TeTRIS: General-purpose Fuzzing for Translation Bugs in Source-to-Source Code Transpilers

Yeaseen Arafat
University of Utah

Stefan Nagy
University of Utah

***Abstract*—Amid the rise of heterogeneous computing and concerns over systems and application security, developers are increasingly embracing *transpilers*: a growing class of tools for converting code from one programming language into another. As languages differ greatly in constructs, syntactic sugar, security mitigations, and more, transpilers face difficulties in faithfully translating software between source and target languages—sometimes causing outright failures, or worse, subtle-yet-incorrect execution behavior. Proactively testing transpilers' correctness is thus critical to the success of code-translation-oriented development tasks, but unfortunately, no effective techniques currently exist. Although fuzz-testing appears a natural fit, current general-purpose fuzzing tools mostly generate invalid, junk code that fails to engage transpilers' core translation logic; while dedicated compiler fuzzers cannot keep pace with the ever-expanding set of languages targeted by existing and emergent transpilers. Thoroughly vetting transpilers' correctness thus demands a fuzzing approach combining the *reach* of general-purpose fuzzing—with the *precision* of dedicated compiler fuzzers.**

This paper presents TETRIS: a general-purpose fuzzer for testing source-to-source code transpilers. At its core, TETRIS bridges the flexibility of general-purpose fuzzing, abstracting away language-level differences into a unified interface for fine-grained code mutations, with the precision of compiler fuzzers by rigorously enforcing syntactic and semantic correctness. Relying solely on minimal language specifications, TETRIS supports fuzzing of *any* transpiler—irrespective of input or output language—producing high-quality programs that extensively probe its underlying translation logic. In an evaluation against four state-of-the-art fuzzers across seven popular transpilers for C, Go, and Haxe, TETRIS is the only solution to uphold both high language validity *and* high transpiler code coverage—whilst supporting the broadest range of transpilers. Moreover, TETRIS reveals the most code translation bugs—all 12 of which were previously unknown—underscoring its effectiveness in vetting today's diverse transpiler ecosystem.

1. Introduction

Transpilers, or source-to-source translators, are an emergent class of code processors designed to rewrite software from one programming language to another—preserving its functionality whilst adapting it to the target language's syntax and semantics. Though early automated code translation primarily centered on web application development (e.g., translating to JavaScript [40] or WebAssembly [60]), transpilers are rapidly gaining popularity in critical systems and application use cases such as platform-to-platform software migration [43], [39], [7], [28], security retrofitting for legacy code [32], and maintaining cross-platform interoperability [19], [46], [5]. Among the most prominent translation efforts today are DARPA's initiatives for rewriting C code into Rust [11], notably through its funding of industrially-developed transpilers [26]. Nowadays, transpilers exist for a wide range of languages, including translating C to Rust [26], Go [52], [27], and Zig [29]; Go to Haxe [54]; as well as Haxe to C++ [22] and Python [23].

Maintaining code equivalence is critical to successful transpilation, necessitating precise handling of syntax and semantics between source and target languages. Yet, transpilation is fraught with difficulty. Different languages face disagreeing constructs, making one-to-one mapping seldom achievable. For example, C's use of `goto` statements is unmatched by Rust, burdening translators with the non-trivial challenge of reconstructing C control flow into equivalent idiomatic Rust. Moreover, differences in typing, such as C's implicit type conversions versus Go's requirement for explicit casting, necessitate careful handling to ensure all edge cases are covered. Thus, as software translation errors risk introducing unexpected program behavior—or producing completely broken or otherwise unusable code—**thoroughly vetting the *correctness* of transpilers is vital to the success of translation-oriented development tasks**.

Fuzzing, or fuzz-testing, has historically seen widespread adoption in automated software bug detection, with numerous successful applications targeting various types of code processors, including compilers [58], decompilers [35], and disassemblers [44]. Unfortunately, **no current fuzzers are well-suited to testing transpilers.** Existing general-purpose fuzzers [16], [20], [9], while supportive of virtually all transpilers today, struggle to create syntactically- and semantically-valid test cases, leaving them unable to exercise transpilers' core translation logic. In contrast, though dedicated compiler fuzzers [58], [36] generate high-quality test cases, their rigid input structure models and mutations restrict them to only individual languages (e.g., C [58])—leaving them unsupportive of the broader transpiler ecosystem that spans multiple programming languages (e.g., Go [27], Haxe [22], Zig [29]). Ensuring correctness of translation-oriented development

tasks thus demands **a comprehensive testing strategy combining both general-purpose fuzzing's reach—with dedicated compiler fuzzers' precision**.

To tackle this challenge, this paper introduces **TETRIS: the first general-purpose fuzzer targeting source-to-source transpilers**. Guided by the key tradeoffs between general-purpose and compiler-dedicated fuzzers, we design TETRIS as an all-in-one platform for flexible transpiler fuzzing. To achieve breadth, TETRIS unifies language-specific differences through a shared intermediate representation, enabling language-aware mutations that systematically explore a wide range of translation edge cases without requiring manual re-engineering per each language. To achieve depth, TETRIS combines language grammars with lightweight semantic constraints, allowing it to generate well-formed, semantically-valid test cases comparable in quality to those produced by compiler fuzzers. Together, these capabilities make TETRIS a practical and extensible solution for fuzzing transpilers—irrespective of their input or output languages—**bringing automated bug discovery to an ever-growing, ever-critical software ecosystem that has, so far, lacked any form of dedicated fuzzing support**.

We evaluate TETRIS on seven real-world transpilers spanning a diverse set of source-to-target language pairings: **C2Rust** [26] (C→Rust), **CxGo** [52] and **C4Go** [27] (C→Go), **Zig Translate-C** [29] (C→Zig), **Go2Hx** [54] (Go→Haxe), **HxCpp** [22] (Haxe→C++), and **HxPy** [23] (Haxe→Python). We compare TETRIS to four state-of-the-art fuzzers: general-purpose AFL++ [16], Polyglot [9], and AFL-Compiler-Fuzzer [20]; as well as compiler fuzzer CSmith [58]. Across five day-long campaigns, TETRIS achieves language validity only slightly below that of C-specific fuzzer CSmith—yet outperforms general-purpose fuzzers Polyglot, AFL++, and AFL-Compiler-Fuzzer by **6–31**×—supporting all seven transpilers without any re-engineering effort for any of them. In total, TETRIS discovers **12 new transpiler bugs** across seven real-world transpilers—the most in our evaluation—**with all since confirmed and/or fixed by developers**.

In summary, we make the following contributions:

- We distill the challenges of fuzzing source-to-source transpilers; and assess the trade-offs faced by conventional general-purpose as well as dedicated compiler fuzzers in supporting them—highlighting the need for a principled approach combining these approaches' strengths to enhance both the breadth and precision of transpiler testing.
- We tackle these challenges through designing TETRIS: a general-purpose fuzzer for testing correctness of source-to-source code translators—irrespective of input language.
- We evaluate TETRIS on seven real-world code transpilers spanning a multitude of input-output language pairs—C to Rust, C to Go, C to Zig, Go to Haxe, Haxe to C++, and Haxe to Python—alongside four state-of-the-art fuzzers: AFL++, Polyglot, AFL-Compiler-Fuzzer, and CSmith.
- We show that, throughout 5×24-hour fuzzing campaigns per transpiler, TETRIS outperforms conventional general-purpose as well as dedicated compiler fuzzing tools in upholding both test case validity and high code coverage, whilst finding the most bugs across multiple transpilers.
- We manually analyze and report all bugs uncovered in our evaluation. Of TETRIS's 12 newly-uncovered bugs, all are confirmed or fixed by their respective developers.
- We open-source our prototype TETRIS, and all evaluation benchmarks and artifacts at the following URL: `https://github.com/FuturesLab/TeTRIS`.

2. Background & Related Work

This section provides an overview of source-to-source transpilers, defects emerging during code translation, and the challenges involved in testing the correctness of transpilers.

2.1. Source-to-Source Code Transpilers

Transpilers (short for "transcompilers") are an emergent class of automated language processors that convert source code from one programming language to another, facilitating easier, automation-assisted migration across differing technology stacks [26] and platforms [22]. Today, a rapidly growing ecosystem of transpilers now addresses a wide range of source-to-source translation needs. C2Rust [26] is a prominent example, translating C to Rust and inspiring follow-on tools focused on concurrency [24], memory safety [13], [12], as well as lighter-weight translation [34]. Other efforts include C-to-Go transpilers (e.g., CxGo [52], C4Go [27]), as well as tools targeting more recent languages such as Zig [29] and Nim [42]. Additionally, some ecosystems—such as Fusion [17] and Haxe [22]—natively support multi-target transpilation to a broad set of languages. While implementations vary, most transpilers share a common structure consisting of four key steps, as outlined below.

2.1.1. Step 1: Source AST Generation. Transpilation begins with parsing the input program's source into an initial abstract syntax tree (AST), representing the code's overall syntactic structure. For example, given C function declaration `double sum(int x, float y)`, the AST is populated with unique nodes for the function name, return type, and all parameters. Analyses are also performed to capture the source language's semantics. For example, static type checking on the expression `(x+y)` may reveal an implicit type conversion on `x`, prompting the AST to be annotated with additional type information to reflect this coercion. Similarly, other analyses (e.g., recognition of higher-level constructs [54], object lifetimes [26]) produce their own annotations, bridging the transpiler's understanding of the source program's syntax *and* semantics.

