Reusing Legacy Code in Wasm: Key Challenges of Compilation and Code Semantics Preservation

Sara Baradaran, Liyan Huang, Mukund Raghothaman, and Weihang Wang
Thomas Lord Department of Computer Science, University of Southern California, Los Angeles, CA, USA
{sbaradar, liyanhua, raghotha, weihangw}@usc.edu

***Abstract*—WebAssembly (Wasm) has emerged as a powerful technology for executing performance-critical code and reusing legacy code in web browsers. With its increasing adoption, ensuring the reliability of WebAssembly code becomes paramount. In this paper, we study how well WebAssembly compilers fulfill code reusability. Specifically, we investigate (*a*) the challenges that arise when cross-compiling a high-level language codebase into WebAssembly, and (*b*) how faithfully WebAssembly compilers preserve code semantics in this new binary. Through a study of open-source codebases, we identify the key challenges in cross-compiling legacy C/C++ code into WebAssembly, highlighting the risks of silent miscompilation and compile-time errors. We categorize these challenges based on their root causes and propose corresponding solutions. We then introduce a differential testing framework, named WASMCHECKER, to check the semantic equivalence between native x86-64 and WebAssembly binaries. Using WASMCHECKER, we provide evidence that WebAssembly compilers do not necessarily preserve original code semantics. Our analysis shows that such miscompilation occurs due to non-uniform implementations of standard libraries, unsupported system calls / APIs, WebAssembly's unique features, and compiler bugs. In particular, we identified 11 new bugs in the Emscripten compiler toolchain. As proof of concept, we publicly release our framework and the collected dataset of open-source codebases.**

***Index Terms*—WebAssembly, differential testing, compilation challenges, code semantic equivalence**

I. INTRODUCTION

WebAssembly, also known as Wasm, is a statically typed binary instruction format that serves as a portable compilation target for high-level programming languages such as C, C++, Rust, and Go [1]. It has been designed to provide a compact, platform-independent bytecode that can be executed within a web browser at a speed close to that of native binaries. This feature encourages web developers to use WebAssembly for performance-intensive applications, such as gaming [2], virtual reality [3], [4], and audio / video processing [5], [6].

Since its inception in 2017, WebAssembly has gained significant adoption and support from vendors of major browsers, including Chrome, Firefox, Safari, and Edge [7]. The WebAssembly ecosystem is currently expanding beyond the browser into a multitude of domains, from edge computing [8], [9] to Internet of Things (IoT) devices [10]–[13] and smart contracts [14]–[16]. This expansion has led to the development of various compilers for translating high-level language programs into low-level WebAssembly code [17]–[22].

Although achieving higher execution speeds is a significant motivation for WebAssembly development, this is not the only objective. WebAssembly also aims to facilitate code reuse in web development by enabling the seamless integration of existing codebases / libraries into web applications instead of writing JavaScript code from scratch. This assumes that programmers can easily compile high-level language code into WebAssembly to use as part of web applications.

However, a review of issue reports in WebAssembly compiler repositories shows that developers frequently encounter difficulties when porting their code into Wasm [23], [24]. Besides, recent studies [25]–[27] indicate that the security behavior of the programs may not be preserved in WebAssembly due to the immaturity of WebAssembly compilers and implementation discrepancies among WebAssembly runtimes [28]. These studies specifically highlight that unexpected memory vulnerabilities [29]–[31] may become exploitable when programs are compiled into Wasm binaries. This mainly occurs because of WebAssembly's linear memory model and the lack of security protection mechanisms, such as stack canaries, in existing WebAssembly compilers.

In addition, Romano et al. [32] provide a systematic analysis of bugs in WebAssembly compilers, which highlights unique challenges that can introduce buggy behaviors specific to this target architecture. For instance, while C/C++ supports a fully synchronous execution model, browsers intrinsically execute JavaScript code asynchronously. Therefore, compiler developers must ensure that synchronous operations in C/C++ code are properly ported to the asynchronous browser environment. Otherwise, the asynchronous execution of such operations may result in bugs and unexpected program behaviors.

Considering these challenges, if WebAssembly is to be a viable method for using legacy code in web applications, the developer community must first ensure that the original code semantics remains consistent when compiling into WebAssembly. Otherwise, unexpected behaviors in WebAssembly code can introduce new, severe bugs into web applications.

In this paper, we aim to understand how effectively contemporary WebAssembly compilers enable code reusability. Unlike previous studies [25]–[27], which exclusively examine the security behavior of programs compiled to WebAssembly, we target the *functional behavior* of code. We specifically focus on the following two questions in our investigation:

RQ1. *What challenges arise when cross-compiling high-level language code into WebAssembly?*

RQ2. *How well do WebAssembly compilers preserve source code semantics in the resulting Wasm binaries?*

We first focus on **RQ1** to investigate whether cross-compiling C/C++ applications into WebAssembly is straightforward and to identify the challenges that arise during the compilation process. We collected 115 open-source C/C++ projects and manually built them into x86-64 and WebAssembly binaries. We observed that the compilation and build process in WebAssembly is not always as straightforward as for native binaries. In many cases, challenges such as unsupported APIs, incorrect compiler settings, and dependency issues cause the compilation and build process in WebAssembly to fail. We divide these failures into 6 categories based on their root causes and discuss corresponding solutions.

Next, to answer **RQ2**, we use a test-driven approach to evaluate code semantics in native (e.g., x86-64) and WebAssembly binaries. Large-scale projects typically contain rich test cases usable for regression testing, refactoring, and code maintenance. We leverage these tests and propose WASMCHECKER, a framework for differential testing between native binaries and their WebAssembly counterparts. Given a C/C++ program and a set of test cases which evaluate its functional correctness, WASMCHECKER first builds the source code alongside the test cases into two distinct binaries, x86-64 and WebAssembly. It then compares the functional behavior of the program in two binaries by executing tests and comparing their outcomes, thereby reporting potential dissimilarities between them.

To assess how faithfully WebAssembly compilers preserve code semantics, we deployed WASMCHECKER on 135 open-source C/C++ projects, comprising 34,480 test cases. This involves 115 projects studied in the first phase and a random sample of 20 new projects. WASMCHECKER reported that out of 5,862 executable tests, 226 exhibited differing outcomes across the two binaries. We analyzed the discrepancies and their underlying root causes to highlight disparities between WebAssembly compilers and traditional C/C++ ones. We also identified and reported 11 new confirmed bugs within the most widely-used WebAssembly compiler, Emscripten [17].

Contributions. This paper makes the following contributions:

- We identify the key challenges of compiling high-level language code into Wasm binaries and the ones that jeopardize the reliability of WebAssembly code (Sections III and IV-A).
- We present WASMCHECKER, a differential testing tool for checking semantic equivalence between x86-64 binaries of C/C++ code and their Wasm counterparts (Section IV-B).
- We evaluate WASMCHECKER on 135 open-source projects with 34,480 test cases from GitHub, showcasing the gaps in code semantics translation between WebAssembly compilers and C/C++ compilers (Section IV-C).
- We identified 11 new bugs within the Emscripten compiler, all of which were confirmed by the Emscripten developers.
- We make the collected dataset of open-source codebases and the source code of WASMCHECKER publicly available.

II. METHODOLOGY

We designed a two-phase study to understand how effectively WebAssembly compilers enable legacy code reuse.

The first phase of our study focuses on identifying potential challenges one may encounter when compiling codebases from high-level languages into WebAssembly using currently available compilers. We attempted to build the codebases of 115 projects collected in a dataset into both x86-64 and WebAssembly. These projects cover a wide range of real-world codebases with diverse code constructs. In doing so, we sought to discover common scenarios where, unlike compiling code into native binaries, cross-compiling it into WebAssembly fails and requires changes to the codebase. We classified observed failures into 6 categories and determined whether each of them needs fundamental changes to the code or whether they can be resolved by simply modifying build scripts.

The second phase of our study focuses on examining the effectiveness of WebAssembly compilers in preserving code semantics. We design WASMCHECKER as a differential testing framework by taking advantage of open-source tests to compare the functional behavior of a given codebase across two different binaries. Our framework first systematically builds the codebase, including the source code and test cases, into two different binaries. Then, it executes executable tests in these binaries and captures their outcomes to subsequently compare and report potential discrepancies. Using this approach, we provide a witness that cross-compiling C/C++ code into WebAssembly does not necessarily preserve code semantics.

A. Compiler Toolchain

In this study, we use Emscripten, the most widely-used compiler toolchain for WebAssembly [32], to build projects in Wasm binaries. We also use the GNU Compiler Collection (GCC) [33] and Clang [34] to build x86-64 binaries.

