RANDOM TESTING OF WEBASSEMBLY INTERFACE TYPE BINDING GENERATORS

by

Ethan Stanley

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Kahlert School of Computing
The University of Utah
May 2025

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Ethan Stanley**

has been approved by the following supervisory committee members:

Eric Norman Eide ,	Chair(s)	**03/14/2025** Date Approved
Mu Zhang ,	Member	**03/14/2025** Date Approved
Stefan Nagy ,	Member	**03/14/2025** Date Approved

by **Mary W. Hall** , Chair/Dean of

the Department/College/School of **Computing**

and by **Darryl P. Butt** , Dean of The Graduate School.

ABSTRACT

We evaluate the effectiveness of random differential testing for finding bugs in WebAssembly Interface Type (WIT) binding generators, which are code generators used within the WebAssembly component ecosystem. Components are composable wrappers around ordinary WebAssembly modules; components specify their imports and exports in a standard format, enabling the interoperation of programs written in different programming languages. A WIT binding generator creates the code that allows the WebAssembly modules within components to communicate, i.e., to invoke each other's exported functions and receive the results of those invocations. It is challenging to produce correct binding code due to potential mismatches in programming language semantics and execution models, and bugs in binding code can undermine the correctness of systems that use WebAssembly components. Therefore, it is necessary to thoroughly and effectively test WIT binding generators.

We implemented a system to perform random differential testing for two WIT binding generators, called wit-bindgen and wit-bindgen-go. Our system uses these binding generators to produce multiple WebAssembly components with the same behavior from programs in different high-level languages. If their run-time behaviors differ, we expect that there is a bug in one of the generated bindings. Using our framework, we discovered four previously unknown code-generation defects in wit-bindgen and wit-bindgen-go. We analyze the nature of these bugs to evaluate the effectiveness of our framework.

CONTENTS

ACKNOWLEDGMENTS

I'd like to thank my advisor, Professor Eric Eide, for invaluable guidance and feedback. I'd also like to thank my family and friends for their support and encouragement.

This material is based upon work supported by the National Science Foundation under Grant Number 2431419. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

CHAPTER 1

INTRODUCTION

In this thesis, we evaluate the effectiveness of random differential testing for detecting bugs in WebAssembly Interface Type (WIT) binding generators. We develop a differential testing framework for WIT binding generators and use it to discover novel bugs in wit-bindgen [5] and wit-bindgen-go [6].

WIT binding generators enable programs written in guest languages (programming languages that support compilation to WebAssembly) to be compiled to WebAssembly components. A component is a wrapper around a core WebAssembly module that declares a public interface using more complex types than those supported by WebAssembly modules (integers and floats). Wit-bindgen and wit-bindgen-go achieve this by generating "binding code" that translates types defined by a guest language program to and from the binary format supported by WebAssembly components. This binary format is called the Canonical ABI [22]. Components compiled from different guest languages can invoke one another's functions and exchange data via function arguments and return values without confusion because the Canonical ABI provides a common format for them to communicate. In this way, the WebAssembly component model greatly increases the portability of WebAssembly code. For example, a software library that is compiled to a WebAssembly component can be used by programs written in any language that also supports the component model.

We claim that it is important to thoroughly test WIT binding generators for the following three reasons. First, these tools support a growing number of target languages, so they have the potential to be widely deployed in many software systems. Second, the generation of binding code is complex and potentially error-prone. Generated bindings may require unsafe language features when languages have different semantics, such as memory-management or error-handling conventions. Third, bugs in binding code can be hard to detect and fix in large software systems.

This thesis provides evidence in support of the following hypothesis:

Random differential testing is an effective method of discovering bugs in WIT binding generators.

Random testing has been shown to be effective for detecting novel bugs in software systems, especially when the space of possible inputs is large. When this is the case, it is difficult to cover all edge cases using traditional methods for testing software, such as writing unit tests. For WIT binding generators, the number of possible component interfaces for which binding code may need to be generated is arbitrarily large. In addition, a single component interface has many possible implementations. Thus, a random testing framework for WIT binding generators could complement traditional testing.

An effective random testing framework for a WIT binding generator should use the generator to produce binding code for arbitrary component interfaces and test the correctness of that code. A WIT binding generator knows what binding code to generate by accepting a description of the intended component's interface as input. This description is written in the WIT (WebAssembly Interface Type) language [23]. WIT is an interface definition language (IDL) for types supported by WebAssembly components. To produce random bindings, a random valid WIT file must be given as input to a WIT binding generator. We implement a test case generator to produce such files. Our test case generator uses wit-smith [2], an existing random WIT file generator developed by the Bytecode Alliance, to obtain raw test cases before performing additional filtering and post-processing.

Once binding code has been generated, various methods may be used to determine its correctness, i.e., to detect "semantic bugs" in the WIT binding generator that produced it. We define a semantic bug as a defect in a WIT binding generator that results in the generation of incorrect binding code, such as binding code that fails to compile or behaves incorrectly at run time. A simple approach to detect semantic bugs is to check if generated binding code can be compiled without raising compile-time errors. A more sophisticated approach is to use the binding code to create a WebAssembly component and check if the functions of the component raise run-time errors when executed. It is also possible for the generated binding code to incorrectly convert data values between languages without failing to compile or raising run-time errors. We need a method of detecting these data conversion bugs in generated binding code.

We achieve this using differential testing: a software testing technique that detects bugs by providing the same input to a set of similar applications and observing differences in their execution [13]. In our case, our framework creates multiple, similar applications for each WIT file produced by our test case generator. For each WIT file, our framework invokes WIT binding generators to produce bindings in multiple programming languages. Our framework uses these bindings to create and compile multiple components with the same behavior. Our framework then executes each component and compares their behaviors. If the outputs of the components differ, then we expect that there is a bug in one or more of the generated bindings.

To generate components that we expect to behave equivalently, we need source programs in each target language that have the same behavior. The source programs must also correctly make use of binding code produced by a WIT binding generator. We implement a source program generator to produce random programs that fulfill these requirements. To automate the steps of our testing pipeline and log any found defects, we implement a test harness.

Using our testing framework, we discovered four unique code-generation defects in wit-bindgen and wit-bindgen-go. We ran our testing framework in extended fuzzing campaigns in a cloud computing testbed [9]. We analyze the bugs we found to evaluate the effectiveness of our testing framework. Based on the bugs we found, we conclude that random differential testing can be an effective method of discovering code-generation bugs in WIT binding generators.

The remainder of this thesis describes the implementation and evaluation of our random testing framework. Chapter 2 provides background information on WebAssembly, the component model, WIT, and the WIT binding generators tested by our framework. Chapter 3 summarizes prior efforts to detect bugs in binding code. Chapter 4 describes the implementation of our framework. Chapter 5 describes our experimental setup, results, and analysis. Chapter 6 concludes and summarizes potential future work.

CHAPTER 2

BACKGROUND

WebAssembly (Wasm) [10] is a binary instruction format with properties that make it useful for implementing both portable and mobile code. It is intended for execution by a stack-based virtual machine. It makes no architectural assumptions, and it is therefore portable across different hardware and operating systems. It runs in a sandboxed, memory-safe environment. Wasm has a compact bytecode representation and can be loaded quickly.

The unit of compilation for Wasm is a module. Conceptually, a module is a collection of functions that share linear memory and global variables. Wasm modules can interface with each other and their runtime environment by importing and exporting functions and linear memory.