2.1.2. Step 2: Intermediate AST Transformation. With the source AST in hand, an *intermediate AST* is constructed to explicitly capture any semantic or structural differences between the source and target languages. Central to this is identifying which source language data types and constructs are unsupported in the target language, and rewriting them into *semantically-equivalent* forms conformant to the target language. For example, Go's stricter type system

prohibits C's implicit type conversions, such as `int p = (4>5)`; to address this, CxGo [52] introduces a custom library function [51] in the intermediate AST to explicitly convert booleans to integers, ensuring compatibility with Go's type requirements. Similarly, Rust lacks C's `goto` unconditional branching, requiring transpilers to rewrite such instances as Rust-supported `match` cases, as shown in Figure 1. Through meticulously handling cross-language nuances, a transpiler's intermediate AST transformations lay the groundwork for accurately *adapting* the original program logic to the syntax and semantics of the target language.

```c
if (y < 5) {
    goto LABEL_RES;
}
if (x < 20) {
    goto LABEL_END;
}
LABEL_RES:
    res = x*y;
LABEL_END:
    return res;
```

(a) Input code in C.

```rust
if y < 5
    {  cur_blk = 1;   }
else if x < 20
    {  cur_blk = 2;   }
else
    {  cur_blk = 1;   }
match cur_blk {
    1 => { res = x*y; }
    _ => {}
} return res;
```

(b) Output code in Rust.

Figure 1: `goto`-using C program and semantically-equivalent Rust version.

2.1.3. Step 3: Final AST Generation & Refinement. After translating the intermediate AST into a preliminary version of the target-language AST, a final refinement pass is performed to ensure that the resulting code conforms to the target language's syntactic rules and semantic requirements, such as type correctness and idiomatic usage. This typically involves AST-level optimizations such as simplifying nested expressions and eliminating dead code for improved readability and performance; resolving type discrepancies, such as inserting explicit casts when required by the target language (e.g., converting integers to floating-point types); and enforcing idiomatic compliance, such as replacing pointer arithmetic in C with safe reference constructs in Rust. Through these careful refinements, transpilers ensure that the generated code not only behaves correctly, but also aligns with the target language's best practices, resulting in cleaner, more efficient, and maintainable output code.

2.1.4. Step 4: Target Code Generation. With the refined target-language AST in hand, transpilers proceed with generating the final target language source code by systematically translating each AST node into its corresponding textual syntax. Unlike the previous step, which focused on adapting program structure to the target language's semantics, this phase emphasizes producing concrete, human-readable code that adheres to the language's grammar, formatting, and naming conventions. Beyond preserving the original program's logic, transpilers strive to meet the performance and stylistic expectations of the target ecosystem, with the aim of enabling seamless integration of the translated code within existing toolchains. This balance between semantic fidelity and idiomatic expression is exemplified by CxGo's translations of Potrace [53] and PortableGL [31], which maintain both compatibility and runtime efficiency.

2.2. Errors in Code Translation

While transpilers ideally perfectly translate programs into their targeted language, accurate code translation is fraught with difficulty. To better understand the challenges faced by transpilers, we survey recent publicly-reported errors across eight mainstream transpilers covering a variety of input and output language pairs. After analyzing each of their root causes, we observe that these transpiler bugs fall into three distinct high-level categories (Table 1): **(1) Syntactic Errors**, **(2) Type Conversion Errors**, and **(3) Code Fragment Omissions**. We detail these each below.

Bug Type	Transpiler	Brief Error Description	Iss. ID
Syntactic	CxGo	Mis-included variable re-declaration	#42
	C2Go	Mis-recovered `switch-case` values	#848
	C2Nim	Mis-recovered `struct` members	#217
	HxCpp	Mis-recovered null `array` in callee	#746
	Zig Translate-C	Mis-recovered `array` type and size	#21192
Type Conversion	C2Rust	Mis-recovered `int`→`float` conversion	#486
	Zig Translate-C	Mis-recovered `bool`→`int` conversion	#20005
	Zig Translate-C	Mis-recovered `int`→`bool` conversion	#20638
Code Fragment	C2Go	Unrecovered `struct` name	#886
	C4Go	Unrecovered `unary` operator	#482
	C2Nim	Unrecovered standard `int` types	#240
	C2Rust	Unrecovered bitfield `struct`	#183
	C2Rust	Unrecovered `call` instruction	#896
	Go2Hx	Unrecovered `array` size	#19
	HxCpp	Unrecovered optional `int` arguments	#225

TABLE 1: Known transpiler translation bugs. `Iss.ID` = GitHub issue ID.

2.2.1. Type 1: Syntactic Errors. Variations in syntax rules between source and target languages can lead to translation errors, such as incorrect ordering or misinterpretation of syntax elements, resulting in non-compilable code. For example, in Figure 2, the translated Go code incorrectly re-declares variable `x` within the *same* scope, causing a compilation error. This issue arises because the CxGo transpiler fails to accurately translate the C macro in lines #1–2. While this macro is indeed valid in C—permitting variable re-declaration within separate block scopes—CxGo mistakenly treats these declarations as occurring within the same scope, producing syntactically-invalid Go code.

```c
#define func(arg)
    { int x = arg; }
if (0) {
    func(5);
    func(6);
}
```

(a) Input C.

```go
if false {
    var x int = 5
    _ = x
    var x int = 6
    _ = x
}
```

(b) CxGo output.

Figure 2: CxGo issue #42: error resulting from re-declaration of local `x`.

2.2.2. Type 2: Type Conversion Errors. Due to differences in how data types are defined, used, and cast across languages, transpilers may struggle to maintain type consistency in the translated code. For example, in Go and Zig, an `if` condition must explicitly be of boolean type. In contrast, C *implicitly* converts integers in `if` conditions to booleans, treating non-zero values as true and zero as false. In Figure 3, consider the C macro on line #1, where the condition is implicitly treated as true in the ternary operation. However, when translated to Zig via its Translate-C transpiler, the `if` condition incorrectly retains the `integer`

type directly, even though Zig requires that conditions be explicitly converted to `bool` (e.g., `@as(bool,2)`) [62]. This mismatch underscores the challenges of accurately type conversions across languages during transpilation.

```
1  #define func(x) (2 ? 4 : 8)
```
(a) Input C.

```
1  fn func(x: anytype) c_int (
2      return if (@as(c_int, 2))
3          @as(c_int, 4);
4      else
5          @as(c_int, 8);
6  )
```
(b) Zig Translate-C output.

Figure 3: Zig issue #20005: mis-recovery of implicit boolean as an integer.

2.2.3. Type 3: Code Fragment Errors. Unlike the errors rooted in syntactically-invalid code or incorrect type conversions, transpilers may also emit code that is syntactically valid—but *semantically* wrong from missing code fragments such as calls, arguments, or operators, resulting in output that fails to match the original code's behavior. For instance, in Figure 4, the C function call `--func();` on line #6 ends up entirely omitted in the Rust code generated by C2Rust. This omission occurs because C2Rust incorrectly identifies the function call as an unnecessary operation, given that its result is not directly used. However, this oversight fails to account for the function's effects at execution time—such as updating the global `g` on line #2—leading to incorrect runtime behavior in the translated Rust code.

```
1  int func() {
2      g = 1;
3      return g;
4  }
5  int main() {
6      -func();
7  }
```
(a) Input C.

```
1  fn func() -> c_int {
2      g = 1 as c_int;
3      return g;
4  }
5  fn main() {
6      //-func();
7  }
```
(b) C2Rust output.

Figure 4: C2Rust issue #896: erroneous omission of call `-func()`.

2.2.4. Takeaways. As Table 1 shows, current transpilers encounter significant challenges, ranging from obvious errors that impede transpiled programs from even being usable (e.g., **Types 1 & 2**), to more subtle semantic errors revealed only through divergent runtime behavior (e.g., **Types 2 & 3**). Moreover, differing implementations further increase the likelihood of inconsistencies; for example, C2Rust relies on a Clang-based preprocessor [38] to parse and type-check C source, while CxGo uses a custom-built C compiler frontend [56]. Without a standard to guide the internal strategies of transpilers, these varied approaches make consistent accuracy across tools difficult to guarantee. Thus, **a comprehensive, automated approach is needed** to accelerate identification—and further, timely resolution—of the increasingly-complex code translation errors across today's ever-growing transpiler ecosystems.

3. Motivation: Obstacles to Transpiler Fuzzing

Although real-world code seems like the natural starting point for testing transpilers, its reliance on unsupported constructs makes it a non-starter for uncovering translation bugs. For example, C2Rust [26] cannot translate C's inlined/variadic functions, SIMD, or packed structs, while Go2Hx [54] omits Go's concurrency features. As these unsupported constructs *halt* transpilation altogether, they obscure the subtle translation errors that occur within otherwise-supported features (e.g., Figure 1). Consequently, relying on real-world code is ineffective for exposing transpiler bugs, **motivating the need for a dedicated testing approach tailored to source-to-source transpilers**.

With extensive adoption across industry, *fuzzing* (or "fuzz-testing") has emerged as by far today's most practical and successful automated testing strategy for large-scale software bug discovery [49], [16]. Yet, despite numerous available fuzzers, **none comprehensively support transpilers**—leaving transpiler developers without an effective means to vet code translation correctness. Below, we examine the fuzzer categories most relevant to transpiler testing—**(1) general-purpose fuzzers** and **(2) compiler fuzzers**—and highlight their limitations in achieving systematic, practical fuzzing of today's diverse source-to-source transpilers.