Figure 1 depicts the internal structure of Emscripten as an LLVM-based compiler. It first takes C/C++ code and uses Clang as the frontend to analyze the syntax and semantics of the source code. Clang translates the parsed code into LLVM Intermediate Representation (IR), which is then optimized for size and efficiency. Next, the LLVM WebAssembly backend translates the optimized IR into WebAssembly code and produces WebAssembly object files. Finally, the wasm-ld linker combines multiple generated object files with required libraries to create a single executable or shared library.

Passing optimization flags to Emscripten at compile time also adds a multi-pass optimization process to the linking phase to improve the performance and reduce the executable code size (dashed step in Figure 1). Specifically, Binaryen [18] is used for optimizations such as code inlining at the level of Wasm modules and the Emscripten's JS optimizer is applied to optimize the glue code. The compiler also optimizes the combined code (Wasm + JavaScript) by minifying imports and exports between the two and eliminating dead code [35].

Since the WebAssembly module produced by Emscripten is not standalone and cannot directly interact with Web APIs, Emscripten also generates JavaScript glue code to instantiate the WebAssembly module. This glue code is responsible for importing necessary functions, allocating memory, emulating the local file system, and translating errors into exceptions.

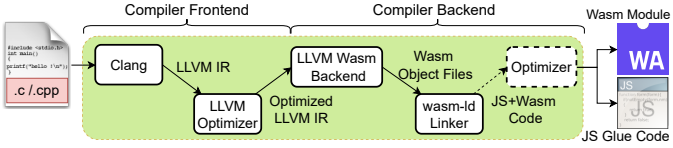


Fig. 1: Emscripten compiler architecture.

TABLE I: Overview of the collected exploratory dataset.

Functionality	Projects	KLOC	Test Cases	Exe.Tests
Utility (e.g., formatting, logging)	40	1,302	14,308	1,929
Data parsing/serialization/management	25	1,436	5,529	712
Algorithms and data structures	23	795	3,684	2,767
Mathematical/graphical/numerical computation	20	4,830	2,339	550
Networking and web	7	2,055	3,042	435
Total	115	10,420	28,902	6,393

B. Data Collection

To gain an insight into the challenges ahead when compiling code in WebAssembly, we collected a dataset of 115 open-source C/C++ projects. This involves a variety of libraries for document parsing (e.g., JSON, YAML, XML), mathematical computation, and data structure implementations. We specifically targeted CMake-based projects on GitHub with (*a*) a set of test cases written either manually or by using testing frameworks (e.g., Catch2 [36], GoogleTest [37]), as evidenced by the `test`/`tests` directory in the repository, (*b*) a CMake build script (i.e., `CMakeLists.txt`) in the root directory, since Emscripten provides `emcmake` utility that facilitates the build process in WebAssembly, and (*c*) over 50 stars, which generally suggest project maturity and higher code quality.

We used the GitHub Search API [38] to collect repositories that meet the above criteria, resulting in an initial dataset of 180 projects. We next conducted a two-phase refinement on the collected dataset. First, by reading the documentation of each project, we filtered out 46 codebases explicitly designed for platform-specific tasks, e.g., efficient memory management for Windows applications. This ensures that we focus on projects designed for general tasks that can be performed on any platform, including the web environment. Second, we built all the remaining projects in native x86-64 binaries using both GCC and Clang, and ruled out 19 projects for which the build process failed, mostly due to outdated build scripts.

Table I summarizes the final dataset. Note that executable tests refer to files that execute one or more test cases. When discussing executable tests in the context of native binaries, we refer specifically to ELF files that execute tests on a native platform, e.g., in our case, a server running Ubuntu 22.04 LTS. Conversely, in the context of WebAssembly, executable tests refer to JavaScript files that instantiate and manage WebAssembly modules produced by cross-compiling test cases.

C. Building the Codebases

Using provided build scripts, we attempted to build 115 projects in both x86-64 and WebAssembly. Consider yaml-cpp [39], a YAML parsing library, for example. We use the following command to build this project into two binaries (command options in [ ] are used only for WebAssembly):

```
$ mkdir build && cd build && [emcmake] cmake ..
    -DYAML_CPP_BUILD_TESTS=ON && [emmake] cmake --build .
```

This way, we could successfully build 81 projects (70%) in WebAssembly using the original build scripts. However, breaking errors were introduced into the build procedure when we tried to build the remaining 34 projects (30%) in Wasm. We analyzed these build errors to understand their root causes, which we categorize in the next section.

III. **RQ1**: Code Compilation Challenges

We now discuss the main challenges we encountered when building real-world projects in WebAssembly and their distribution as shown in paragraph headings. We encountered a total of 44 compilation and linking errors, which could be divided into 6 categories based on their underlying reasons. Note that one may need to address *multiple* challenges when compiling a C/C++ codebase into WebAssembly.

1) Undefined symbols (11 of 44): If the code refers to functions / variables not defined within the program and linked libraries, Emscripten will generate an *undefined symbol* error, which indicates the object file / library containing the symbol definition is not properly linked. A frequently observed case is the *undefined symbol:__stack_chk_guard* error, which results from the lack of support for Stack Smashing Protection (SSP), a security mechanism to detect buffer overflows [40].

Upon calling a function, the SSP inserts a value, called the stack canary, into the call stack before inserting the local variables of the callee procedure. This value is stored in the `__stack_chk_guard` variable. The SSP also adds instructions before the function's return, checking whether the canary value was preserved intact. If it fails to pass the sanitary check, a failure function, namely `__stack_chk_fail`, is called which terminates the program to prevent stack-based attacks.

Traditional C/C++ compilers like GCC and Clang have an option to enable SSP (e.g., `-fstack-protector`), which adds the necessary stack protector support when compiling C/C++ code. As depicted in Figure 2, this option declares the external variable `__stack_chk_guard` and the function `__stack_chk_fail` in the code alongside defining the `canary` variable. It also adds a code section to check the canary value and call the failure handler function. The compiler then produces object files for the resulting code, which are later combined by the linker and linked against the stack protector runtime library (e.g., `libssp` in GCC) in order to resolve references to external symbols.

Since Emscripten internally uses Clang to compile C/C++ code into LLVM IR, it supports most Clang compiler options, including those that enable SSP. However, Emscripten does not offer a runtime library to support stack protection. As a result, the wasm-ld linker introduces an error when a program tries to link an object file with stack protection calls inserted.

Besides the lack of SSP support, other limitations in the Emscripten compiler also result in undefined symbol errors. For

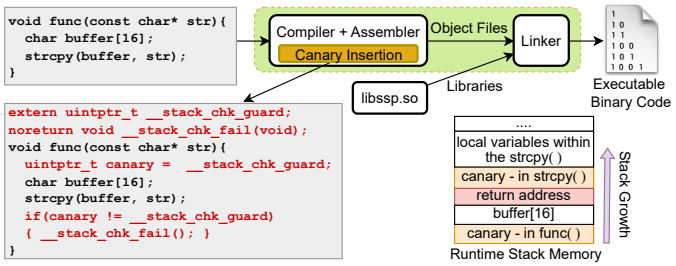


Fig. 2: Stack smashing protection in C/C++ compilers.

example, since WebAssembly is executed in a virtual machine, the functionality of POSIX APIs that depend on access to the operating system or hardware must be emulated without real access to system resources. Currently, Emscripten only partially emulates certain POSIX APIs (e.g., for file system access) and does not support the remaining ones. Hence, an undefined symbol error may arise when the source code contains calls to these API functions.

WebAssembly's official documentation [41] leaves it to the compiler to adapt the standard interfaces to the available imports of the host environment. Nevertheless, current versions of WebAssembly compilers do not thoroughly perform this adaptation. In the case of Emscripten, unsupported API calls must be removed from code to make it portable to Wasm.

2) Missing third-party libraries (6 of 44): Reusing third-party libraries as dependencies is a common practice to save time and manpower in software development [42]. Large-scale codebases often use third-party libraries such as Boost [43], which extends the C++ standard library with functions and structures for linear algebra, random number generation, etc. When building such codebases in WebAssembly, CMake is unable to locate the header files for these third-party libraries, thereby raising errors. To build these codebases, one needs to first build the required libraries in WebAssembly and link them with the main program. Only a limited number of useful libraries have currently been ported to WebAssembly, which reside in emscripten-ports repositories [44]. Each ported library can be used via a specific compiler flag. For example, to compile a codebase that uses the Boost library, the compiler option `USE_BOOST_HEADERS` should be set. This instructs Emscripten to fetch the Boost library from the remote server, set it up, build it locally, and link it against the project.

