PROGnosticator: Testing Source-to-Source Code Translators via Construct-Oriented Fuzzing

YEASEEN ARAFAT, University of Utah, USA
STEFAN NAGY, University of Utah, USA

To ease software interoperability and migration, developers are increasingly embracing *transpilers*: automated tools for converting source code from one language to another. Unfortunately, differences in language constructs, syntax, and semantics leave transpilers facing many translation bugs, emitting incorrect or non-functional translations. Thorough, proactive transpiler testing is thus critical to the reliability of emergent translation-oriented development. While fuzzers excel in testing adjacent classes of language processors (e.g., compilers), current fuzzers remain tied to only the specific code patterns expressed in their inputs—hardcoded grammars or seed programs—which are costly to curate and extend, constraining their testing to just narrow subsets of language constructs. Worse yet, their generated programs are often overly complex, requiring non-trivial reduction to pinpoint the exact code patterns behind transpiler errors. Evaluating current and future transpilers thus demands a rigorous, input-independent fuzzing strategy—systematically exercising languages' broad range of code constructs *without* needing costly per-language expertise or re-engineering.

To bridge this gap, we present *Construct-oriented Fuzzing*: a language-agnostic yet construct-aware approach for systematically testing transpilers. Motivated by our insights from past transpiler bugs, revealing most translation errors embody construct-specific mishandling, our approach explicitly targets the vast space of code patterns derived from core language constructs. Harnessing large language models' code understanding and synthesis, we (1) automatically enumerate a language's core constructs, before (2) generating self-contained programs exercising them individually—and combinations thereof—precisely testing transpilers' many edge-cases whilst eschewing cumbersome grammars or seeds. In evaluating our prototype, PROGnosticator, against four state-of-the-art compiler and transpiler fuzzers across seven transpilers for C, Go, and JavaScript, we show how our approach attains high per-language validity as well as construct-usage diversity—exposing 77 total transpiler bugs, of which 64 are previously unknown, with 63 since confirmed or fixed by developers.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

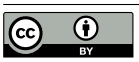
Additional Key Words and Phrases: Fuzzing, Transpilers, Differential Testing

ACM Reference Format:
Yeaseen Arafat and Stefan Nagy. 2026. PROGnosticator: Testing Source-to-Source Code Translators via Construct-Oriented Fuzzing. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE040 (July 2026), 23 pages. https://doi.org/10.1145/3797135

1 Introduction

Transpilers (or source-to-source translators) are a growing class of tools built for automatically converting software from one programming language to another. With continued evolution of modern language ecosystems, transpilers now see widespread adoption in various software domains, including web app development (e.g., Emscripten [83], Babel [17]), software migration (e.g., IBM's Mono2Micro [51]), security retrofitting (e.g., DARPA's TRACTOR [36] initiative), and cross-platform interoperability (e.g., Haxe [23]). Among popular transpilers are efforts translating C to Go [73],

Authors' Contact Information: Yeaseen Arafat, University of Utah, Salt Lake City, USA, y.arafat@utah.edu; Stefan Nagy, University of Utah, Salt Lake City, USA, snagy@cs.utah.edu.



ACM 2994-970X/2026/7-ARTFSE040
https://doi.org/10.1145/3797135

Rust [47], and Zig [53]; Go to Haxe [74], and WASM [27]; as well as translation between different versions of JavaScript [17, 26]—supporting a wide range of cross-language translation needs.

Importantly, transpilers must carefully preserve original programs' functionality while upholding the syntactic and semantic standards of their targeted languages. However, not all language constructs have direct one-to-one mappings to other languages, making translation inherently fraught with difficulty. For example, C's variable-length arrays are non-existent in Rust, necessitating alternative implementations for accommodating dynamically-sized arrays. Similarly, Go-unique implicit interfaces remain unsupported in Haxe. These and other language-to-language disparities introduce substantial risk of translation errors, leading to mistranslated or otherwise broken code—underscoring the need for proactive methods to rigorously assess transpilers' correctness.

Fuzzing, an automated testing approach based on random input generation, has been highly successful for other language processors [34, 81, 82], making it a promising candidate for vetting transpilers. Unfortunately, existing fuzzers see only limited success on transpilers: they rely heavily on hardcoded language grammars [81] or curated seed programs [40, 41], which are not only labor-intensive to obtain and extend, but also inherently restrict their generated programs to a narrow range of supported languages and language constructs—leaving large portions of transpilers' source-language handling unexplorable. Worse yet, many fuzzers are further hindered by the high density and complexity of their generated programs, complicating efforts to pinpoint the specific code patterns behind transpiler crashes or mistranslations. Effective transpiler testing therefore demands a fundamentally different approach—one that is **input-independent, language-agnostic, and capable of systematically exercising language constructs in a targeted, interpretable way**.

To address these challenges and enable practical transpiler testing, we introduce *Construct-oriented Fuzzing*: a method for exhaustively vetting transpilers' fidelity on the vast space of code patterns rooted in core input-language constructs. Guided by our insights from prior transpiler defects, Construct-oriented Fuzzing automatically generates minimal programs exercising individual constructs—and interactions thereof—effectively revealing incorrect handling or omission of specific language features. To realize our approach, we leverage large language models (LLMs) for their code understanding and synthesis capabilities, inheriting syntax, semantics, and usage patterns of diverse programming constructs from pretraining on vast code corpora and language specifications. Namely, our approach harnesses LLMs to perform (1) construct exploration, enumerating a wide range of language patterns, followed by (2) construct-oriented program generation to synthesize valid, self-contained programs targeting these patterns—**eliminating the need for seed inputs, hardcoded language specifications, predefined grammars, or post-hoc program reduction**.

We implement Construct-oriented Fuzzing as a prototype, PROGnosticator, and evaluate it on seven transpilers spanning six different input–output language pairs: **C2Rust** [47] (C→Rust), **CxGo** [73] (C→Go), **Zig Translate-C** [53] (C→Zig), **Go2Hx** [74] (Go→Haxe), **TinyGo** [27] (Go→WASM), as well as **Babel** [17] and **SWC** [26] (ES6+→ES5 JavaScript). We compare PROGnosticator against four state-of-the-art compiler and transpiler fuzzers: Polyglot [34], AFL-Compiler-Fuzzer [41], CSmith [81], and TransFuzz [33]. Across five 24-hour campaigns per transpiler, PROGnosticator attains the highest code pattern diversity—exceeding AFL-Compiler-Fuzzer by **23.26**×, CSmith by **4.23**×, Polyglot by **4.15**×, and TransFuzz by **3.85**×—while maintaining high language validity across its test cases. Moreover, PROGnosticator finds the most transpiler bugs—**77** in total—including **64 previously unknown bugs, with 63 since confirmed or fixed** by developers.

In summary, this paper makes the following contributions:

- We introduce *Construct-oriented Fuzzing*: the first systematic, language-agnostic fuzzing approach for evaluating the correctness of source-to-source transpilers, motivated by our empirical analysis of previously reported transpiler bugs that reveals the dominant cause of translation errors is the mishandling of code exercising core input-language constructs.

- We show how our approach uses LLMs' code-reasoning abilities to automate both the exploration of core construct-usage patterns, and the generation of valid, minimal test programs that explicitly target them—eliminating the need for seed inputs, formal specifications, or program reduction.
- We implement this approach in a prototype tool, PROGnosticator, which leverages GPT-4 to test the correctness of source-to-source code translators.
- We evaluate PROGnosticator on seven real-world transpilers against four state-of-the-art fuzzers, showing that it achieves the highest construct diversity while maintaining high validity and uncovering the most bugs—77 in total, including 64 previously unknown and 63 confirmed.
- We open-source our prototype of PROGnosticator as well as our evaluation benchmarks and artifacts at the following URL: https://github.com/FuturesLab/PROGnosticator.

2 Background & Motivation

This section outlines the fundamentals of source-to-source transpilers and their underlying bugs, as well as the limitations of existing fuzzing techniques relevant to testing transpilers.

2.1 Transpilers and Transpiler Bugs

Transpilers are a class of tools for converting source code from one programming language to another, while preserving the original software's full functionality in the new language. At its core, transpiling involves three key steps [32]: (1) parsing the input source code into an abstract syntax tree (AST), (2) translating it into an intermediate representation that bridges the input-to-output language divide, and (3) applying language-specific refinements before ultimately generating the output code. While this high-level workflow is standard across most transpilers (e.g., C2Rust [47], Go2Hx [74]), transpiling in practice is fraught with challenges that typically fall into three categories of failures: **transpiler crashes**, **invalid code**, and **semantic divergences**. We detail each below.

```c
1 #define foo(a) (a ? ((a ? 1 : 2) | 1) : 1)
2 int iflags = foo(1);
```

(a) Original C input.

```rust
1 let mut iflags: libc::c_int =
2 if 0 != 1i32 {
3 if 0 != 1i32 {1i32} else {2i32} | 1i32
4 } else {1i32};
```

(b) C2Rust's Rust output.

Fig. 1. **Invalid code:** C2Rust's mistranslation of nested conditional macros, resulting in an unmatched closing brace that impedes the program from being compilable.

```go
1 x := [2]int{0,0}
2 if x != [2]int{0,0} { ... }
```

(a) Original Go input.

```haxe
1 var x = new Array<Int>([0,0]);
2 if (x != new Array<Int>([0,0])) { ... }
```

(b) Go2Hx's Haxe output.

Fig. 2. **Semantic divergence:** Go2Hx's mistranslation of Go array comparisons, causing the conditional check to wrongly return *true*.

- **Transpiler Crashes:** Transpiler crashes are failures in which translation unexpectedly halts prior to or during output code generation. Crashes may arise from malformed inputs that violate the input language's syntax, but more critically from valid yet *unsupported* input language constructs triggering front-end errors or internal exceptions within the transpiler. For example, CxGo aborts [3] when parsing C code such as int a = sizeof(int), emitting an undefined: size_t error due to its failure to resolve the standard size_t typedef during front-end parsing.
- **Invalid Code:** Invalid-code failures occur when translation finishes, yet the translated code is syntactically-incorrect or incomplete—such as missing declarations, malformed constructs, or misplaced tokens—and subsequently cannot be compiled and/or run. For example, in Figure 1a, the C macro on line #1 defines a nested conditional macro that evaluates to ((a ? 1 : 2) | 1) when a is nonzero and 1 otherwise. Yet, line #2 of C2Rust's translated code in Figure 1b omits parentheses around the inner if statement before the bitwise-or operator. As a result, | binds at the wrong level, leaving an unmatched closing brace and producing an unexpected token } syntax error [1].

