Finding Bugs in WebAssembly Interface Type Binding Generators

Ethan Stanley
University of Utah
Salt Lake City, UT, USA
ethan.stanley@utah.edu

Eric Eide
University of Utah
Salt Lake City, UT, USA
eeide@cs.utah.edu

***Abstract*—The WebAssembly Component Model is an emerging standard for assembling applications from parts that are implemented in WebAssembly. Unlike ordinary WebAssembly modules, components implement their interfaces in a standardized way, enabling the interoperation of software parts compiled to WebAssembly from different programming languages. A component essentially wraps an ordinary module with *binding code*, created by a *WebAssembly Interface Type (WIT) binding generator*, to adapt the module to the WebAssembly component standard. Errors in the generation of binding code can lead to hard-to-diagnose run-time errors, including crashes and silent data corruption, in applications built from WebAssembly components. Prior published work on WebAssembly testing has focused on finding bugs in WebAssembly compilers and runtimes, and has overlooked the potential for bugs in the generation of binding code. In this experience paper, we detail and evaluate our approach to addressing this oversight.**

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 written in different high-level languages. If the components' run-time behaviors differ, we expect that there is a bug in one of the generated bindings. Using our framework, we discovered seven previously unknown code-generation defects in wit-bindgen and wit-bindgen-go. We analyze these bugs in this paper and, in addition, share lessons learned that can guide future efforts to test binding generators.

***Index Terms*—components, fuzzing, WebAssembly, WIT**

I. Introduction

The *WebAssembly Component Model* [1] makes it possible to implement software systems as assemblies of individual but interoperable parts (i.e., components), even when those parts are compiled to WebAssembly from different source programming languages. For example, a WebAssembly component that was compiled from Rust source code can invoke functions and exchange data with a WebAssembly component that was compiled from Go. The WebAssembly Component Model enables this because it defines a *Canonical ABI* (Application Binary Interface) [2], which specifies how messages and data are exchanged between components. The properties of WebAssembly components—source-language independence, and sandboxed code execution via WebAssembly—make them increasingly popular for implementing systems that involve mobile and/or untrusted code. For example, the wasmCloud [3] and Spin [4] platforms allow users to upload applications as WebAssembly components, which are then executed in the cloud. NGINX Unit [5] is a web app server that supports applications implemented as WebAssembly components. Visual Studio Code [6] and the Zed editor [7] support extensions that are implemented as WebAssembly components.

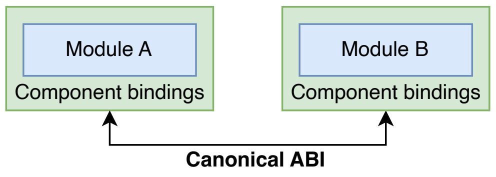

Fig. 1. Illustration of two WebAssembly components communicating via the Canonical ABI.

A WebAssembly component is essentially a wrapper around an ordinary WebAssembly module, as illustrated in Figure 1. The code of the ordinary module is written in a language such as C, Rust, or Go, and compiled to WebAssembly. The wrapper is specified in an interface definition language called *WIT* (WebAssembly Interface Type) [8]. The wrapper is an adapter [9] that translates between the Canonical ABI (at the outer boundary of the wrapper) and whatever conventions are implemented by the compiler that produced the WebAssembly module being wrapped (at the inner boundary of the wrapper). The code within the wrapper is called *binding code*, and it is created from a WIT specification by a *WIT binding generator*.

Previous work on testing WebAssembly implementations has aimed to find bugs in WebAssembly compilers (which produce native code), interpreters, and runtimes [10–16]. This past work *does not test the creation of binding code* by WIT binding generators, and therefore cannot discover bugs in these tools, even though binding generators are an essential part of the WebAssembly component ecosystem. It is important to thoroughly test WIT binding generators for three reasons. First, they support a growing number of component-implementation 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.

In this paper, we report our experience in using random testing to find code-generation 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 can 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. To produce random bindings, a random valid WIT file must be given as input to a WIT binding generator. We implemented a test case generator to produce such files. Our test case generator uses wit-smith [17], an existing random WIT file generator developed by the Bytecode Alliance, to obtain raw test cases before performing additional filtering and post-processing (§III-A).

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 needed a method of detecting these data-conversion bugs in generated binding code.

We achieved 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 [18]. In our case, our framework creates multiple, similar applications for each WIT file produced by our test case generator (§III-B). 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 needed 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 implemented a source program generator to produce random programs that fulfill these requirements (§III-B). To automate the steps of our testing pipeline and log any found defects, we implemented a test harness (§III-C). Our complete testing framework is available as open-source software [19].