3) Target-dependent warnings (7 of 44): The compiler option `-Werror` is used to treat warnings as errors. When this flag is enabled, warnings which are generated during the compilation process cause the compiler to halt and prevent it from generating executable code. In our dataset, some projects enforce a stricter code standard by enabling `-Werror` in their build scripts. This can result in build failures in WebAssembly, even though the codebase can still be compiled into a native binary. Due to different compilation targets (i.e., x86-64 and Wasm), a compiler may generate warnings while the other smoothly compiles the same code without introducing warnings. We call such warnings *target-dependent*.

For example, enabling the `-mtune` flag allows traditional compilers to optimize code performance for a given processor. In particular, setting `-mtune=native` optimizes the binary code for the processor on which the compilation takes place without restricting it to run only on that processor. Since Emscripten cross-compiles code into platform-independent WebAssembly, this setting is disregarded by the compiler, followed by producing an *unused compilation argument* warning. This warning alongside the enabled `-Werror` results in build failures. In such cases, disabling `-Werror` or warning-triggering options (e.g., `-mtune`) allows the compilation to proceed without affecting the functionality of the resulting code.

4) Architecture- and platform-specific code (5 of 44): Codebases that use architecture- or platform-specific instructions fail to be built in WebAssembly using Emscripten. For example, C/C++ code with architecture-specific inline assembly (e.g., an `asm()` containing x86-64 code) is not portable. We found that some codebases in the dataset were designed and optimized to run on 64-bit platforms. Also, some codebases are platform-dependent and use Linux-specific headers. Such codebases need fundamental changes to their source code before they can be ported to WebAssembly.

5) Incompatible compiler options (11 of 44): Traditional C/C++ compilers provide flags specific to code compilation for native platforms and hardware-specific optimizations. These flags are not available in Emscripten as it targets WebAssembly, which is platform-independent and runs in web environments. For instance, the flag `-march` instructs the compiler to generate optimized code for a user-specified processor by leveraging processor-specific instructions. The optimization flag `-Ofast` is another compiler option which is unavailable on Emscripten. It enables optimizations not permitted by the standard C and C++ specifications, aiming to improve performance potentially at the cost of standards compliance.

Conversely, Emscripten introduces new compiler options for memory management, multithreading, and exception handling, which differ from those available in traditional compilers. Although these default-disabled options often lead to runtime failures rather than build failures, we observed cases where failing to activate Emscripten-specific options at compile time could also trigger build errors. We discuss these options in detail in the subsequent sections.

6) WebAssembly compiler bugs (4 of 44): We also observed that the build process for a few codebases failed due to compiler bugs. We identified 5 new bugs in Emscripten versions 3.1.54–63, which triggered build errors across 4 projects in the dataset. Specifically, 4 bugs refer to implementation defects in the LLVM `libc++` headers utilized by Emscripten as the C++ standard library, and 1 bug results from the incomplete implementation of functions within the `xlocale.h` header. We also uncovered 6 other bugs which did not break the compilation process. These bugs instead affect the semantics of the Wasm binaries and compromise their runtime reliability. We will elaborate on an example of such bugs in Section IV-C.

Takeaways: Porting legacy C/C++ code to WebAssembly often involves challenges whose resolution either requires appropriate adjustments to the compiler settings (e.g., deletion of troublesome compiler options) or necessitates significant changes to the source code (e.g., adaptation of unsupported APIs and architecture-specific instructions).

IV. RQ2: WebAssembly Code Reliability

To investigate whether the compiled WebAssembly code is semantically identical to the native x86-64 binary, we first need a systematic approach to compare the semantics of two different binaries produced from the same codebase. We note that two executables are semantically equivalent if they generate the same outputs for every possible input [45]. However, because an exhaustive analysis is not feasible in practice, we have to confine our scope and rely on the set of test cases provided by developers to dynamically assess the functional behavior of the codebase.

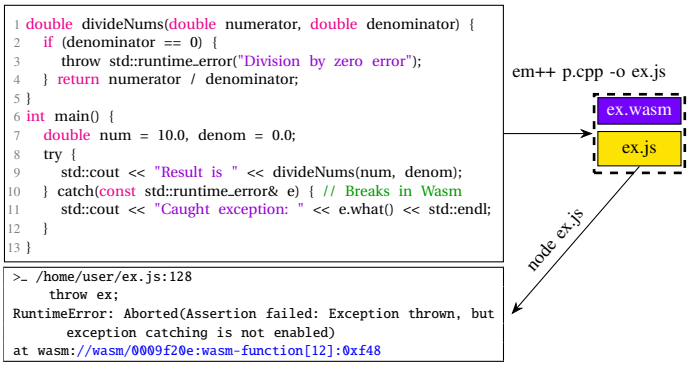
Given a source program P and its test set T, we compile both P and T into two different binaries. A test case $t \in T$ might *pass* or *fail* when executed in each binary. Let o_t and o'_t be boolean variables taking values corresponding to the outcome of test case t in native binary and WebAssembly, respectively (i.e., equal to 1 if t passes and 0 otherwise). With these definitions, if the native binary and WebAssembly versions of both P and T are semantically equivalent, then for all $t \in T$, it must be the case that $o_t = o'_t$.

As mentioned in Section II-C, Emscripten could successfully build the codebases of 81 projects in WebAssembly. Specifically, for these codebases, we enabled the testing option to instruct the build process to compile test cases alongside the source code (e.g., `YAML_CPP_BUILD_TESTS` in yaml-cpp project). When we run executable tests of 81 projects in x86-64 binary, 4,687 out of 4,691 tests pass and yield the expected outcomes. However, the results surprisingly vary when we run the corresponding JavaScript tests produced by Emscripten[1]. Only 4,316 tests pass in WebAssembly, conveying 371 test inconsistencies across the two binaries. A deeper analysis shows that a significant portion of Wasm test failures can be resolved by appropriately modifying compiler settings.

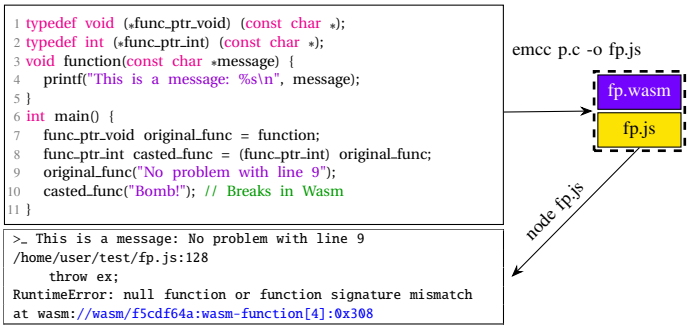
A. Fixable WebAssembly Test Failures

We now discuss the 3 types of WebAssembly test failures that can be addressed by changing compiler settings.

1) Default-disabled compiler options: First, the default settings of Emscripten are intended to generate high-performance WebAssembly code. It omits support of constructs that introduce additional overhead. For example, exception handling in WebAssembly can lead to performance overhead, as the mechanism required to handle exceptions, such as stack unwinding and state management, is complicated and slows down the program execution. Exception handling is therefore disabled in Emscripten by default. In this case, if a program throws an exception, it will not be caught, leading to a program crash (see Figure 3a). To avoid this issue, the compiler flag `NO_DISABLE_EXCEPTION_CATCHING` should be set to enable runtime exception handling.



(a) Compiling code by the default settings of Emscripten disables exception handling at runtime. Throwing the exception in Line 3 will abort the program without catch statement in Line 10 being executed.



(b) Type casting of function pointers before function invocations does not work in WebAssembly. Unlike the native binary, the invocation on Line 10 triggers a runtime error when executed in WebAssembly.

Fig. 3: Sample code snippets to show how Emscripten with its default settings produces high-performance WebAssembly code by sacrificing the runtime reliability of the resulting code.

As another example, function pointers / references in Wasm code must be invoked with the same type as declared. This is enforced by the runtime check of type signatures for indirect calls. While calling a function pointer after casting it to a different type works in native binary, it leads to failures in WebAssembly. Instead, one can change such invocations in the code by writing an adapter procedure that calls the original function without needing to cast, which requires familiarity with the codebase. Emscripten also provides the compiler flag `EMULATE_FUNCTION_POINTER_CASTS` as a wrapper to each function in the WebAssembly table to dynamically fix the parameters and return type at runtime. Since runtime correction incurs overhead, the default settings of Emscripten disable this flag, which leads to runtime errors in such cases (see Figure 3b).