3.1. Transpiler-relevant Fuzzing Techniques

With no dedicated fuzzer for transpilers today, transpiler developers frequently turn to general-purpose options like AFL++ [16] and Polyglot [9], or to compiler-dedicated ones such as CSmith [58]. Table 2 provides an overview of these techniques, which we further discuss below.

3.1.1. General-purpose Fuzzers. At their core, general-purpose fuzzers provide broad testing capabilities for nearly any program input format—or in the case of transpilers—any input language. Most of these fuzzers rely on mutational input generation, dynamically modifying inputs to create diverse and expansive test case corpora. AFL++ [16] and libFuzzer [48] are among today's most popular general-purpose fuzzers, having identified thousands of bugs across a wide range of software domains. Moreover, recent years have also seen the emergence of newer general-purpose fuzzers specifically tailored to language processors—primarily compilers, though equally applicable to transpilers. AFL-Compiler-Fuzzer [20], for example, performs syntax-aware string mutations using regular expressions; while Polyglot [9] instead utilizes language grammars to generate programs conformant to the given target's input language syntax.

3.1.2. Dedicated Compiler Fuzzers. Efforts to test language compilers have increasingly turned to dedicated *compiler fuzzers*—tools that generate well-formed inputs designed to rigorously probe compilers' front-ends (e.g., code parsing) and back-ends (e.g., code optimizations). These fuzzers typically perform grammar-aware mutations that conform to the syntax of the target language, incorporating additional specialized restrictions to also maximize semantic validity. Examples include CSmith [58] and YARPGen [36], which have uncovered numerous bugs in C compilers such as LLVM [33]. While primarily built for compilers, these

tools also see occasional use in transpiler testing. For example, CSmith-generated programs have been used to evaluate C-to-Rust transpilers such as C2Rust [25], given their ability to produce diverse, semantically-sound input programs.

3.2. Fundamental Design Considerations

While fuzzing offers a promising way of systematically uncovering translation errors, existing fuzzing tools were not designed with transpiler testing in mind. To understand how current fuzzers can be adapted—or what new capabilities are required—we survey current techniques and identify three fundamental design considerations that directly impact their suitability for transpiler fuzzing (Table 2): **(1) Language Agnosticism**, **(2) Minimal Language Specification**, and **(3) Code Validity**. Below, we discuss the significance of each—and derive a criteria for defining fuzzing approaches best-suited to practical, effective transpiler testing.

Fuzzer Name	C1: Lang Agnostic	C2: Min Langspec	C3: Code Validity
AFL++ [16]	✔	n/a	✗
libFuzzer [48]	✔	n/a	✗
AFL-Compiler-Fuzzer [20]	✔	n/a	✗
Polyglot [9]	✔	✔	✔
CSmith [58]	✗	✗	✔
YARPGen [36]	✗	✗	✔
TETRIS (our approach)	✔	✔	✔

TABLE 2: Conventional fuzzers and their fundamental tradeoffs with respect to transpiler fuzzing. n/a: specifications irrelevant to that fuzzer.

3.2.1. Consideration 1: Language Agnosticism. A central challenge in fuzzing transpilers is the need to support a wide range of source–to-target language pairs. General-purpose fuzzers like AFL++ [16] treat inputs as raw byte streams, making them broadly applicable but prone to generating invalid code. Similarly, general-purpose language-processor fuzzers like Polyglot [9] apply mutations based on user-provided specifications, offering flexibility to any language. In contrast, dedicated compiler fuzzers such as CSmith [58] remain hardcoded to individual languages and require significant non-trivial re-engineering to be extended to others. This lack of portability severely limits compiler fuzzers' utility for transpiler fuzzing, where multi-language support is essential. A practical solution must strike a balance: it should remain **independent of any specific language** while retaining enough structural awareness to generate meaningful, valid test cases across diverse language pairs.

Fuzzer	Fuzzer Component Size (LOC)			
	Core	C Spec	Go Spec	Haxe Spec
CSmith [58]	38,988		n/a	n/a
Polyglot [9]	7,016	1,508	n/a	n/a
TETRIS	5,601	811	797	785

TABLE 3: Total code lines of fuzzing core and per-language specifications for compiler fuzzer CSmith, general-purpose language processor fuzzer Polyglot, and TETRIS. n/a: language unimplemented by that fuzzer.

3.2.2. Consideration 2: Minimal Language Specification. Most language processor fuzzers rely on hand-crafted *specifications* to bootstrap their generation of syntactically- and semantically-valid programs. As shown in Table 3, CSmith [58] includes tens of thousands of lines of C-specific code generation logic—making reuse difficult and extension to new languages impractical. In contrast, Polyglot [9] requires lightweight specifications consisting of only language grammars and accompanying semantic annotations, enabling far easier adaptation across languages. For transpiler fuzzing, where broad language coverage is essential, such specification overhead is a major scalability barrier. A practical solution must **minimize per-language specification effort**, ideally leveraging existing grammars to support high-quality generation with minimal customization.

3.2.3. Consideration 3: Code Validity. While compiler fuzzers such as CSmith [58] remain the gold standard for code validity due to their language-specific designs, general-purpose fuzzers like AFL++ mutate inputs indiscriminately at the string level—ignoring language syntax and semantics (e.g., `floatx=3.1;`→`f#âtx>3.1;`). AFL-Compiler-Fuzzer exhibits similar issues, often producing syntax-breaking mutations (e.g., `inty=3;`→`inty=;`), resulting in test cases that rarely reach the transpiler's core logic. While Polyglot improves syntactic validity via its specification-guided mutation, its *semantic* validity remains low—as little as 20%—due to errors such as duplicate declarations. As a result, it only marginally improves test coverage of transpiler translation behavior. For effective transpiler fuzzing, it is essential to generate inputs that are **both syntactically valid and semantically sound**, ideally through lightweight, generalizable mechanisms rather than brittle, language-specific infrastructure.

Motivation: Current fuzzers either fail to generate valid code, or cannot easily support today's many transpilers across different programming languages, leaving transpilers' core logic—and bugs—occluded. Effectively testing transpilers thus demands a versatile fuzzing approach that combines **(1) the flexibility of general-purpose fuzzers** with **(2) the correctness of compiler-dedicated ones**—extending comprehensive testing across today's growing transpiler ecosystems.

4. TeTRIS: Design & Implementation

To make systematic transpiler fuzzing practical and effective, we introduce **TETRIS**—**Te**sting **T**ranspilers **R**egardless of **I**nput **S**ource—a language-agnostic transpiler fuzzer capable of generating syntactically- *and* semantically-valid inputs comparable to those of dedicated compiler fuzzers. Unlike real-world code, which often breaks translation outright due to transpiler-*unsupported* language features (§ 3), TETRIS automatically generates synthetic, targeted programs that focus on transpilers' *supported* features, uncovering the subtle bugs that would otherwise remain hidden. In the following section, we detail our underlying transpiler fuzzing methodology, as well as the design decisions behind our prototype implementation of TETRIS.

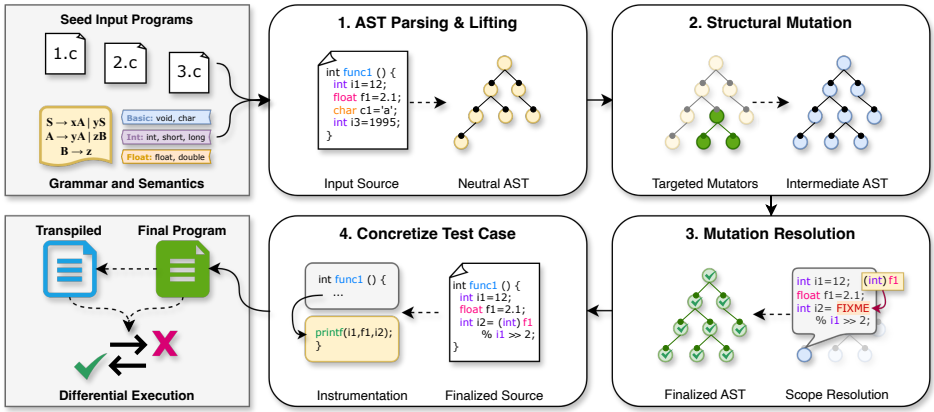


Figure 5: TETRIS: our general-purpose transpiler fuzzer. Shown are TETRIS's four steps in a setup targeting an arbitrary C-processing transpiler.

4.1. Overview

Figure 5 illustrates TETRIS's overall workflow. TETRIS prioritizes syntactic validity by applying construct-level mutations directly on a unified, language-agnostic abstract syntax tree (AST) representation of the program (**Step 1**), enabling seamless support for diverse transpiler input languages. To preserve semantic validity, it applies targeted mutators (**Step 2**) guided by precise scope and type resolution (**Step 3**). The mutated AST is then re-rendered into a complete source program (e.g., Go for Go2Hx [54]), with differential-testing instrumentation inserted to detect runtime divergences between the original and transpiled outputs (**Step 4**). In addition to behavioral mismatches, TETRIS captures other failure modes—such as transpiler crashes and post-translation errors—**systematically identifying all observable indicators of translation defects**. We describe each of these phases in detail below.