Using our testing framework, we discovered seven unique code-generation defects across two WIT binding generators, called wit-bindgen [20] and wit-bindgen-go [21]. In addition, we triggered one bug in TinyGo [22], the compiler we used for Go-based components. In this paper, we describe the bugs we found (§IV-B). We reflect on our experience (§V) and conclude that random differential testing can be a useful method of discovering code-generation bugs in WIT binding generators (§V-A). Additionally, we draw lessons learned from our experience that can inform future testing efforts (§V-D).[1]

II. Background

WebAssembly (also known as Wasm) [24] 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* [1] 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

[1]The work presented in this paper is a major expansion of work presented in the MS thesis of the first author [23]. With respect to the thesis, this paper describes • an improved source program generator that (1) exercises binding code more thoroughly and (2) supports WIT resource types while (3) being easier to understand (§III-B); • all-pairs testing of Wasm components, including pairs implemented in different programming languages (§III-B); • an improved, AST-based test case reduction method (§III-D); • a greater number of code-generation bugs discovered in WIT binding generators, with two newly discovered bugs detailed (§IV-B); and • a bug discovered in TinyGo (§IV-B).

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 [2]. 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 knowing that, internally, their types are represented 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 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 C 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 interface and outputs type definitions and function stubs for the component's imported and exported functions. 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 metadata describing the component's interface. This step may be done automatically by toolchains that natively support compilation to Wasm components.

Wit-bindgen [20] and *wit-bindgen-go* [21] are two software tools that 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. 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. In this paper, we test C and Rust bindings produced by wit-bindgen and Go bindings produced by wit-bindgen-go.

As mentioned in Section I, WIT binding generators accept a description of a component model interface written in *WIT* [8]. 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 work, 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.

III. Methodology

In this section, we describe the differential testing framework that we created 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, Wasm components are created from 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.

A. 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 the generators' 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 [17]. 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 [25]. 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 *foo* that implements an arbitrary (and randomly generated) interface, we derive from it the following two worlds. First, we define a world named *driver* that has the same imports as *foo* but exports only the entry-point function *run*. Second, we define a

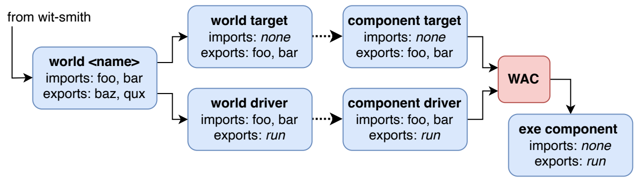

Fig. 2. Process by which an arbitrary WIT interface is turned into an executable component

world named *target* that exports the imports of *driver* and has no imports of its own: i.e., *target* satisfies the dependencies of *driver*. Now we can create a Wasm component that implements each world definition (*driver* and *target*). Composing these two components yields 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 [26]. Figure 2 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 at least one function. We call a world *testable* if it satisfies this criterion. If a world is not testable, we cannot compose implementations of *driver* and *target* because the components have no functions in common.

To implement the process described above, we developed *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 testable worlds or uses unsupported features of WIT, return to step 1.
3) Select a testable world, *w*, from *t* at random.
4) Remove all world definitions from *t*, except *w*.
5) Modify the type definitions in *w*, if necessary.
6) Replace *w* with definitions of *driver* and *target*.
7) Modify each world in *t* to import a checksum function.
8) Insert the seed used to generate *t* in a comment at the top of *t*.
9) Output *t*.

At steps 1 and 2, Atlas repeatedly invokes wit-smith until it produces a test case containing a testable 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.[2] For example, the *use* keyword in WIT facilitates definition reuse, but alters the dependencies of a world in a way that conflicts with our strategy for creating executable components. Thus, we filter out test cases containing the *use* keyword.

[2]Our framework supports all the primitive and user-defined data types in the current version of the WebAssembly Component Model, known as “WASI Preview 2” [27].

At steps 3 and 4, Atlas selects a testable world from the test case at random and deletes all other world definitions.

At step 5, Atlas makes any necessary corrections to the type definitions in *w*. For example, a world may import or export *resource types*, which can be thought of as remote object references [28]. Atlas ensures that all resource type definitions contain valid constructor and identifier methods, so that they can be exercised in subsequent stages of our framework.

At step 6, Atlas replaces the definition of *w* with definitions of *driver* and *target*, as described earlier in this section. At step 7, 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 III-B.

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 III-D.