2) File access paradigms in C/C++ vs. JavaScript: The second issue that causes many tests to fail in WebAssembly comes from different file access paradigms in native C/C++ and JavaScript code. Native C/C++ code usually calls synchronous (blocking) file APIs in `libc` / `libcxx`, while JavaScript only allows asynchronous file access (with callbacks). Moreover,

[1]Throughout this study, we utilized Node.js as the runtime to execute WebAssembly modules and their corresponding JavaScript glue code.

JavaScript code does not have direct access to the host's file system when running in the sandbox environment provided by a web browser.

Emscripten provides a Virtual File System (VFS) that emulates the local file system, allowing cross-compiled C/C++ code to access files through normal `libc stdio` and `libcxx fstream` APIs [46]. As a limitation, files that need to be accessed by C/C++ code should be preloaded / embedded into the virtual file system when compiling into WebAssembly. Emscripten does not offer an automated approach for this task. The user is thus required to manually preload files into the correct paths within the VFS so that Wasm code can access them at runtime.

3) Memory restrictions: The third root cause of test failure in WebAssembly involves memory restrictions. WebAssembly programs divide their address space into two segments which separately maintain the execution stack and program data. One is managed by the WebAssembly module, and the other is under the control of the VM where the Wasm code is executed. The latter is commonly referred to as the control stack, which is used to manage the execution flow of the program. It primarily holds the function call frames, including the return address and the base frame pointers.

For C/C++ programs with deeply recursive function calls, the WebAssembly modules produced by Emscripten may trigger an *exceeded maximum call stack size* error when executed in the Node.js runtime environment. This error indicates that the control stack is fully consumed and no longer able to store call frames. To ensure reliable compilation of these programs, the Wasm code can be optimized by using the `-Oz` flag which aggressively minimizes code size and the depth of nested calls by performing transformations such as selective inlining.

The default settings of Emscripten also impose restrictions on the size of the heap and stack memory which are under the control of WebAssembly modules. The default stack size for a WebAssembly module is 64KB, which is significantly smaller than that of native binaries (e.g., 8MB on Linux and 1MB on Windows). In the case of programs that heavily store data in the stack, the limited stack space leads to *out-of-bounds memory access* errors at runtime. Since there is no way to enlarge the stack size at runtime, one must increase it at compile time using the compiler option `STACK_SIZE` to make it large enough for the program requirements. Otherwise, it silently breaks down the execution while running Wasm code. Furthermore, Emscripten restricts the size of heap memory allowed to be allocated by the WebAssembly module. If it attempts to allocate further than the default size, then a runtime error will occur. Passing the compiler flag `ALLOW_MEMORY_GROWTH` to Emscripten at compile time enables the memory array to grow at runtime. These memory limitations also align with the Emscripten's goal of generating high-performance WebAssembly code.

Takeaways: Compilers are usually assumed to be trusted software. Nevertheless, the default settings of Emscripten as a widely-used WebAssembly compiler may not reliably preserve code semantics due to (*a*) default-disabled compiler options, (*b*) file access paradigms in WebAssembly, and (*c*) memory limitations imposed when generating Wasm code.

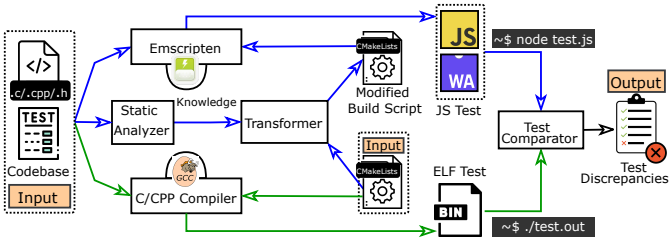


Fig. 4: Architecture of WASMCHECKER.

B. WASMCHECKER: A Differential Testing Framework

Since manually discovering and applying the necessary compiler settings is time-consuming and error-prone, we instead design and implement an automated pipeline for end-to-end building and testing C/C++ codebases in WebAssembly. This involves addressing the challenges identified in Section III and the issues which lead to unreliable code semantics translation and test failures, as discussed in Section IV-A. With this pipeline, we build an integrated framework named WASMCHECKER for differential testing between C/C++ programs compiled into native x86-64 binaries and their WebAssembly counterparts.

1) Overview: The high-level workflow of WASMCHECKER is illustrated in Figure 4. Given a codebase, including source code and test cases, WASMCHECKER builds the codebase in native binary using a C/C++ compiler and the CMake toolchain. It then executes executable tests to capture their outcomes (see green steps in Figure 4).

On the other hand, to build the codebase in WebAssembly, WASMCHECKER first performs a lightweight static analysis to extract source-level information. The code snippets in Figure 3, for example, showed unexpected behavior at runtime since the two intended flags, `NO_DISABLE_EXCEPTION_CATCHING` and `EMULATE_FUNCTION_POINTER_CASTS`, were not enabled at compile time. The source-level static analysis identifies that these code snippets encompass exception handling and type casting. This enables programmers who are not familiar with the codebase to automatically turn on the required flags when compiling into WebAssembly. By leveraging static analysis, WASMCHECKER extracts language constructs from the codebase and passes this knowledge to a transformer that enables required flags accordingly. After modifying compiler settings, WASMCHECKER builds the codebase in WebAssembly using Emscripten and subsequently executes JavaScript tests that instantiate Wasm modules using Node.js to capture test outcomes (see blue steps in Figure 4).

As the final step, WASMCHECKER compares test outputs in two different binaries and reports discrepancies, if any. The next sections provide detailed descriptions of the static analysis phase and the transformer, which clarifies the entire pipeline.

2) Static analyzer: The static analysis component uses CodeQL [47], a code analysis engine that converts the codebase into a database format, and makes it easier to query complex

information. It also provides its own declarative query language, similar to SQL, allowing users to write arbitrary queries to specify what patterns they look for. The query Q1 in the following is an example that extracts cast expressions of function pointers and their locations from a codebase:

```
Q1 { from Cast expr
     where expr.getUnderlyingType() instanceof FunctionPointerType
     select expr, expr.getLocation().getFile()
```

Correspondingly, we use a set of predefined CodeQL queries to extract codebase constructs. These queries, for example, determine whether a codebase uses multithreading (to set `-pthread`) or relies on exception handling (to set `NO_DISABLE_EXCEPTION_CATCHING`), function pointer / reference type casting (to set `EMULATE_FUNCTION_POINTER_CASTS`), and specific data types, e.g., long double to set `PRINTF_LONG_DOUBLE`.

To preload the required files into the VFS, the static analyzer extracts constant strings defined in source and test files, which represent valid absolute or relative paths in the local file system on which the compilation takes place. For example, given the assignment `const char* t = "../data/test.xml"` in source code, static analyzer identifies `../data/test.xml` as the path from which `test.xml` file must be accessible at runtime. With this path as `dst`, it localizes the absolute path to `test.xml` in the local file system as `src`. This mapping is finally passed to the transformer to preload the `test.xml` file from `src` path in the local file system to the `dst` path in the VFS by using `--preload-file src@dst` as a compile-time option.

3) Transformer: Based on the source-level information extracted by the static analyzer, WASMCHECKER transforms the CMake build script. For example, if CodeQL outputs a non-empty table for the predefined query Q1, then the transformer enables the `EMULATE_FUNCTION_POINTER_CASTS` option to ensure that problems around translating function pointers are properly handled by the compiler.

Besides CodeQL-guided modifications, the transformer also applies general changes to build scripts. It specifically disables `-Werror` option to avoid build failures which result from target-dependent warnings. Moreover, we omit incompatible compiler options, such as `-march` and `-Ofast`, and disable stack protection flags (e.g., `-fstack-protector`) to avoid undefined symbol errors. Finally, to avoid runtime memory errors, the transformer sets the stack size to 1MB alongside enabling the `ALLOW_MEMORY_GROWTH` option and the `-Oz` optimization flag.

4) Evaluation: To evaluate the effectiveness of our framework in finding and fixing build errors, faithfully compiling the codebases into WebAssembly, and identifying semantic discrepancies between native and Wasm binaries, we tested WASMCHECKER on our collected exploratory dataset, and on a random sample of additional C/C++ projects distinct from the initial dataset to minimize evaluation bias.

As discussed in Section IV, manually compiling 115 codebases in our dataset with the default settings of Emscripten resulted in the successful build of 81 projects with 4,691 executable tests overall. We also observed 371 test discrepancies across 58 codebases when running executable tests in two distinct binaries, x86-64 and WebAssembly.

TABLE II: Testing results on the two collected datasets.