4.2. AST Lifting and Structural Code Mutations

As Figure 5 shows, TETRIS begins by selecting an input program from the user-provided seed corpus, and lifting its source into a corresponding AST—represented as a binary tree with each node containing an order, type, operator, and value, along with contextual details such as data types (e.g., if it is a variable or function prototype) and its scope hierarchy (e.g., global-, function-, or statement-level scope). While many fuzzers are built around hardcoded, language-specific AST parsing (e.g., GrayC's [14] reliance on LLVM's C-to-AST parsing [37]), our approach prioritizes *generality*: we implement AST parsing atop the Bison [10] and Flex [45] frameworks, enabling language-agnostic parsing requiring only a BNF (Backus–Naur form) source language grammar.

In the following, we detail TETRIS's subsequent program mutation procedures and explain how they work together to produce diverse, well-formed programs that effectively test the core translation logic of transpilers.

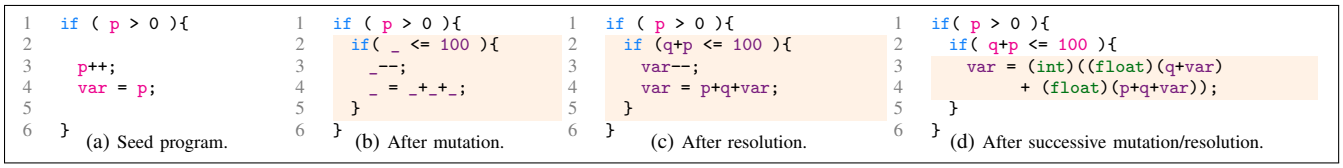
4.2.1. Source Construct Dictionary. As other language processor fuzzers [14], [59], TETRIS performs mutations to mix-and-match different source language constructs. To support this, it performs AST parsing on all user-provided seed programs to populate an internal dictionary of source construct samples, analogous to a biological "gene pool" of code samples. Examples of recovered constructs include string and numeric literals (e.g., `int x = 0x4000`), control-flow statements (e.g., `goto LABEL`), and sub-expressions (e.g., `dim_x + dim_y`). At a high level, these samples serve as the fundamental building blocks for TETRIS's subsequent code mutations, described below.

Mutator Name	Description and Targeted Source Constructs
REPLACE-OPERATOR	Replace operator with language-permitted operator. **Targets:** arithmetic/bitwise/unary/logical/relational ops.
REPLACE-LITERAL	Replace string and numerical literals with a random value. **Targets:** all global and local literals.
RECAST-EXPLICIT	Replace explicit cast with language-permitted alternative. **Targets:** all global and local variables.
EXPAND-EXPRESSION	Expand a left or right sub-expression with a new one. **Targets:** all expressions' left or right sub-expressions.
SWAP-STATEMENT	Replace an entire statement with another. **Targets:** all statements.
DELETE-STATEMENT	Delete an entire statement. **Targets:** all statements.

TABLE 4: Overview of TETRIS's source construct-level mutators.

4.2.2. Construct-level Mutation. After populating its internal dictionary of source elements, TETRIS performs randomized construct-level mutations (Table 4). These mutations operate at the granularity of syntactically-meaningful AST nodes—such as `if` statements, `goto` branches, or expressions—to preserve syntax and avoid the errors that plague general-purpose fuzzers (§ 3.2.3). TETRIS targets a wide range of such constructs, drawn from categories common across many languages, including statements, expressions, literals, operators, and typecasts. For example, TETRIS's SWAP-STATEMENT replaces full statements (e.g., substituting `p++` with an `if` block, Figure 6), while REPLACE-LITERAL substitutes literal values with randomly chosen alternatives. While additional language-specific mutators are possible (e.g., manipulating C dispatch tables), our current implementation focuses on constructs common across *many* languages to maximize generality.

4.2.3. Language Specifications. Beyond syntactic correctness, TETRIS must also preserve the *semantics* of the input



(a) Seed program. (b) After mutation. (c) After resolution. (d) After successive mutation/resolution.

Figure 6: Side-by-side example visualizing TETRIS's successive operations on a single seed input.

```
BasicIntTypes: [
  int, short, short int,
  unsigned short int,
  long int, long long int,
  unsigned long int,
  unsigned long long int
],
BasicFloatTypes: [
  float, double
],
```

(a) C Semantics.

```
BasicIntTypes: [
  int, uint,
  int8, uint8,
  int16, uint16,
  int32, uint32,
  int64, uint64,
],
BasicFloatTypes: [
  float32, float64
],
```

(b) Go Semantics.

Figure 7: Example data type semantics for both C and Go.

language, ensuring that source constructs interact meaningfully to produce well-formed programs that transpilers will accept and correctly process. Inspired by prior specification-guided fuzzers [9], [61], TETRIS augments off-the-shelf language grammars with lightweight semantic *annotations* to enforce both construct-level semantic rules—applicable across multiple languages—as well as language-level constraints specific to individual languages.

As illustrated in Figure 7, these annotations encode valid type combinations (e.g., permissible integer types in C and Go) and are consulted during mutation (e.g., RECAST-EXPLICIT in Table 4) to ensure that type compatibility is preserved. This helps avoid many of the semantic errors, particularly type-related ones, that frequently undermine prior general-purpose fuzzers [9], [20]. We further impose a set of lightweight restrictions—external to TETRIS's core mutation engine—that guard against common semantic pitfalls. For example, as shown below, rule #1 prevents the duplication of declaration-bearing statements, which represent a frequent source of re-declaration errors in tools like Polyglot [9]. Others, shown below, enforce language-specific constraints, such as Go's ban on prefix increment operators and Haxe's requirement for using `cast()` syntax:

- **Construct-level:** Any statements bearing variable or function declarations must not be duplicated.
- **Construct-level:** Any inserted `else` statements must be immediately preceded by an `if` statement.
- **Construct-level:** To avoid infinite loops, any inserted `gotos` must target a downstream branch label.
- **Construct-level:** To avoid infinite recursion, any inserted `call` must not target the current function.
- **Language-level:** Accommodate Go's prohibition of prefixed increments and decrements (e.g., `++x`) [55].
- **Language-level:** Accommodate Haxe's requirement of wrapping all typecasts via its `cast()` function [21].

Importantly, these annotations impose minimal overhead within TETRIS, irrespective of the target language—especially when compared to the extensive, multi-thousand-line specifications at the core of compiler fuzzers such as CSmith (Table 3). TETRIS relies only on lightweight, language-specific inputs: a grammar and a corresponding set of semantic annotations. Together, these enable broad language support with **fewer than 1,000 lines** of configuration per target. This low specification burden enables TETRIS to scale efficiently across diverse transpilers, requiring only a few hours of effort per specification in our experience.

4.3. Scope and Type Resolution

Since TETRIS's mutators insert code constructs directly from its construct dictionary, the resulting intermediate syntax tree often includes references to *out-of-scope* variables. Left unresolved, these can lead not only to undeclared identifiers, but also to *type errors*, as expressions may reference variables with unknown or mismatched types. To address this, we implement a scope and type resolution system that accurately tracks both function-local and global program scopes. We describe its design below.

4.3.1. Scope and Type System. TETRIS's scope and type systems work synergistically to correctly populate all variable slots carved-out by our construct-level AST mutations (e.g., Figure 6b). A *type map* assigns each type a unique identifier: primitive types (e.g., `int`) are drawn from the semantic specification (Figure 7), while composite types (e.g., functions) record richer structure such as names, arguments, and return values. In parallel, a *scope tree* organizes visibility, with the global scope as the root and each scope-creating construct introducing a child. Each scope node maintains a symbol table of declared variables annotated with their type identifiers, and variables are visible along the path from the current node to the root.

To resolve each variable slot, TETRIS processes expressions one at a time by traversing the scope chain and collecting declared variables by type from the current scope's symbol table and from enclosing scopes (i.e., outer conditionals, the parent function, and the global scope). In Figure 6b, this yields three integer variables `p`, `q`, and `var`, which TETRIS then uses to populate randomly-constructed expressions. When resolving the assignment expression at line 4, TETRIS first selects a type identifier from the available set and then chooses a variable of that type, excluding function identifiers with matching return types. Through this process, the type map disambiguates variables from functions and enables type-correct, scope-consistent expressions (e.g., "`var = p + q + var`" in Figure 6c), which in turn evolve into more complex forms after successive mutations, resolutions, and typecasts, as shown in Figure 6d.

Although we considered reusing Polyglot's scope and type system, its limited coverage and frequent inference

errors prompted us to develop a more robust and accurate implementation. TETRIS's scope and type systems improve on Polyglot's in several key ways:

- **Retrieval:** TETRIS retrieves all code objects, while Polyglot maxes-out at 15 statements. Unlike Polyglot, TETRIS adds additional parsing to also retrieve function-level `struct` and `pointer` declarations, and `arrays`.
- **Disambiguation:** TETRIS disambiguates code identifiers (e.g., function names) from data, while Polyglot often mistakenly treats code as data (e.g., `main += 1`).
- **Structs:** TETRIS recovers `struct` objects, their members, and pointers to them; while Polyglot misinterprets and erroneously treats them as functions and variables.