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, *name*.

```
package name:name1;

world name {
  import name: interface {
    resource xx5 {
      constructor(xz: s16, name2: u32);
    }
  }
  import x: func(l: char) -> string;
}
```

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

```
// seed: 5932601428561635873
package name:name1;

world driver {
  import hash:crc/checksum;
  import name: interface {
    resource xx5 {
      constructor(xz: s16, name2: u32);
      whoami: func() -> u32;
    }
  }
  import x: func(l: char) -> string;

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

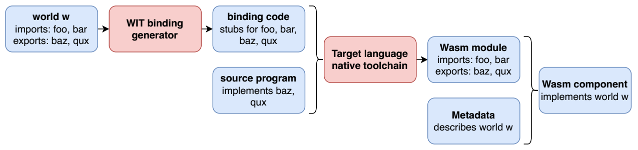

Fig. 3. Pipeline to create a Wasm component targeting an arbitrary WIT world

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

  export name: interface {
    resource xx5 {
      constructor(xz: s16, name2: u32);
      whoami: func() -> u32;
    }
  }
  export x: func(l: char) -> string;
}
... hash:crc/checksum and wasi:cli/run defs ...
```

The world *name* was replaced with definitions for *driver* and *target*. In addition, the identifier function *whoami* was added to the definition of resource *xx5*. 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.

B. Source Program Generation

As discussed in Section II, 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 3 depicts the 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 in several steps. First, our framework invokes the source program generator two times—once for each world defined in a test case produced by Atlas (*driver* and *target*). Doing so yields **six** source programs: implementations of *driver* in C, Rust, and Go, and implementations of *target* in C, Rust, and Go. Next, our framework uses the process depicted in Figure 3 to create a component from each of the six source programs. Specifically, our framework generates binding

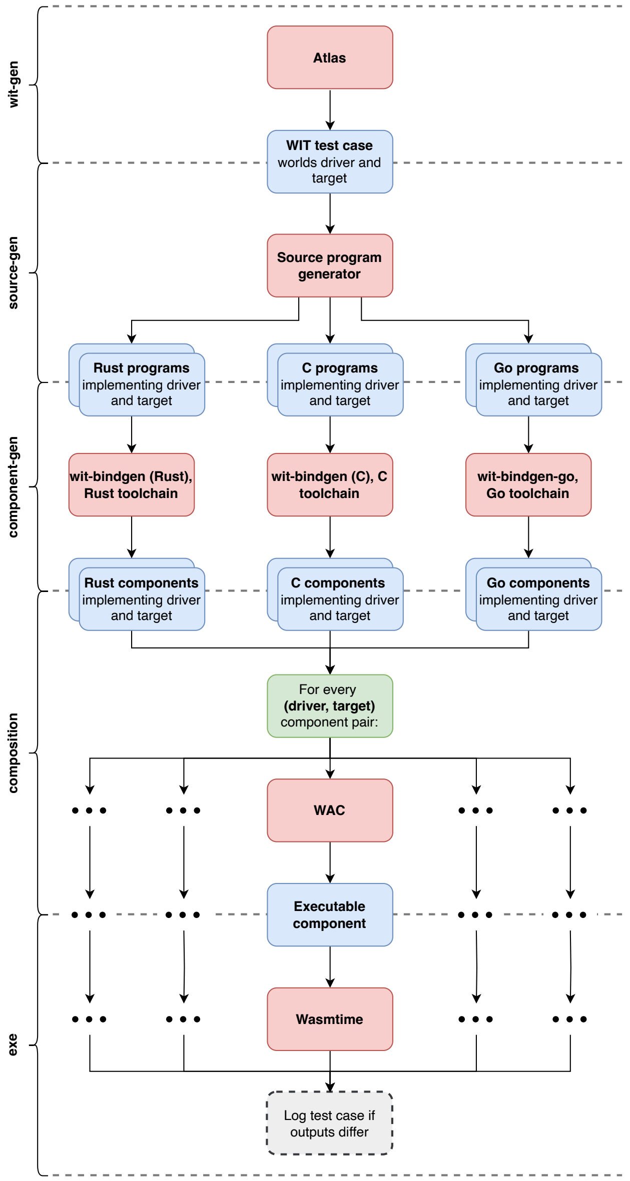
wit-gen
source-gen
component-gen
composition
exe
Atlas
WIT test case
worlds driver and target
Source program generator
Rust programs
implementing driver and target
C programs
implementing driver and target
Go programs
implementing driver and target
wit-bindgen (Rust), Rust toolchain
wit-bindgen (C), C toolchain
wit-bindgen-go, Go toolchain
Rust components
implementing driver and target
C components
implementing driver and target
Go components
implementing driver and target
For every
(driver, target)
component pair:
WAC
Executable component
Wasmtime
Log test case if outputs differ

Fig. 4. Overview of entire differential testing pipeline, grouped by harness stages

code for each source program using a WIT binding generator and compiles the resulting binding-code/source-program pairs into Wasm components. This yields six components: three implementations of *driver* and three implementations of *target*. Finally, our framework composes together all possible pairs of components implementing *driver* and *target*. Composing an implementation of *driver* with an implementation of *target* yields an executable component. Therefore, the previous step yields **nine** executable components. This strategy of creating executable components using our source program generator is visualized in the *source-gen*, *component-gen*, and *composition* sections of Figure 4.