Dataset	Compilation Approach	Compiled Projects	Total Exe.Tests	Projects *w/* Test Diffs	Total Test Diffs
Exploratory (size: 115)	Manual	81	4,691	58	371
	WASMCHECKER	99	5,638	23	223
		(81 + 18)	(4,691 + 947)	(19 + 4)	(36 + 187)
Validation (size: 20)	Manual	18	221	14	34
	WASMCHECKER	20 (18 + 2)	224 (221 + 3)	2 (2 + 0)	3 (3 + 0)

TABLE III: Overview of the collected validation dataset.

Functionality	Projects	KLOC	Test Cases	Exe.Tests
Utility (e.g., formatting, logging)	7	40	2,560	58
Data parsing/serialization/management	5	212	501	141
Mathematical/graphical/numerical computation	4	46	186	5
Algorithms and data structures	4	72	2,331	20
Total	20	370	5,578	224

Once again, we utilized WASMCHECKER across our entire dataset to build and test 115 codebases in WebAssembly. Table II presents the results of this experiment.

We observed that WASMCHECKER resolved 25 (57%) build errors and compiled 99 (86%) codebases, comprising 5,638 executable tests, into WebAssembly without errors. This involves 81 projects (having 4,691 tests) for which the manual compilation was also successful and 18 previously uncompilable ones (having 947 tests). WASMCHECKER also reported that among these 99 codebases, 23 have at least one test with inconsistent outputs in two binaries. Altogether, it reported 223 test discrepancies across these 23 codebases. Specifically, 36 discrepancies appear in 19 of 81 projects, and 187 arise from 4 of those 18 projects that WASMCHECKER built beyond the manual compilation.

Accordingly, WASMCHECKER could compile 18 (22%) more codebases, showcasing its effectiveness in fixing build issues. A comparison over 81 projects shows that these codebases generate approximately 10× fewer test inconsistencies when compiled using WASMCHECKER (36 vs. 371). This indicates the higher reliability of Wasm binaries which are produced by WASMCHECKER and its effectiveness for automatic adjustment of compiler settings, as explained in Section IV-A.

We also evaluated WASMCHECKER on a separate set of unseen codebases to confirm that our results are generalizable beyond the exploratory dataset. We used the same API-based querying employed in Section II-B to randomly collect 35 new C/C++ projects hosted on GitHub, apart from the exploratory dataset which we used to learn from. We also applied a similar two-phase refinement on the new validation dataset, resulting in a final dataset of 20 projects with 5,578 test cases, as summarized in Table III. Table II also shows the performance of WASMCHECKER on this validation dataset.

Observe that WASMCHECKER could build all 20 codebases in WebAssembly, while manual compilation succeeded for 18 projects. Furthermore, running executable tests, which were obtained by manual cross-compilation of 18 codebases, resulted

```
1  std::unordered_map<vertex_id_t, int> welsh_powell_coloring(const GRAPH& graph) {
2    using deg_vertex_pair = std::pair<int, vertex_id_t>;
3    // Sort vertices by degree in descending order
4    std::vector<deg_vertex_pair> deg_vertex_pairs;
5    for (const auto& [vertex_id, _] : graph.get_vertices()) {
6      int degree = properties::vertex_degree(graph, vertex_id);
7      deg_vertex_pairs.emplace_back(degree, vertex_id);
8    }
9    std::sort(deg_vertex_pairs.rbegin(), deg_vertex_pairs.rend());
10   std::unordered_map<vertex_id_t, int> color_map;
11   for (const auto [_, curr_vertex] : deg_vertex_pairs) {
12     int color = 0; // Start with color 0 & check colors of adjacent vertices
13     for (const auto& neighbor : graph.get_neighbors(curr_vertex)) {
14       // Increment the color if a neighbor is already colored with this color
15       if (color_color_map.contains(neighbor) && color_map[neighbor] == color)
16         color = color + 1;
17     } color_map[curr_vertex] = color; // Assign the color to the current vertex
18   } return color_map;
19 }
20 TYPED_TEST(WelshPowellTest, CompleteGraph) {
21   /* ... Code section to create a complete graph with 4 vertices ... */
22   auto coloring = welsh_powell_coloring(graph);
23   std::unordered_map<vertex_id_t, int> expected = {{3,0},{2,1},{1,2}{0,3}};
24   ASSERT_EQ(coloring, expected); // Check if the coloring is as expected
25 }
```

Test execution in WebAssembly	Test execution in x86-64
10: color_map←{} 11–12: curr_vertex←3, color←0 13–16: !color_map.contains(0) 13–16: !color_map.contains(1) 13–16: !color_map.contains(2) 17: color_map[3]←0, color_map: {(3,0)} 11–12: curr_vertex←2, color←0 13–16: !color_map.contains(0) 13–16: !color_map.contains(1) 13–16: color_map.contains(3)∧ color_map[3]=0, color←1 17: color_map[2]←1, color_map: {(2,1) ,(3, 0)} 11–12: curr_vertex←1, color←0 13–16: !color_map.contains(0) 13–16: color_map.contains(2)∧ color_map[2]!=0 13–16: color_map.contains(3)∧ color_map[3]=0, color←1 17: color_map[1]←1, color_map: {(1,1) ,(2, 1) ,(3, 0)} 11–12: curr_vertex←0, color←0 13–16: color_map.contains(1)∧ color_map[1]!=0 13–16: color_map.contains(2)∧ color_map[2]!=0 13–16: color_map.contains(3)∧ color_map[3]=0, color←1 17: color_map[0]←1; 17: color_map: {(0,1) ,(1, 1) ,(2, 1) ,(3, 0)}	color_map←{} curr_vertex←3, color←0 !color_map.contains(2) !color_map.contains(1) !color_map.contains(0) color_map[3]←0, color_map: {(3,0)} curr_vertex←2, color←0 color_map.contains(3)∧ color_map[3]=0, color←1 !color_map.contains(1) !color_map.contains(0) color_map[2]←1, color_map: {(2,1) ,(3, 0)} curr_vertex←1, color←0 color_map.contains(3)∧ color_map[3]=0, color←1 color_map.contains(2)∧ color_map[2]=1, color←2 !color_map.contains(0) color_map[1]←2, color_map: {(1,2) ,(2, 1) ,(3, 0)} curr_vertex←0, color←0 color_map.contains(3)∧ color_map[3]=0, color←1 color_map.contains(2)∧ color_map[2]=1, color←2 color_map.contains(1)∧ color_map[1]=2, color←3 color_map[0]←3; color_map: {(0,3) ,(1, 2) ,(2, 1) ,(3, 0)}

Fig. 5: Implementation of a graph coloring algorithm adapted from the Graaf library [48] as an example to represent how non-uniform implementations of data structures across different standard libraries affect the program's flow and the test output.

in 34 test discrepancies across 14 projects. This however dropped to only 3 test inconsistencies across 2 projects when we ran tests compiled using WASMCHECKER.

We carefully analyzed 226 test discrepancies reported by WASMCHECKER in both experiments and identified 220 as true alarms which denote semantic divergence among binaries. This shows the effectiveness of WASMCHECKER in precisely identifying cases where Emscripten inevitably fails to preserve original code semantics. We devote Section IV-C to elaborating on the root causes of these cases.

5) Limitations: One important limitation of our framework, which leads to 6 false alarms, is its heuristic-based method for automatic file embedding. Currently, WASMCHECKER does not use advanced types of analysis (e.g., string analysis [49]) to discover which files in the local file system need to be embedded in the virtual file system and where. Note that the correct path to a file in the VFS may not be simply defined within the codebase as a constant / string literal and may instead be dynamically calculated (e.g., by concatenating the path to a base directory with the expected filename).

Also note that WASMCHECKER does not perform source code adaptation and only addresses build failures through the adjustment of compiler settings and build commands. Among the challenges identified in Section III, WASMCHECKER can fully address build failures arising from incompatible compiler options and target-dependent warnings. It also partially addresses missing third-party libraries and undefined symbol errors. The former is confined to the libraries available on [44], and the latter is limited to the errors arising from (*a*) enabling SSP or (*b*) the missing definition of symbols that are supported in the case of enabling specific compiler options.

On the other hand, WASMCHECKER cannot address failures occurring due to architecture- or platform-specific instructions within the source code and those stemming from unsupported API calls. Indeed, addressing these issues requires significant changes in the source code, which falls outside the scope of this paper but would be a worthwhile future direction.

C. Semantic Divergence Among Binaries

We now discuss the root causes of true test discrepancies identified by WASMCHECKER which were confirmed through manual analysis.

