm2c* [13], *w2c2* [31], and *wasm-decompile* [4].

wasm2c is part of the WABT WebAssembly tool set [11]. It converts a WASM binary file to C source code along with the auxiliary header and supports various experimental WASM features, such as exceptions and threading, by incorporating specific command-line options. *wasm2c* can generate bug-free C programs that can be compiled and executed, and the behavior of the recompiled program is expected to adhere to the original program; that is, the correctness of the program shall hold.

wasm-decompile [4] is also a component of WABT toolset. The reason to include another tool from the WABT toolset is that *wasm-decompile* and *wasm2c* have very different design goals, and the difference is significant enough to include both decompilers in our study (see Sec. 5). *wasm-decompile* "is aimed at users that want to be able to 'read' large volumes of WASM code [4]." Decompiled code generated by *wasm-decompile* is not designed to "be a programming language," that is, the functionality of recompilation is currently not provided, and the execution correctness of the decompiled code is not guaranteed. In con-

clusion, *wasm-decompile* focuses on the decompiled code's readability and not correctness.

w2c2 [31] is a standalone tool that translates WASM modules to portable C. While it supports basic WebAssembly features, it also includes three experimental WASM features: bulk memory operations, sign-extension operators, and non-trapping float-to-int conversions.

In the case of native binary decompilation, we employed two open-source decompilers that were previously evaluated in the DecFuzzer paper: *Ghidra* [35] and *RetDec* [21]. These tools have been widely used and evaluated in various decompilation tasks, making them suitable candidates for our evaluation.

3.5 Benchmarks

We use the 1,000 CSmith synthesized C programs from DecFuzzer for correctness. In the original work of DecFuzzer, the authors further mutated them to test decompilers rigorously. We considered 1,000 programs reasonable for our attempt to examine WASM decompilers, and we did not mutate them further.

For evaluating structural similarity and readability, we employed two widely used C benchmark suites: PolyBenchC [38] and CHStone [17]. These benchmark suites contain real-world program implementations that are likely to be used in the WASM environment, such as scientific visualization, encryption, simulation, image recognition, etc.

We opted not to include the programs from DecFuzzer for structural similarity and readability evaluation for two main reasons. Firstly, we found that since DecFuzzer's synthesized programs may contain dead code, some portions of the code are automatically removed during compiling, even when setting the compiler optimization level to 0. This results in the decompilers not handling the exact same program as the original one, making it less reasonable to include these programs. For the same reason, the evaluation of correctness is limited to zero compiler optimization for the current implementation.

Secondly, the original synthesized code from DecFuzzer is not designed for human readers and contains no specific meaning or intended functionality. Consequently, including these programs in the evaluation would not be meaningful for assessing decompiled code's structural similarity and readability.

In contrast, PolyBenchC and CHStone are not used in the correctness evaluation due to their inclusion of library calls, which are challenging for both native and WASM decompilers to handle correctly during runtime. Therefore, we reserved these benchmark suites to assess the structural similarity and readability aspects of the decompiled programs.

4 Implementation

The total scripts for compiling benchmarks and generating metrics contain around 5,000 lines of Python and shell scripts.

4.1 Correctness

DecFuzzer utilized CSmith [47], a widely recognized C-code synthesizer for compiler testing, to generate C code for evaluating decompilers. In its original form, the generated code operated on global variables and computed a checksum to verify execution results. DecFuzzer consolidated the synthesized code into a single function, `func`, and operated on local copies of the original global variables. As `func` contains no library calls or global variable access, it can be fully compiled and decompiled by state-of-the-art decompilers. To generate re-compilable code, DecFuzzer deployed a simple rewriter to fix the syntax errors in decompiled code. Finally, the rewritten code was re-compiled and executed to test against the original binary.

To adapt DecFuzzer for WASM, we wrapped `func` into an importable module that can be called and receive output from the main function. This module-based approach was chosen because the WASM code handles memory differently, making it challenging to create a complete program that can be executed natively. Luckily, as a standalone WASM module, the code can be imported and called within a testing framework simulating a browser environment. This modified code was then compiled to WASM, decompiled back to C, re-compiled to object files, and eventually imported and called by our sandbox testing framework.