Our framework performs differential testing by running each of the executable components 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. Below is the algorithm implemented by each exported function *f* in a generated source program:

1) Call each imported function *N* times.[3]
2) Compute a CRC-32 checksum using (1) the values of *f*'s arguments and (2) the return values of the function calls performed in step 1.
3) Write the CRC-32 value to standard output.
4) Return a (randomly chosen) value from *f*.

Our source program generator constructs an AST containing all the information needed to create a program whose functions have the structure above. For each exported function, the AST records which imported functions are called, the values of function arguments, and a return value. (Our framework can also implement and exercise methods for WIT resource types [28], but we omit discussion of this due to space.) Two functions that implement the algorithm above using the same arguments and return values are semantically equivalent: i.e, they are expected to print the same CRC value and return equivalent values. Thus, our source program generator creates semantically equivalent programs in different target languages by rendering programs from the same AST.

When our source program generator is invoked and provided with a world definition, it initializes a PRNG using a seed passed as a command line parameter. The PRNG is used to construct an AST, describing the implementations of the world's exported functions, by randomly generating values for function arguments and return values. Values in the AST are sampled from the set of values that are valid for a given type, including boundary values (such as the minimum or maximum values of a 32-bit integer) with a configurable probability. Then, the AST is used to render a source program 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 are implementations of the same AST. 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 all pairs of compatible components, as shown in the composition section of Figure 4, yields nine executable components that we expect to behave equivalently. This enables our framework to perform differential testing by executing each component and comparing their outputs.

[3]*N* is a command line parameter of the source program generator.

The algorithm implemented by every exported function (shown above) is simple, yet it allows us to detect semantic defects in binding code. 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 triggering 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 constructs an AST describing the implementation of *foo*. Assume the AST specifies that *foo* should call *bar* one time with 76, true, and "hello" as arguments and return 'q'. Below is an implementation of *foo* in Rust according to the AST:

```
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 the equivalent function implemented in Go:

```
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(asUint(bool(a.F1))))
  data = append(data,
     checksum.DatumUint(asUint(!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 III-A, 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 the components that we test.

The primary inputs to our source program generator are a WIT 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, and therefore, our source program generator must be able to predict all identifiers produced by a WIT binding generator. Although we did not face issues with unpredictable binding code generation, there are cases in which some identifiers depend on the order and location of WIT definitions in a test case. For example, under certain rare conditions, the C generator of wit-bindgen will represent multiple types defined in a WIT file with a single C type definition. We account for this only for *result* types, because they make up the majority of shared type definitions. Rather than implement this logic for all possible types, 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. These cases were very uncommon in our testing campaigns and are easy to detect, because compilation of the source program and binding code fails in a predictable way. We leave achieving perfect source program generation for future work.

C. Test Harness

We created a testing harness to automate the steps of our differential testing framework (Challenge 5). The test harness maintains a log of evaluated test cases, which 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 TOML 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, all pairs of components implementing *driver* and *target* are composed using WAC. Finally, each composed component is executed in the *exe* stage. Figure 4 illustrates our complete testing pipeline, organized by the stages of our test harness.

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 two 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 two times per test case (once for the world *driver* and once for the world *target*). 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.

D. Test Case Validation and Reduction

Section III-C 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 (Challenge 6). 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 (SUTs). 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 [29].

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 minimize the effort required to simplify test cases. A notable example is C-Reduce, a C and C++ program reducer [30]. 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 sometimes possible to trigger a bug in generated binding code without making a call to another component, yet each test case produced by our framework is a composition of two 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 corresponding changes are made to other world definitions.

Instead of implementing a test case reducer that takes this property into account, we reduce WIT files manually. We configure the harness to skip WIT generation while we make valid simplifications to the WIT file by hand. 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. Our source program generator facilitates this by providing an option to serialize and deserialize the AST used to generate source programs to JSON. This allows a user to make simplifications to the serialized AST and have them automatically reflected in all generated source programs.

When the WIT test case and generated source programs are sufficiently reduced, it is often possible to identify the location of the bug through manual inspection of the test case's output. To assist 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.

IV. Evaluation

In this section, we evaluate the effectiveness of our differential testing framework for discovering code-generation bugs in WIT binding generators.

A. Experimental Setup

We used CloudLab [31], a testbed for cloud computing research, to run large-scale testing campaigns. We allocated multiple machines within CloudLab and installed our testing framework on each one. We used Dell PowerEdge R430 (d430) and Dell PowerEdge C6620 (c6620) servers running Ubuntu 22 in our campaigns. Each d430 machine had two 2.4 GHz 64-bit 8-Core Xeon E5-2630v3 processors, 64 GB of RAM, and a 200 GB SSD. Each c6620 machine had one 2.1 GHz 64-bit 28-Core Xeon Gold 5512U processor, 128 GB of RAM, and an 800 GB NVMe 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 three 64-bit seeds.