Post-resolution, TETRIS's generated AST is syntactically and semantically finalized (Figure 6c), and ready for concretization. Overall, TETRIS's scope and type systems facilitate realistic intertwining of many different program data sources (e.g., variables, expressions, and globals)—challenging transpilers' meticulous parsing and recovery of their semantics—and as our evaluation shows (§ 5), **revealing more transpiler bugs than any other fuzzer**.

4.4. Test Case Concretization

With the finalized test case AST in hand, TETRIS begins generating the final concrete program that will be passed to the transpiler. During this process, it instruments the code to support *differential execution*, which serves as one of our eventual bug oracles (§ 4.5). We detail this below.

4.4.1. Differential Testing Instrumentation. Like other differential fuzzers [50], [59], TETRIS introduces a suite of global "shadow" variables initialized with random values; and additionally, it designates a primary *target* function and injects a routine to capture the runtime states of these shadow variables, enabling semantic equivalence checks between the original and transpiled programs. As mutations proceed, various logical operations modify these global variables' states in diverse data-dependent ways, rigorously testing transpilers' accuracy in program logic reconstruction.

Should transpilation succeed, we instrument both the source and translated programs with their *language-specific* printing routines: `printf()` for C, `cout` for C++, `fmt.Printf()` for Go, `sys.println()` for Haxe, `print()` for Python, `println!()` for Rust, and `std.debug.print()` for Zig. By instrumenting the source and transpiled code *separately*, we avoid burdening transpilers with the impractical task of translating entire shared libraries (e.g., `libc`) that contain source-specific print functions—an issue that would derail TETRIS's fuzzing process. Once instrumentation is complete, both programs are ready for differential execution and comparison.

4.5. Identifying Translation Bugs

Following test case generation, TETRIS evaluates transpiler correctness using a series of pre- and post-translation checks designed to uncover potential translation bugs—even when no output code is successfully produced.

- **Runtime Divergences:** As in prior compiler [58] and decompiler fuzzing [59] works, we classify cases where the original and transpiled programs produce differing runtime outputs as likely code translation errors, reflecting semantic divergence introduced during transpilation.
- **Post-translation Failures:** Many known transpiler bugs (Table 1) result in code that either fails to compile or crashes at runtime. We log such post-translation build or execution failures as probable translation errors, even in the absence of runtime output comparison.
- **Intra-translation Failures:** Finally, we flag any cases where the transpiler itself crashes or raises an internal error during translation. While not producing output code, such failures still expose bugs within the transpiler's translation logic or front-end analysis.

By capturing these distinct error classes, TETRIS facilitates detection of both obvious and subtle translation bugs. Combined with its language-agnostic design, TETRIS enhances the reach and effectiveness of transpiler fuzzing, **establishing a robust foundation for uncovering diverse translation flaws** across multiple programming languages.

4.6. Implementation

We implement TETRIS atop grey-box fuzzer AFL v2.56b, integrating our custom mutators, type and scope system, and BNF grammars with corresponding annotations for the three transpiler input languages TETRIS currently targets: C, Go, and Haxe. We modify AFL's test case generation and post-processing phases to incorporate our modifications. Where possible, we utilize AFL's QEMU-based [6] code coverage tracing to enable feedback-guided *grey-box* transpiler fuzzing, but also support a feedback-agnostic *black-box* where not.

5. Evaluation

We evaluate TETRIS's effectiveness in fuzzing a suite of real-world transpilers against four leading general-purpose and compiler-dedicated fuzzers. Our overall evaluation is guided by the following fundamental research questions:

- **RQ1:** Is TETRIS effective at generating valid inputs?
- **RQ2:** Can TETRIS more thoroughly test transpilers?
- **RQ3:** Does TETRIS uncover more translation bugs?

5.1. Experimental Setup

We evaluate TETRIS on seven real-world transpilers spanning six input–output language pairings: **C2Rust** [26] (C→Rust); **CxGo** and **C4Go** (C→Go); **Go2Hx** [54] (Go→Haxe); **HxCpp** [22] (Haxe→C++); **HxPy** (Haxe→Python); and **Zig Translate-C** [29] (C→Zig). We evaluate it against two state-of-the-art general-purpose language-processor fuzzers: Polyglot [9] and AFL-Compiler-Fuzzer [20]; general-purpose conventional fuzzer

Transpiler	TETRIS		Polyglot		AFL-Compiler-Fuzzer		AFL++		CSmith	
	TOTAL	%VALID	TOTAL	%VALID	TOTAL	%VALID	TOTAL	%VALID	TOTAL	%VALID
C2Rust	2,933	77.39%	19,998	19.56%	23,047	7.48%	66,092	4.94%	2,774	95.57%
CxGo	8,101	71.84%	203,474	8.18%	360,984	1.53%	121,694	2.72%	n/a	n/a
C4Go	332	55.41%	126,610	6.67%	229,563	0.82%	297,919	1.13%	n/a	n/a
HxCpp	2,717	75.72%	n/a	n/a	189,633	0.83%	465,746	0.79%	n/a	n/a
HxPy	3,430	77.80%	n/a	n/a	55,982	1.95%	134,511	1.72%	n/a	n/a
Mean:		**71.63%**		**11.47%**		**2.52%**		**2.26%**		**95.57%**

TABLE 5: Mean total and valid test cases per transpiler. We omit black-box-only transpilers Go2Hx and Zig Translate-C since the evaluated fuzzers dump generated test cases only in grey-box fuzzing mode. n/a: transpiler unsupported by that fuzzer.

Transpiler	TETRIS	Polyglot		AFL-Compiler-Fuzzer		AFL++		CSmith	
	TOTAL	TOTAL	Δ COV	TOTAL	Δ COV	TOTAL	Δ COV	TOTAL	Δ COV
C2Rust	72,384.4	62,075.4	+16.61%	49,662.4	+45.75%	49,658.2	+45.77%	80,648.8	-10.25%
CxGo	42,413.6	40,006.4	+6.02%	33,379.8	+27.06%	36,534.4	+16.09%	n/a	n/a
HxCpp	48,405.2	n/a	n/a	42,939.4	+12.73%	42,326.0	+14.36%	n/a	n/a
HxPy	54,046.8	n/a	n/a	49,612.8	+8.94%	48,868.8	+10.60%	n/a	n/a
Mean:			**+11.31%**		**+23.62%**		**+21.70%**		**-10.25%**

TABLE 6: Mean basic block coverage per transpiler and TETRIS's relative differences (Δ COV). All comparisons to TETRIS are statistically-significant (i.e., Mann-Whitney U p-values < 0.05). We omit transpilers unsupportive of coverage tracing. n/a: transpiler unsupported by that fuzzer.

AFL++ [16]; and dedicated C compiler fuzzer CSmith [58], as it is reportedly used in development of C2Rust [25].

Seed Inputs: We seed all fuzzers with the same 2–5 seed programs per each targeted transpiler input language. For C-consuming transpilers, we auto-generate the seeds via CSmith; while for others where no CSmith-like program generator exists (e.g., Haxe), we manually crafted the seeds ourselves to be structurally similar to CSmith's: namely, all seeds contain diverse source code construct usage—statements, expressions, literals, control-flow constructs, and more—to ensure that all fuzzers benefit from a high initial source code diversity for their eventual program mutations.

Grey- and Black-box Fuzzing: We configure TETRIS and all AFL-based fuzzers (i.e., Polyglot, AFL-Compiler-Fuzzer, and AFL++) in *grey-box* mode, using standard QEMU-based [6] coverage-tracing tools for all supportive transpilers: C2Rust, CxGo, C4Go, HxCpp, and HxPy. For QEMU-incompatible transpilers Go2Hx and Zig Translate-C, we instead conduct *black-box*, coverage-agnostic fuzzing. Accordingly, our test case validity (§ 5.2) and code coverage (§ 5.3) measurements rely on the *saved* (i.e., coverage-increasing) test case corpora that AFL stores exclusively in grey-box mode. Because AFL does not save such corpora in *black*-box mode, we cannot replicate these experiments for Go2Hx or Zig-Translate-C. However, since AFL supports saving bug-triggering inputs in black-box mode—detecting them independently of code coverage—our bug-finding experiments (§ 5.4) include both Go2Hx and Zig-Translate-C.

Trials and Resources: Following the evaluation standard set by Klees et al. [30], we conduct each fuzzing campaign experiment over five 24-hour trials per each competing fuzzer. We assess statistical significance among all TETRIS versus competitor coverage comparisons using the Mann-Whitney U test with a $p = 0.05$ significance level. We distribute all experiments across three Ubuntu 22.04 machines, each with an Intel Core i9-12900K CPU and 64GB RAM, and reuse these systems for all post-processing.

Excluded Fuzzers: We exclude several fuzzers from our evaluation due to incompatibility with transpiler testing. RustSmith [50], which generates Rust programs for compiler testing, is omitted because no transpilers exist that translate Rust into other languages. Similarly, GoFuzz [18] produces library-dependent programs that cannot be consumed by Go2Hx. Finally, while we explored GoSmith [57], we observe that it consistently fails to generate valid code, with none of its programs compiling successfully.

Excluded Comparisons and Justifications: We omit comparisons where competing fuzzers are *incompatible* with specific transpilers. For example, Polyglot lacks Haxe or Go specifications, limiting its use to C-based transpilers. Similarly, CSmith's test cases depend on non-removable library features, rendering them incompatible with C-based transpilers that lack library translation support (e.g., CxGo); thus, we evaluate CSmith only against C2Rust. All incompatibilities are marked as n/a in our evaluation tables.