1) Different standard libraries: Emscripten relies on `musl` and LLVM `libc++` as the C and C++ standard libraries respectively when compiling code into WebAssembly. On the other hand, native C/C++ compilers typically utilize the operating system's native standard libraries, in our case, `glibc` and `libstdc++`. Non-identical implementations of these standard libraries can result in semantic discrepancies between Wasm and native binaries. For example, `musl` has subtle differences in memory management, thread handling, and locale support compared to `glibc`. Similarly, `libc++` handles certain data structures and exceptions differently from `libstdc++`. The code compiled into WebAssembly may therefore show different functional behaviors compared to its native x86-64 counterpart, thereby affecting the program's logic.

Consider the code in Figure 5 which implements a graph coloring algorithm. We expect this code to assign a different color to each vertex when the given test is executed over a complete graph. The `welsh_powell_coloring` function behaves correctly in native x86-64 binary since the `get_neighbors` method provides adjacent vertices in descending order of their `vertex_ids`. In WebAssembly, however, this method returns a differently ordered set, which changes the loop traversal and the code execution flow.

Consider the test case in Figure 6 as another example. This test is designed to ensure that the logging process behaves as expected in the case of an error in file access operations. It attempts to open a file which does not exist in the file system. If the file unexpectedly opens, the code triggers a failure by calling the `FAIL()` handler function. Otherwise, it logs a message using the provided logging macro `PLOG(INFO)`.

```
1 TEST(PLogTest, WriteLog) {
2   std::fstream file("/a/file/that/does/not/exist.txt", std::fstream::in);
3   if (file.is_open()) { FAIL(); /* We do not expect to open file */ }
4   PLOG(INFO) << "The plog";
5   std::string expected = BUILD_STR(getDate() << " The plog: No such file or directory [2]\n");
6   std::string actual = tail(1);
7   EXPECT_EQ(expected, actual); // Check if the log is as expected
8 }
```

Fig. 6: Example test case adapted from the Easylogging library [50], which yield different outcomes in x86-64 and Wasm binaries due to inconsistent error numbers in `errno.h`.

The recorded log is expected to be a string including the current date and the appropriate message followed by a specific error number `2` which corresponds to `ENOENT` macro defined at `errno.h` in `libc`. On the other hand, the `musl` library which is used by Emscripten borrows the error numbers from the WebAssembly System Interface (WASI). It therefore defines a different error number (i.e., `44`) for the errors which occur when attempting to open non-existent files. Accordingly, the log statement retrieved by `tail(1)` does not match the string `expected`, and the test fails in WebAssembly.

Overall, different implementations of standard libraries may rely on different hashing mechanisms for data structures such as sets and maps, which affect the order in which elements appear within these structures. In addition, error numbers and static members of template classes such as `std::numeric_limits`, which represents properties of arithmetic types, may be assigned different values by different libraries. We observed 13 discrepancies among test outcomes, which stem from implementation inconsistencies between the standard libraries used by GCC / Clang and Emscripten.

2) Unsupported system calls and APIs: Native binaries are directly executed by the operating system on the hardware, and take full advantage of the processor and system features available on a platform. On the other hand, WebAssembly code is executed within a sandboxed VM that restricts resource access (e.g., to file systems and network interfaces). WebAssembly instead relies on the browser APIs or the embedding environment for file access, networking, and other system-level operations.

As mentioned in the previous sections, there is a class of system-level POSIX APIs that are currently unsupported by Emscripten. If a codebase uses such APIs, Emscripten generates an undefined symbol error when compiling that code. There also exist unimplemented system calls (e.g., `fork`, `exec`) that do not break the compilation process by introducing errors. In contrast, they always fail when executed. For example, the code fragment in Figure 7 relies on the `fork` system call. Although this code is smoothly compiled into WebAssembly, the `if`-condition on Line 3 evaluates to `false` at runtime and the code on Line 4 remains unexecuted. This occurs since Emscripten does not emulate the functionality of the `fork` system call, which thereby causes the invocation on Line 2 to return -1.

```
1 int get_child(){
2   int pid = fork();
3   if (pid > 0) {
4     return pid;
5   } return 0;
6 }
```

```
1 extern int get_child();
2 int child_pid = get_child();
3 if (child_pid > 0) {
4   /* some lines of code */
5 }
```

Fig. 7: Code with the `fork` system call.

As another example, given C programs that utilize TCP networking using POSIX socket APIs, Emscripten tries to mimic the connection over WebSockets. Nevertheless, this emulation is currently incomplete, which causes runtime failures of unsupported socket primitives. Overall, we identified 197 discrepancies in test outcomes which occurred due to unsupported system calls / APIs.

3) WebAssembly language features: The core specification of WebAssembly is intended to address the problem of safe, fast, portable, low-level code on the web [1]. To achieve this goal, WebAssembly introduces new features which result in differences in code execution compared to native binaries. For example, both direct and indirect calls are subject to dynamic type checking in WebAssembly, which prevents the execution of functions in the case of a signature mismatch. Accordingly, running the Wasm binary of the code in Figure 8 results in a runtime error whose description is the string *unreachable*. This issue specifically arises from a mismatch between the declaration and the definition of the function `func`. In contrast, we observed smooth execution of the corresponding x86-64 binaries which were produced by GCC and Clang. We identified 1 test with inconsistent behavior in the two binaries, which occurs due to similar runtime type checks.

```
1 int func();
2 int main() { func(); }
```

```
1 void func() {
2   /* some lines of code */
3 }
```

Fig. 8: Example of inconsistent function definition in C/C++.

4) Compiler bugs: We also identified several bugs within the Emscripten compiler toolchain which are responsible for unexpected Wasm code behaviors. In these cases, although the WebAssembly code was successfully built, discrepancies were introduced in their semantics, which resulted in different test outcomes in WebAssembly and x86-64 binaries.

For example, an implementation bug in the file system emulation layer prevents empty directories from being loaded into the virtual file system. Consider the Figure 9 using which we describe how this bug affects the program behavior. The right-hand portion of this figure visualizes the file system's structure starting from the `dir` directory, which includes a text file and two subdirectories, `empty` and `non-empty`. Given this hierarchy, the code snippet on the left opens the `dir` directory and iterates over the entries within it. This program also executes the code section on Line 5 when it encounters the `empty` directory.

Notice that on a native platform which allows the execution of x86-64 binaries, the `if`-condition on Line 4 evaluates to `true`. This allows the code on Line 5 to be executed as well. On the other hand, when we use Emscripten and preload the entire `dir` in its VFS using `--preload-file` option, the `empty` directory is excluded, and therefore the cross-compiled Wasm code does not execute the section which resides on Line 5.

Overall, we identified 7 bugs within the Emscripten compiler, including 6 new bugs confirmed by developers, which resulted in a total of 14 test inconsistencies between the two binaries.

```
1 DIR* _dir = ::opendir("/path/to/dir"); dirent* _entry;
2 if (!_dir) {std::cout << "dir is null"; exit(0);}
3 while((_entry = ::readdir(_dir)) != nullptr) {
4   if (!strcmp(_entry->d_name, "empty")) {
5     /* some lines of code */
6   }
7 } ...
```

```
>_:/path/to/dir$ tree
.
|-- data.txt
|-- empty
|-- non-empty
    `-- program.c
2 directories, 2 files
```

Fig. 9: Example code fragment to show the impact of a bug in the Emscripten's VFS emulation on the program behavior.

Takeaways: The Emscripten compiler with user-specified settings still does not necessarily preserve original code semantics in Wasm binaries because of several reasons: (*a*) Traditional C/C++ compilers and WebAssembly compilers use different standard libraries with differing implementations of data structures and algorithms. (*b*) WebAssembly compilers do not fully emulate the functionality of all system-level APIs / system calls. (*c*) WebAssembly provides new features (e.g., dynamic check of function signatures), which may differ from native binaries, to ensure the safe and fast execution of code. (*d*) WebAssembly compilers still contain bugs, which can affect the semantics of the resulting binaries.

V. Discussion and Future Work

We have discussed the main findings and key takeaways of this study within each section. We now review further insights from our work that suggest promising directions for future research. We hope these insights highlight avenues to improve WebAssembly compilers and streamline high-level language code reuse in widespread WebAssembly applications.

First, the current version of WebAssembly imposes a tradeoff between execution performance and code reusability. Although recent proposals extend WebAssembly's capabilities to support various C/C++ language features, such as exceptions, these proposals need to be improved in terms of performance cost. Developing compiler-level techniques to mitigate execution overhead without sacrificing the runtime reliability of code is also another direction for future research.

Second, WebAssembly code runs in a sandbox environment without direct access to underlying system resources. The file system access operations therefore must be emulated to perform through a virtual file system over browser APIs. As we noted earlier, identifying files which need to be embedded into this virtual file system complicates WebAssembly compilation process. Advanced static / dynamic analysis techniques can be leveraged to perform precise file embedding.