In our campaigns, we attempted to detect bugs in two WIT binding generators: wit-bindgen [20] and wit-bindgen-go [21]. As described in Section II, 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, 0.37.0, and 0.41.0). Wit-bindgen-go is a WIT binding generator for Go that is separate from wit-bindgen. We tested wit-bindgen-go versions 0.5.0 and 0.6.2.

We ran four 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. Campaign 4 was another extended campaign on CloudLab, testing wit-bindgen 0.41.0 and wit-bindgen-go 0.6.2.[4]

Campaigns 2 and 3 were each executed using five d430 machines for seven days. Campaign 4 was carried out on 10 c6620 machines for 10 days. This gives a total of 4,080 node-hours of compute 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.

B. 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 method to distinguish correct from incorrect (non-crashing) outputs, such as differential testing.

Through our testing campaigns, we discovered seven code-generation bugs in WIT binding generators, summarized in Table I. All were previously unknown. Five of the bugs we discovered are also *latent* code-generation bugs, as indicated in the table. All seven bugs have been confirmed by developers,

[4]The supporting software in our experiments included wit-smith 0.221.2 (Campaigns 1–3) and 0.222.1 (Campaign 4); Wasmtime 28.0.x (Campaigns 1–3) and 32.0.0 (Campaign 4); WAC 0.6.0 (all campaigns); rustc 1.83.0 (all campaigns); and TinyGo 0.35.0 (Campaigns 1–2) and 0.37.0 (Campaign 4).

TABLE I
SUMMARY OF REPORTED WIT BINDING GENERATOR BUGS

System	Defect Synopsis	Status
wit-bindgen 0.36.0 (Rust)	Corrupted list-of-tuples argument (*latent*, ●1) [32]	Fixed
wit-bindgen 0.37.0 (Rust)	Record field name collision (*latent*, ●2) [33]	Confirmed
wit-bindgen-go 0.5.0	Boolean and integer type confusion (*latent*) [34]	Fixed
wit-bindgen-go 0.5.0	Variant with list case causes crash [35]	Fixed
wit-bindgen-go 0.6.2	Type confusion with boolean discriminant (*latent*) [36]	Fixed
wit-bindgen-go 0.6.2	Monotypic tuple used as placeholder (*latent*, ●3) [37]	Fixed
wit-bindgen-go 0.6.2	Err type larger than placeholder (●4) [38]	Fixed

and at the time of writing, six have been fixed. Below, we describe the four bugs marked with "●."

The first bug (●1) [32] was discovered in the Rust generator of wit-bindgen 0.36.0 during Campaign 1. 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 an unaccounted-for difference between the encoding of tuples in Rust and the Canonical ABI. 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 mishandled 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 (●2) [33] 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. Like the first bug (●1), this bug was discovered by comparing the outputs of components created using wit-bindgen for Rust and wit-bindgen for C. The corrupted function argument resulted in a difference between the hash values printed by the C and Rust targets. Also 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 (●3) [37] was found in wit-bindgen-go 0.6.2 during Campaign 4. This bug was triggered when initializing a value with the following WIT type:

```
result<tuple<bool, bool, bool>, s16>
```

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 tuple value, while the error case is associated with a signed 16-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 either of the associated values. The placeholder value is either the largest associated type (the tuple in this example) or an explicit placeholder type if the largest associated type cannot be used (such as if it has a sparse encoding). The cause of this bug was an assumption that monotypic tuples (tuples where all elements are the same type) can be used as placeholder values, since they have packed encodings. However, the LLVM IR to which this program was compiled ignored the seven most-significant bits of each byte in the associated value. This caused the integer value of the *Err* case to be silently corrupted. This bug was found because the component created using wit-bindgen-go printed a different hash value than the other targets.

The fourth bug (●4) [38] was also found in wit-bindgen-go 0.6.2 during Campaign 4. It was triggered when initializing a value of the following type:

```
result<
  option<tuple<string, string, string>>,
  tuple<u16, result<u64>, u8>>
```

The type definition generated by wit-bindgen-go to represent this type failed validation. Specifically, the associated type of the *Err* case exceeded the size of the chosen placeholder type. As a result, a function that validates the signatures of all initialized result values raised an exception. This test case was described as uncovering a "subtle bug" in how certain type sizes were calculated. Although differential testing is not needed to detect this bug because it raises an exception, it is still necessary to generate, implement, and execute binding code for a complex WIT definition.

Our framework also triggered a bug in TinyGo 0.37.0 [22], the tool our framework uses to compile Go source programs to Wasm. The bug [39], related to the allocation of memory in external components, was known about in theory to the developers of wit-bindgen-go, but did not yet have a reliable reproducer. This reflects the value of random testing for triggering bugs that require complex test cases. This bug has been confirmed but is not yet fixed.