As an assembly-like compilation target, Wasm modules support four basic types: 32- and 64-bit integers and 32- and 64-bit floats. More complex data types must be encoded in linear memory and passed between functions using indexes into that memory (i.e., pointers). Compilers for high-level languages targeting Wasm are not constrained in how they represent values of complex types in linear memory. For example, a C program compiled to Wasm might encode a string differently than a Go program. This inhibits meaningful function calls between Wasm modules compiled from different high-level languages.

The WebAssembly component model [24] was recently introduced to remedy this. A WebAssembly component is a wrapper around a traditional Wasm module that defines its public interface using more complex types than those supported by Wasm natively. Examples of component model types include strings, lists, records, variants, tuples, and options. Notably, components are not allowed to export or import linear memory. Components are composable, meaning that compatible imported and exported functions of different components can be "plugged" together to form a new component—analogous to how traditional object files are linked. To ensure that linked functions of different components can successfully pass and receive arguments, all components are required to adhere to the Canonical ABI for externally visible functions [24]. The Canonical ABI specifies how component model functions are implemented as core module functions, including function names, function signatures, and the encoding of data values in linear memory. As a result, two components with compatible interfaces may call one another's functions without ever knowing that, internally, their types are represented and manipulated differently.

While Wasm components enable function calls between different high-level languages, it is unreasonable to expect all compiler implementations to modify how they encode data types in order to adhere to the Canonical ABI. An alternative approach is to use "binding code" to translate data types between the Canonical ABI and their natural representation in a high-level language. For example, an option type is naturally represented in C as a pointer to a value. When the pointer is null, there is no value; otherwise, the value is obtained by dereferencing the pointer. In contrast, the Canonical ABI represents an option type with a value placeholder and a flag indicating if the placeholder contains the value. C binding code would automatically translate between these representations so that it is not the responsibility of the compiler or the programmer.

In practice, binding code is produced using a tool called a binding code generator. For compiled languages, a binding code generator accepts a description of the component's intended interface and outputs type definitions and function stubs for the imported and exported functions of the component. A user implements the stubs of exported functions and may optionally invoke the stubs of imported functions. When the user-implemented function stubs are compiled using the language's native Wasm toolchain, the binding code guarantees that the functions imported and exported by the module match the Canonical ABI's specifications. Next, the module is wrapped into a component with additional metadata describing the component's interface. This step may be done automatically by toolchains that natively support compilation to Wasm components.

Wit-bindgen and wit-bindgen-go both facilitate the creation of Wasm components by generating binding code that is intended to be used in this manner. Wit-bindgen is a collection of binding code generators for high-level languages [5]. It is written in Rust, and the binding generators share a common back end. Wit-bindgen-go is a separate binding generator for Go, written entirely in Go [6]. In this thesis, we test C and Rust bindings produced by wit-bindgen and Go bindings produced by wit-bindgen-go.

As mentioned in Chapter 1, WIT binding generators accept a description of a component model interface written in WIT. Using WIT, one can define component model types, function signatures, interfaces (collections of types and functions), worlds (all the imported and exported functions or interfaces of a single component), and packages (groups of related WIT definitions to enable their reuse). In this thesis, we focus on world definitions because they correspond to the external interface of a Wasm component. Therefore, they are the smallest WIT definition for which we can generate and implement bindings to create a component.

CHAPTER 3

RELATED WORK

Brown et al. described binding code as having "... the dangerous distinction of being both hard to avoid and hard to get right" [7]. Binding code is ubiquitous because it must be generated any time function calls are made across language boundaries. For example, the runtimes of high-level languages often use bindings to low-level system libraries to implement functionality such as file system and network access. Binding code has a high risk of containing bugs and vulnerabilities. It must translate values of various types, calling conventions, memory-management semantics, and error-handling across language boundaries. Because binding code is prevalent and liable to error, a variety of approaches have been used to ensure its correctness, including static analysis, dynamic checking, and random testing.

Effective static analysis of binding code is difficult because the analyzer must understand both the guest and host language and be able to operate over the boundary between them. Tan et al. introduced ILEA (Inter-LanguagE Analysis): a framework that enables static analyzers for Java to understand the behavior of C code [20]. This enables static analysis of foreign function calls from Java to C using the Java Native Interface (JNI). The authors used ILEA to detect dozens of null-related bugs in Java applications that use the JNI.

Python specifies a foreign function interface that allows extension modules to be programmed in C or C++ [15]. Python uses reference counting to manage heap objects, but extension modules are outside of Python's memory management system. Therefore, a developer of an extension module is responsible for correctly updating reference counts. Li et al. introduced Pungi, a tool that statically detects reference counting errors in Python/C interface code [12]. They were able to detect over 150 errors in a collection of Python programs that use extension modules.

JavaScript runtimes are implemented in low-level languages such as C and C++. Runtimes such as Node.js give JavaScript code access to the filesystem and network using a binding-code layer. Brown et al. introduced a suite of static checkers for detecting errors in JavaScript binding code [7]. They used the checkers to formulate 81 exploits of security flaws introduced by binding code in widely used JavaScript runtimes.

However, not all rules can be enforced statically. Lee et al. showed how to formulate dynamic checkers of FFI violations using state machines [11]. They used this approach to create Jinn: a dynamic bug detection tool for the Java Native Interface. This approach can be used to enforce FFI rules at run time for many languages.

Fuzzing has also been used to test binding code. Notably, much work has been done on effective browser fuzzing [25, 27, 28]. Browser fuzzing includes testing the APIs that are exposed by the JavaScript runtime, such as the API to the DOM (Document Object Model). These APIs are implemented using a binding layer to code written in a lower-level language that implements their functionality. Thus, any framework for testing browser APIs is also testing the correctness of the binding code used to implement them.

A notable example of this is Favocado, a fuzzing framework for testing the binding layers of JavaScript runtimes [8]. Favocado parses IDL files that describe the APIs exposed by the JavaScript runtime under test. It uses the extracted information to generate test cases that use the bindings in semantically valid ways.

Rust-bindgen, a tool that generates bindings from Rust to C libraries, serves a similar purpose to wit-bindgen and is tested using fuzzing [21]. Random semantically valid C programs are generated using Csmith [26] and passed as input to rust-bindgen. If rust-bindgen crashes or the emitted bindings fail to compile, a bug is detected.

Wit-smith [2] is easily turned into a random testing framework for WIT binding generators using a similar strategy. For example, a framework could repeatedly provide WIT test cases generated by wit-smith to a WIT binding generator and monitor for crashes or attempt to compile the outputted bindings. The framework presented in this thesis uses wit-smith to randomly generate WIT files, but goes beyond testing that the generated bindings compile correctly. Instead, our framework performs random differential testing to test the run-time behavior of generated bindings.

We are not aware of any previous work to test binding generators using differential testing. Instead, previous frameworks detect bugs by monitoring for compilation failures, crashes, or undefined behavior using specialized tools such as address sanitizers. These frameworks necessarily fail to detect subtle issues in generated bindings, such as the corruption of data when translating between languages. This kind of bug may cause the program to produce an incorrect result without crashing or failing to compile, therefore going undetected by existing tools.

In summary, the work presented in this thesis differs from past work because it is the first random testing framework for WIT binding generators that evaluates the run-time behavior of generated bindings. It is also, to our knowledge, the first differential testing framework for a language binding generator.

CHAPTER 4

METHODOLOGY

In this chapter, we describe our approach to implementing a differential testing framework for WIT binding generators. At a high level, our framework repeats the following steps. First, a random valid WIT test case is generated. Second, the test case is provided to wit-bindgen and wit-bindgen-go, which output binding code in each target language (C, Rust, and Go). Third, a Wasm component is created from each of the generated bindings. Finally, the components are executed and their behaviors are compared to detect any defects in the generated bindings.