Besides the above efforts, we upgraded DecFuzzer's syntax rewriters to match the latest compiler and decompiler versions. We also switched from using GCC to clang, which aligned better with the WASM toolchain and resulted in fewer errors during preliminary testing. To make the DecFuzzer code compatible with WASM, we created a Python converter with approximately 300 lines of code. Additionally, we patched the legacy RLBox [50] code to match the latest WASM toolchain and integrated it into our evaluation framework for sandbox testing. Besides, *w2c2* is skipped for our correctness evaluation due to the current lack of support for sandboxing the *w2c2* decompiled code.

4.2 Readability & Structural Similarity

For readability and structural similarity, we built the analysis on top of *clang Python API* [9] and *cppcheck* [26]. As clang API relies on the clang's preprocessor, the under-analyzing target must be C syntactically correct. We leveraged the syntax rewriter of DecFuzzer to try to make the native binary's decompiled code re-compilable. Due to the limitation of the preprocessor, it cannot parse the decompiled code generated by *wasm-decompile* under optimization level 1 or 2, as the code would break the type system during the analysis progress.

5 Evaluation Results

As introduced in the previous chapters, we evaluated the state-of-the-art WASM decompilers, *w2c2*, *wasm2c*, and *wasm-decompile*, in three aspects: correctness, readability of the decompiled code, and structural similarity of the decompiled

code with the source code. Based on decompiler characteristics introduced in Section 3.4, *w2c2* is excluded from the correctness evaluation, and the structural similarity evaluation of *wasm-decompile* is limited.

To establish a baseline for comparison, we also included decompilers of native binaries, namely *Ghidra* and *RetDec*. All experiments were performed on an Intel-i5 machine with four cores, 8GB RAM, and running ArchLinux. To ensure the validity of the results, we used the latest versions of all software.

For the selected benchmarks, we observed that the compiling and decompiling processes could be completed within five seconds without significant memory consumption. Given the swift execution of these processes, we did not measure and compare the decompilers' runtime performance. Instead, our focus was solely on evaluating the quality of the generated code.

In each section of the evaluation, we address the following questions through the use of proposed metrics:

– Can proposed metrics help evaluate and characterize decompilers?
– How does each decompiler perform in terms of the specific metrics?

5.1 Correctness

Table 2 shows the result of our correctness evaluation. Factors are based on DecFuzzer, including execution results and the number of decompile or re-compile failures.

From the table, we can clearly see that *wasm2c* outperformed native binary decompilers by achieving a 100% correctness rate. We attribute this success to the fact that *wasm2c* adopts a conservative approach when translating WASM to decompiled code: Performing minimal optimizations. Consequently, when the execution environment was appropriately set up, the decompiled code faithfully reproduced the original functionality. This high correctness rate can also be attributed to the more uncomplicated instruction set in WASM compared to native binary code. Therefore, it's relatively simple to translate WASM bytecode to C code directly, whereas native binary decompilers often face challenges in handling low-level instructions. Propagating these instructions as functions in the decompiled code without providing corresponding runtime implementations often leads to re-compile failures.

Compared to *wasm2c*, the other three decompilers all suffer from compilation failure and execution discrepancies. First, as presented to be a decompiler to improve readability for WASM, *wasm-decompile* is only able to achieve 649 out of 1000 program correct. We manually reviewed the semantically incorrect code, and summarized the reason for failures into two main reason:

– **Unfaithful representation of the original WASM workflow** In order to improve readability, *wasm-decompile* translates WASM's stack-based instructions into SSA formats, which is theoretically more readable for human than long lines of C code in *wasm2c*. However, this transformation also introduces errors in the value passing and data flow, which leads to the incorrect execution of the decompiled code.

Bench	Success	Re-compile Failure	Exec. Discrepancy	Total
wasm2c	1,000	0	0	
wasm-decompile	649	6	345	
Ghidra	818	13	169	1,000
RetDec	775	0	225	

Table 2: Correctness results. **Success** implies the numbers achieving same execution results after successful recompilation.

– **False identification of structures and pointers** In the synthesized benchmark for testing correctness, we introduce no pointer or special data structure to reduce the complexity of evaluation. However, *wasm-decompile* falsely grouped some of the variables as data structures and accessed through pointer offsets on stack. This not only results in violation of C syntax that resulted in failure to compile, but potentially introduce rooms for more error to preserve the original program's semantic. Moreover, it conversely increases the difficulty for understanding the decompile code.