- **Semantic Divergences:** Semantic divergences emerge when translated code compiles and runs error-free, but exhibits different *runtime* behavior than the original input-language program. For example, Go arrays are compared element-by-element, so both [2]int{0, 0} arrays in line #2 of Figure 2a are considered equal. However, Go2Hx's Haxe translation instead uses *reference*-level rather than element-level array comparison, causing the same expression—which evaluates to *false* in Go—to instead evaluate to *true* in Haxe (Figure 2b) due to distinct object references [15].

In practice, transpiler failures directly threaten the reliability of transpiled software. As transpilers evolve to increasingly take on much-needed tasks such as porting critical libraries and legacy systems [35, 37, 69, 75], undetected errors risk allowing subtle defects to slip into production—**underscoring the need for a rigorous automated testing methodology that exposes transpilers' language-level errors *before* they compromise the quality of translated software**.

2.2 Transpiler-relevant Fuzzing Techniques

Fuzzing, among today's most widely used automated testing techniques [40], has long proven effective in testing many important classes of language processors such as code compilers [39, 59] and decompilers [58, 82]. Broadly, current fuzzing techniques for language processors fall under one of two categories: language-*specific* and language-*agnostic*. We briefly detail these below.

- **Language-specific Fuzzers:** Language-specific fuzzers fall into two subtypes: *generational*, which synthesize programs using built-in language grammars and semantic constraints; and *mutational*, which start from pre-created "seed" programs and iteratively apply syntax- and semantics-conformant transformations to derive new programs. On the generational side, CSmith [81] and YARPGen [59]—both widely used to fuzz C compilers (e.g., LLVM [56]), with CSmith also reportedly used by the developers of C-to-Rust transpiler C2Rust [46]—implement a significant portion of the C/C++ specification, enabling them to generate valid test cases out of the box. Similarly, GoSmith [78] targets Go compilers such as gc [20]. On the mutational side, TransFuzz [33] applies ECMAScript-conformant mutations to pre-created JavaScript seed programs, proving effective in testing JavaScript-converting transpilers such as Babel [17] and SWC [26].
- **Language-agnostic Fuzzers:** Unlike language-specific fuzzers, which commonly encode the semantics of just a single programming language, language-agnostic fuzzers are designed to work across *many* languages. They typically start from existing seed programs and apply random mutations, but differ in what code-structure analysis they implement. AFL-Compiler-Fuzzer [41], for example, augments fuzzing-standard mutation strategies (e.g., token-level deletions, insertions, and splicing) with syntax awareness through regular expressions. Polyglot [34] goes further by lifting seed programs into a common abstract syntax tree (AST) derived from user-provided grammars, enabling AST-level mutations that preserve structural validity across diverse languages. Historically, these techniques have revealed many front-end-crashing bugs in compilers [34, 41].

2.3 Limitations of Existing Fuzzers for Transpiler Testing

Despite the successes of both language-specific and language-agnostic fuzzers on other classes of language processors, **existing tools remain universally ill-suited for testing transpilers**. Following our survey in Table 1, we examine the tradeoffs of these approaches and the key obstacles that limit their effectiveness in uncovering transpilers' underlying code translation bugs.

- **Challenge 1: Broad Language Support.** Language-specific fuzzers are tightly coupled to their target languages, which in practice requires developing a distinct fuzzer for *each* transpiler's source language. Yet, designing and implementing language-specific fuzzers is non-trivial given their complexity (e.g., CSmith consists of **80K** lines of code [81]) and the growing number of transpilers today [49]. In contrast, despite their generality, language-agnostic fuzzers such

Fuzzer	Broad Support	Seed Agnostic	Program Gen. Validity	Program Gen. Reduced	Code Diversity
AFL-Comp-Fuzz [41]	✔	✗	✗	✔	✗
Polyglot [34]	✗	✗	✗	✔	✗
YARPGen [59]	✗	✔	✔	✗	✗
CSmith [81]	✗	✔	✔	✗	✗
GoSmith [78]	✗	✔	✔	✗	✗
TransFuzz [33]	✗	✗	✔	✔	✗
PROGnosticator	✔	✔	✔	✔	✔

Table 1. State-of-the-art language processor fuzzers and their key tradeoffs with respect to testing transpilers.

Construct Origin	Problematic Patterns
C Language Core Total bugs: **57**	Function-like macros; Bitfield/packed structs; Type mismatches; String operations; Strings within macros
Platform Specific Total bugs: **21**	SIMD intrinsics; Calling conventions; Built-in functions; Kernel APIs/macros

Table 2. Taxonomy of prior bugs in the C2Rust [47] transpiler by construct.

as Polyglot [34] still require detailed grammar annotations and semantic specifications per each supported language, limiting their scalability. Yet in our experiments (§ 4.1.4), even *with* a compatible Go specification, Polyglot still failed to successfully parse Go code—underscoring the difficulty of supporting multiple diverse programming languages within a single unifying fuzzer.

- **Challenge 2: Dependence on Seed Programs.** While some language-specific fuzzers [81] generate programs from scratch, virtually all mutation-based fuzzers (e.g., TransFuzz [33], AFL-Compiler-Fuzzer [41], Polyglot [34]) depend on **pre-created** seed programs. Yet without sufficiently diverse seeds, these fuzzers struggle to cover the full range of language constructs, limiting their effectiveness in systematically testing transpilers. Obtaining seeds that are both valid *and* construct-diverse is highly labor-intensive, often requiring substantial manual curation [2, 65]. For more recent programming languages such as Go, where comparable curated corpora are largely absent, seed availability becomes an even greater bottleneck to transpiler testing.
- **Challenge 3: Generating Valid & Reduced Programs.** In the absence of language-specific guardrails, language-agnostic fuzzers often fail to generate *valid* code conforming to the syntax and semantics of the transpiler's input language. For example, AFL-Compiler-Fuzzer's [41] string-level mutations frequently break syntax (e.g., `var p int = 4` ⟶ `var p inX = 4`); while Polyglot, despite using grammars, faces self-reported program validity **as low as 20%** [34] from semantics-breaking mutations (e.g., deleting a necessary `return`). Such invalid, language-nonconforming programs are overwhelmingly rejected by transpilers, leaving their translation logic untestable by these fuzzers. Moreover, language-specific fuzzers [59, 78, 81] typically emit large, logically dense programs [77], obscuring the code responsible for triggering transpiler bugs. For example, our efforts to reduce a CSmith program that exposed a C2Rust bug [6] via C-Reduce [68] took 17 minutes from **over 2,800** re-transpilation and re-compilation attempts. This challenge is even more significant for modern languages like Go, which currently lack dedicated reducer tooling.
- **Challenge 4: Code Pattern Diversity.** To better understand the nature of translation bugs, we analyzed all of C2Rust's [47] previously reported issues and identified a common root cause: **over 73% of errors** stem from problematic code patterns exercising *core* language constructs. As Table 2 shows, these embody common examples such as string operations, bitfield-containing unions or structs, and function-like macros. However, current fuzzers' coverage of code patterns remains tied to hardcoded grammar rules (e.g., CSmith, GoSmith, YARPGen) or to the code patterns found in their seeds (e.g., AFL-Compiler-Fuzzer, Polyglot, TransFuzz). For example, CSmith supports just a subset of C11, neglecting many constructs in real-world codebases such as variable-length arrays, function pointers, and compound literals. Likewise, AFL-Compiler-Fuzzer and Polyglot are constrained to the syntactic and semantic patterns of their seed programs, covering only a fraction of all possible construct usage. As a result, these fuzzers leave most of the input language's code patterns unexplored, making many transpiler bugs out of their reach.

Motivation: Existing language processor fuzzers derive their power from their predefined grammars or curated seeds, making their effectiveness largely reliant on the richness of these resources. Beyond other limitations (Table 1), this dependence overwhelmingly yields low-diversity programs, leaving many transpiler bugs undiscoverable by existing fuzzing approaches. As we observe that most transpiler bugs stem from distinct code patterns involving core language constructs, addressing this gap requires **a fuzzing approach systematically exploring core language constructs *and* their interactions**—without relying on seeds or grammars.

3 Our Approach: Construct-oriented Fuzzing

To address the mismatch between existing fuzzers and the unique challenges of testing transpilers, we introduce *Construct-oriented Fuzzing*: the first fuzzing technique purpose-built for exploring the core language constructs—and interactions thereof—that underlie most translation bugs in practice. Below, we detail our approach alongside the design of our prototype, PROGNOSTICATOR.

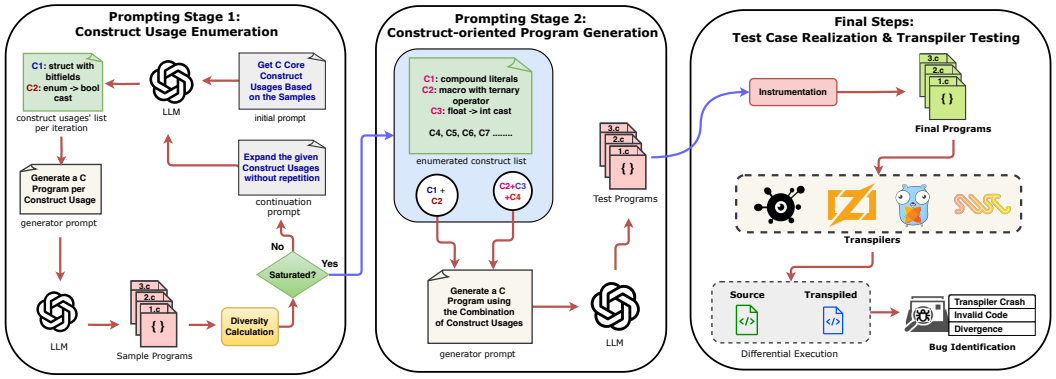


Fig. 3. Flow of *Construct-oriented Fuzzing*: our LLM-driven approach for language-agnostic transpiler testing.