Finally, as we discussed in Section IV-B5, developing automatic techniques for reliable source code adaptation is also an important direction which further addresses the challenges in Section III and streamlines legacy code reuse in WebAssembly.

VI. Threats to Validity

Our study might be potentially subject to several threats, including the representativeness of the chosen codebases and the correctness of the analysis methodology.

Regarding the representativeness of the codebases in the dataset, we selected real-world C/C++ projects hosted on GitHub which contain test cases and a CMake build script (i.e., `CMakeLists.txt`). While our collected dataset offers valuable insights, it might not be fully representative of the broader landscape of the challenges which are encountered when using legacy code in WebAssembly.

Another threat concerns the correctness of the analysis methodology. We confine the analysis of code semantics to those functional behaviors that can be evaluated using a set of predefined test cases. To alleviate this threat, we used real-world codebases from public repositories with over 50 stars. This serves as a proxy for project maturity to ensure that we use codebases with diverse and comprehensive test suites.

VII. Related Work

1) Analyzing binary security of WebAssembly: There is a large body of research on WebAssembly binary analysis, particularly for security purposes [25]–[27], [51]–[67]. The closest work to ours is by Stiévenart et al. [26], who conducted a comparative analysis of vulnerable code behavior in WebAssembly and x86-64. Through manual compilation of benchmarks in the Juliet C/C++ test suite [68] into Wasm, their work showed that negligent code compilation may expose security vulnerabilities in WebAssembly applications. Instead of targeting small, synthetic, vulnerable programs, we analyze the functional behavior of real-world codebases in WebAssembly and x86-64 using an automated differential testing framework and by taking advantage of open-source tests.

2) Differential testing for bug diagnosis in compilers: Differential testing—which involves comparing multiple implementations of the same specification—has been established as a highly effective technique for detecting software bugs [69]–[74]. Csmith [70] is a randomized code generation tool that leverages differential testing for bug discovery in C compilers. Inspired by Csmith, RustSmith [71] randomly generates programs that conform to the type system of Rust in terms of rules related to borrowing and lifetimes, guaranteed to yield a well-defined output for differential testing of Rust compilers. These approaches use differential testing to detect bugs in compilers of a specific language. In contrast to these papers, which focus on multiple compilers that target the same architecture, we use differential testing to identify discrepancies between compilers targeting different architectures.

3) Differential testing among WebAssembly runtimes: Researchers have also broadly used differential testing to analyze discrepancies among Wasm runtimes [75]–[82]. Since WebAssembly does not mandate a specific implementation, different WebAssembly engines may vary in how they apply the specification. Perényi et al. [75] and Hamidy et al. [76] employed differential fuzz testing to identify bugs and discrepancies in browser-based WebAssembly engines. WADIFF [77] generates sufficient test cases by combining a DSL transformer with a symbolic execution engine. WASMaker [80] creates diverse WebAssembly codes by assembling basic elements from real-world Wasm binaries and applying mutation strategies. WarpDiff [78] applies differential testing in a different domain to detect performance issues in server-side Wasm runtimes.

VIII. Conclusion

We conducted a two-phase study to investigate the capability of WebAssembly compilers to realize code reusability. We categorized failures that one might encounter when compiling legacy C/C++ code into Wasm and developed a differential testing framework, WasmChecker, to test the semantic equivalence of WebAssembly and x86-64 binaries. Using WasmChecker and a set of open-source C/C++ codebases, we showed that WebAssembly compilers do not necessarily preserve source code semantics in the resulting Wasm binaries.

Data Availability

Our dataset of open-source projects, WasmChecker's source code, and the replication scripts are available at [83].

Acknowledgments

We thank the anonymous SANER reviewers for their constructive feedback. This research was supported in part by the U.S. National Science Foundation (NSF) under grants 2409005, 2321444, and 2146518. Note that any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web Up to Speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017, p. 185–200.

[2] R. Battagline, *Hands-On Game Development with WebAssembly: Learn WebAssembly C++ Programming by Building A Retro Space Game*. Packt Publishing Ltd, 2019.

[3] K. Liu, N. Wu, and B. Han, "Demystifying Web-based Mobile Extended Reality Accelerated by WebAssembly," in *Proceedings of the 2023 ACM on Internet Measurement Conference (IMC)*, 2023, p. 145–153.

[4] B. B. Khomtchouk, "WebAssembly Enables Low Latency Interoperable Augmented and Virtual Reality Software," *arXiv preprint arXiv:2110.07128*, 2021.

[5] D. Pisanò and A. Servetti, "Audio-aware Applications at the Edge Using in-browser WebAssembly and Fingerprinting," in *Proceedings of the 4th International Symposium on the Internet of Sounds*, 2023, pp. 1–9.

[6] L. V. S. Kaluva and A. Hossain, "WebAssembly for Video Analysis: An Explorative Multi-method study," 2020.

[7] "Can I use: up-to-date browser support tables for support of front-end web technologies on desktop and mobile web browsers," 2008. [Online]. Available: https://caniuse.com/wasm

[8] P. Mendki, "Evaluating WebAssembly Enabled Serverless Approach for Edge Computing," in *2020 IEEE Cloud Summit*, 2020, pp. 161–166.

[9] M. N. Hoque and K. A. Harras, "WebAssembly for Edge Computing: Potential and Challenges," *IEEE Communications Standards Magazine*, vol. 6, no. 4, pp. 68–73, 2022.

[10] P. P. Ray, "An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions," *Future Internet*, vol. 15, no. 8, 2023. [Online]. Available: https://www.mdpi.com/1999-5903/15/8/275

[11] E. Wen and G. Weber, "Wasmachine: Bring IoT up to Speed with A WebAssembly OS," in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2020, pp. 1–4.

[12] B. Li, H. Fan, Y. Gao, and W. Dong, "Bringing WebAssembly to Resource-constrained IoT Devices for Seamless Device-cloud Integration," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2022, p. 261–272.

[13] K. Moron and S. Wallentowitz, "Benchmarking WebAssembly for Embedded Systems," *ACM Trans. Archit. Code Optim.*, no. 3, 2025.

[14] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, and L. Wu, "WASAI: Uncovering Vulnerabilities in Wasm Smart Contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022, p. 703–715.

[15] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang, "EOSAFE: Security Analysis of EOSIO Smart Contracts," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1271–1288. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/he-ningyu

[16] J. Zhou and T. Chen, "Wasmod: Detecting vulnerabilities in wasm smart contracts," *IET Blockchain*, vol. 3, no. 4, pp. 172–181, 2023. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/blc2.12029

[17] "Emscripten: a complete open source LLVM-based compiler toolchain for WebAssembly," 2010. [Online]. Available: https://emscripten.org

[18] "Binaryen: an optimizer and toolchain library for WebAssembly," 2015. [Online]. Available: https://github.com/WebAssembly/binaryen

[19] "AssemblyScript: a TypeScript-like language for WebAssembly," 2017. [Online]. Available: https://github.com/AssemblyScript/assemblyscript

[20] "Cheerp: an enterprise-grade C++ compiler for the web," 2019. [Online]. Available: https://leaningtech.com/cheerp

[21] "Asterius: a Haskell to WebAssembly compiler," 2017. [Online]. Available: https://github.com/tweag/asterius

[22] "Wasm-Bindgen: a Rust library and CLI tool for facilitating high-level interactions between WebAssembly modules and JavaScript," 2017. [Online]. Available: https://github.com/rustwasm/wasm-bindgen

[23] "Emscripten GitHub issues." [Online]. Available: https://github.com/emscripten-core/emscripten/issues

[24] "AssemblyScript GitHub issues." [Online]. Available: https://github.com/AssemblyScript/assemblyscript/issues

[25] Q. Stiévenart, C. De Roover, and M. Ghafari, "The Security Risk of Lacking Compiler Protection in WebAssembly," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 132–139.

[26] Q. Stiévenart, C. De Roover, and M. Ghafari, "Security Risks of Porting C Programs to WebAssembly," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC)*, 2022, p. 1713–1722.

[27] D. Lehmann, J. Kinder, and M. Pradel, "Everything Old is New Again: Binary Security of WebAssembly," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 217–234.

[28] Y. Zhang, M. Liu, H. Wang, Y. Ma, G. Huang, and X. Liu, "Research on webassembly runtimes: A survey," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 8, 2025.

[29] O. Draissi, T. Cloosters, D. Klein, M. Rodler, M. Musch, M. Johns, and L. Davi, "Wemby's Web: Hunting for Memory Corruption in WebAssembly," *Proceedings of the 34rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, vol. 2, pp. 1326–1349, 2025.