We showed a code snippet of the decompile output for *wasm-decompile* in Listing 1.6. The first part of the code is how *wasm-decompile* translates function epilogue. As all function parameters are stored on stack and then loaded during function calls, *wasm-decompile* may falsely group some parameters as one data structure, while the original program actually only passes integer values. The latter part of the listing shows how variable type casting is represented with *wasm-decompile*. The variable `n` is a 32-bit integer casting to 8-bit and stored into `q`. This kind of operation sequence is conducted through out the decompile code, making the code not only long and hard to read, but also introducing correctness errors throughout the long dependency chain.

```
export function func_1():int {
  var a:int = stack_pointer;
  var b:int = 16;
  var c:{ a:int, b:int /*...*/ }
        = a - b;
  stack_pointer = c;

  var f:int = -1051244671;
  c.b = f;
  var g:int = 1902055121;
  c.a = g;
  // ....
  var o:int = 24;
  var p:int = n << o;
  var q:int = p >> o;
  // ....
}
```

Listing 1.6: Code snippet of decompile code generated by wasm-decompile

```
export function KeySchedule(a:int_ptr,
      /*...*/):int {
    // ...
    c = b - (g = b / i) * i;
    if (eqz(c)) {
        f = Sbox;
        c = f + ((e = (c = ((a = word + (a
              << 2)) + 1440)[0]:int) / 16)
              << 6) + (c - (e << 4) << 2);
        e = f + ((d = (e = a[240]) / 16)
              << 6) + (e - (d << 4) << 2);
        h = (f + ((h = (d = a[120]) / 16)
              << 6))[d - (h << 4)]:int ^
            (Rcon0 + (g << 2) - 4)[0]:int;
        g = a[0];
      goto B_q;
    }
    // ....
}
```

Listing 1.7: Decompiled code by wasm-decompile with compiler optimization

Metrics	Opt Level	Original Code	Decompilers				
			w2c2	wasm2c	wasm-decompile	Ghidra	RetDec
		Total					
Lines of code	O0	2,619	74,473	72,362	23,744	10,376	4,818
	O1		24,828	28,332	7,975	11,290	7,487
	O2		27,440	30,461	7,771	15,575	9,249
# of goto statements	O0	0	767	576	768	98	21
	O1		738	296	774	175	63
	O2		584	257	455	405	131
# of type casting	O0	39	0	0	627	138	2,872
	O1		0	0	382	99	2,520
	O2		0	0	642	92	2,958
# of variables	O0	194	16,109	19,368	18,804	2,883	519
	O1		1,496	1,591	619	3,261	760
	O2		1,622	1,711	741	4,952	1,110
# Lines of dead code	O0	4	64	207	7	4	31
	O1		143	25	14	4	7
	O2		513	140	35	8	8
		Average Stdev	Average Stdev		Average Stdev	Average Stdev	Average Stdev
Maximum nesting depth	O0	0.952 1.163	1.221 1.534	0.655 0.477	1.455 1.409	1.634 1.798	1.338 1.464
	O1		1.228 1.558	0.600 0.492	1.041 1.241	1.855 2.010	2.000 1.911
	O2		1.766 2.447	1.462 1.514	1.710 2.065	2.097 2.428	1.745 1.813
Cyclomatic complexity	O0	14.329 18.067	3.634 3.853	4.924 4.667	6.283 6.139	6.772 8.465	4.683 4.412
	O1		5.152 7.295	6.766 8.030	6.710 8.475	7.517 9.430	7.883 8.920
	O2		5.883 8.192	7.359 8.530	6.228 8.180	9.255 12.438	9.048 10.252
Halstead complexity measure	O0	3.996 4.144	43.177 31.158	52.026 24.492	26.084 9.363	30.956 30.761	36.106 44.118
	O1		46.341 54.848	52.563 51.660	30.778 26.185	31.434 26.192	35.865 36.894
	O2		55.253 62.667	66.963 63.842	36.559 32.038	30.818 25.237	42.948 42.853