3.1 High-level Overview

Motivated by our observation that most transpiler bugs arise from mishandling specific usage of core language constructs (Table 2), we aim to systematically test transpilers by **(1)** identifying distinct construct-usage code patterns (e.g., nested conditionals, array slicing) and then **(2)** generating programs incorporating them—alone and in combination (e.g., array slicing *within* nested conditionals)—feeding each to the transpiler under test. Whereas prior fuzzers inherit code patterns from pre-defined artifacts (e.g., hardcoded [33, 59, 78, 81] or human-written [34] grammars or seed inputs [33, 34, 41]), our approach aims for **full decoupling from *any* language-level tailoring**.

To this end, we draw on recent advancements in large language models (LLMs), which highlight their potential for *prompt-driven code synthesis* [80]: broadly generating minimized, well-formed programs encompassing specific language features. Accordingly, **we harness LLMs for Construct-oriented Fuzzing through a *two-stage* prompting strategy** (Figure 3), which we detail below.

3.2 Prompting Stage 1: Construct Usage Enumeration

To synthesize code patterns without relying on grammars or seed programs, we design a *multi-round* prompting strategy that, given a desired transpiler input language (e.g., C for C2Rust [47]), queries LLMs for varied usage patterns of its core code constructs. Retrieved patterns are then organized into high-level categories, which are iteratively expanded across subsequent rounds until additional prompting yields diminishing returns in construct-usage diversity. Below, we detail our code pattern grouping, prompting methodology, and approach to detecting code pattern saturation.

3.2.1 Code Pattern Grouping. To better structure LLM-retrieved construct-usage patterns amid today's diverse language syntaxes, we organize them into *two* high-level categories, beginning with **language-*agnostic* patterns** that capture foundational aspects of language behavior:

- **Primitive Data Usage:** Manipulation of primitive constructs (e.g., int, float, bool), including their declaration, initialization, and use in expressions like arithmetic and comparison operations.
- **Composite Data Usage:** Manipulation of composite constructs (e.g., array, map, slice, struct), spanning patterns like indexed access, field assignment, and iteration over collections.
- **Control Flow Usage:** Manipulation of control-flow constructs like functions, if-else, switch-case, and loops (for, while, do-while), such as nested conditionals and loop exits using break.

Building on our retrieval of language-agnostic patterns, we also identify **language-*specific* patterns** encompassing idiomatic features unique to individual languages, including:

- **For C:** Features such as qualifiers (e.g., const, volatile), and preprocessor directives and macros.
- **For Go:** Features such as multiple return values, short variable declarations, type inference, composite literals, and implicit interface implementations.
- **For ES6+ JavaScript:** Features such as object and array destructuring, rest/spread operators, arrow functions, async/await, iterators, and generators.

Through this grouping, we achieve representation of both universal *and* language-specific code patterns, ensuring that neither common nor rarer constructs are overlooked [45]. This foundation sets the stage for our subsequent code pattern retrieval, where prompts are iteratively refined to cover an expanding share of the language—and the complex cases transpilers encounter in practice.

3.2.2 Prompting-driven Pattern Retrieval. As Figure 3 shows, our approach follows a multi-round prompting strategy to broadly explore a language's many distinct construct-usage patterns. Each round, we query the LLM to propose natural-language examples of core construct usage, guided by the grouping described in § 3.2.1, which are then analyzed for their syntactic and structural diversity. This continues until code pattern diversity *stagnates*, per our saturation criterion detailed in § 3.2.3.

You are an expert in the **$LANGUAGE** programming language. Your task is to design a diverse and representative set of code-level core **$LANGUAGE** construct usages for testing transpilers. The goal is to cover a wide range of construct usages—including edge cases and uncommon usages—to ensure transpilers correctly handle various usage patterns in **$LANGUAGE**.

Group these constructs under high-level categories. Some categories are language-agnostic (e.g., Primitive Data Usage, Control Flow Usage), while others are language-specific, such as **Qualifiers** in C, **Interfaces & Embedding** in Go, and **Iterators & Generators** in JavaScript.

For each category, provide a list of short, descriptive phrases that capture typical usage patterns or constructs. These phrases should be brief, like:
"declare int variables" **"switch-case inside a loop"** **"array destructuring"**
"pointer difference" **"float to int cast"**. **"for...of over generators"**

Include both common and distinctive usages, especially those that are semantically rich or known to be challenging for translation. Avoid repeating similar items across categories; each usage example **must** reflect a distinct usage pattern of core constructs within its category.

Return the result as a JSON object in the following format:

{ "Category_1": ["construct usage 1", "construct usage 2"],
"Category_2": ["construct usage 1", "construct usage 2"], ... }

Fig. 4. Structure of our code pattern exploration prompt.

```
Primitive Data: [ "let and const
  variables", "BigInt arithmetic
  operation", "template literals" ],
Control Flow: [ "for...of loop over
  an array", "do..while in function",
  "early return in switch-case" ],
Iterators & Generators: [ "spread (...)
  over iterable", "generator with
  yield", "custom iterator in Symbol" ]
```

Fig. 5. Resulting patterns for ES6+ JS.

Focusing strictly on core language constructs, we begin with an initial **exploration prompt** (shown in Figure 4) that outlines several examples of both language-agnostic and language-specific construct-usage code patterns, and additionally instructs the LLM to return as many distinct examples as possible. Responses are parsed as JSON objects labeled by their § 3.2.1 construct grouping sub-category (e.g., Primitive Data), each containing lists of minimal natural-language descriptions of code patterns utilizing constructs in that grouping (e.g., "BigInt arithmetic operation").

In subsequent rounds, we issue **continuation prompts** that include *all* previously retrieved pattern descriptions, and instruct the LLM to both (1) avoid repeating them and (2) produce 90 *new* construct-usage patterns. Figure 5 showcases several returned code patterns for ES6+ JavaScript.

3.2.3 Diversity-guided Saturation Detection. To determine when further prompting no longer yields meaningful novelty, we introduce a diversity-guided saturation criterion. For each natural-language code pattern description returned by the LLM, we (1) synthesize a minimal program incorporating it in isolation, before (2) extracting the following AST-level structural diversity metrics that quantify the variety and depth of construct usage, aggregated **cumulatively across all prompting rounds**:

- D_1**:** unique standalone construct usage (e.g., a goto statement by itself);
- D_2**:** unique composite two-level nestings (e.g., a goto inside an if block);
- D_3**:** unique three-level nestings (e.g., a goto inside an if, which itself is in a for loop);
- D_4**:** unique four-level nestings (e.g., enclosing the previous pattern in a switch-case);
- D_5**:** unique five-level nestings (e.g., embedding the above within an outer while loop); and
- D_6**:** unique six-or-greater nesting levels (e.g., wrapping the full chain from above in a function).

To determine when to **stop** prompting for new construct-usage patterns, we define *saturation* as the point at which additional rounds plateau in producing novel construct usage. This reflects our observation that, as rounds progress, LLMs increasingly reproduce *previously seen* patterns rather than finding new ones, leading to a diminishing rate of change across our D_1–D_6 diversity metrics.

Accordingly, we quantify saturation as follows: let $D_i^{(r)}$ denote the value of metric D_i at round r, with $i \in \{1, 2, 3, 4, 5, 6\}$. We define the relative growth of each metric as:

$$\Delta D_i^{(r)} = \frac{D_i^{(r)} - D_i^{(r-1)}}{D_i^{(r-1)}},$$

which captures the additional diversity gained in round r relative to the *previous* round. Saturation is declared when relative growth across all six metrics falls below a pre-set threshold ϵ:

$$\forall i \in \{1, 2, 3, 4, 5, 6\}, \quad \Delta D_i^{(r)} < \epsilon.$$

In our prototype, PROGnosticator, we set $\epsilon = 0.10$ (10%) based on observed structural-diversity trends. As shown in our evaluation (Figure 6), diversity growth initially increases and then decays; once it falls below 10%, additional enumeration rounds yield only marginal new construct combinations with negligible structural variation. In our testing, varying this threshold between 5–15% affects construct-usage enumeration cost only marginally. Nevertheless, ϵ is fully user-configurable, enabling users to adjust the trade-off between exploration depth and enumeration cost.

As our subsequent analysis shows (§ 4.2), we observe that the saturation point varies by language: **nine** rounds for C, **six** for Go, and **eight** for ES6+ JavaScript. Each round, including all pattern retrieval (§ 3.2.2) and diversity analysis (§ 3.2.3) steps, requires approximately 10 minutes, yielding a total Construct Usage Enumeration time no greater than **90 minutes** per language.

3.3 Prompting Stage 2: Construct-oriented Program Generation

With construct-usage code patterns in hand, we next begin synthesizing minimal *test case programs* exercising individual and *multiple* patterns at a time. This process reflects our core motivation: transpiler bugs overwhelmingly stem not just from isolated constructs, but also from **the many unique interactions among them** (Table 2). These programs, produced via LLM prompting, form the foundational test cases in Construct-oriented Fuzzing's subsequent transpiler testing stage (§ 3.4).

3.3.1 Generation Prompting. Concretely, our program generation combines *up to three* random construct-usage patterns from our previously enumerated list (§ 3.2.2), as our analysis of prior transpiler bugs indicates that real-world transpiler bugs typically arise from interactions between one or two constructs, and at most three interacting constructs, with diminishing bug-finding returns expected beyond this point. We thus design a *generation prompt* that instructs the LLM

to: **(1)** combine the selected patterns into a *composite* usage; **(2)** wrap the result in a standalone function, producing self-contained programs akin to those generated by prior language-processor fuzzers [58, 78, 81, 82]; and **(3)** add a corresponding natural-language description as a code comment. When auxiliary definitions are required (e.g., C's typedef, Go's interface), our prompt ensures their placement outside the function body and updates the function accordingly.

3.3.2 Prompt Guardrails. To reduce risk of generating invalid code that would otherwise be *rejected* by transpilers, our prompt enforces strict language-specific constraints: for example, any generated C programs must avoid undefined behavior (e.g., uninitialized variables, out-of-bounds array access, invalid pointer dereferencing); while Go programs must avoid construct usage leading to compile-time errors (e.g., unused variables, malformed declarations). Moreover, across all languages, our prompt prohibits main-like functions, which would conflict with our inserted standalone function; I/O operations (e.g., printf() or fmt.Println()), which we introduce only in our final stage (§ 3.4); and nonstandard library headers, which we find seldom produce transpilable code.

3.4 Final Steps: Test Case Realization & Transpiler Testing

After code generation, our mixed-pattern programs are realized as final test cases for the targeted transpiler. These are then submitted for translation, where any encountered failures—**transpiler crashes**, **invalid code**, and **semantic divergences**—are caught and classified accordingly.