We identify the following six challenges in creating a random differential testing framework for WIT binding generators:

1. Obtaining random valid WIT definitions.

2. Executing a Wasm component with an arbitrary interface.

3. Obtaining source programs that correctly implement arbitrary bindings.

4. Creating components that we expect to behave equivalently when executed.

5. Automating the testing pipeline and logging any found defects.

6. Verifying, reducing, and reporting bug-triggering test cases.

Section 4.1 describes solutions to Challenges 1 and 2. Section 4.2 provides solutions for Challenges 3 and 4. Section 4.3 and Section 4.4 address Challenges 5 and 6, respectively.

4.1 WIT Generation

Given a WIT world definition, which describes the interface of a single component, a WIT binding generator creates binding code that implements the definition in some target programming language. Therefore, to randomly test WIT binding generators, we need a way to obtain random world definitions (Challenge 1). Our goal is to find defects in binding code produced by WIT binding generators rather than to test their mechanisms for parsing and validating WIT files. Therefore, we want to obtain *valid* world definitions. Fortunately, a tool already exists for this purpose. Wit-smith is a tool for generating valid WIT files developed by the Bytecode Alliance [2]. We solve Challenge 1 by invoking wit-smith in our testing framework and manipulating its output, as described below.

For any valid WIT world definition, a component can be created that implements it. However, a component that implements an arbitrary interface cannot generally be executed "as is" (Challenge 2). This is because, at the time of writing, the only Wasm runtime that supports component execution is Wasmtime. Wasmtime only accepts components that (1) have no imports and (2) export a single function named *run*, which serves as the entry point of execution.

To resolve this, we use the fact that Wasm components are composable. Specifically, if we want to test a world w that implements an arbitrary (and randomly generated) interface, we define two additional worlds, e and i. The world e is the "exporter" for w because it exports all of w's dependencies and has no dependencies of its own. The world i is the "importer" for w because it imports the functions exported by w and exports only the function *run*. For each world definition (e, w, and i), we create a Wasm component that implements it. If we compose these three components, we obtain a component that can be executed using Wasmtime, i.e., it has no imports and only exports *run*. To compose Wasm components, we use WAC (WebAssembly Composition), a tool created by the Bytecode Alliance for this purpose [1]. Figure 4.1 visualizes this method of creating an executable component from an arbitrary world definition.

To simplify our testing framework, we place the following constraint on which worlds we test with our framework: a world is suitable only if it imports and exports at least one function. We call a world "chainable" if it satisfies this criterion. If a world is not chainable, it is invalid to compose it with "importer" and "exporter" components, since they have no functions in common.

With this in mind, we develop Atlas, a tool for generating WIT test cases suitable for the subsequent stages of our testing pipeline. Atlas is a command line application written in Rust. Below is the algorithm that Atlas implements:

1. Generate a WIT test case, t, using wit-smith.
2. If t contains no chainable worlds or has unimplemented features of WIT, return to step 1.
3. Select a chainable world, w, from t at random.
4. Remove all world definitions from t, except w.
5. Add definitions for worlds e and i to t.
6. Modify each world in t to import a checksum function.
7. Insert the seed used to generate t in a comment at the top of t.
8. Output t.

At steps 1 and 2, Atlas repeatedly invokes wit-smith until it produces a test case containing a chainable world. In addition, Atlas filters out test cases containing certain keywords. The list of keywords that Atlas searches for is configurable as a command line parameter. This allows Atlas to exclude test cases containing features of WIT that are incompatible with our testing framework. For example, the *use* keyword in WIT facilitates definition reuse, but alters the dependencies of a world in a way that conflicts with our three-world strategy for creating executable components. Thus, we filter out test cases containing the *use* keyword.

At steps 3 and 4, Atlas selects a chainable world from the test case at random and deletes all other world definitions. At step 5, Atlas inserts appropriate exporter and importer world definitions, as described earlier in this section. At step 6, Atlas modifies each world definition to import a function named *checksum*. This is a helper function for computing CRC-32 checksums whose utility will be explained in Section 4.2.

Finally, Atlas records the seed used to generate the test case in an inline comment. A user may pass a seed as a command line parameter to Atlas. If provided, the seed is used to initialize a pseudo-random number generator (PRNG). The PRNG is used to populate a buffer of random bytes that are provided as input to wit-smith. Wit-smith uses the buffered bytes to guide generation of a WIT test case. If the created test case is suitable, it is processed and output by Atlas. Otherwise, Atlas indicates that the provided seed cannot be used to generate a test case. If no seed is provided, Atlas repeatedly uses random seeds obtained from the Rust runtime system to generate test cases until an acceptable test case is created. The ability to specify a seed is useful for reproducing test cases that trigger bugs, as subsequently determined by our random testing framework. We discuss test case reproduction in Section 4.4.

Below is a simplified example of a WIT test case generated by wit-smith, before being processed by Atlas. It contains a package declaration and a single world definition, *x1*.

```
package name:name1;

world x1 {
  import xx: func(l: char, x: u8, x2: string);

  export v: func(name: string, name1: u16) -> bool;
}
```

Atlas will accept the test case above for post-processing because it contains a chainable world definition. Below is the same test case after being processed by Atlas.

```
// seed: 6653103418428468414

package name:name1;

world w {
  import xx: func(l: char, x: u8, x2: string);
  import hash:crc/checksum;

  export v: func(name: string, name1: u16) -> bool;
}
world e {
  import hash:crc/checksum;

  export xx: func(l: char, x: u8, x2: string);
}
world i {
  import v: func(name: string, name1: u16) -> bool;
  import hash:crc/checksum;
```

```
  export wasi:cli/run@0.2.0;
}

... hash:crc/checksum and wasi:cli/run definitions ...
```

The test case now contains three world definitions instead of one. The world *x1* was renamed to *w* and definitions were added for worlds *e* and *i*. All worlds import the *checksum* function, which is normally defined in the test case but omitted above for brevity. Thus, we have a randomly generated WIT test case from which we can create executable components.

4.2 Source Program Generation

As discussed in Chapter 2, creating a Wasm component from a world definition requires multiple steps. First, the world definition must be provided to a WIT binding generator. The WIT binding generator outputs binding code (function stubs, helper functions, and type definitions) for the world. Next, one must write a source program that implements the exported function stubs, optionally invoking imported functions. After that, the binding code and source program must be compiled together (to Wasm) using the source language's native toolchain. Finally, the resulting Wasm module must be turned into a component by adding metadata describing the world (the component's public interface). Figure 4.2 depicts this generic process of creating a Wasm component that implements a given world definition using a WIT binding generator.

To create a differential testing framework that automates the steps above, we need a way to obtain source programs that implement arbitrary world definitions produced by Atlas (Challenge 3). Our solution is to implement a source program generator. Specifically, our source program generator accepts a world definition and outputs three programs that implement it—one Rust program, one C program, and one Go program.