V. Discussion and Insights

In this section, we discuss our random testing framework's effectiveness for discovering bugs in WIT binding generators, the kinds of bugs found by our framework, and lessons learned from our work that can be transferred to other testing efforts.

A. *Effectiveness of the Random Testing Framework*

Overall, we found our random testing framework to be effective for discovering 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.

B. *Finding Bugs Early in the Pipeline*

While we focused on discovering 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 improve the non-*exe* stages of our testing pipeline to eliminate known causes of certain "false positives." For example, as described in Section III-B, 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, potentially making it more difficult to identify test cases logged in this stage that actually trigger a bug in a WIT binding generator.

C. *Bugs Not Found*

We can describe a bug in a WIT binding generator as *self-consistent* if it results in the creation of a Wasm component that exhibits incorrect behavior only when composed with a component compiled from a different source language. For example, a WIT binding generator might produce binding code that serializes a data value incorrectly (according to the specification of the Canonical ABI) yet deserializes the data value to the correct (original) value. Our framework enables the detection of self-consistent bugs by composing components from all possible pairs of source languages. Despite this, none of the bugs detected by our framework to date were self-consistent. Further work is necessary to determine if these kinds of bugs are present in WIT binding generators and whether they can be detecting using random testing.

We also note that, to date, we have found no bugs in the C binding generator of wit-bindgen. We speculate that this is because generated C bindings are simple in comparison to bindings for Rust and Go, but more investigation is required.

D. *Lessons Learned*

We identify four generalizable lessons from this work that can inform future efforts in testing binding code generators.

First, increased community efforts are needed toward testing binding code generators and similar code-generation tools. We found that the middleware enabling the interoperation of Wasm components contains significant defects (§IV-B). While considerable effort has been made to ensure the correctness of WebAssembly runtimes (§VI), the correctness of the binding-code layer has been neglected. We speculate that the situation is similar for other software component ecosystems.

Second, testing the binding layer requires specialized fuzzing tools. Compiler-fuzzing tools are not readily able to determine the correctness of code produced by binding generators: Binding code cannot be thoroughly tested without being executed, and executing binding code requires implementations of *driver* (client) and *target* (server) components. Further, detecting non-crashing bugs in binding code requires a test oracle, such as the one implicitly provided by differential testing. We could not have uncovered the bugs we found if we had not implemented our custom fuzzing framework (§III).

Third, implementing a specialized fuzzing tool for a binding code generator is both feasible and worthwhile. Our framework contains ∼6.6 KLOC of Rust and ∼1.1 KLOC of Python, and it utilizes suitable internal representations to generate driver and target programs in three programming languages, enabling significant "mix-and-match" testing of component pairs (§III-B). Our tool enabled the discovery of significant bugs that had gone undetected by other means of testing.

Fourth, the design of our testing framework can serve as a blueprint for constructing fuzzing tools targeting other middleware and binding code generators. OpenAPI Generator [40], gRPC [41], and SWIG [42] are all examples of heavily used systems that generate client and server stubs to enable the interoperation of disparate software components. We believe the approach we developed in this paper can be reused to create differential testing frameworks for these and other systems.

VI. Related Work

Brown et al. described binding code as having "...the dangerous distinction of being both hard to avoid and hard to get right" [43]. 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. A variety of approaches have been used to ensure the correctness of binding code, 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 [44]. 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 written in C or C++ [45]. Python uses reference counting to manage heap objects, but extension modules are outside of Python's memory management system. Therefore, the developer of an extension 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 [46]. 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 [43]. They used the checkers to formulate 81 exploits of security flaws introduced by binding code in widely used JavaScript runtimes.

Not all rules can be enforced statically; Lee et al. showed how to formulate dynamic checkers of FFI violations using state machines [47]. They used this approach to create Jinn, a dynamic bug-detection tool for the Java Native Interface. Their 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 [48–50]. 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 that provides access to code written in lower-level languages. Thus, any framework for testing browser APIs is also testing 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 [51]. 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 [52]. Random semantically valid C programs are generated using Csmith [53] and passed as input to rust-bindgen. If rust-bindgen crashes or the emitted bindings fail to compile, a bug is detected.

Wit-smith [17] 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 and/or attempt to compile the bindings. The framework presented in this paper uses wit-smith to randomly generate WIT files, but it does much more than test that the generated bindings compile correctly. Our framework performs random differential testing [18] to test the run-time behavior of generated bindings.