3.4.1 Preparation for Differential Testing. Although transpiler crashes and invalid-code errors occur *within* the translation process, semantic divergences can only be observed at *runtime* by comparing the behavior of the original and transpiled code. To detect such errors, we build on the strategy of prior differential fuzzers [58, 59, 81, 82], which instrument test cases with lightweight behavioral oracles. In our approach, the return value of our inserted standalone function (§ 3.3.1) serves as the behavioral oracle for detecting semantic divergences.

To expose this return value at runtime, both the original and transpiled programs are instrumented post-translation with a language-specific main routine that invokes the standalone function, printing its return value accordingly: via printf() for C, fmt.Printf() for Go, and std.debug.print() for Zig. After this, both program variants are ready for semantic comparison via differential execution.

3.4.2 Detecting Transpiler Bugs. With both the original and transpiled test case programs prepared, we finally perform differential execution on each pair to evaluate the target transpiler's correctness. Failures are captured and grouped by the three canonical transpiler bug categories (§ 2.1) as follows:

- **Transpiler Crashes:** detected when the transpiler itself terminates abnormally due to unhandled constructs, internal errors, or assertion violations, typically reflecting front-end issues.
- **Invalid Code:** detected when the transpiler emits target code that fails to compile or otherwise fails to be executed, revealing back-end code-generation flaws.
- **Semantic Divergences:** detected when both the original and transpiled programs execute successfully but produce differing outputs, exposing translation-induced behavioral mismatches.

Following the design of most other fuzzers [40], all bug-triggering test cases are saved to disk for subsequent manual root cause analysis and reporting to transpiler developers.

3.5 Implementation: PROGnosticator

We implement Construct-oriented Fuzzing as a prototype transpiler fuzzer, PROGnosticator, spanning approximately 1,100 lines of Python code. For Stage-1 Construct Usage Enumeration (§ 3.2) and Stage-2 Construct-oriented Program Generation (§ 3.3), PROGnosticator leverages OpenAI's GPT-4 model [30] which we call programmatically via OpenAI's Python API [24]. LLM queries are issued through a lightweight wrapper that manages structured inputs, response parsing, and

retry logic, ensuring robustness across long-running campaigns. For measuring D_1–D_6 structural diversity (§ 3.2.3), we extract each program's abstract syntax trees (AST) using the Tree-sitter code AST parser [29], and implement our own logging routines to enumerate unique subtree patterns.

Our Solution: By decoupling program generation from predefined grammars and seeds, Construct-oriented Fuzzing enables unbiased exploration of a language's vast space of code patterns, allowing even subtle corner cases to emerge naturally during testing. Collectively, these capabilities position Construct-oriented Fuzzing as a principled and practical methodology for systematically uncovering correctness flaws in today's diverse source-to-source transpilers.

4 Evaluation

We evaluate Construct-oriented Fuzzing through our prototype implementation, PROGnosticator, guided by the following high-level research questions:

- **RQ1:** Can PROGnosticator capture diverse code patterns without grammars or seeds?
- **RQ2:** Does PROGnosticator yield higher-quality programs compared to other fuzzers?
- **RQ3:** Is PROGnosticator capable of finding translation bugs in real-world transpilers?

4.1 Experimental Setup

4.1.1 Transpiler Benchmarks. We evaluate PROGnosticator across seven state-of-the-art real-world transpilers spanning six distinct input–output language pairs (Table 3): **C2Rust** [47] (C→Rust), **CxGo** [73] (C→Go), **Zig Translate-C** [53] (C→Zig), **TinyGo** [27] (Go→WebAssembly), **Go2Hx** [74] (Go→Haxe), as well as **Babel** [17] and **SWC** [26] (ES6+→ES5 JavaScript).

Input → Output	Transpiler	Version	Commit	Release Date
C → Rust	C2Rust	0.19.0	b9339f4	Sept. 2024
C → Go	CxGo	0.5.0	d23185c	Mar. 2025
C → Zig	Zig Translate-C	0.14.0	bd237bc	Nov. 2024
Go → WASM	TinyGo	0.36.0	8c54e3d	Mar. 2025
Go → Haxe	Go2Hx	0.1.0	bbc65e0	Feb 2025
ES6+ → ES5 JavaScript	Babel	7.28.3	ef155f5	Aug. 2025
ES6+ → ES5 JavaScript	SWC	1.13.5	0c53d85	Aug. 2025

Table 3. Our benchmark transpilers alongside their input and output languages, version, and commit hash.

4.1.2 Competing Fuzzers. We evaluate PROGnosticator against four state-of-the-art fuzzers for language processors, spanning both language-specific and language-agnostic testing approaches:

- **Language-specific Fuzzers:** We include TransFuzz [33], a fuzzer targeting JavaScript transpilers, as well as C compiler fuzzer CSmith [81] given its reported adoption by C2Rust's developers [46].
- **Language-agnostic Fuzzers:** While many language-agnostic fuzzers for language processors exist today [31, 76], we select Polyglot [34] due to its design goal of generating valid code as well as its frequent use as a baseline in prior works on fuzzing language processors [39, 44, 85]. We additionally include AFL-Compiler-Fuzzer [41], a standard compiler-fuzzing baseline [39, 44, 61, 63].
- **Excluded Fuzzers:** For Go2Hx and TinyGo, we also examined the Go compiler fuzzer GoSmith [78]; however, because it targets Go v1.5 and does not support later versions (e.g., v1.23, the version used by both transpilers at the time of writing), we therefore omit it from our evaluation. Consistent with prior reports [13, 14], we were also unable to reproduce a functional deployment of the LLM-driven compiler fuzzer Fuzz4All [80], and thus exclude it from our evaluation.

4.1.3 Fuzzer Setup. As Table 1 shows, Polyglot, AFL-Compiler-Fuzzer, and TransFuzz all require initial seed program corpora. **For C transpilers**, we follow prior work [71] in using the "c-testsuite" [2], which contains 220 hand-curated C programs covering a wide range of constructs.

For JavaScript transpilers, we follow prior efforts [43] in using the DIE corpus [66, 67], which provides 597 ES6+ JavaScript programs. **For Go transpilers**, because no standardized program corpus exists, we assemble 25 Go seed programs from public sources [16, 21]; as with the C and JavaScript corpora, these Go programs exercise diverse expressions, statements, and control-flow structures, while intentionally avoiding translation-derailing artifacts such as external dependencies.

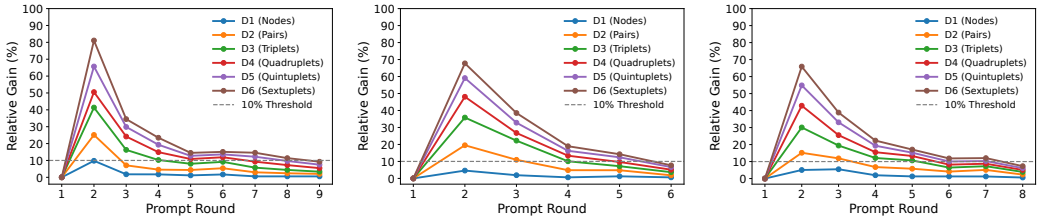
For PROGnosticator, we configure its large language model (OpenAI's GPT-4 [30]) with a default sampling temperature of 1.0. While adjusting sampling temperature is possible, we expect limited benefit from higher or lower settings, as discussed in § 5.2.

4.1.4 Incompatibilities. Despite being supplied with a compatible Go grammar, Polyglot fails to generate any valid Go programs, hence we exclude it from comparisons on Go2Hx and TinyGo. While CSmith supports C2Rust and Zig Translate-C, it unfortunately remains incompatible with CxGo due to its reliance on tool-specific header files (e.g., csmith.h, safe_math.h) that CxGo fails to transpile, even in isolation. More broadly, whenever a competitor cannot support a given transpiler (e.g., CSmith on Go2Hx, TinyGo, Babel, or SWC), **we mark these cases as n/a in our results**.

4.1.5 Infrastructure. We distribute all experiments and results post-processing tasks across two Ubuntu 22.04 workstations, each equipped with an Intel Core i9-12900K CPU and 64GB RAM.

4.2 RQ1: Code Pattern Diversity

Because Construct-oriented Fuzzing aims to explore varied usage patterns of language constructs without relying on grammars or seeds (Table 1), we evaluate this capability by first analyzing the diversity of code patterns returned by PROGnosticator's Stage-1 *Construct Usage Enumeration* (§ 3.2). As § 3.2.2 and § 3.2.3 detail, our approach prompts the LLM to propose 90 *new* construct-usage patterns per round—synthesizing minimal programs that incorporate each—and stops once the rate of change across D_1–D_6 AST-level diversity for these programs falls below our $\epsilon = 10\%$ saturation threshold. To measure how diversity evolves and when saturation occurs per language, we calculate D_1–D_6 growth across all of PROGnosticator's Stage-1 programs over five trials for C, Go, and ES6+ JavaScript (i.e., the *input* languages for our benchmark transpilers in Table 3), shown in Figure 6.



(a) C (b) Go (c) ES6+ JavaScript

Fig. 6. Mean per-round growth in D_1–D_6 AST-level diversity during PROGnosticator's Stage-1 *Construct Usage Enumeration* (§ 3.2). We compute each prompting round's growth in construct-usage pattern diversity across five independent trials, terminating each once the $\epsilon = 10\%$ diversity saturation threshold is reached.

Overall, PROGnosticator's relative D_1–D_6 diversity growth falls below our $\epsilon = 10\%$ threshold after **nine** prompting rounds for C, **six** for Go, and **eight** for JavaScript. As Figure 6 shows, diversity peaks in the second round across all three languages, after which further rounds yield diminishing returns in code-pattern uniqueness. To further analyze the impact of explicitly seeking *novel* code patterns (§ 3.2.2), we also evaluate an alternative configuration, **random-PROGnosticator**, where prompting selects construct-usage patterns at *random*, irrespective of prior outputs. Figure 7 compares the resulting C language D_1–D_6 AST-level diversity for PROGnosticator versus random-PROGnosticator, averaged over five trials with each capped at 1,000 total construct-usage patterns.