Our testing framework uses our source program generator to create executable components that are implemented by code in each of our target languages (Rust, C, and Go). Our testing framework does this in several steps. First, our framework invokes the source program generator three times—once for each world defined in a test case produced by Atlas (*e*, *w*, and *i*). Doing so yields a source program implementation of each world definition in all of the target languages. Next, our framework uses the process depicted in Figure 4.2 to create a component from each source program generated in the previous step. Specifically, our framework generates binding code for each source program using a WIT binding generator and compiles the resulting binding code-source program pairs into Wasm components. Finally, our framework composes together all the components that are compiled from the same target language: i.e., the Rust implementations of e, w, and i are composed, the C implementations of e, w, and i are composed, and the Go implementations of e, w, and i are composed. By design, composing the implementations of e, w, and i yields an executable component. Therefore, the previous step yields three executable components (one compiled from each target language). Figure 4.3 provides an overview of our framework's strategy for creating executable components from source programs.

Our framework performs differential testing by executing each component and comparing their outputs. If the implementation of each component were arbitrary, we could not expect them to behave equivalently. This would prevent us from performing meaningful differential testing (Challenge 4). Our solution is to generate source programs that are semantically equivalent. Specifically, when our source program generator receives a world definition, it outputs an equivalent source program in each target language. It achieves this by forcing all source programs to have the same structure. In addition, it initializes identical values (function arguments, return values) in programs that must behave equivalently. Below is the algorithm that exported functions in generated source programs implement:

1. Initialize arguments for each imported function.

2. Call each imported function and store their return values, if any.

3. Combine the return values with its own arguments using a CRC-32 checksum.

4. Write the CRC value to standard output.

5. Initialize a value and return it.

Two functions that implement the algorithm above are semantically equivalent if they initialize the same arguments and return values. Thus, our source program generator creates semantically equivalent programs in different target languages by generating functions that implement the algorithm above and initialize the same values. When our source program generator is invoked (provided with a world definition), it initializes a PRNG using a seed passed as a command line parameter. The PRNG is used to choose arbitrary values for all function arguments and return values needed to create a source program that implements the inputted world definition. Then, a source program that uses the chosen values is generated in each target language.

In summary, given a world definition, our source program generator outputs one Rust program, one C program, and one Go program. The three programs perform the same computations (they implement the same algorithm and initialize the same values). Each of these programs is compiled to a Wasm component (using a WIT binding generator and the program's toolchain), yielding three components that should behave equivalently. Invoking our source program generator on each world defined in a test case and composing components derived from the same source language, as shown in Figure 4.3, yields executable components in each target language that we expect to behave equivalently. This enables our framework to perform differential testing by executing each component and comparing their outputs.

The algorithm implemented by every exported function (shown above) is simple, yet still allows us to detect semantic defects in binding code. For example, if an argument is corrupted when calling an imported function, the hash value printed by the called function will be affected. Similarly, if a function's return value is corrupted, the hash value printed by the calling function will be affected. If other testing targets do not corrupt these values, our testing framework will detect the difference in printed hash values and flag the test case as a bug.

Below is a simplified example of a WIT world produced by Atlas for which we can generate semantically equivalent source programs in different target languages.

```
world w {
    import hash:crc/checksum;

    import bar: func(a: u8, b: bool, c: string) -> option<s64>;
```

```
    export foo: func(a: tuple<s16, bool>) -> char;
}
```

To implement *w*, our source program generator first obtains random argument values for *bar* and a random return value of *foo*. Assume that the arguments for *bar* are 76, true, and "hello", and the return value of *foo* is 'q'. Below is an implementation of *foo* in Rust using these values:

```
fn foo(a: (i16, bool)) -> char {
    let bar_a: u8 = 76;
    let bar_b: bool = true;
    let bar_c: String = String::from("hello");
    let bar_result = bar(bar_a, bar_b, &bar_c);

    let mut data: Vec<Datum> = Vec::new();

    data.push(Datum::Int(a.0 as i64));
    data.push(Datum::Uint(a.1 as u64));
    match bar_result {
        None => data.push(Datum::Uint(false as u64)),
        Some(v) => {
            data.push(Datum::Uint(true as u64));
            data.push(Datum::Int(v as i64));
        }
    }

    let hash = hash::crc::checksum::crc(&data);
    println!("{hash}");
    'q'
}
```

Here is an implementation of *foo* in Go using the same values:

```
W.Exports.Foo = func(a cm.Tuple[int16, bool]) (result rune) {
    w_bar_a := uint8(76)
    w_bar_b := bool(true)
    w_bar_c := string("hello")
    w_bar_ret := W.Bar(w_bar_a, w_bar_b, w_bar_c)

    var data []checksum.Datum = []checksum.Datum{}

    data = append(data, checksum.DatumInt(int64(a.F0)))
    data = append(data, checksum.DatumUint(boolToUint(bool(a.F1))))
```

```
        data = append(data,
                      checksum.DatumUint(boolToUint(!w_bar_ret.None())))
        if ptr := w_bar_ret.Some(); ptr != nil {
            val := *ptr
            data = append(data, checksum.DatumInt(int64(val)))
        }
        println(checksum.Crc(cm.ToList(data)))
        result = rune('q')
    }
```

The two functions above, although different syntactically, should have the same output as long as *checksum* has the same behavior in each language. We guarantee this by implementing a single Wasm component that exports an implementation of the CRC-32 checksum algorithm. As mentioned in Section 4.1, every world produced by Atlas imports a function named *checksum*. This allows us to share a single implementation of the CRC-32 algorithm across all components. An alternative solution would be to implement a checksum helper function in each target language. Guaranteeing the equivalence of each helper function would be difficult because the CRC-32 algorithm operates over bytes, and different target languages may encode the same data values differently. Instead, we use a WIT binding generator to produce binding code that converts values from different target languages to the same encoding.

The only input to our source program generator is a world definition and a list of target languages. This means that all identifiers, such as type, function, and namespace names, must be generated solely using information in the WIT test case. These identifiers are chosen by the WIT binding generator for each target language. Therefore, our source program generator must be able to predict all identifiers produced by a WIT binding generator. Alternatively, a source program generator could parse and use symbols from generated binding code. Although we did not face issues with unpredictable binding code generation, there are cases in which some identifiers depend on the order in which WIT definitions appear in the test case. For example, if an empty result type is defined in multiple namespaces in a WIT file, wit-bindgen will only generate one C type definition, selecting one of the namespaces based on the order in which they are processed. Rather than emulate this and other similar behaviors, we accept that our source program generator will sometimes produce programs that are not correct implementations of binding code produced by a WIT binding generator, due to identifier mismatches. However, these cases were relatively uncommon in our testing campaigns and are easy to detect, because compilation of the source program and binding code fails. We leave achieving perfect source program generation for future work.

4.3 Test Harness

We created a testing harness to automate the steps of our differential testing framework. The test harness maintains a log of evaluated test cases. This allows us to run and interpret the results of extended testing campaigns.

We implemented our harness as a Python script. The harness uses Python's subprocess library to execute shell commands that drive the testing pipeline. A JSON file is used to configure the parameters of a testing campaign, such as target languages, paths to command line tools, and debug options.

Our test harness divides operations into six stages: WIT generation (wit-gen), WebAssembly generation (wasm-gen), source file generation (source-gen), component generation (component-gen), component composition (composition), and execution (exe). In wit-gen, a WIT test case is generated by Atlas. In wasm-gen, the WIT test case is compiled to Wasm bytecode. In source-gen, semantically equivalent programs in each target language are generated by our source program generator. During component-gen, components are created from the generated source programs and bindings produced by the WIT binding generators being tested. In the composition stage, components of each target language are composed using WAC. Finally, each component is executed in the exe stage.

Figure 4.4 illustrates our complete testing pipeline, organized by the stages of our test harness. It omits details about the creation of equivalent executable components from source programs, described in Section 4.2.