5.2. RQ1: Test Case Validity

Transpilers, like compilers and decompilers, process highly-structured inputs, requiring adherence to language syntax and semantics to effectively test their core logic. Accordingly, we measure all fuzzers' percentage of *valid* test cases: namely, the proportion of their final saved corpora of test cases that are *accepted* by the targeted transpiler—and therefore un-rejected for syntactic or semantic errors.

For all AFL-based fuzzers (i.e., TETRIS, Polyglot, AFL-Compiler-Fuzzer, and AFL++), we apply validity checking on their saved test case corpora (i.e., their populated queue directories) per trial. For CSmith, we manually collect five trials' worth of test cases capped at 24 hours of transpiler runtime each. As AFL-based fuzzers unfortunately cannot store inputs in black-box mode, we omit transpilers Go2Hx and Zig Translate-C in this experiment. Table 5 shows each fuzzer's mean total test cases and validity rates.

5.2.1. Results. As Table 5 shows, TETRIS outperforms Polyglot, AFL-Compiler-Fuzzer, and AFL++ with a mean **71.63% validity** versus these competitors' 11.47%, 2.52%, and 2.26% validity rates, respectively. Unsurprisingly, general-purpose fuzzers see higher test case throughputs (total test cases generated) from their indiscriminate mutations (§ 3.2.3), which—while simpler and faster than TETRIS's

conservative mutation and scope/type resolution—produce *lower-quality* programs overall, as evidenced by their diminished code coverage (Table 6). Moreover, on C2Rust, TETRIS achieves validity not far off from language-specific CSmith's 95.57%, underscoring the overall robustness of TETRIS's more generalizable approach.

Interestingly, three fuzzers—TETRIS, Polyglot, and AFL-Compiler-Fuzzer—see lowest validity specifically on C4Go. Upon our investigation, we find C4Go has more limited language-level support than its counterpart, CxGo—for example, failing on `constant` definitions [2], logical NOTs on typedefs [1], and anonymous `structs` [3]—leading to a higher rejection rate of test cases across all fuzzers. Nevertheless, TETRIS's balanced mutation strategies achieve the highest test case validity rate among all general-purpose fuzzers, **highlighting TETRIS's effectiveness across multiple diverse transpiler input languages**.

RQ1: TETRIS trades-off speed for code validity, bringing correctness comparable to language-specific compiler fuzzing to a much wider range of transpilers.

5.3. RQ2: Transpiler Code Coverage

In fuzzing, higher code coverage of the target naturally leads to a higher likelihood of bug discovery. To evaluate all fuzzer's transpiler code coverage, we replay their generated inputs using the standard coverage-tracing tool AFL-QEMU-Cov [15], counting their unique basic blocks reached in each fuzzing trial. We exclude transpilers in this experiment that remain incompatible with AFL-QEMU-Cov coverage tracing: C4Go, Go2Hx, and Zig Translate-C.

Transpiler	TETRIS			CSmith		
	OLD	NEW	CONF	OLD	NEW	CONF
C2Rust	1	0	0	2	0	0
CxGo	0	7	7	n/a	n/a	n/a
C4Go	0	2	2	n/a	n/a	n/a
Go2Hx	1	2	2	n/a	n/a	n/a
HxCpp	0	0	0	n/a	n/a	n/a
HxPy	0	0	0	n/a	n/a	n/a
Zig Translate-C	2	1	1	n/a	n/a	n/a
Total:	**4**	**12**	**12**	**2**	**0**	**0**

TABLE 7: Total old (known), new, and new confirmed bugs. `n/a`: transpiler unsupported by that fuzzer. Fuzzers that are not shown find *zero* bugs.

5.3.1. Results. Table 6 reports the mean five-trial coverage per fuzzer, along with TETRIS's coverage relative to its competitors (Δ COV). Overall, TETRIS surpasses every general-purpose fuzzer, attaining relative coverage improvements of **11–23%**. All Mann-Whitney U comparisons yield statistically-significant differences: apart from CSmith, TETRIS sees statistically-significant *improvements* over all general-purpose fuzzers, with p-values of 0.00794 for all. Moreover, as our appendix coverage plots illustrate (Figure 11), TETRIS consistently outranks general-purpose approaches throughout the entire duration of fuzzing. Though TETRIS faces 10.25% lower coverage than CSmith on C2Rust, it ultimately trades-off a minimal coverage loss for far greater flexibility—**expanding thorough fuzzing to a much wider range of language transpilers**.

RQ2: TETRIS's higher-quality test cases consistently outrank general-purpose fuzzers in transpiler coverage, enabling deeper testing of transpilers' translation logic.

5.4. RQ3: Transpiler Bug Discovery

Lastly, we evaluate TETRIS's ability to uncover translation errors among our seven real-world transpilers. After collecting all fuzzers' test cases, we manually analyze and deduplicate them into a final set of unique transpiler bugs. For all competitors, we apply TETRIS's own correctness-checking procedures (§ 4.5) on their saved test case corpora. Table 7 reports TETRIS's and CSmith's total number of previously-known (old) bugs, newly discovered bugs, and developer-confirmed bugs following our disclosure. We omit all other competitors from Table 7 as each finds *zero* bugs.

5.4.1. Results. Overall, TETRIS's higher validity and coverage enabled its discovery of the most transpiler bugs in our evaluation—16 total, including **12 new bugs** found exclusively by TETRIS, all of which are confirmed by their respective developers. While not shown in Table 7, five TETRIS-found CxGo bugs and one Go2Hx bug are since patched, demonstrating TETRIS's impact in driving real-world transpiler improvements. Comparatively, CSmith finds two bugs in C2Rust, yet both are previously-known defects. Nevertheless, as Table 8 shows, **TETRIS's discovered defects exercise a multitude of source constructs and semantic translation challenges**—demonstrating the breadth and depth of TETRIS's transpiler fuzzing capabilities.

RQ3: TETRIS's high-quality test cases and high transpiler coverage enhance discovery of a broader range of transpiler defects than previous fuzzing methodologies.

5.4.2. Transpiler Bug Case Studies. In the following, we showcase several interesting case studies of TETRIS-discovered bugs that reveal distinct challenges among mainstream source-to-source transpilers.

```
1  x += 100 + (4 < 4.5);
2  x += 9.9 + (4 < 4.5);
```

(a) Original C.

```
1  x += 100 + float32(int(libc.BoolToInt(4 < 4.5)))
2  x += 9.9 + float32(float64(4 < 4.5)) Error!
```

(b) CxGo Go Output.

Figure 8: Type conversion bug for `bool→float` in Go.

CxGo: Implicit Bool-to-Float Conversion. Figure 8 shows a CxGo bug stemming from an unhandled conversion between boolean and floating-point types. In C, relational expressions (e.g., `4 < 4.5`) evaluate to `1` (true) or `0` (false), allowing implicit conversion to floats within expressions (e.g., `(4 < 4.5) + 9.9`). In contrast, Go does *not* permit direct boolean-to-float conversion, yet CxGo erroneously attempts to *cast* such expressions as floats, thereby resulting in compilation errors in the transpiled

Transpiler	Bug Type	How Detected	Brief Error Description	Iss. ID	New?
C2Rust	Syntactical	Post-translation Failure	Mis-recovering bitfields contained in `union`	#881	
CxGo	Syntactical	Post-translation Failure	Mis-recovering 3-D `array` element pointer deref	#78	✔
CxGo	Syntactical	Post-translation Failure	Mis-recovering compound literal (e.g., `&(int)1`)	#81	✔
CxGo	Type Conversion	Post-translation Failure	Mis-recovering implicit conversion (`bool` → `float`)	#76	✔
CxGo	Type Conversion	Post-translation Failure	Mis-recovering explicit conversion (`bool` → `float`)	#75	✔
CxGo	Type Conversion	Post-translation Failure	Mis-recovering explicit conversion (`float` → `int`)	#79	✔
CxGo	Code Fragment	Runtime Divergence	Mis-recovering operations on `static` variable	#77	✔
CxGo	Code Fragment	Intra-translation Failure	Crash parsing unreachable `switch` body code	#80	✔
C4Go	Type Conversion	Post-translation Failure	Unsupported type conversion (`bool` → `int`)	#516	✔
C4Go	Code Fragment	Post-translation Failure	Unrecovered initialization of value in `union`	#515	✔
Go2Hx	Syntactical	Intra-translation Failure	Unrecovered `generic` types for function args	#178	✔
Go2Hx	Code Fragment	Runtime Divergence	Mis-recovering arch-specific type (`haxe.Int64`)	#26	
Go2Hx	Code Fragment	Runtime Divergence	Mis-recovering zeroed `array` comparisons	#179	✔
Zig Translate-C	Syntactical	Post-translation Failure	Mis-recovering bitfields contained in `struct`	#1499	
Zig Translate-C	Syntactical	Post-translation Failure	Mis-including unreachable post-`return` code	#19716	
Zig Translate-C	Code Fragment	Post-translation Failure	Unrecovered compound literal (e.g., `&(int)1`)	#21855	✔

TABLE 8: TETRIS-found transpiler bugs, how they were detected (§ 4.5), and corresponding GitHub issue IDs. ✔= bug is newly found by TETRIS.

code. However, CxGo *correctly* handles boolean-to-*integer* conversions by using an intermediate integer cast (e.g., `int(libc.BoolToInt(4 < 4.5))`). Yet, while such approach works for integers, it fails when booleans interact with floating points—though reapplying similar casting could resolve this. By examining these complex data-type relationships, TETRIS identifies numerous implicit and explicit conversion errors, uncovering key blind spots in mainstream transpilers' implementations.

```
return x += y*3;
x++;
return 0 ;
```
(a) Original C.

```
return blk: {
  const ref = &x;
  ref.* += y * @as(c_int,3);
  break :blk ref.*;
};
x += 1;
return 0;
```
(b) Zig Translate-C Output.