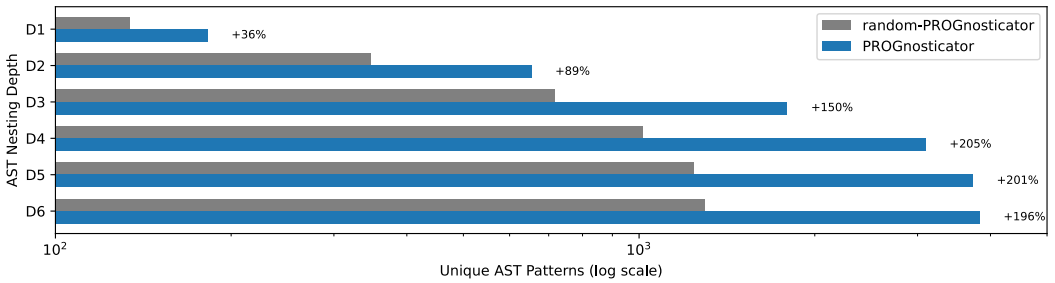


Fig. 7. Mean C language D_1–D_6 AST-level diversity for PROGnosticator vs. random-PROGnosticator across five Stage-1 *Construct Usage Enumeration* (§ 3.2) trials capped at 1,000 construct-usage patterns each. Values are log-scaled, with PROGnosticator's relative improvements over random-PROGnosticator on the right.

As Figure 7 shows, PROGnosticator's resulting code patterns consistently outrank random-PROGnosticator's in structural diversity across *all* D_1–D_6 AST-level diversity depth metrics. In total, PROGnosticator's relative diversity gains include **36% more** distinct standalone constructs (D_1), **89% more** two-level construct nestings (D_2), and up to **150% more** four-level construct nestings (D_4), with similarly strong improvements at other nesting depths. Together, these results showcase the effectiveness of Construct-oriented Fuzzing's guided, LLM-driven construct exploration in producing **structurally diverse and construct-rich programs**.

RQ1: Construct-oriented Fuzzing's guided construct enumeration uncovers a broad range of code patterns, providing a solid foundation for systematically generating diverse programs to test transpilers' correctness.

4.3 RQ2: Generated Program Quality

We next evaluate the overall quality of Construct-oriented Fuzzing's resulting transpiler test case programs versus those of state-of-the-art compiler and transpiler fuzzers. Following the fuzzing evaluation standard of Klees et al. [55], we perform five 24-hour fuzzing campaigns, collecting all generated test case programs from PROGnosticator's Stage-2 *Construct-oriented Program Generation* (§ 3.3.1) as well as from each competing fuzzer; and analyze their language validity rate, code density (total lines of code), and D_1–D_6 AST-level diversity (§ 3.2.3).

As PROGnosticator and CSmith operate as standalone program generators, we run each on all supported languages (i.e., C for CSmith) for five 24-hour program generation campaigns. Conversely, Polyglot, AFL-Compiler-Fuzzer, and TransFuzz only retain their test case programs *during* transpiler fuzzing. To obtain representative per-language corpora for these fuzzers, we merge their fuzzing-generated program corpora trial by trial across all supported transpilers per language (§ 4.1.4). Despite our efforts, Polyglot and AFL-Compiler-Fuzzer could not retain test cases on Babel or SWC, nor could AFL-Compiler-Fuzzer on TinyGo or Go2Hx, as these transpilers are out-of-the-box incompatible with their expected mode of operation (i.e., grey-box fuzzing [84]). Thus, we omit these fuzzers from the corresponding language-level program quality evaluations.

4.3.1 Code Validity and Density. We measure fuzzers' validity rates as the proportion of their generated programs that successfully compile and successfully execute pre-translation, reflecting the practical constraint that transpilation proceeds only for *well-formed*, language-conforming programs. As shown in Table 4, PROGnosticator averages high validity across all three languages—**73.01%** for C, **70.05%** for Go, and **91.66%** for JavaScript—second only to language-specific fuzzers CSmith (93.09% for C) and TransFuzz (99.97% for JavaScript). In contrast, we observe that the syntax- and semantics-breaking mutations of AFL-Compiler-Fuzzer leave it reaching just 1.30% average

validity; while Polyglot's similarly leave it struggling at an average 21.26% validity, on-par with its own self-reported semantic validity of ~20% [34]. Additionally, as Table 4 shows, PROGnosticator exhibits small standard deviations and narrow confidence intervals for every language, indicating stable validity across all program generation campaigns per language.

Lang.	Fuzzer	Mean #Programs	Language Validity Mean	StDev	[CI_{low}, CI_{high}]	Program Size (LoC) [Min, Max]	Mean
C	AFL-Compiler-Fuzzer	3,182.40	1.30%	0.10	[1.21, 1.39]	[3, 126]	19.17
	Polyglot	8,373.60	21.26%	1.93	[19.57, 22.95]	[3, 222]	86.06
	CSmith	108,164.40	93.09%	0.19	[92.92, 93.26]	[18, 14,990]	1403.50
	PROGnosticator	10,305.00	73.01%	1.56	[71.64, 74.38]	[7, 87]	22.64
Go	PROGnosticator	10,120.00	70.05%	0.80	[69.35, 70.75]	[7, 96]	27.93
JS	TransFuzz	11,397.70	99.97%	0.01	[99.96, 99.98]	[5, 164]	14.28
	PROGnosticator	10,810.00	91.66%	0.36	[91.34, 91.98]	[6, 68]	22.31

Table 4. Per-language statistics of competing fuzzers' resulting transpiler test case programs versus PROGnosticator's across five 24-hour program generation campaigns. We compute 95% confidence intervals for program validity, measured as the proportion of resulting programs that compile and run prior to transpilation.

With respect to program size, PROGnosticator's test cases exhibit more compactness, with a mean **22.64** lines of code (LoC) for C, **27.93** for Go, and **22.31** for JavaScript, compared to CSmith's and Polyglot's far larger programs (over 1,400 and over 86 LoC, respectively). Overall, these results underscore that Construct-oriented Fuzzing upholds high validity while producing smaller, more targeted test case programs, **striking a balance between correctness and structural diversity**.

4.3.2 Code Pattern Diversity. Table 5 summarizes each fuzzer's per-language D_1–D_6 AST-level diversity across five 24-hour program generation campaigns. As our competitors either do not support Go, or cannot retain their generated Go programs when fuzzing Go transpilers (§ 4.3), we use our Go seed corpus (§ 4.1.3) as the baseline for comparing PROGnosticator's Go program diversity.

Lang.	Fuzzer		D1	D2	D3	D4	D5	D6	Σ D1–D6
C	AFL-Compiler-Fuzzer	**Mean**	127	330	610	791	850	789	3497
		StDev	4.16	17.05	39.10	41.68	58.66	78.08	221.07
		CI	[123, 131]	[315, 345]	[576, 644]	[754, 828]	[799, 902]	[721, 858]	[3304, 3691]
	Polyglot	**Mean**	135	411	1357	2961	4886	6724	16474
		StDev	1.10	6.38	52.22	244.96	595.81	997.72	1892.57
		CI	[134, 136]	[405, 417]	[1311, 1403]	[2746, 3176]	[4363, 5409]	[5849, 7599]	[14814, 18131]
	CSmith	**Mean**	109	281	704	1746	3991	9374	16205
		StDev	0.00	0.84	2.17	3.05	4.34	14.67	20.07
		CI	[109, 109]	[280, 282]	[702, 706]	[1743, 1749]	[3987, 3995]	[9361, 9387]	[16186, 16221]
	PROGnosticator	**Mean**	197	1111	5002	13309	26236	38999	84854
		StDev	2.80	8.30	66.90	320.70	1059.70	2069.70	3479.60
		CI	[194, 200]	[1104, 1119]	[4943, 5061]	[13028, 13591]	[25307, 27165]	[37185, 40814]	[81804, 87904]
Go	Go seed programs	**Mean**	110	241	384	502	532	528	2297
	PROGnosticator	**Mean**	174	956	3524	8282	14500	22276	49712
		StDev	1.79	10.24	102.22	647.84	719.55	1262.33	2563.05
		CI	[172, 176]	[947, 965]	[3434, 3614]	[7714, 8850]	[13869, 15131]	[21170, 23383]	[47464, 51957]
JS	TransFuzz	**Mean**	140	538	1688	3607	5976	8341	20290
		StDev	0.89	1.64	13.16	44.54	80.38	126.13	262.24
		CI	[139, 141]	[536, 540]	[1676, 1700]	[3567, 3647]	[5905, 6047]	[8230, 8452]	[20060, 20519]
	PROGnosticator	**Mean**	174	933	4564	13517	29203	50218	98609
		StDev	1.64	7.79	35.39	107.24	258.63	378.30	752.55
		CI	[172, 176]	[926, 940]	[4532, 4596]	[13423, 13611]	[28976, 29430]	[49886, 50550]	[97948, 99267]

Table 5. Per-language mean, standard deviation, and 95% confidence intervals (CI) of D_1–D_6 AST-level code pattern diversity for the transpiler test-case programs generated by each fuzzer across five 24-hour campaigns.

Overall, PROGnosticator achieves the highest structural diversity across *all* languages: outperforming AFL-Compiler-Fuzzer by **23.26**×, CSmith by **4.23**×, Polyglot by **4.15**×, and TransFuzz by **3.85**× in *cumulative* D_1–D_6 diversity, computed as the sum of all mean D_1–D_6 values per language.

These improvements hold consistently from D_1 diversity (i.e., single-construct patterns) to D_3–D_6 diversity (i.e., multi-construct patterns), indicating that Construct-oriented Fuzzing enriches both surface-level construct usage as well as deeper structural composition. Moreover, the standard deviations and confidence intervals in Table 5 show that these gains are stable across trials, with PROGnosticator exhibiting consistently tight confidence bounds even at greater structural depths, and no overlap with the confidence intervals of other fuzzers.

Interestingly, while CSmith sees the highest C-level program validity and generates over an order of magnitude more programs per trial on average than PROGnosticator (Table 4), its lack of support for much of C's syntax (§ 2.3) ultimately limits the diversity of its resulting programs relative to PROGnosticator's. TransFuzz, though attaining near-perfect validity on JavaScript, faces the same constraint, as its grammar-based mutations repeatedly recycle many of the same JavaScript code patterns. Even when compared to our Go seed program corpus (§ 4.1.3), PROGnosticator still produces substantially richer AST-level structures in Go (Table 5). Collectively, these results show Construct-oriented Fuzzing's potential for synthesizing **structurally diverse programs in a fully language-agnostic manner**, free of grammar- or seed-specific tailoring.