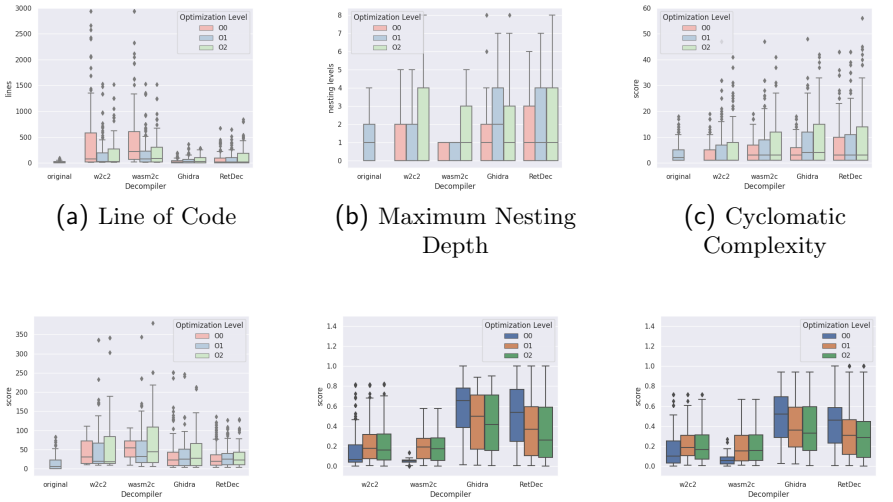
Table 3: Results of readability evaluation

For native-binary decompilers, both *Ghidra* and *RetDec* achieved slightly better results than *wasm-decompile*, but still have a significant number of failures. The main reason for the failures is the decompilers' capabilities of translating low-level instructions to high-level C code. For such instructions, the decompilers often directly translate them to function calls, mimicking the original instructions. The runtime environment does not support such cases and will either lead to re-compile failures or be falsely rewritten by the DecFuzzer, leading to incorrect execution results. For example, *Ghidra* places `CONCAT` as a pseudo function to translate assembly code instructions such as `mov AH, 2`. While serving a purpose in the original code, the existing DecFuzzer implementation falsely removed it from the decompiled code to make the decompiled program re-compilable.

Summary: With the presented approach for evaluating correctness, we highlight the remarkable 100% accuracy of *wasm2c*'s decompiled code. The limitation of *wasm-decompile* in correctness not only emphasizes the importance of evaluating decompilers from multiple perspectives but also underscores the challenges in recovering WASM's stack-based instructions into high-level C code. While native binary decompilers present some failures, it is essential to note that some may be attributed to the evaluation approach rather than inherent issues with the decompilers themselves. More refined evaluation methodologies and techniques can address and improve these specific cases separately.

5.2 Readability

We conducted a comprehensive evaluation by compiling each C code in PolyBenchC and CHStone at optimization levels 0, 1, and 2 to both WASM and



(a) Line of Code (b) Maximum Nesting Depth (c) Cyclomatic Complexity

(d) Halstead Complexity Measure (e) Node Quantity Compare (f) Tree Edit Distance

Fig. 3: Readability and structural similarity results consistent with Tables 3 and 4.

native binaries and subsequently decompiled outputs to compare their differences.

The complete evaluation results are in Table 3 and Figure 3. Regarding the readability metrics, higher values generally indicate more challenging to understand.

First, by comparing the results for *Lines of code*, *Maximum nesting depth*, *Cyclomatic complexity*, and *Halstead complexity measure*, it is evident that WASM decompilers generated significantly different code compared to others. The decompiled WASM code contains more lines of code and variables. Structurally, it exhibits a "flattened" code with a relatively low maximum nesting depth and Cyclomatic complexity. However, as the Halstead complexity measure indicates, the data flows are notably complicated. The low nesting depth also comes from the fact that WASM's decompiled code relies heavily on goto statements. This discrepancy is mainly due to the fundamental differences between WASM's stack-based instruction set and C's register-based language. The state-of-the-art native binary decompilers attempt to translate the code to resemble the semantics of C code closely, while the WASM decompilers lack such design as observed in the motivation decompiled code (Section 2), all three WASM decompilers primarily convert WASM's stack-based instructions into C syntax directly.

Although *wasm-decompile* is designed to generate readable decompiled code, it is interesting that the readability metrics do not necessarily support this state-

ment. With optimization level 0, decompiled code generated by *wasm-decompile* has a significantly larger number in most metrics evaluated because *wasm-decompile* attempts to recover the WASM semantics into readable format faithfully. However, as WASM has only a limited number of types, a large portion of the decompiled code is related to type conversion between variables in support of the original C programs' semantics. Faithfully translating these operations could help to understand small-scale analysis of variable dependency better, but may not necessarily improve the whole program's readability. This is highly reflected in the differences in the total LoC and type castings between *wasm-decompile* and *wasm2c*. *wasm2c* does not specify the casting of variables in code while preserving the correct semantics as shown in Section 5.1, while *wasm-decompile* explicitly shows such program flow and resulted in higher numbers in these two metrics.