During each stage, relevant information for the reproduction and evaluation of the current test case is added to a JSON-formatted log. For example, after a test case is generated in wit-gen, the seed of the test case is added to the log. Similarly, during source-gen, the three PRNG seeds passed as command line parameters to the source program generator are logged. Specifically, one PRNG seed is logged per invocation of the source program generator, and the source program generator is invoked three times per test case (once each for worlds e, w, and i). In addition, the standard outputs and standard errors of all spawned processes are recorded in the log, grouped by stage, to aid debugging. If there is a failure during any stage or the component outputs differ in exe, the test case is logged in a directory identified by the current stage and the name of the testing campaign. Grouping logged test cases this way allows us to distinguish different kinds of bugs. For example, if a WIT binding generator produces binding code that fails to compile, the test case will be logged during the component-gen stage. In contrast, if generated binding code contains a semantic defect that affects the output of an executed component, the test case will be logged in the exe stage.

4.4 Test Case Validation and Reduction

Section 4.3 describes how our test harness can be used to run fuzzing campaigns. This section describes how our testing harness can be used to validate and reduce a logged test case. Validating a test case means repeating a test case described in the log and verifying that its output matches what was recorded in the log. Validating a test case is the first step in determining if it triggers a bug in one of the systems under test (SUT). Reducing a test case means simplifying a test case (such as deleting lines of a file) while preserving its ability to trigger a bug in a SUT.

When verifying a test case, the harness uses the seeds for WIT and source file generation recorded in the log file. It also allows the outputs of all spawned processes to go to standard output instead of capturing and logging them. This allows a user to confirm that all stages of the testing pipeline behave as expected and that the output of a test case matches the log.

Randomly generated test cases are often large and opaque. Therefore, it is usually necessary to simplify a test case before including it in a bug report. Fuzzing tools are often paired with automated program reducers, which reduce the effort required to simplify test cases. A notable example is C-Reduce, a C and C++ program reducer [14]. Program reducers typically operate by making a random deletion from a source file and running a predicate to check if a property of interest has been preserved, undoing the deletion if not.

In our case, several files in a test case may require simplification before a bug report can be submitted. First, the WIT test case often contains definitions that are unrelated to the triggered bug. Next, each source program may be unnecessarily complex. For example, a list may be initialized with 50 elements when a single element is sufficient for triggering an observed bug. Finally, it is usually possible to trigger a bug using one or two components, yet each test case produced by our framework is a composition of three components.

At the time of writing, no automated program reducers for WIT exist. Our framework also places unique constraints on what reductions are valid. For example, all the worlds in the test case must be compatible and yield an executable component when implemented and composed. Therefore, changes to one world definition are invalid unless the same change is made to other world definitions.

Instead of implementing a test case reducer that takes this property into account, we reduce WIT files manually. Specifically, we configure the harness to skip WIT generation and make valid simplifications to the WIT file. After each simplification, we run the harness to confirm that the incorrect behavior is still triggered. We repeat this process until no further simplifications can be made to the WIT test case.

Reducing the WIT test case also reduces the complexity of the generated source files. If source programs are further reduced, it is necessary to modify the source programs for all target languages in the same way to preserve their semantic equivalence. At this point, it is often possible to identify the location of the bug through manual inspection of the test case's output. To facilitate this, the harness can be configured to use a checksum function that sends all received values to standard output. The additional output makes it easier to trace corrupted values. When the bug has been identified, it can be moved into a self-contained test case that can be executed without the test harness. Such a test case can be included in a bug report along with a description of the incorrect behavior.

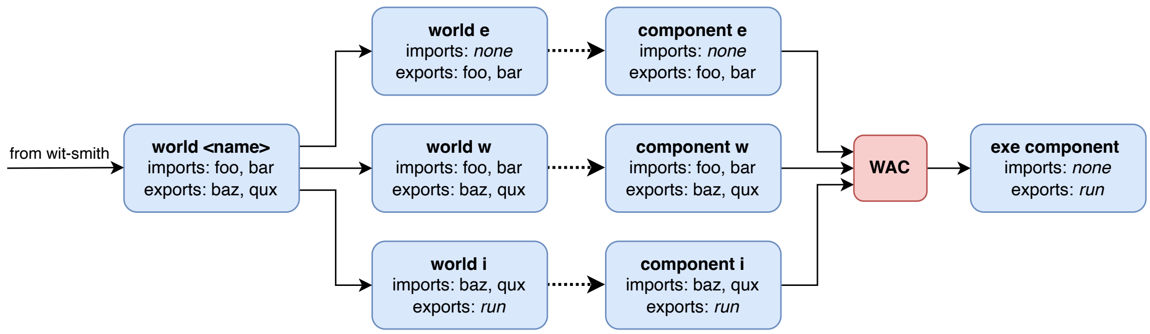

Figure 4.1. Process by which an arbitrary WIT interface is turned into an executable component