RQ2: Construct-oriented Fuzzing's generated programs uniquely balance validity, compactness, and high structural diversity—in every language—facilitating more rigorous evaluation of transpilers' correctness.

4.4 RQ3: Transpiler Bug Discovery

Finally, we evaluate Construct-oriented Fuzzing's effectiveness in uncovering translation defects in the real-world transpilers shown in Table 3. Following our experiment procedure from § 4.3, we run five 24-hour transpiler testing campaigns per each competing fuzzer, feeding their resulting test case programs to their supported transpiler(s) and monitoring for the canonical indicators of potential bugs (§ 3.4.2): transpiler crashes, invalid-code failures, or semantic divergences.

While CSmith is equipped with its own built-in differential testing mechanism for catching semantic divergences, we extend PROGnosticator's differential-testing instrumentation (§ 3.4.1) to all other fuzzers such as Polyglot and AFL-Compiler-Fuzzer, since they do not provide such functionality themselves. Following manual root cause analysis and deduplication, we report all newly discovered bugs and reproducer programs to their respective transpiler developers. Table 6 summarizes our bug-finding results: the number of previously known, newly discovered, as well as developer-confirmed new bugs found per transpiler. Fuzzers not listed uncover **zero** bugs, while **n/a** denotes CSmith's incompatibilities (§ 4.1.4) or unsupported (i.e., non-C) transpilers.

Transpiler	PROGnosticator			CSmith		
	Old Bugs	New Bugs	Confirmed	Old Bugs	New Bugs	Confirmed
C2Rust	8	15	14	2	0	0
Zig Translate-C	4	9	9	3	1	1
CxGo	0	11	11	n/a	n/a	n/a
TinyGo	1	2	2	n/a	n/a	n/a
Go2Hx	0	15	15	n/a	n/a	n/a
Babel	0	2	2	n/a	n/a	n/a
SWC	0	10	10	n/a	n/a	n/a
Total:	13	64	63	5	1	1

Table 6. Total *old* (previously known), *new*, and newly-found bugs that are since *confirmed* by transpiler developers after reporting. **n/a**: transpiler unsupported by that fuzzer. Fuzzers not shown (i.e., AFL-Compiler-Fuzzer, Polyglot, and TransFuzz) find **zero bugs** across our entire evaluation and are therefore omitted for brevity.

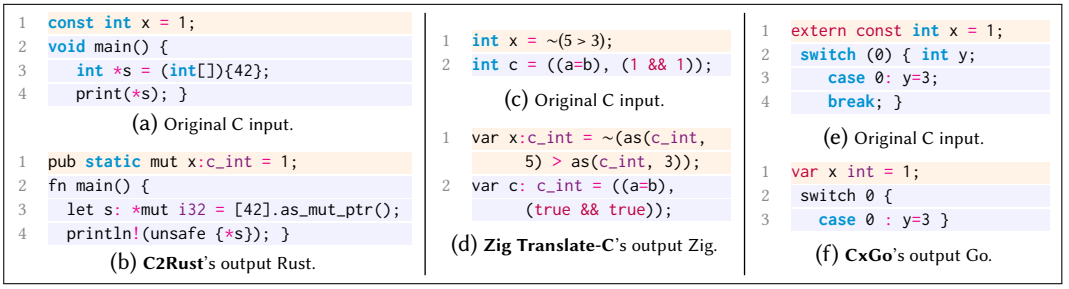
4.4.1 Discovered Bugs. Overall, PROGnosticator finds the most transpiler bugs—**77** in total—including **64 new bugs**, of which **63** are so far confirmed or fixed by transpiler developers following our reporting. Beyond uncovering the most bugs, PROGnosticator rediscovers eight known issues in C2Rust [5–7, 19], four in Zig Translate-C [4, 8–10], and one in TinyGo [28], highlighting Construct-oriented Fuzzing's broad effectiveness across diverse transpilers. In contrast, CSmith finds four bugs in Zig Translate-C, including one new bug (issue #23975) and three known bugs [4, 8, 9], as well as two known bugs in C2Rust [6, 7]; while all other competing fuzzers each find zero bugs in total. Table 7 lists all new bugs found by PROGnosticator, including their type, how they were detected, bug-triggering code patterns, and reporting status.

Target	Type	Cause	Implicated Construct Pattern	Iss. ID
C2Rust	TC	UC	(++/--) in compound literal	#1165†
	IC	SE	Invalid compound literal address	#1166†
	TC	UC	enum in compound literal	#1168†
	TC	UC	struct init via expressions	#1171†
	SD	TM	const treated as mutable	#1184*
	IC	SE	Misuse of max_align_t	#1208*
	SD	CM	array decay misrepresented	#1217†
	SD	CM	compound literal comparison	#1231†
	TC	UC	anon enum in anon union	#1233†
	TC	UC	Macro-gen anon enum init	#1234†
	IC	TM	enum to array cast	#1236
	IC	TM	Ptr decrement on struct field	#1237*
	IC	TM	Array cast to void*	#1238†
	IC	CM	packed struct mistranslation	#1239†
	IC	TM	volatile int* to literal	#1240†
Go2Hx	SD	CM	Interface w/ slice, map, func	#183†
	SD	CM	slice/map comparison	#184†
	SD	CM	func ptr comparison wrong	#185†
	IC	SE	Nil to array conversion	#186*
	IC	SE	double-panic mishandled	#187†
	SD	CM	recover in defer mishandled	#188*
	SD	CM	len(*nilPtrToArray) mishandled	#222†
	SD	CM	Double pointer reassignment	#225†
	SD	CM	Tuple to struct* error	#226*
	SD	CM	Nil pointer iteration bug	#230*
	SD	CM	channel data translation	#231†
	SD	CM	Ambiguous interface method	#232*
	SD	CM	_ in tuple unpacking mishandled	#256†
	SD	CM	goto mishandling in loop exit	#276†
	SD	CM	struct map key mishandled	#277†
TinyGo	TC	UC	len() on nil array pointer	#4786†
	TC	UC	generic type alias crash	#4819†
Zig T-C	IC	TM	Cast: pointer to 2D array	#22045*
	SD	CM	string vs literal compare	#22136*
	IC	SE	Addr of struct literal immutable	#22139*
	IC	UC	anon enum not translated	#22148*
	IC	TM	bool to pointer cast	#22964*
	IC	TM	Bad function pointer type	#23283*
	IC	CM	(expr1, bool_expr) mishandled	#23975*
	IC	CM	Bitwise NOT on bool wrong	#23987*
	IC	CM	Bad pointer cast (int*)1	#24010*
SWC	SD	CM	proxy property loss in rest	#11039†
	SD	CM	object rest misordering	#11040†
	SD	CM	Missed TDZ ReferenceError	#11045*
	SD	CM	Nested object yield	#11046†
	SD	CM	Destructuring array hole	#11047†
	SD	CM	default array hole missing	#11048†
	SD	CM	Promise leaked in async array	#11049†
	SD	CM	Async short-circuit break	#11050†
	SD	CM	Object unpack in condition	#11051†
	SD	CM	Missing write-back across await	#11052†
CxGo	IC	SE	compound string initializer	#90*
	TC	UC	switch-based loop unrolling	#91*
	SD	CM	%zu format mishandled	#92*
	IC	SE	Null function pointer wrong	#93*
	TC	UC	max_align_t unrecognized	#95*
	TC	UC	_Complex unsupported	#96*
	IC	SE	Invalid pointer arithmetic	#97*
	IC	TM	Bad char array init	#98*
	SD	TM	External const var wrong	#99*
	IC	CM	Vars before switch mishandled	#100*
	SD	CM	function pointer error	#101*
Babel	SD	CM	...rest yields subclass	#17502†
	IC	TM	Incorrect BigInt exponentiation	#17190*

Table 7. PROGnosticator's 64 newly-found transpiler bugs and their corresponding types, root causes, implicated construct patterns, public GitHub issue IDs, and reporting statuses. **Bug Type:** *TC*=Transpiler Crash, *IC*=Invalid Code, *SD*=Semantic Divergence. **Cause:** *SE*=Syntax Error, *TM*=Type Mismatch, *CM*=Construct Mistranslation, *UC*=Unhandled Construct. **Status:** †fixed post-reporting, *confirmed and awaiting fix.

4.4.2 Transpiler Bug Case Studies. Below, we present several notable transpiler bugs discovered by PROGnosticator that highlight key challenges in source-to-source translation.

- **C2Rust:** Figure 8 (a–b) shows two C2Rust bugs. In the first (issue #1184), C2Rust translates the constant integer x in line 1 of Figure 8a into a mutable global in Rust (line 1 of Figure 8b), turning an immutable constant into a mutable global and introducing potential data races. In the second (issue #1217), line 3 of Figure 8a initializes s with a compound literal array of automatic storage duration, so the array remains valid for the entire function and thus prints 42. Instead, C2Rust takes a pointer s to a temporary array [42] whose lifetime ends with the expression in line 3 of

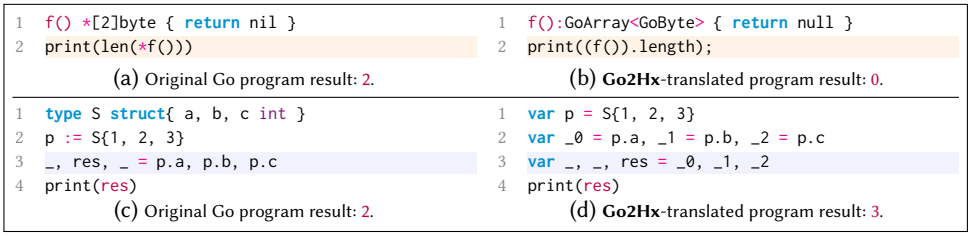


(a) Original C input. (b) **C2Rust**'s output Rust. (c) Original C input. (d) **Zig Translate-C**'s output Zig. (e) Original C input. (f) **CxGo**'s output Go.