In comparing native binary decompilers, *Ghidra* generally generated more compact code and was closer to the original. This is attributed to *Ghidra*'s continuous development and successful heuristics in decompiling. In contrast, *RetDec* produced more lines of code and variables. This is due to its failures in recovering many native binary instructions into high-level C code and simply migrating instructions to C functions. E.g., *RetDec* directly used `v8 = __asm_movsd(v3)` in the decompiled code as the translation of the instruction `movsd`.

For other metrics, some may not directly reflect the code's readability but offer insights into the characteristics of the generated decompiled code. For instance, *Ghidra* had excessive type casting. This is due to its strategy for pointers and data-structure recovery, but it may not essentially affect how we interpret the code.

Interestingly, higher compilation optimization levels have different effects on different decompilers. WASM decompilers tend to produce more compact code at higher optimization levels, mainly because highly optimized WASM files are smaller in size, resulting in shorter decompiled code when directly translated. However, this does not necessarily mean that the decompiled code more closely resembles the original program nor improve readability. For example, in Listing 1.7, we show a code snippet of decompile code generated by *wasm-decompile* with optimization level 2. It is observable that although the total lines of code could be less as operations are shrinking into single lines, but the code is syntactically incorrect and semantically incomprehensible for the complicated value assignments between operations.

On the contrary, higher optimization levels do not necessarily lead to more compact code for native binary decompilers. Native compilers use aggressive algorithms to optimize binary size and performance, while these optimized operations may not be easily translated into C semantics. Decompilers typically create highly readable code through heuristic approaches. When a binary is highly optimized, it may not fit within the existing heuristics, resulting in larger decompiled code with more direct assembly-to-C translations.

Summary: The evaluation results unveil substantial differences in the decompiled code generated by WASM decompilers. Notably, the decompiled code exhibits excessive lines of code, frequent usage of goto statements, and intricate data flows, all of which negatively impact code readability.
Moreover, these metrics provide valuable insights into other interesting aspects, such as the influence of compiler optimization levels on each decompiler's performance. The interplay between these factors can be considered to be a robust indicator for assessing the quality of decompiled code generated by the decompilers.

5.3 Structural Similarity

The evaluation of structural similarity yields similar observations to the readability metrics. The results of structural similarity can be found in Table 4. As we have mentioned before, due to parser limitation, *wasm-decompile*-generated code can only be analyzed under optimization level 0. The evaluation compared original and decompiled functions by normalizing the computed *Node Quantity Compare* (NQC) and *Tree Edit Distance* (TED) scores to a range of $[0, 1]$. Higher scores indicate greater structural similarity between the functions.

In general, the structural similarity results align with the findings from the readability evaluation. WASM decompiled code exhibits significant differences compared to native binary decompiled code, and the trend of structural similarity varies with different optimization levels. Higher optimization levels lead to increased structural similarity for WASM decompilers, while for native binary decompilers, the opposite trend is observed.

Notably, the structural similarity evaluation revealed insights that are not evident in the readability evaluation. Specifically, the improvements introduced by *w2c2* are more apparent. We observed several exceptional high scores for *w2c2* decompiled code, corresponding to *w2c2*'s outliers in Figure 3e and Figure 3f. Our manual inspection revealed that these scores are primarily associated with pure arithmetic functions like `mul64To128`, which do not contain pointers or data structures. These functions exhibit minimal structural differences between the compiled WASM and native binary. As a result, *w2c2* successfully decompiled them with high NQC and TED scores that are closer to what native binary decompilers obtained. In contrast, *wasm2c* still produced long and complicated code for these functions.