[30] S. Baradaran, M. Heidari, A. Kamali, and M. Mouzarani, "A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes," *International Journal of Information Security*, vol. 22, no. 5, pp. 1277–1290, 2023.

[31] M. Mouzarani, A. Kamali, S. Baradaran, and M. Heidari, "A unit-based symbolic execution method for detecting heap overflow vulnerability in executable codes," in *International Conference on Tests and Proofs (TAP)*. Springer, 2022, pp. 89–105.

[32] A. Romano, X. Liu, Y. Kwon, and W. Wang, "An Empirical Study of Bugs in WebAssembly Compilers," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022, p. 42–54.

[33] "GCC, the GNU Compiler Collection," 1987. [Online]. Available: https://gcc.gnu.org

[34] "Clang Compiler." [Online]. Available: https://clang.llvm.org/docs/UsersManual.html#introduction

[35] "Code Optimizations in Emscripten," 2017. [Online]. Available: https://emscripten.org/docs/optimizing/Optimizing-Code.html

[36] "Catch2," 2010. [Online]. Available: https://github.com/catchorg/Catch2

[37] "GoogleTest," 2008. [Online]. Available: https://github.com/google/googletest

[38] "GitHub REST API," 2013. [Online]. Available: https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28

[39] "yaml-cpp: A YAML parser and emitter in C++," 2008. [Online]. Available: https://github.com/jbeder/yaml-cpp

[40] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *7th USENIX Security Symposium (USENIX Security 98)*, vol. 98, 1998, pp. 63–78.

[41] "Portability At The C/C++ Level," 2017. [Online]. Available: https://webassembly.org/docs/portability

[42] W. Tang, Z. Xu, C. Liu, J. Wu, S. Yang, Y. Li, P. Luo, and Y. Liu, "Towards Understanding Third-party Library Dependency in C/C++ Ecosystem," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[43] "Boost C++ Libraries," 2000. [Online]. Available: https://github.com/boostorg/boost

[44] "Emscripten ported libraries," 2014. [Online]. Available: https://github.com/emscripten-ports

[45] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, 2009, p. 81–92.

[46] "File System Overview in Emscripten," 2017. [Online]. Available: https://emscripten.org/docs/porting/files/file_systems_overview.html

[47] O. d. Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble, "Keynote Address: .QL for Source Code Analysis," in *Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2007, p. 3–16.

[48] "Graaf lib: A general-purpose lightweight graph library," 2023. [Online]. Available: https://bobluppes.github.io/graaf

[49] D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond, "String Analysis for Java and Android Applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015, p. 661–672.

[50] "Easylogging++: A logging library for C++ applications," 2012. [Online]. Available: https://github.com/abumq/easyloggingpp

[51] G. Perrone and S. P. Romano, "WebAssembly and security: A review," *Computer Science Review*, vol. 56, p. 100728, 2025.

[52] A. Hilbig, D. Lehmann, and M. Pradel, "An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases," in *Proceedings of the Web Conference (WWW)*, 2021, p. 2696–2708.

[53] X. Wu, J. He, L. Huang, C. Fu, and W. Wang, "Wbsan: Webassembly bug detection for sanitization and binary-only fuzzing," in *Proceedings of the ACM on Web Conference (WWW)*, 2025, p. 3311–3322.

[54] L. Huang, J. He, C. Wang, and W. Wang, "Wascr: A webassembly instruction-timing side channel repairer," in *Proceedings of the ACM on Web Conference (WWW)*, 2025, p. 4562–4571.

[55] D. Lehmann, M. T. Torp, and M. Pradel, "Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly," 2021. [Online]. Available: https://arxiv.org/abs/2110.15433

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

[57] T. Brito, P. Lopes, N. Santos, and J. F. Santos, "Wasmati: An efficient static vulnerability scanner for webassembly," *Comput. Secur.*, vol. 118, no. C, 2022.

[58] F. Breitfelder, T. Roth, L. Baumgärtner, and M. Mezini, "Wasma: A static webassembly analysis framework for everyone," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023, pp. 753–757.

[59] N. He, Z. Zhao, J. Wang, Y. Hu, S. Guo, H. Wang, G. Liang, D. Li, X. Chen, and Y. Guo, "Eunomia: Enabling user-specified fine-grained search in symbolically executing webassembly binaries," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023, p. 385–397.

[60] A. Romano and W. Wang, "Automated webassembly function purpose identification with semantics-aware analysis," in *Proceedings of the ACM Web Conference (WWW)*, 2023, p. 2885–2894.

[61] A. Romano, Y. Zheng, and W. Wang, "Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection," in *Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2020, p. 1129–1140.

[62] W. Fang, Z. Zhou, J. He, and W. Wang, "Stacksight: Unveiling webassembly through large language models and neurosymbolic chain-of-thought decompilation," in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

[63] W.-C. Wu, Y. Yan, H. D. Egilsson, D. Park, S. Chan, C. Hauser, and W. Wang, "Is this the same code? a comprehensive study of decompilation techniques for webassembly binaries," in *Security and Privacy in Communication Networks*, S. Alrabaee, K.-K. R. Choo, E. Damiani, and R. H. Deng, Eds. Springer Nature Switzerland, 2026, pp. 108–130.

[64] X. She, Y. Zhao, and H. Wang, "Wadec: Decompiling webassembly using large language model," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, p. 481–492.

[65] H. Huang and J. Zhao, "Multi-modal learning for webassembly reverse engineering," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024, p. 453–465.

[66] D. Lehmann and M. Pradel, "Finding the dwarf: Recovering precise types from webassembly binaries," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2022, p. 410–425.

[67] A. Benali, "An initial investigation of neural decompilation for webassembly," Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2022.

[68] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java Test Suite," *Computer*, vol. 45, no. 10, pp. 88–90, 2012.

[69] W. M. McKeeman, "Differential Testing for Software," *Digit. Tech. J.*, vol. 10, pp. 100–107, 1998. [Online]. Available: https://api.semanticscholar.org/CorpusID:14018070

[70] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, p. 283–294.

[71] M. Sharma, P. Yu, and A. F. Donaldson, "RustSmith: Random Differential Compiler Testing for Rust," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023, p. 1483–1486.

[72] H. Zhong, "Enriching Compiler Testing with Real Program from Bug Report," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[73] S. Li and Z. Su, "Finding Unstable Code via Compiler-Driven Differential Testing," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023, p. 238–251.

[74] G. Ofenbeck, T. Rompf, and M. Püschel, "RandIR: Differential Testing for Embedded Compilers," in *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala (SCALA)*, 2016, p. 21–30.

[75] Á. Perényi and J. Midtgaard, "Stack-Driven Program Generation of WebAssembly," in *roceedings of the 18th Asian Symposium on Programming Languages and Systems (APLAS)*. Springer, 2020, pp. 209–230.

[76] G. Hamidy, "Differential fuzzing the webassembly," 2020.

[77] S. Zhou, M. Jiang, W. Chen, H. Zhou, H. Wang, and X. Luo, "WADIFF: A Differential Testing Framework for WebAssembly Runtimes," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 939–950.

[78] S. Jiang, R. Zeng, Z. Rao, J. Gu, Y. Zhou, and M. R. Lyu, "Revealing Performance Issues in Server-Side WebAssembly Runtimes Via Differential Testing," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 661–672.

[79] S. Cao, N. He, X. She, Y. Zhang, M. Zhang, and H. Wang, "WRTester: Differential Testing of WebAssembly Runtimes via Semantic-Aware Binary Generation," *arXiv preprint arXiv:2312.10456*, 2023.

[80] Cao, Shangtong and He, Ningyu and She, Xinyu and Zhang, Yixuan and Zhang, Mu and Wang, Haoyu, "WASMaker: Differential Testing of WebAssembly Runtimes via Semantic-Aware Binary Generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024, p. 1262–1273.

[81] S. Zhou, J. Wang, H. Ye, H. Zhou, C. Le Goues, and X. Luo, "LWDIFF: an LLM-Assisted Differential Testing Framework for WebAssembly Runtimes," in *IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 769–769.

[82] Y. Hu, W. Zhang, B. Xiao, Q. Kong, B. Yi, S. Ji, S. Wang, and W. Wang, "WASIT: Deep and Continuous Differential Testing of WebAssembly System Interface Implementations," in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP)*, 2025, pp. 719–735.

[83] "WasmChecker: A differential testing framework." [Online]. Available: https://github.com/SaraBaradaran/WasmChecker