Figure 9: Zig Translate-C's mis-inclusion of unreachable post-return code.

Zig Translate-C: Unreachable Code. Beyond type errors, TETRIS uncovers numerous parsing-related edge cases in transpilers. In Figure 9, TETRIS generated a C program with several lines of dynamically-*unreachable* code following the initial `return` statement. Given Zig's strict enforcement against unreachable code, Translate-C focuses extensively on finding and culling unreachable code to ensure successful compilation, as evidenced by its past fixes of related bugs (e.g., GitHub issues #7585, #8002, #15155). However, Translate-C mistakenly retains the post-`return` code in TETRIS's generated program, leading to compile errors on the transpiler-emitted Zig code. Thus, TETRIS's ability to stress-test subtle, edge-case semantics helps uncover bugs in areas where considerable transpiler development effort has been spent, empowering transpiler creators with a better, more thorough means of vetting their tools' correctness.

```
x := [4]int{0, 0, 0, 0}
if x != [4]int{0, 0, 0, 0} {
    panic("Error: not equal!")
}
```
(a) Original Go.

```
var x = new Array<Int>([0, 0, 0, 0]);
if (x != new Array<Int>([0, 0, 0, 0])) {
    throw "Error: not equal!";
}
```
(b) Go2Hx Haxe Output.

Figure 10: Go2Hx's mis-translation of comparisons on zeroed arrays.

Go2Hx: Comparing Two Zeroed Arrays. In Go, comparing arrays value-by-value is straightforward; two arrays with identical elements, such as `[4]int{0, 0, 0, 0}`, are considered equal. Yet, after translating this code to Haxe via Go2Hx, a runtime error occurs as Haxe instead compares arrays by *reference*—rather than by value (Figure 10). This difference creates unexpected behavior: a Go expression such as `x != [4]int0, 0, 0, 0`, which would evaluate to *false*, is instead translated into a Haxe expression evaluating to *true* due to reference inequality. Ultimately, TETRIS identifies a non-trivial challenge in Go2Hx's translation of Go array semantics to Haxe, as a faithful translation requires implementing *element*-wise comparison support.

Security Implications of Transpiler Bugs. Translation errors indeed can undermine security of translated code. For example, a previously-fixed bug in sudo's Rust port introduced an information-leakage vulnerability not present in sudo's C code [4]. To this end, TETRIS fills an important gap in the ability to vet transpilers against these (and other) security-affecting translation errors. For example, a TETRIS-found Go2Hx bug (Figure 10) introduces erroneously-inverted conditional checks; subtle data-logic-affecting bugs like these may introduce unexpected crashes (i.e., denial-of-service vulnerabilities)—or worse, create opportunities for attacks exploiting unintended differences from the original code. We anticipate TETRIS and future transpiler-testing efforts will be critical as our world increasingly embraces transpiler-driven automated code translation—with initiatives like DARPA's TRACTOR [11] highlighting this shift as already on the horizon.

Takeaways: Through rigorous exploration of language edge cases relating to type conversions, control flow, data structures, and more, TETRIS reveals complex translation challenges often missed by transpiler developers. This breadth facilitates TETRIS's discovery of **both obvious *and* subtle translation errors**, revealing critical gaps in transpilers' correctness.

6. Discussion

In the following, we weigh several limitations of our approach and prototype implementation, TETRIS.

6.1. Supporting Other Transpiled Languages

Deploying TETRIS to other transpilers requires a specification comprising of two components: a BNF grammar with accompanying semantic annotations. While grammars are often freely available, semantic annotations remain a manual task. Our per-language semantics span several hundred lines of code (Table 3), however, most are boilerplate—such as specifying basic type compatibilities (Figure 7)—requiring minimal effort to craft. In practice, we spent only a few hours integrating C, Go, and Haxe into TETRIS.

While TETRIS targets statically-typed languages like C, it currently lacks support for dynamically-typed ones like Python and JavaScript. Ensuring correctness for dynamically-typed languages requires additional guardrails to prevent runtime type errors: for example, an `int` variable later erroneously treated as a `string`, producing an invalid program. To this end, we envision leveraging recent advancements in type inference [41], [8] to recover declared variables' types—and from there, allowing TETRIS's mutation resolution (Figure 5) to piece-together appropriately typed expressions as it currently does for statically-typed languages. We anticipate these changes will primarily center on TETRIS's Scope and Type System (§ 4.3), with optimizations essential to prevent excessive runtime type-checking overhead. We leave exploring this to future work.

6.2. Supporting Other Language Features

Although our goal is to thoroughly test transpilers' correctness across a language's full feature set, we defer scrutinizing their handling of more advanced language features (e.g., Rust's borrow-checker, Haxe's macros) to future work, as we observe that our evaluated transpilers cannot yet handle them. For example, C2Rust [26] lacks support for Rust's ownership model, while Go2Hx [54] emits rudimentary Haxe without macros. Nevertheless, by uncovering 12 new bugs, our work shows that even common-case language constructs remain problematic for current transpilers, underscoring the need for approaches like TETRIS to rigorously test their handling of these features.

6.3. Unspecified and Undefined Behavior

Although TETRIS finds many transpiler bugs, it cannot avoid certain undesirable semantic behaviors inherent to the target transpiler's input language. For example, we see several cases where TETRIS-discovered runtime divergences are actually from unspecified operation ordering rather than transpiler errors. We also encountered some instances of undefined behavior such as integer overflows in TETRIS-generated C programs; and while such errors are catchable via static analyzers like `UndefinedBehaviorSanitizer` [47], similar tools are not common in other languages such as Haxe.

While we foresee potential for more semantic guardrails in TETRIS—as CSmith [58] and other language-specific fuzzers use—we leave this engineering to future work. In practice, we easily filtered-out such test cases through automated scripts during our post-fuzzing bug triage.

6.4. Test Case Throughput

As Table 5 shows, TETRIS generates fewer test cases on average compared to general-purpose fuzzers AFL++, AFL-Compiler-Fuzzer, and Polyglot. This lower throughput is largely due to the overhead of TETRIS's mutation engine, which retains numerous source constructs in memory; and its pre-transpilation validation step, where each generated program is tested. While TETRIS's throughput is on-par with compiler-dedicated fuzzer CSmith, we see potential for several performance optimizations in TETRIS—such as restructuring our mutation engine to leverage multi-threading—but leave the requisite engineering and re-evaluation to future work. Ultimately, as TETRIS is built atop the AFL fuzzer [16], which already supports parallelization, we posit that TETRIS is also easily parallelized for larger-scale transpiler fuzzing.

6.5. Ethical Considerations

Like other fuzzers, TETRIS carries a dual-use risk: while intended to strengthen the correctness and security of transpilers, it could also be misused to identify vulnerabilities in transpiled code or in the transpilers themselves. To mitigate this risk, we will release TETRIS as an open-source artifact with ethical usage guidelines that emphasize responsible disclosure, ensuring that transpiler developers and the broader research community benefit from its use. These guidelines will align with established security community norms for coordinated vulnerability disclosure.

7. Conclusion

Transpilers' automated source-to-source translation is increasingly driving cross-platform interoperability, technology migration, and security hardening of legacy code. Unfortunately, the complexity of transpiling often leads to insidious errors compromising translation accuracy, resulting in broken or otherwise incorrect code. To expedite discovery of transpiler bugs, we present the first generic, language-agnostic transpiler fuzzing framework, capable of generating high-quality test cases comparable to those from specialized, language-specific fuzzers. Our evaluation demonstrates that this approach achieves superior code coverage, test case validity, and bug discovery compared to prior techniques—uncovering **12 new transpiler bugs**, with all confirmed.

Acknowledgments

This material is based upon work supported by the National Science Foundation (NSF) under Award No. 2419798, and by the Defense Advanced Research Projects Agency (DARPA) under Award No. FA8750-24-2-0002, Subaward No. GR105409-SUB00001384. 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 NSF or DARPA.

References

[1] C4go issue #381: operation not: typedef. https://github.com/Konstantin8105/c4go/issues/381.

[2] C4go issue #469: #define as constant. https://github.com/Konstantin8105/c4go/issues/469.

[3] C4go issue #51: F: struct declaration. https://github.com/Konstantin8105/c4go/issues/51.

[4] sudo-rs issue #577: Don't leak information about what files exist on filesystem. https://github.com/trifectatechfoundation/sudo-rs/issues/577.

[5] Bastidas F. Andrés and María Pérez. Transpiler-based architecture for multi-platform web applications. In *IEEE Ecuador Technical Chapters Meeting*, ETCM, 2017.

[6] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, ATC, 2005.

[7] Alexander Bergmayr, Hugo Brunelière, Javier Luis Cánovas Izquierdo, Jesús Gorroñogoitia, George Kousiouris, Dimosthenis Kyriazis, Philip Langer, Andreas Menychtas, Leire Orue-Echevarria, Clara Pezuela, and Manuel Wimmer. Migrating Legacy Software to the Cloud with ARTIST. In *European Conference on Software Maintenance and Reengineering*, CSMR, 2013.