Table 4: Results of structural similarity evaluation

Metrics	**Opt Level**	**Decompilers**									
		w2c2		**wasm2c**		**wasm-decompile**		**Ghidra**		**RetDec**	
		Average	Stdev	Average	Stdev	Average	Stdev	Average	Stdev	Average	Stdev
Node quantity compare	O0	0.170	0.206	0.047	0.022	0.134	0.066	0.573	0.260	0.506	0.310
	O1	0.230	0.195	0.197	0.140	N/A	N/A	0.470	0.284	0.378	0.291
	O2	0.223	0.201	0.193	0.147	N/A	N/A	0.436	0.304	0.357	0.300
Tree edit distance	O0	0.166	0.169	0.062	0.046	0.149	0.103	0.495	0.236	0.439	0.286
	O1	0.219	0.158	0.184	0.151	N/A	N/A	0.403	0.236	0.343	0.257
	O2	0.210	0.166	0.191	0.152	N/A	N/A	0.377	0.256	0.325	0.267

Summary: Overall, the structural similarity evaluation provides additional insights into the performance of decompilers. Specifically, it highlights the effectiveness of *w2c2* in handling certain types of functions that exhibit minimal structural differences between compiled WASM and native binary versions.

6 Discussion and Future Work

In this study, we evaluated and compared state-of-the-art WASM decompilers. To achieve our goal, we collected and integrated multiple metrics from previous works and created a comprehensive framework for evaluating decompilers. This framework assesses correctness, readability, and structural aspects, and includes a case study. Through our analysis, we gained valuable insights into the performance and capabilities of existing decompilers. However, our investigation also revealed certain limitations and disadvantages that warrant attention in future research endeavors.

6.1 Correctness

We evaluated the correctness of WASM decompilers that generated compilable C code. Still, the process was limited by the absence of a substantial portion of the original features in C programming. It would benefit the community if a framework were developed to assess the correctness of decompilers for a broader range of C programs.

Further investigations could also explore the correctness of WASM decompilers for different languages, platforms, or applications, extending the scope of our findings. Assessing our findings' generalizability across different WASM compilers and decompilers is crucial to comprehensively understanding correctness challenges.

6.2 Readability and Structural Similarity

We encountered several challenges when delving further into the readability and structural similarity. Distinguishing whether an issue lies within the scope of the "WASM decompiler" or is a broader "WASM" or "decompiler" problem posed a difficulty. For example, the inherent design of WASM as a stack-based machine lacks concepts of arrays. Consequently, a straightforward comparison between the decompiled code and native binaries becomes problematic due to their structural differences. Devising fair evaluation methods to handle such disparities and identifying the appropriate properties for comparison is a compelling direction.

7 Related Works

General WebAssembly Study. WebAssembly has been used for Crypto mining [20,30], games [5], software libraries [33,19], computer vision [36,49], and

encryption [3]. Researchers have analyzed topics including the presence of WebAssembly in the wild [3], bugs in WebAssembly [39], and the performance of WebAssembly.

Reverse Engineering in WebAssembly. *wasmdec* [45] is an open-source WebAssembly decompiler. However, it stopped updating in 2018 and didn't support many new features of WebAssembly standards. *JEB* [42] is a commercial decompiler that provides the WebAssembly decompilation function. Brandefelt et al. [7] implemented a Datalog-based WebAssembly decompiler and found that all generated programs can be decompiled. More than 97% of decompiled programs are recompilable, while only 70% of the lowest complexity programs maintained correctness, and when the complexity increased, this percentage fell below 20%. Benali et al. [6] investigated the viability of applying machine learning techniques, i.e., Neural Machine Translation (NMT), for decompiling WebAssembly binaries to C source code.

General Reverse Engineering For native binary decompilers, we also considered *angr* [2] and *snowman* [48] besides the two open-source decompilers (*Ghidra* [35] and *RetDec* [21]) in the study. However, they can only generate C pseudocode or C programs containing syntax errors; thus, decompiled programs are not recompilable. Thus, these native decompilers are excluded from the study. To measure the complexity of decompiled programs, in addition to the widely-used Cyclomatic complexity metric [27], previous decompiler research by Khaled et al. [46] highlighted that complicated goto statements in decompiled C code could be a significant obstacle for developers to understand the control flow. To improve readability, they attempted to reduce the number of such statements. Subsequently, a later work by Eric et al. [41] introduced more code properties, including the number of casts and dead assignments.