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

Extensive efforts have been made to find bugs in Wasm compilers and runtimes through random testing, using both generative [10, 15] and mutational [11–14, 16] methods. These approaches typically provide Wasm test cases to several Wasm runtimes and use techniques such as differential testing to detect crash and code-generation bugs. These fuzzing tools differ from the one presented in this paper because they test a separate part of the Wasm ecosystem: i.e., the compilers and runtimes. In contrast, our framework tests binding code generators, which input WIT specifications and produce the code necessary to create WebAssembly components. To be correct, WIT-derived binding code must provide a correct adapter between the Canonical ABI [2] of the WebAssembly Component Model and the interface of an encapsulated WebAssembly module (§I). Although binding code is eventually compiled to Wasm and executed by a runtime, its correctness is distinct from the correctness of the runtime itself.

In summary, the work presented in this paper differs from prior 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.

VII. Conclusion

WIT binding generators are essential tools for the creation of WebAssembly components. We set out to uncover code-generation bugs in WIT binding generators by implementing a random differential testing framework for exercising these generators. With four testing campaigns, we discovered five latent code-generation bugs and two regular code-generation bugs in WIT binding generators. We believe the lessons learned from our successful bug-hunting experience can be reused for finding bugs in other binding generators and similar tools.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Number 2431419.

References

[1] WebAssembly Community Group, "Component model," 2025. [Online]. Available: https://github.com/WebAssembly/component-model

[2] ——, "Canonical ABI," 2025. [Online]. Available: https://github.com/WebAssembly/component-model/blob/main/design/mvp/CanonicalABI.md

[3] wasmCloud LLC, "wasmCloud - a CNCF project," 2025. [Online]. Available: https://wasmcloud.com/

[4] The Spin Framework Contributors, "Introducing Spin," 2025. [Online]. Available: https://spinframework.dev/

[5] NGINX, Inc., "Universal web app server – NGINX Unit," 2025. [Online]. Available: https://unit.nginx.org/

[6] D. Bäumer, "Using WebAssembly for extension development," May 2024. [Online]. Available: https://code.visualstudio.com/blogs/2024/05/08/wasm

[7] Zed Industries, "Zed — the editor for what's next," 2025. [Online]. Available: https://zed.dev/

[8] WebAssembly Community Group, "WIT," 2025. [Online]. Available: https://github.com/WebAssembly/component-model/blob/main/design/mvp/WIT.md

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Adapter," in *Design Patterns: Elements of Reusable Software*. Addison-Wesley, 1994, pp. 139–150.

[10] Á. Perényi and J. Midtgaard, "Stack-driven program generation of WebAssembly," in *Programming Languages and Systems (APLAS 2020)*, 2020, pp. 209–230. [Online]. Available: https://doi.org/10.1007/978-3-030-64437-6_11

[11] B. Jiang, Z. Li, Y. Huang, Z. Zhang, and W. K. Chan, "WasmFuzzer: A fuzzer for WebAssembly virtual machines," in *Proceedings of the 34th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2022, pp. 537–542. [Online]. Available: https://doi.org/10.18293/SEKE2022-165

[12] 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. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00188

[13] S. Cao, N. He, X. She, Y. Zhang, M. Zhang, and H. Wang, "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, pp. 1262–1273. [Online]. Available: https://doi.org/10.1145/3650212.3680358

[14] W. Zhao, R. Zeng, and Y. Zhou, "Wapplique: Testing WebAssembly runtime via execution context-aware bytecode mutation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1035–1047. [Online]. Available: https://doi.org/10.1145/3650212.3680340

[15] J. Park, Y. Kim, and I. Yun, "RGFuzz: Rule-guided fuzzer for WebAssembly runtimes," in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 920–938. [Online]. Available: https://doi.org/10.1109/SP61157.2025.00003

[16] L. Zhang, B. Zhao, J. Xu, P. Liu, Q. Xie, Y. Tian, J. Chen, and S. Ji, "Waltzz: WebAssembly runtime fuzzing with stack-invariant transformation," in *Proceedings of the 34th USENIX Security Symposium*, 2025, pp. 6159–6178. [Online]. Available: https://www.usenix.org/conference/usenixsecurity25/presentation/zhang-lingming

[17] The Bytecode Alliance, "wit-smith," 2025. [Online]. Available: https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wit-smith

[18] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[19] E. Stanley, "WIT bindgen fuzzer," 2025. [Online]. Available: https://gitlab.flux.utah.edu/ethanstanley12/wit-bindgen-fuzzer

[20] The Bytecode Alliance, "wit-bindgen," 2025. [Online]. Available: https://github.com/bytecodealliance/wit-bindgen

[21] ——, "wit-bindgen-go," 2025. [Online]. Available: https://github.com/bytecodealliance/go-modules

[22] The TinyGo Organization, "TinyGo," 2025. [Online]. Available: https://github.com/tinygo-org/tinygo

[23] E. Stanley, "Random testing of WebAssembly Interface Type binding generators," Master's thesis, University of Utah, 2025. [Online]. Available: https://www.flux.utah.edu/paper/stanley-thesis