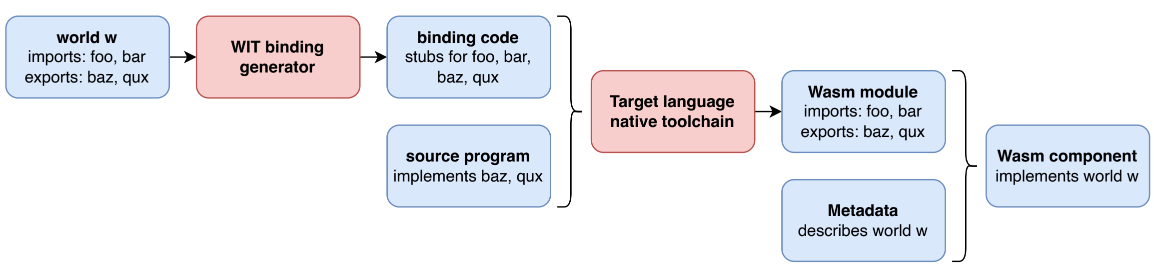

Figure 4.2. Pipeline to create a Wasm component targeting an arbitrary WIT world

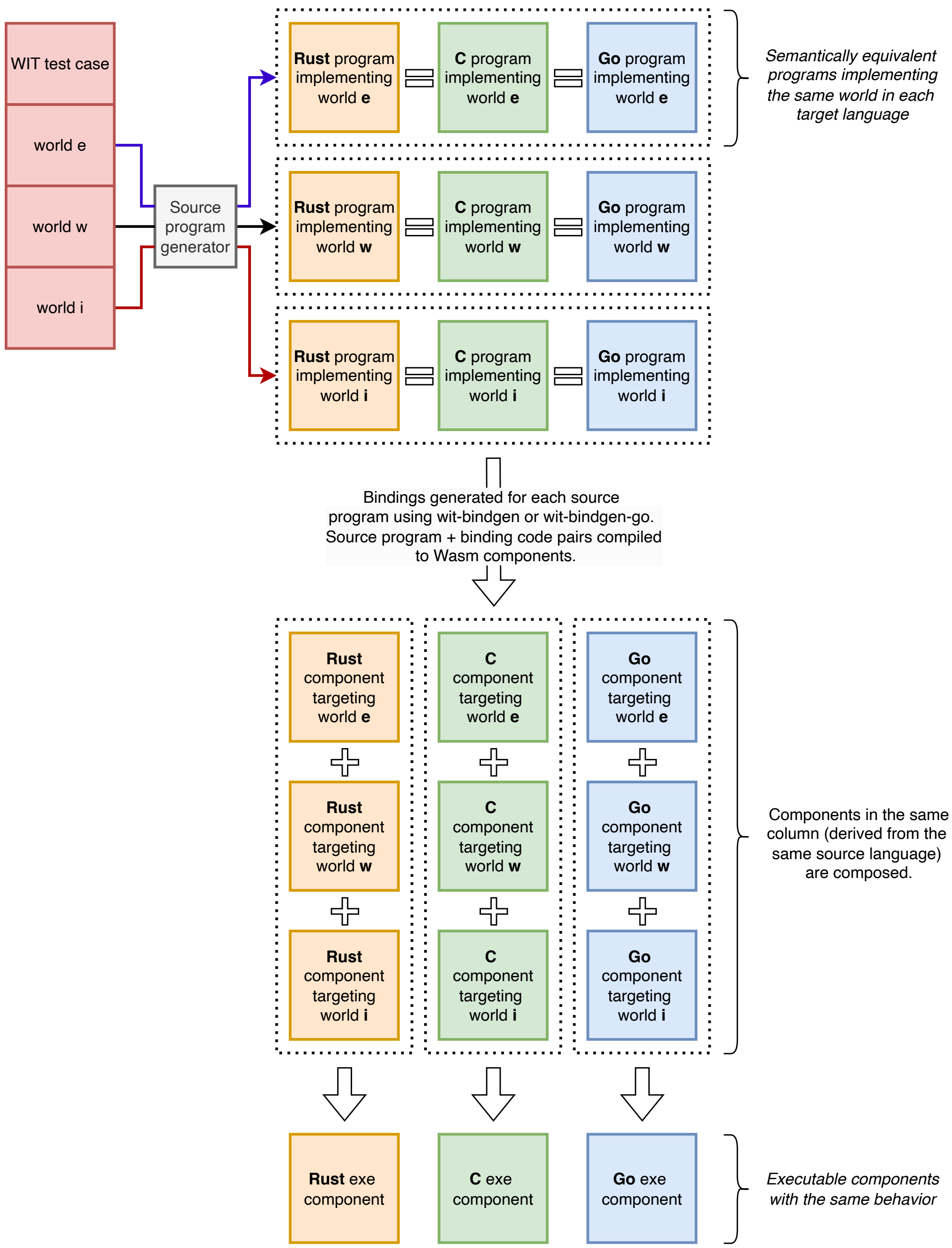

Figure 4.3. Process of creating semantically equivalent, executable components from source programs

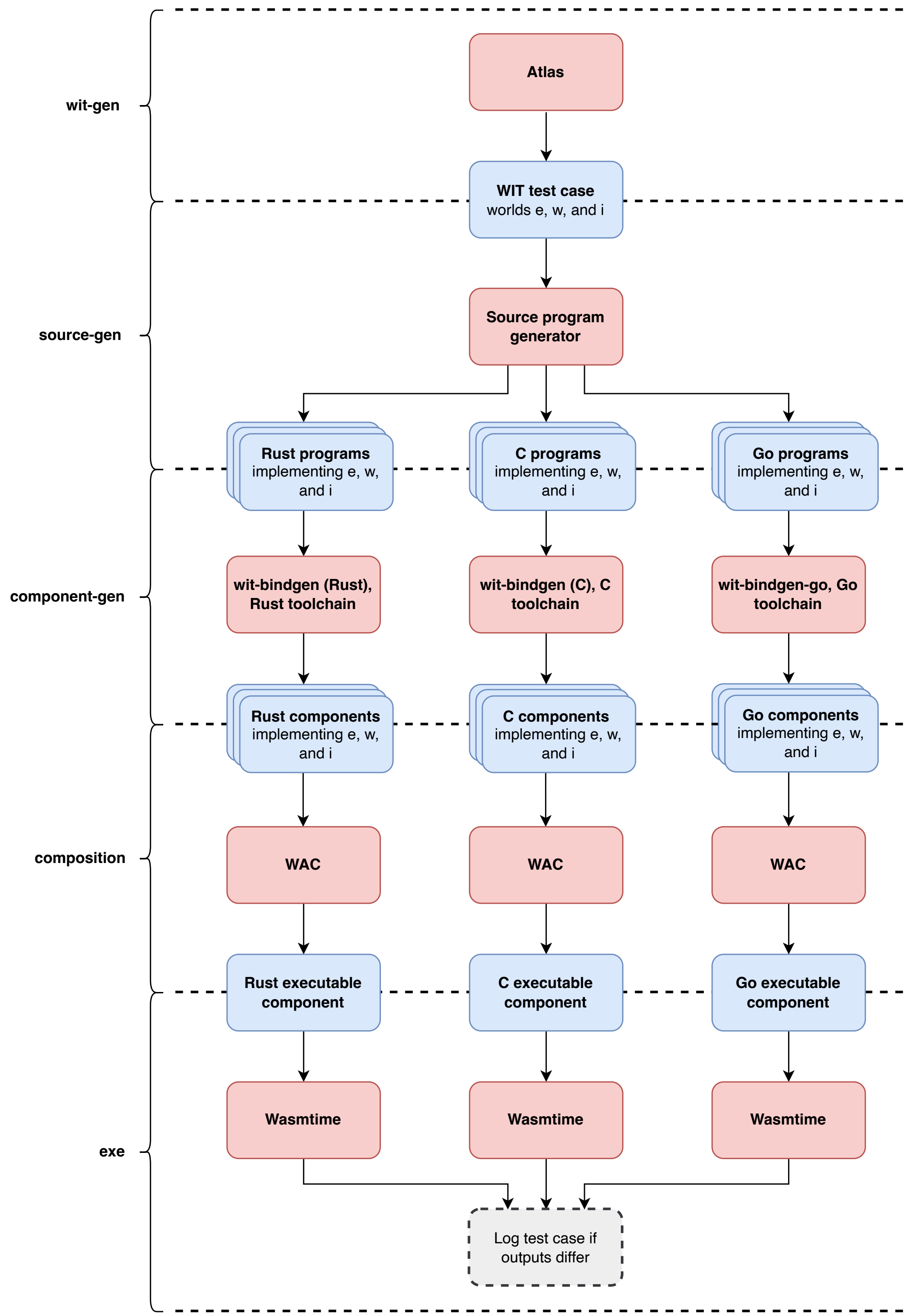

Figure 4.4. Overview of entire differential testing pipeline, grouped by harness stages

CHAPTER 5

EVALUATION

In this chapter, we evaluate the effectiveness of our differential testing framework for discovering bugs in WIT binding generators. Section 5.1 describes the setup of testing campaigns we ran using our framework. Section 5.2 describes the results of these campaigns. Section 5.3 draws conclusions about our framework's effectiveness and limitations.

5.1 Experimental Setup

We used CloudLab [9], a testbed for cloud computing research, to run large-scale testing campaigns. We allocated multiple machines on CloudLab and installed our testing framework on each one. The machines we allocated were Dell PowerEdge R430 servers running Ubuntu 22. Each machine had two 2.4 GHz 64-bit 8-Core Xeon E5-2630v3 processors, 64 GB of RAM, and a 200 GB 6 Gbps SSD. We did not implement any methods of synchronization between the nodes, noting that redundant evaluations of test cases are rare because each test case is parameterized by four 64-bit seeds.

In our campaigns, we attempted to detect bugs in two WIT binding generators: wit-bindgen [5] and wit-bindgen-go [6]. As described in Chapter 2, wit-bindgen is a collection of binding generators for various programming languages. We tested the Rust and C binding generators of wit-bindgen (versions 0.36.0 and 0.37.0). Wit-bindgen-go is a WIT binding generator for the Go programming language that is separate from wit-bindgen. We tested wit-bindgen-go version 0.5.0.