Fig. 8. New translation bugs discovered by PROGNOSTICATOR in C-consuming transpilers. **C2Rust** (a–b): semantic divergence caused by incorrect handling of constant mutability and compound-literal lifetimes (issues #1184 and #1217, respectively). **Zig Translate-C** (c–d): invalid code caused by incorrect handling of boolean-to-integer conversions (issues #23987 and #23975, respectively). **CxGo** (e–f): semantic divergence and invalid code caused by incorrect treatment of qualifiers and switch-local declarations, respectively (issues #99 and #100, respectively). Matching highlighting denotes corresponding original and translated code fragments.

Figure 8b, leaving the pointer dangling in the subsequent print statement. Although the program may still print 42, the translation introduces undefined behavior, which Rust's built-in checks correctly detect. A correct translation would instead allocate a local array with `let mut s = [42];` and then obtain its pointer via `s.as_mut_ptr()`, ensuring the array is valid for the function's scope.

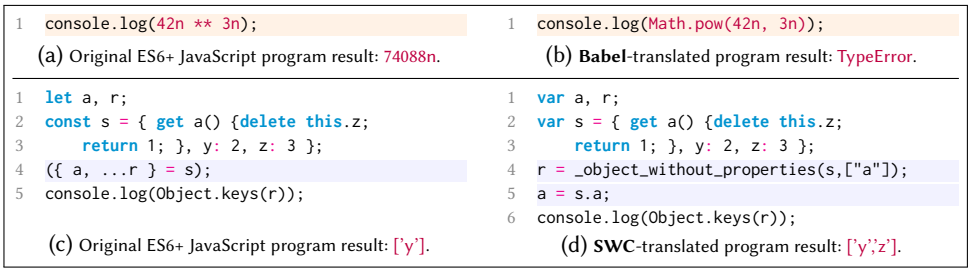
- **Zig Translate-C:** Figure 8 (c–d) illustrates two Zig Translate-C bugs. In the first (issue #23987), C expression `int x = ~(5 > 3);` in line 1 of Figure 8c performs bitwise negation (`~`) only after evaluating the comparison to 0 or 1. Without the negation, Zig Translate-C handles the conversion correctly, generating `intFromBool(as(c_int, 5) > as(c_int, 3));`. However, with the negation, Zig Translate-C emits `~(as(c_int, 5) > as(c_int, 3))` in line 1 of Figure 8d, producing an illegal bitwise negation of a boolean result. Correct translation must first convert the comparison result to an integer *prior* to negation (e.g., `~(intFromBool(as(c_int, 5) > as(c_int, 3)))`). In the second (issue #23975), the comma expression in line 2 of Figure 8c evaluates `(1 && 1)` and yields the integer value 1. Zig Translate-C instead wrongly renders this expression as `(true && true)`, whose result has type `bool` in Zig. The subsequent assignment to `c: c_int` in line 2 of Figure 8d therefore incorrectly attempts to store a `bool` in an integer variable. To match C's implicit conversion, the correct translation must instead utilize an explicit boolean-to-integer cast (e.g., `intFromBool(true && true)`).
- **CxGo:** Figure 8 (e–f) highlights two CxGo bugs. In the first (issue #99), the C variable declaration `extern const int x = 1;` in line 1 of Figure 8e combines external linkage with immutability. On translation, CxGo instead emits `var x int = 1` in line 1 of Figure 8f, dropping both qualifiers so that `x` becomes mutable and loses external linkage, thereby diverging from the original's C semantics. In the second (issue #100), line 2 of Figure 8e declares `y` inside a `switch` block before the first case label, which is permitted in C. CxGo omits this declaration and emits only the assignment `y = 3` in line 3 of Figure 8f, leaving `y` undefined in the generated Go code. To uphold correctness, the translation must instead hoist the declaration of `y` outside the `switch`.
- **Go2Hx and TinyGo:** Figure 9 presents several bugs in Go-consuming transpilers. In the first (issue #222 for Go2Hx and issue #4786 for TinyGo), the expression `len(*f())` in line 2 of Figure 9a evaluates to the constant `2` in Go because the length of an array is determined by its type, even when the pointer returned by `f()` is `nil`. Go2Hx instead emits `print((f()).length);` in line 2 of Figure 9b, producing the result `0` and diverging from the original Go semantics, while TinyGo panics on the same input. In the second (Go2Hx issue #256), line 3 of Figure 9c performs multiple assignment with the blank identifier `_`, correctly binding the second field of `p` to `res` and producing the result `2`. Go2Hx instead emits `var _, _, res = _0, _1, _2` in line 3 of Figure 9d, incorrectly binding the third field to `res`, causing the code to produce the divergent runtime output `3`.



(a) Original Go program result: 2. (b) **Go2Hx**-translated program result: 0.

(c) Original Go program result: 2. (d) **Go2Hx**-translated program result: 3.

Fig. 9. New translation bugs discovered by PROGnosticator in Go-consuming transpilers. **Go2Hx and TinyGo** (a–b): semantic divergence and runtime panic (issue #222 and issue #4786, respectively) caused by incorrect handling of array length computation on a dereferenced nil pointer. **Go2Hx** (c–d): semantic divergence caused by incorrect handling of multiple assignment with the blank identifier (issue #256). Matching highlighting denotes corresponding original and translated code fragments.



(a) Original ES6+ JavaScript program result: 74088n. (b) **Babel**-translated program result: TypeError.

(c) Original ES6+ JavaScript program result: ['y']. (d) **SWC**-translated program result: ['y','z'].

Fig. 10. New translation bugs discovered by PROGnosticator in ES6+ JavaScript-consuming transpilers. **Babel** (a–b): invalid code caused by incorrect handling of BigInt exponentiation (issue #17190). **SWC** (c–d): semantic divergence caused by incorrect evaluation order in object destructuring (issue #11040). Matching highlighting denotes corresponding original and translated code fragments.

- **Babel:** Figure 10 (a–b) shows a Babel bug (issue #17190) in the handling of BigInt exponentiation. In line 1 of Figure 10a, the original ES6+ JavaScript program evaluates 42n ** 3n to 74088n. Babel instead lowers the operator to Math.pow, emitting Math.pow(42n, 3n) in line 1 of Figure 10b, which throws a TypeError at runtime because Math.pow does not accept BigInt arguments.
- **SWC:** Figure 10 (c–d) illustrates an SWC bug (issue #11040) in object destructuring. In the original ES2018+ JavaScript program, the destructuring assignment { a, ...r } = s in line 4 of Figure 10c triggers the getter for a (line 2), which deletes property z before the rest object is constructed, leaving only ['y']. SWC instead constructs the rest object in line 4 of Figure 10d before invoking the getter in line 5, producing ['y','z'] and thereby diverging from the specified evaluation order.

4.4.3 Construct Interaction Analysis. We further analyze the construct composition of the programs that triggered the 64 transpiler bugs found by PROGnosticator. Among these, **12** arise from usage of a **single** core language construct, **30** from interactions between **two** constructs, and **22** from **three**. Thus, the majority of transpiler bugs emerge from interactions among *multiple* constructs, supporting Construct-oriented Fuzzing's explicit targeting of multi-construct program generation § 3.3.1. Consistent with this observation, PROGnosticator uncovered five distinct bugs in C2Rust and CxGo where C compound literals triggered failures only when combined with other constructs, such as enums, strings, increment/decrement expressions, or pointers (Table 7).

Importantly, competing state-of-the-art language processor fuzzers struggle to exercise the nuanced construct-usage patterns underlying these bugs, limiting their ability to expose real-world transpiler failures. For example, because CSmith cannot presently generate programs containing compound literals (without substantial refactoring), the multi-construct bugs found by PROGnosticator were missed by C2Rust's own in-house CSmith-based testing [46]. In summary, by

systematically exploring the vast space of how core language constructs can be *used*—in isolation and in combination—Construct-oriented Fuzzing uncovers complex translation defects often overlooked by developers, **bringing critical challenges in source-to-source translation to light**.

RQ3: By systematically exercising core language constructs in varied ways, Construct-oriented Fuzzing generates programs that expose a much wider range of transpiler failures than prior fuzzing approaches.

5 Discussion & Threats to Validity

In the following, we weigh several potential limitations of both Construct-oriented Fuzzing as well as our prototype implementation, PROGnosticator.

5.1 Bug Impacts and Developer Responses

While transpiler crashes and invalid-code errors manifest as obvious failures—blocking translation and/or code execution outright—semantic divergences pose a deeper risk: they introduce subtle semantic correctness issues that may remain hidden until long after translation. This risk is magnified as transpilers gain increasing adoption, underscored by initiatives such as DARPA's TRACTOR [36], which seeks to accelerate automated C-to-Rust translation with the explicit goal of rewriting critical legacy software such as Gzip and Apache. Amidst this landscape, Construct-oriented Fuzzing effectively uncovers *every* canonical class of transpiler failure (§ 2.1), revealing defects across all seven real-world transpilers we evaluated.

While some bugs listed in Table 7 remain pending developer confirmation, the majority of those found by PROGnosticator were promptly acknowledged, with **33** already patched and another **30** reportedly with fixes underway. In our correspondence, transpiler developers noted that many failures involved troublesome yet common code patterns, with many PROGnosticator-generated programs since being adopted within transpilers' internal test suites (e.g., [22, 25]).

Notably, one bug affecting TinyGo (issue #4786) was traced back to the Go compiler—a defect in SSA handling of len(*f()) when f() returns a nil pointer to an array—showing that Construct-oriented Fuzzing can surface subtle code translation edge cases that stem even beyond the transpiler itself.

5.2 Code Hallucinations, Undefined Behavior, and Sampling Randomness.

Because Construct-oriented Fuzzing relies on large language models (LLMs), it inherits their well-known tendency to *hallucinate* [86], potentially producing incomplete or otherwise incorrect code. For example, generated programs may erroneously lack PROGnosticator's required wrapping function (§ 3.3.1), resulting in invalid code that cannot be compiled. Nevertheless, PROGnosticator still attains a mean code validity rate **above 70%** across all languages (Table 4), ultimately yielding more transpiler bugs than all other evaluated fuzzers.

Moreover, while our prompting strategy also attempts to explicitly forbid generation of code with *undefined behavior* (§ 3.3.2), such as returning a pointer to a local variable in C, these patterns still occasionally arise. However, our manual analysis using LLVM's UndefinedBehaviorSanitizer [56] reveals that out of PROGnosticator's total 51,525 generated C programs, only 1,426 (**2.76%**) exhibit undefined behavior, indicating that such cases are rare and largely inconsequential.