[24] WebAssembly Community Group, "WebAssembly," 2019. [Online]. Available: https://webassembly.org/

[25] The Bytecode Alliance, "Wasmtime," 2025. [Online]. Available: https://github.com/bytecodealliance/wasmtime

[26] ——, "WebAssembly Compositions," 2025. [Online]. Available: https://github.com/bytecodealliance/wac

[27] WebAssembly Community Group, "WASI Preview 2," 2025. [Online]. Available: https://github.com/WebAssembly/WASI/blob/main/wasip2/README.md

[28] ——, "WIT resource types," 2025. [Online]. Available: https://github.com/WebAssembly/component-model/blob/main/design/mvp/WIT.md#item-resource

[29] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, Feb. 2002. [Online]. Available: https://doi.org/10.1109/32.988498

[30] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 335–346. [Online]. Available: https://doi.org/10.1145/2254064.2254104

[31] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/duplyakin

[32] E. Stanley, "Wit-bindgen issue 1112," 2025. [Online]. Available: https://github.com/bytecodealliance/wit-bindgen/issues/1112

[33] ——, "Wit-bindgen issue 1134," 2025. [Online]. Available: https://github.com/bytecodealliance/wit-bindgen/issues/1134

[34] ——, "Go-modules issue 284," 2025. [Online]. Available: https://github.com/bytecodealliance/go-modules/issues/284

[35] ——, "Go-modules issue 288," 2025. [Online]. Available: https://github.com/bytecodealliance/go-modules/issues/288

[36] ——, "Go-modules issue 344," 2025. [Online]. Available: https://github.com/bytecodealliance/go-modules/issues/344

[37] ——, "Go-modules issue 352," 2025. [Online]. Available: https://github.com/bytecodealliance/go-modules/issues/352

[38] ——, "Go-modules issue 350," 2025. [Online]. Available: https://github.com/bytecodealliance/go-modules/issues/350

[39] ——, "Go-modules issue 348," 2025. [Online]. Available: https://github.com/bytecodealliance/go-modules/issues/348

[40] OpenAPI-Generator Contributors, "Hello from OpenAPI Generator," 2025. [Online]. Available: https://openapi-generator.tech/

[41] gRPC Authors, "grpc: A high performance, open source universal RPC framework," 2025. [Online]. Available: https://grpc.io/

[42] The SWIG Developers, "Welcome to SWIG," 2025. [Online]. Available: https://www.swig.org/

[43] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan, "Finding and preventing bugs in JavaScript bindings," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 559–578. [Online]. Available: https://doi.org/10.1109/SP.2017.68

[44] G. Tan and G. Morrisett, "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 (OOPSLA)*, 2007, pp. 39–56. [Online]. Available: https://doi.org/10.1145/1297027.1297031

[45] A. Rigo and M. Fijalkowski, "CFFI documentation," 2024. [Online]. Available: https://cffi.readthedocs.io/en/stable/

[46] S. Li and G. Tan, "Finding reference-counting errors in Python/C programs with affine analysis," in *European Conference on Object-Oriented Programming (ECOOP)*, 2014, pp. 80–104. [Online]. Available: https://doi.org/10.1007/978-3-662-44202-9_4

[47] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. S. McKinley, "Jinn: Synthesizing dynamic bug detectors for foreign language interfaces," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 36–49. [Online]. Available: https://doi.org/10.1145/1806596.1806601

[48] C. Zhou, Q. Zhang, L. Guo, M. Wang, Y. Jiang, Q. Liao, Z. Wu, S. Li, and B. Gu, "Towards better semantics exploration for browser fuzzing," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, Oct. 2023. [Online]. Available: https://doi.org/10.1145/3622819

[49] C. Zhou, Q. Zhang, M. Wang, L. Guo, J. Liang, Z. Liu, M. Payer, and Y. Jiang, "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 (ESEC/FSE)*, 2022, pp. 1135–1147. [Online]. Available: https://doi.org/10.1145/3540250.3549107

[50] J. Wang, P. Qian, X. Huang, X. Ying, Y. Chen, S. Ji, J. Chen, J. Xie, and L. Liu, "Tacoma: Enhanced browser fuzzing with fine-grained semantic alignment," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1174–1185. [Online]. Available: https://doi.org/10.1145/3650212.3680351

[51] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé, and Y. Shoshitaishvili, "Favocado: Fuzzing the binding code of JavaScript engines using semantically correct test cases," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*, 2021. [Online]. Available: https://doi.org/10.14722/ndss.2021.24224

[52] The Rust Team, "Fuzzing bindgen with Csmith," 2022. [Online]. Available: https://github.com/rust-lang/rust-bindgen/blob/main/csmith-fuzzing/README.md

[53] 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, pp. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532