We ran three campaigns to evaluate the effectiveness of our random testing framework. Campaign 1 consisted of periodic executions of our framework during its development to verify that it was functioning as intended. It primarily tested wit-bindgen 0.36.0 and wit-bindgen-go 0.5.0. Campaign 2 was an extended campaign carried out on CloudLab. It tested wit-bindgen 0.37.0 and wit-bindgen-go 0.5.0. Campaign 3 was also carried out on CloudLab and tested wit-bindgen 0.37.0. In Campaign 2, our framework filtered out test cases containing a specific WIT type to avoid repeatedly triggering a bug in wit-bindgen-go discovered in Campaign 1. Campaign 3 allowed the type definition that was filtered out during Campaign 2, because wit-bindgen-go was not a target of Campaign 3.

Campaigns 2 and 3 were executed using five machines for seven days. This gives a total of 1,680 hours of computation time. This is an upper bound because testing was temporarily halted during Campaign 3 due to the machines running out of disk space. This was caused by not configuring a maximum cache size for Wasmtime. The issue was fixed and testing was resumed after several hours of downtime.

5.2 Results

To categorize the bugs we discovered using our testing framework, we define two kinds of defects in WIT binding generators: "code-generation bugs" and "latent code-generation bugs." A code-generation bug is any defect in a WIT binding generator that results in the generation of incorrect binding code. Latent code-generation bugs are a subcategory of code-generation bugs where the generated bindings behave incorrectly without raising an exception during component compilation, linking, or execution. We are especially interested in finding latent code-generation bugs. Detecting latent code-generation bugs requires a mechanism to distinguish correct from incorrect (non-crashing) outputs, such as differential testing.

Through our testing campaigns, we discovered four code-generation bugs in WIT binding generators. Three of the bugs we discovered are also latent code-generation bugs. At the time of writing, all four bugs have been confirmed by developers, and three have been fixed. Table 5.1 provides an overview of the status of each bug as well as citations of the submitted reports.

The first bug was discovered in the Rust generator of wit-bindgen 0.36.0 during Campaign 1. We did not have to filter out test cases that trigger this bug in Campaign 3, because a fix for this bug was released in wit-bindgen 0.37.0. Wit-bindgen produced binding code that incorrectly encoded a Rust data value into its representation in the Canonical ABI. This manifested as a corrupted function argument between two components. Below is the WIT definition of the corrupted data value's type:

```
list<tuple<s8, s64, s8>>
```

The cause of the bug was determined to be an unaccounted difference between the encoding of tuples in Rust and the Canonical ABI [18]. Notably, this bug can only be triggered by initializing a list of tuples with fields similar to the ones shown above. We found this bug through differential testing by comparing the outputs of components created using the Rust and C generators of wit-bindgen. Both target components executed successfully, but the mistranslated function argument caused one of the hash values printed by the Rust component to differ from the correct hash printed by the C component. The difference was flagged by our harness and the test case was logged. This highlights the utility of differential testing for detecting latent code-generation bugs. The binding code generated by the Rust target did not raise any compile- or run-time errors, but we were still able to detect incorrect behavior.

The second bug was found in the Rust generator of wit-bindgen 0.37.0 during Campaign 3. Similar to the first bug, an argument to a function call between components was corrupted. The cause of this bug was a name collision caused by the poor naming of internal variables in generated code that processes fields of a record type. The name of a generated internal variable collided with the name of a parameter of a generated function, causing the parameter value to be overwritten [19]. This bug was also discovered by comparing the outputs of components created using wit-bindgen for Rust and wit-bindgen for C. The corrupted function argument value resulted in a difference in the printed hash values between the C and Rust targets. Like the previous bug, this bug manifested as an incorrect computation, rather than a code-generation-time, compile-time, or run-time error. Thus, differential testing was useful to detect it.

The third bug was found in wit-bindgen-go during Campaign 1. This bug was triggered when initializing a value with the following WIT type:

```
result<bool, s8>
```

The result type in WIT is an alias for a variant type with two cases: *OK* and *Err*. In the type above, the *OK* case is associated with a boolean value, while the error case is associated with a signed 8-bit integer. Because a variant value can only have one associated value at a time, binding code produced by wit-bindgen-go stores the associated values of each case at the same memory location. To achieve this, a placeholder value is allocated in memory that is large enough to contain any of the possible associated values. Since both associated types in the example above (*bool* and *s8*) occupy one byte in Go, a boolean value was arbitrarily chosen as the placeholder value allocated in memory. However, this caused the Go compiler to treat the associated value of the *Err* case (an 8-bit signed integer) as a boolean value [16]. This caused the wrong value to be computed for the integer case of the result type. This bug was found because the component created using wit-bindgen-go printed a different hash value than the other targets.

The fourth bug was also found in wit-bindgen-go but was detected during Campaign 2. It was triggered when the return value of a function was the following WIT type:

```
variant var {
  a(list<u8>),
  b(u64),
}
```

The underlying cause was an incorrect setting of the ABI alignment of list types [17]. This issue manifested when a list was the associated type of a variant case. Initially, our testing framework detected this bug as a latent code-generation bug: i.e., the hash values printed by the Go component were incorrect. Reducing the test case caused the Go component to crash upon executing an unreachable instruction. Therefore, differential testing was not strictly necessary to detect this bug as other test cases can reproduce it such that a runtime error is raised. However, this bug cannot be triggered without implementing and composing multiple components. Thus, the component-creation strategy of our framework was useful to detect this bug.

5.3 Discussion

In this section, we discuss our random testing framework's effectiveness for discovering bugs in WIT binding generators as well as its limitations.

5.3.1 Effectiveness of the Random Testing Framework

Overall, we found our random testing framework to be effective for finding bugs in wit-bindgen and wit-bindgen-go. Particularly, our harness made it straightforward to detect and reduce code-generation bugs in the WIT binding generators we tested. We expect that the bugs found by our framework would have been difficult to detect using other methods of software testing. Most of them were only triggered by initializing specific, nested WIT types to particular values. In addition, most of the bugs we discovered could only be triggered by a function call between components, meaning it was necessary to generate and compose multiple Wasm components to find them.

While we focused on evaluating test cases that trigger code-generation bugs in WIT binding generators, we believe that our framework could be effective for finding other kinds of bugs in the component creation pipeline. To evaluate this claim, it would be helpful to optimize other stages of our testing pipeline to reduce the number of "false positives." For example, as described in Section 4.2, our source program generator occasionally produces source programs that fail to compile due to identifier mismatches with generated bindings. These test cases are logged in the component-gen stage of our testing pipeline. This makes it more difficult to identify test cases logged in this stage that actually trigger a bug in a WIT binding generator. Minimizing the number of false positives would reduce the effort necessary to validate and reduce these test cases.

5.3.2 Limitations

We identify four major limitations of our random testing framework. Some of these limitations could potentially be addressed in future work.

First, it can only test WIT worlds that have at least one exported function and one imported function. This reduces the generality of our framework. For example, there may be defects in the component creation pipeline that occur when a component has no exports or no imports.

Second, our random testing framework only composes components that were compiled from the same source language. This means that, currently, our framework cannot detect defects in binding code that cannot be triggered when a calling component uses the same binding code as a called component. For example, binding code may encode a data value incorrectly according to the Canonical ABI, yet correctly reconstruct the value because the lifting and lowering code use the same incorrect encoding. This category of bugs cannot be detected with our current testing methodology but is an obvious avenue for future investigation.