[8] Giuseppe Castagna, Mickaël Laurent, and Kim Nguy˜ên. Polymorphic type inference for dynamic languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2024.

[9] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. One engine to fuzz'em all: Generic language processor testing with semantic validation. In *IEEE Symposium on Security and Privacy*, Oakland, 2021.

[10] Robert Corbett. GNU Bison: a general-purpose parser generator. https://www.gnu.org/software/bison/, 2024.

[11] DARPA. TRACTOR: Translating all c to rust. https://www.darpa.mil/research/programs/translating-all-c-to-rust, 2024.

[12] Mehmet Emre, Peter Boyland, Aesha Parekh, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Aliasing limits on translating c to safe rust. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2023.

[13] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Translating c to safer rust. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2021.

[14] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2023.

[15] Andrea Fioraldi. Afl-qemu-cov: Measure basic blocks coverage of all testcases in the afl queue using a patched qemu. https://github.com/andreafioraldi/afl-qemu-cov, 2024.

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies*, WOOT, 2020.

[17] Fusion Language. Fusion programming language: Transpiling to c, c++, c#, d, java, javascript, python, swift, typescript and opencl c. https://github.com/fusionlanguage/fut, 2024.

[18] Google. Gofuzz. https://github.com/google/gofuzz, 2022.

[19] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, Mikel Luján, and Hanspeter Mössenböck. Cross-language interoperability in a multi-language runtime. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(2), 2018.

[20] Alex Groce, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. Making no-fuss compiler fuzzing effective. In *ACM SIGPLAN International Conference on Compiler Construction*, CCC, 2022.

[21] Haxe Foundation. Haxe: Casting. https://haxe.org/manual/expression-cast.html, 2024.

[22] Haxe Foundation. Hxcpp: Runtime files for c++ backend for haxe. https://github.com/HaxeFoundation/hxcpp, 2024.

[23] Haxe Foundation. Hxpy: Native bindings for python. https://github.com/Vortex2Oblivion/hxpy, 2024.

[24] Jaemin Hong and Sukyoung Ryu. Concrat: An automatic c-to-rust lock api translator for concurrent programs. In *IEEE/ACM International Conference on Software Engineering*, ICSE, 2023.

[25] Immunant. C2rust: Csmith testing scripts. https://github.com/immunant/c2rust/blob/master/scripts/csmith.py, 2024.

[26] Immunant. C2rust: Migrate c code to rust. https://github.com/immunant/c2rust, 2024.

[27] Konstantin Izyumov. C4go: Transpiling c code to go code. https://github.com/Konstantin8105/c4go, 2024.

[28] Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. Mono2Micro: a practical and effective tool for decomposing monolithic Java applications to microservices. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, 2021.

[29] Andrew Kelley. Zig translate-c: automatic translation from c source code. https://zig.guide/working-with-c/translate-c/, 2024.

[30] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2018.

[31] Jarrett Kuklis. Portablegl translated with cxgo. https://github.com/TotallyGamerJet/pgl, 2024.

[32] Per Larsen. Migrating c to rust for memory safety. *IEEE Security & Privacy*, 22(04), 2024.

[33] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, CGO, 2004.

[34] Kornel Lesinski. Citrus: Convert c to rust. https://gitlab.com/citrus-rs/citrus, 2024.

[35] Zhibo Liu and Shuai Wang. How far we have come: testing decompilation correctness of C decompilers. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2020.

[36] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2020.

[37] LLVM. Libclang. https://clang.llvm.org/docs/LibClang.html, 2024.

[38] LLVM. LibTooling: a library to support writing standalone tools based on clang. https://clang.llvm.org/docs/LibTooling.html, 2024.

[39] Bruno Góis Mateus, Matias Martinez, and Christophe Kolski. Learning migration models for supporting incremental language migrations of software applications. *Information and Software Technology*, 153, 2023.

[40] Sebastian McKenzie. Babel: a javascript compiler. https://babeljs.io/docs/, 2024.

[41] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4py: Practical deep similarity learning-based type inference for python. In *International Conference on Software Engineering*, ICSE, 2022.

[42] Nim Language. C2nim: a tool to translate ansi c code to nim. https://github.com/nim-lang/c2nim, 2024.

[43] Gabriel Noaje, Christophe Jaillet, and Michaël Krajecki. Source-to-Source Code Translator: OpenMP C to CUDA. In *IEEE International Conference on High Performance Computing and Communications*, HPCC, 2011.

[44] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. N-version disassembly: differential testing of x86 disassemblers. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2010.

[45] Vern Paxson. Flex: The fast lexical analyzer. https://github.com/westes/flex, 2024.

[46] Micha Reiser and Luc Bläser. Accelerate javascript applications by cross-compiling to webassembly. In *ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL, 2017.

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, ATC, 2012.

[48] Kosta Serebryany. Continuous fuzzing with libfuzzer and address-sanitizer. In *IEEE Cybersecurity Development Conference*, SecDev, 2016.

[49] Kostya Serebryany. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *USENIX Security Symposium*, USENIX, 2017.

[50] Mayank Sharma, Pingshi Yu, and Alastair F Donaldson. Rustsmith: Random differential compiler testing for rust. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2023.

[51] Denys Smirnov. Cxgo: Booltoint(). https://github.com/gotranspile/cxgo/blob/main/runtime/libc/conv.go, 2024.

[52] Denys Smirnov. Cxgo: Tool for transpiling c to go. https://github.com/gotranspile/cxgo, 2024.

[53] Denys Smirnov. Potrace translated with cxgo. https://github.com/gotranspile/gotrace, 2024.

[54] Elliott Stoneham. Go2hx: Go to haxe source-to-source compiler. https://github.com/go2hx/go2hx, 2024.

[55] The Go Programming Language. Go: Operators. https://go.dev/ref/spec#Operators, 2024.

[56] The Go Project. cc/v3: a c99 compiler front end. https://pkg.go.dev/modernc.org/cc/v3, 2024.

[57] Dmitry Vyukov. Gosmith. https://github.com/dvyukov/gosmith, 2015.

[58] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI, 2011.

[59] Zao Yang and Stefan Nagy. Bin2Wrong: a Unified Fuzzing Framework for Uncovering Semantic Errors in Binary-to-C Decompilers. In *USENIX Annual Technical Conference*, ATC, 2025.

[60] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA, 2011.

[61] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. Squirrel: Testing database management systems with language validity and coverage feedback. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2020.

[62] Zig Software Foundation. Zig: If expressions. https://zig.guide/language-basics/if/, 2024.

Appendix

Appendix: Code Coverage Graphs

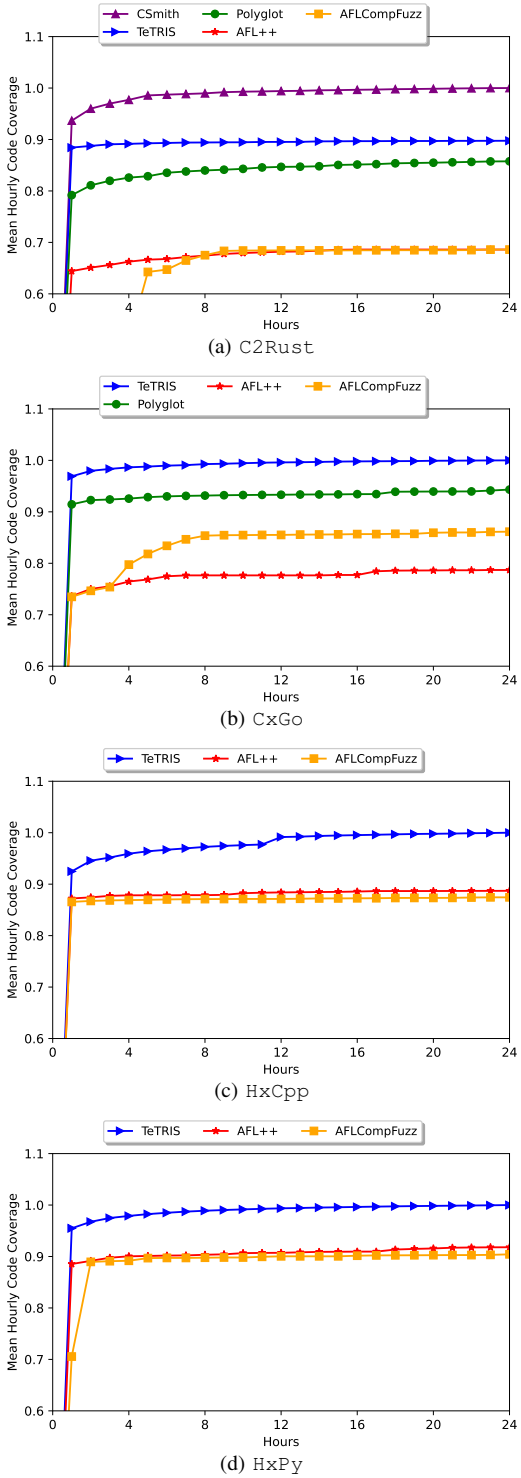


Figure 11: Mean hourly code coverage plots per fuzzer for each supported transpiler. We omit transpilers incompatible with coverage tracing.