8 Conclusion

In this paper, we performed an empirical analysis using a selection of diverse metrics to evaluate the generated code from WASM decompilers. Our evaluation framework covers multiple perspectives, including correctness, readability, and structural aspects. By employing these metrics, we have demonstrated their usability in assessing C-based decompilers and provided valuable insights into the properties and limitations of current decompiled code. We believe that our findings and the framework we presented will serve as a useful guide for future researchers in the field. Our work aims to foster the development of more sophisticated decompilers and contribute to the enhancement of both decompiling and WebAssembly toolchains.

References

1. Akbary, S.: Wasmer (Jan 2023), `https://wasmer.io/`
2. Angr: Angr (2023), `https://angr.io/`

3. Attrapadung, N., Hanaoka, G., Mitsunari, S., Sakai, Y., Shimizu, K., Teruya, T.: Efficient two-level homomorphic encryption in prime-order bilinear groups and a fast implementation in webassembly. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security. pp. 685–697 (2018)
4. Auten, J.: Github - wwwg/wasmdec: WebAssembly to C decompiler. (2023), https://wwwg.github.io/web-wasmdec/
5. Battagline, R.: Hands-On Game Development with WebAssembly: Learn WebAssembly C++ programming by building a retro space game. Packt Publishing Ltd (2019)
6. Benali, A.: An initial investigation of neural decompilation for webassembly (2022)
7. Brandefelt, L.: Decompilation of webassembly using datalog (2022)
8. Chen, G., Wang, Z., Zhang, R., Zhou, K., Huang, S., Ni, K., Qi, Z., Chen, K., Guan, H.: A refined decompiler to generate c code with high readability. In: 2010 17th Working Conference on Reverse Engineering. pp. 150–154. IEEE Computer Society, Beverly, MA (2010). https://doi.org/10.1109/WCRE.2010.24
9. Clang: Clang Indexing Library Bindings — libclang 15.0.6 documentation (2023), https://libclang.readthedocs.io/en/latest/#
10. contributors, E.: Emscripten documentation, https://emscripten.org/index.html
11. developers, W.: The WebAssembly Binary Toolkit (2023), https://github.com/WebAssembly/wabt
12. developers, W.: Wabt documentation (2023), https://webassembly.github.io/wabt/doc/wasm-decompile.1.html
13. developers, W.: Wasm2c Documentation (2023), https://webassembly.github.io/wabt/doc/wasm2c.1.html
14. Gohman, D.: (Feb 2023), https://wasmtime.dev/
15. Gurdeep Singh, R., Scholliers, C.: Warduino: A dynamic webassembly virtual machine for programming microcontrollers. In: Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes. p. 27–36. MPLR 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3357390.3361029, https://doi.org/10.1145/3357390.3361029
16. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 185–200. PLDI 2017, Association for Computing Machinery, New York, NY, USA (Jun 2017). https://doi.org/10.1145/3062341.3062363, https://dl.acm.org/doi/10.1145/3062341.3062363
17. Hara, Y., Tomiyama, H., Honda, S., Takada, H.: Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. Journal of Information Processing **17**, 242–254 (2009). https://doi.org/10.2197/ipsjjip.17.242, http://www.jstage.jst.go.jp/article/ipsjjip/17/0/17_0_242/_article
18. Hilbig, A., Lehmann, D., Pradel, M.: An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In: Proceedings of the Web Conference 2021. pp. 2696–2708. ACM, Ljubljana Slovenia (Apr 2021). https://doi.org/10.1145/3442381.3450138, https://dl.acm.org/doi/10.1145/3442381.3450138