Third, the rate at which our framework evaluates test cases is slow. For example, a single node on CloudLab is able to evaluate approximately 500 test cases per hour when evaluating all three target languages—or about one test case every seven seconds. This is primarily because our framework has multiple compiler toolchains in the critical path of testing. For example, with three languages under test, a compiler toolchain must be invoked to compile nine source programs (three Rust, three C, and three Go programs).

Fourth, our framework filters out test cases containing certain WIT constructs. As described in Section 4.1, our framework filters out WIT test cases containing the *use* keyword. This keyword allows a type definition defined in an interface to be reused by a different world or interface. If a world contains a *use* statement, the interface it references is implicitly imported by the world. This is problematic when a generated world, w, defines one of its exports using a reused type. Doing so forces i to also reuse the type definition (to define its imports) and therefore have an extra dependency that is not exported by w. As a result, the final composed component will have an unsatisfied dependency. Therefore, the *use* keyword is currently incompatible with our testing pipeline. Our framework also filters out test cases containing the *resource* keyword, which is a WIT type that represents a reference to an object that implements a specific interface and should not be copied by value. Future enhancements to our framework could allow WIT test cases containing these constructs to be evaluated.

Table 5.1. Summary of reported bugs

System	Defect Synopsis	Status	Ref.
wit-bindgen 0.36.0 (Rust)	Corrupted list-of-tuples argument	Fixed	[18]
wit-bindgen 0.37.0 (Rust)	Record field name collision	Confirmed	[19]
wit-bindgen-go 0.5.0	Type confusion of Result cases	Fixed	[16]
wit-bindgen-go 0.5.0	Variant with list case causes crash	Fixed	[17]

CHAPTER 6

CONCLUSION

The WebAssembly component model is a promising technology, and WIT binding generators are a key tool for the creation of WebAssembly components. Our goal was to evaluate the effectiveness of random testing for finding bugs in WIT binding generators. To do this, we implemented a differential testing framework for exercising these generators. With three testing campaigns, we discovered three latent code-generation bugs and one regular code-generation bug in WIT binding generators. We conclude that random testing can be effective for discovering bugs in WIT binding generators and that differential testing is a useful mechanism for detecting semantic bugs in WIT binding generators.

We identify several areas of future work for our random testing framework. First, one could implement additional source program generators to increase the number of languages tested by our framework. For example, wit-bindgen also supports generation of C#, Java, and MoonBit bindings. Other WIT binding generators can create component model bindings for Python and JavaScript [3, 4]. Incorporating these programming languages and binding generators into our framework would increase its bug-finding power.

A second area of future work is to add support for WIT types that are currently filtered out by our framework, such as resources and *use* statements. This would increase the variety of test cases produced by our framework, potentially increasing its effectiveness.

Another potential modification is to implement a strategy for composing components compiled from different source languages. This is feasible because component interfaces are agnostic to the source language from which they were compiled. Doing so would resolve the limitation discussed in Section 5.3.2.

Finally, one could develop an automated framework for reducing test cases. This would involve implementing program reducers for multiple languages, including WIT. It would also require a strategy to preserve the semantic equivalence of different programs during reduction. Doing so would greatly reduce the manual effort required to evaluate the findings of large-scale testing campaigns. This would be especially valuable if more target languages are added to the framework, increasing the volume and complexity of logged test cases.

REFERENCES

[1] The Bytecode Alliance. 2024. WAC. `https://github.com/bytecodealliance/wac`.

[2] The Bytecode Alliance. 2024. wit-smith. `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wit-smith`.

[3] The Bytecode Alliance. 2025. componentize-py. `https://github.com/bytecodealliance/componentize-py`.

[4] The Bytecode Alliance. 2025. ComponentizeJS. `https://github.com/bytecodealliance/ComponentizeJS`.

[5] The Bytecode Alliance. 2025. wit-bindgen. `https://github.com/bytecodealliance/wit-bindgen`.

[6] The Bytecode Alliance. 2025. wit-bindgen-go. `https://github.com/bytecodealliance/go-modules`.

[7] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and Preventing Bugs in JavaScript Bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*. 559–578. `https://doi.org/10.1109/SP.2017.68`

[8] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. *Proceedings 2021 Network and Distributed System Security Symposium*. `https://dx.doi.org/10.14722/ndss.2021.24224`

[9] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, Renton, WA, 1–14. `https://www.usenix.org/conference/atc19/presentation/duplyakin`

[10] WebAssembly Community Group. 2019. WebAssembly. `https://webassembly.org/`. Accessed: 2025-17-2.

[11] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. 2010. Jinn: Synthesizing Dynamic Bug Detectors for Foreign Language Interfaces. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association

for Computing Machinery, New York, NY, USA, 36–49. https://doi.org/10.1145/1806596.1806601

[12] Siliang Li and Gang Tan. 2014. Finding Reference-Counting Errors in Python/C Programs with Affine Analysis. In *European Conference on Object-Oriented Programming (ECOOP)*. 80–104. http://www.cse.psu.edu/~gxt29/papers/refcount.pdf

[13] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107. http://dblp.uni-trier.de/db/journals/dtj/dtj10.html#McKeeman98

[14] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

[15] Armin Rigo and Maciej Fijalkowski. 2024. CFFI documentation. https://cffi.readthedocs.io/en/stable/

[16] Ethan Stanley. 2025. Go-modules Issue 284. https://github.com/bytecodealliance/go-modules/issues/284.

[17] Ethan Stanley. 2025. Go-modules Issue 288. https://github.com/bytecodealliance/go-modules/issues/288.

[18] Ethan Stanley. 2025. Wit-bindgen Issue 1112. https://github.com/bytecodealliance/wit-bindgen/issues/1112.

[19] Ethan Stanley. 2025. Wit-bindgen Issue 1134. https://github.com/bytecodealliance/wit-bindgen/issues/1134.

[20] Gang Tan and Greg Morrisett. 2007. ILEA: Inter-Language Analysis across Java and C. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 39–56. https://doi.org/10.1145/1297027.1297031

[21] The Rust Team. 2024. https://github.com/rust-lang/rust-bindgen/blob/main/csmith-fuzzing/README.md

[22] W3C WebAssembly Community Group. 2024. Canonical ABI. https://github.com/WebAssembly/component-model/blob/main/design/mvp/CanonicalABI.md.

[23] W3C WebAssembly Community Group. 2024. WIT. https://github.com/WebAssembly/component-model/blob/main/design/mvp/WIT.md.

[24] W3C WebAssembly Community Group. 2025. Component Model. https://github.com/WebAssembly/component-model.

[25] Jiashui Wang, Peng Qian, Xilin Huang, Xinlei Ying, Yan Chen, Shouling Ji, Jianhai

Chen, Jundong Xie, and Long Liu. 2024. Tacoma: Enhanced Browser Fuzzing with Fine-Grained Semantic Alignment. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1174–1185. https://doi.org/10.1145/3650212.3680351

[26] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[27] Chijin Zhou, Quan Zhang, Lihua Guo, Mingzhe Wang, Yu Jiang, Qing Liao, Zhiyong Wu, Shanshan Li, and Bin Gu. 2023. Towards Better Semantics Exploration for Browser Fuzzing. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2, Article 243 (Oct. 2023), 28 pages. https://doi.org/10.1145/3622819

[28] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. 2022. Minerva: Browser API Fuzzing with Dynamic Mod-ref Analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1135–1147. https://doi.org/10.1145/3540250.3549107