19. Jeong, H., Jeong, J., Park, S., Kim, K.: Watt: A novel web-based toolkit to generate webassembly-based libraries and applications. In: 2018 IEEE International Conference on Consumer Electronics (ICCE). pp. 1–2. IEEE (2018)
20. Konoth, R.K., Vineti, E., Moonsamy, V., Lindorfer, M., Kruegel, C., Bos, H., Vigna, G.: 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. pp. 1714–1730 (2018)
21. Křoustek, J., Matula, P., Zemek, P.: Retdec: An open-source machine-code decompiler. In: July 2018 (2017)
22. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: International symposium on code generation and optimization, 2004. CGO 2004. pp. 75–86. IEEE (2004)
23. Lehmann, D., Kinder, J., Pradel, M.: Everything old is new again: Binary security of {WebAssembly}. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 217–234 (2020)
24. Liu, R., Garcia, L., Srivastava, M.: Aerogel: Lightweight access control framework for webassembly-based bare-metal iot devices. In: 2021 IEEE/ACM Symposium on Edge Computing (SEC). pp. 94–105. Institute of Electrical and Electronics Engineers, New York City, NY (2021). https://doi.org/10.1145/3453142.3491282
25. Liu, Z., Wang, S.: How far we have come: Testing decompilation correctness of c decompilers. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 475–487 (2020)
26. Marjamäki, D.: Cppcheck, `https://cppcheck.sourceforge.io/`
27. McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering **SE-2**(4), 308–320 (1976)
28. McCallum, T.: The future of ewasm (2019), `https://hackernoon.com/diving-into-ethereums-virtual-machine-the-future-of-ewasm-wrk32iy`
29. McConnell, S.: Code Complete. Microsoft Press, United States, 2 edn. (2004)
30. Musch, M., Wressnegger, C., Johns, M., Rieck, K.: Thieves in the browser: Web-based cryptojacking in the wild. In: Proceedings of the 14th International Conference on Availability, Reliability and Security. pp. 1–10 (2019)
31. Müller, B.: W2c2 (2023), `https://github.com/turbolent/w2c2`
32. Narayan, S., Disselkoen, C., Garfinkel, T., Froyd, N., Rahm, E., Lerner, S., Shacham, H., Stefan, D.: Retrofitting fine grain isolation in the firefox renderer. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 699–716. USENIX Association, Berkeley, CA (Aug 2020), `https://www.usenix.org/conference/usenixsecurity20/presentation/narayan`
33. Narayan, S., Garfinkel, T., Lerner, S., Shacham, H., Stefan, D.: Gobi: Webassembly as a practical path to library sandboxing. arXiv preprint arXiv:1912.02285 (2019)
34. Nguyen, V., Deeds-Rubin, S., Tan, T., Boehm, B.: A sloc counting standard. In: Cocomo ii forum. vol. 2007, pp. 1–16. Citeseer (2007)
35. NSA: Ghidra (2023), `https://ghidra-sre.org/`
36. OpenCV: Opencv: Build opencv.js, `https://docs.opencv.org/4.7.0/d4/da1/tutorial_js_setup.html`
37. Pop, V.A.B., Niemi, A., Manea, V., Rusanen, A., Ekberg, J.E.: Towards securely migrating webassembly enclaves. In: Proceedings of the 15th European Workshop on Systems Security. p. 43–49. EuroSec '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3517208.3523755, `https://doi.org/10.1145/3517208.3523755`
38. Pouchet, L.N.: PolyBench/C – Homepage of Louis-Noël Pouchet (2023), `https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`

39. Romano, A., Liu, X., Kwon, Y., Wang, W.: An empirical study of bugs in webassembly compilers. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 42–54. IEEE (2021)
40. Rossberg, A.: Introduction - webassembly 2.0 (draft 2023-03-01) (Mar 2023), https://webassembly.github.io/spec/core/intro/introduction.html
41. Schulte, E., Ruchti, J., Noonan, M., Ciarletta, D., Loginov, A.: Evolving exact decompilation. In: Workshop on Binary Analysis Research (BAR) (2018)
42. Software, P.: Webassembly analysis (2023), https://www.pnfsoftware.com/jeb/manual/webassembly/
43. Wagner, L.: Webassembly consensus and end of browser preview (Feb 2017)
44. Weyuker, E.J.: Evaluating software complexity measures. IEEE transactions on Software Engineering **14**(9), 1357–1365 (1988)
45. Wwwg: Wwwg/wasmdec: Webassembly to c decompiler (2023), https://github.com/wwwg/wasmdec
46. Yakdan, K., Eschweiler, S., Gerhards-Padilla, E., Smith, M.: No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In: NDSS. Citeseer (2015)
47. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. In: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. pp. 283–294 (2011)
48. Yegord: Yegord/snowman: Snowman decompiler (2023), https://github.com/yegord/snowman
49. Yuan, A., Dukhan, M.: Supercharging the TensorFlow.js WebAssembly backend with SIMD and multi-threading (9 2020), https://blog.tensorflow.org/2020/09/supercharging-tensorflowjs-webassembly.html
50. Zakai, A.: WasmBoxC: Simple, Easy, and Fast VM-less Sandboxing, https://kripken.github.io/blog/wasm/2020/07/27/wasmboxc.html