Additionally, our current work does not evaluate PROGnosticator under varying LLM sampling configurations, relying only on a default sampling temperate of 1.0. Although PROGnosticator can be indeed configured to operate deterministically by minimizing temperature, doing so only yields near-identical programs across runs—counter to fuzzing's fundamental reliance on *randomness* to produce diverse test cases that surface distinct bugs [42]. Further, as Table 4 and Table 5 show,

PROGnosticator exhibits a low overall cross-run variability despite using stochastic sampling, suggesting that Construct-oriented Fuzzing remains relatively stable in the face of sampling noise.

5.3 Testing New Transpilers and Language Features.

Our evaluation targets a representative set of popular transpilers spanning diverse input and output languages pairs (Table 3). While this selection provides broad coverage of different syntactical and semantic translation challenges, extending PROGnosticator to other languages remains straightforward, requiring minimal input-language-specific prompt tailoring and supporting infrastructure. Accordingly, we leave extension of PROGnosticator and Construct-oriented Fuzzing to other transpiler ecosystems as future work.

Additionally, our testing for Rust-targeting transpilers (e.g., C2Rust [47]) omits memory-safety properties like ownership, borrowing, or aliasing. We posit that this scope aligns with existing transpilers' current limitations at the time of writing: C2Rust, despite being today's most mature Rust-targeting transpiler, has yet to fully support generation of safe and idiomatic Rust. Accordingly, our work focuses on language features currently *supported* by existing transpilers, deferring memory-safety-aware testing to future work as transpiler toolchains evolve.

Finally, our current prototype of PROGnosticator remains restricted to generating *sequential* programs, and does not yet support concurrency-related code constructs (e.g., C's threading or Go's goroutines). While we have preliminary support for concurrency-aware program generation, we leave the engineering and requisite reevaluation as a future update to PROGnosticator.

5.4 Costs of LLM Utilization.

Unlike the other fuzzers in our evaluation, our current prototype of PROGnosticator incurs an additional financial cost due to its reliance on an LLM that is, at the time of writing, only commercially available (i.e., OpenAI's GPT-4 [30]). In analyzing PROGnosticator's runtime costs, we calculate that its Stage-1 Construct Usage Enumeration (§ 3.2.2) averages per-trial costs of $10.65, $10.27, and $12.67 for C, Go, and JavaScript, respectively; while its Stage-2 Construct-oriented Program Generation (§ 3.3) averages $0.015 per generated program (~$150 per 10,000-program trial)—on-par with the costs seen in other recent LLM-driven program generation approaches for fuzzing (e.g., PromptFuzz's reported $0.016 per generated program [60]).

Although cost reductions are possible from adopting more economical or free-to-use LLMs, we leave the requisite reevaluation to future work. To maximize community benefit from our initial investment in deploying PROGnosticator, **we publicly release all PROGnosticator-generated corpora** (§ 8), enabling their reuse for bootstrapping testing of current and future transpilers.

6 Related Work

Below we discuss related work on large language models, their applications in software testing, and fuzzing techniques for language processors beyond source-to-source transpilers.

Large Language Models. Advancements in natural language processing [62] have led to an explosion of large language models (LLMs) for both natural-language tasks [87] as well as code comprehension and synthesis [50]. Modern LLMs generally span one of two types: *fine-tuned* models, which are trained on domain-specific corpora (e.g., code repositories or bug-fix datasets) and thus require substantial labeled data; and *prompt-based* models [64] (e.g., GPT-4 [30], Claude [18]), which achieve strong generalization without task-specific training. While our current prototype, PROGnosticator, leverages prompt-based models, we see strong potential in further leveraging models fine-tuned for the tasks of language-specific code construct reasoning and code synthesis.

LLM-driven Software Testing. Recent work applies LLMs to software testing tasks, including unit test generation [57, 70] and fuzzing test case synthesis [38, 48, 80]. For example, TestPilot [70] constructs prompts containing function source code and example usage to generate unit tests and repair incorrect ones. Fuzz4All [80], by contrast, focuses on *emergent* language features (e.g., C++23's if consteval [11]) and standard-library APIs (e.g., Go's atomic package [12]), leveraging code documentation and snippets to guide code synthesis. By contrast, PROGnosticator is the first to systematically explore *core* language constructs—the predominant source of today's transpiler bugs (Table 2)—enabling broad evaluation of transpilers' handling of fundamental code patterns.

Fuzzing for Language Processors. Fuzzing has long been applied to test different language processors: compilers [33, 34, 41, 59, 81], decompilers [72, 82], assemblers [52, 54], and instruction decoders [79]. While CSmith [81] and YARPGen [59] synthesize testcases using hardcoded C grammar rules to target C/C++ compilers such as LLVM [56], TransFuzz [33] instead applies ECMAScript-compliant mutations to existing JavaScript programs to test JavaScript-targeting transpilers such as Babel [17] and SWC [26]. AFL-Compiler-Fuzzer [41] employs syntax-aware, string-level mutations guided by regular expressions, whereas Polyglot [34] parses seed programs using the target language's grammar, lifts them into a unified abstract syntax tree (AST), and applies structure-preserving mutations, filtering its generated programs via type-inference-guided semantic validation.

To uncover bugs in binary decompilers, Cornucopia [72] mutates compiler optimization flags to generate diverse binaries, while Bin2Wrong [82] mutates source code, compilers, optimizations, and executable formats in combination. TheHuzz [52] generates random instruction sequences to uncover processor bugs, while AsFuzzer [54] infers assembly grammar rules to better guide fuzzing of assemblers. Mishegos [79] performs differential testing of instruction decoders by cross-checking competing implementations for inconsistencies.

To our knowledge, PROGnosticator remains the first language-agnostic approach to bring fuzzing to source-to-source *transpilers*, extending the strengths of LLMs to systematically test this long-undervetted class of language processors.

7 Conclusion

As transpilers see wider adoption for source-to-source translation, early detection of their underlying defects is critical to preserving translated software quality. Analyzing historical bug reports, we find that most failures arise from mishandled core language constructs. Accordingly, we introduce *Construct-oriented Fuzzing*, which harnesses LLMs to systematically target these constructs and their interactions to expose translation errors—without relying on language-specific seed inputs or grammar specifications. Our evaluation demonstrates that our approach achieves both the highest structural diversity and the highest transpiler bug discovery, culminating in **64 newly discovered defects across 7 real-world transpilers, of which 63 have since been confirmed or fixed**.

8 Data Availability

We open-source our prototype of PROGnosticator as well as our evaluation benchmarks and artifacts at the following URL: https://github.com/FuturesLab/PROGnosticator.

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] 2019. C2Rust Issue #81. https://github.com/immunant/c2rust/issues/81.
[2] 2020. C-testsuite. https://github.com/c-testsuite/c-testsuite.
[3] 2021. CxGo Issue #24. https://github.com/gotranspile/cxgo/issues/24.
[4] 2022. Zig Translate-C Issue #11992. https://github.com/ziglang/zig/issues/11992.
[5] 2023. C2Rust Issue #1201. https://github.com/immunant/c2rust/issues/1201.
[6] 2023. C2Rust Issue #881. https://github.com/immunant/c2rust/issues/881.
[7] 2023. C2Rust Issue #887. https://github.com/immunant/c2rust/issues/887.
[8] 2023. Zig Translate-C Issue #143. https://github.com/ziglang/translate-c/issues/143.
[9] 2023. Zig Translate-C Issue #17427. https://github.com/ziglang/zig/issues/17427.
[10] 2023. Zig Translate-C Issue #208. https://github.com/ziglang/translate-c/issues/208.
[11] 2024. Fuzz4All Documentation: C++23. https://github.com/fuzz4all/fuzz4all/blob/main/config/documentation/cpp/cpp_23.md.
[12] 2024. Fuzz4All Documentation: Go Atomic. https://github.com/fuzz4all/fuzz4all/blob/main/config/documentation/go/go_atomic.md.
[13] 2024. Fuzz4All Issue #8. https://github.com/fuzz4all/fuzz4all/issues/8.
[14] 2024. Fuzz4All Issue #9. https://github.com/fuzz4all/fuzz4all/issues/9.
[15] 2024. Go2Hx Issue #179. https://github.com/go2hx/go2hx/issues/179.
[16] 2025. A Tour of Go. https://go.dev/tour/basics/6.
[17] 2025. Babel: A Compiler for Writing Next Generation JavaScript. https://github.com/babel/babel.
[18] 2025. Build with Claude. Anthropic. https://www.anthropic.com/api.
[19] 2025. C2Rust Manual: Known Limitations. https://c2rust.com/manual/docs/known-limitations.html.
[20] 2025. Go. https://github.com/golang/go.
[21] 2025. Go by Example. https://gobyexample.com/.
[22] 2025. Go2Hx: functioninterfaceequality.go. https://github.com/go2hx/go2hx/blob/master/tests/unit/funcinterfaceequality.go.
[23] 2025. Haxe: The Cross-platform Toolkit. https://haxe.org/.
[24] 2025. OpenAI API. https://openai.com/index/openai-api/.
[25] 2025. SWC: 11051/input/input.js. https://github.com/swc-project/swc/blob/main/crates/swc/tests/fixture/issues-11xxx/11051/input/input.js.
[26] 2025. SWC: Rust-based Platform for the Web. https://swc.rs/.
[27] 2025. TinyGo: A Go Compiler for Small Places. https://tinygo.org/.
[28] 2025. TinyGo Issue #4816. https://github.com/tinygo-org/tinygo/issues/4816.
[29] 2025. Tree-sitter AST Parser. https://tree-sitter.github.io/tree-sitter/.
[30] Josh Achiam, Steven Adler, Sandhini Agarwal, Sam Ahmad, Shyamal Anadkat, et al. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).
[31] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Network and Distributed System Security Symposium (NDSS)*.
[32] Andrés Bastidas Fuertes, María Pérez, and Jaime Meza Hormaza. 2023. Transpilers: A Systematic Mapping Review of their Usage in Research and Industry. *Applied Sciences* 13, 6 (2023).
[33] Le Chen, Zhide Zhou, Xiaochen Li, and He Jiang. 2023. Detecting JavaScript Transpiler Bugs with Grammar-guided Mutation. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.
[34] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. In *IEEE Symposium on Security and Privacy (Oakland)*.
[35] Stephen Crane. 2024. Porting C to Rust for a Fast and Safe AV1 Media Decoder. https://www.memorysafety.org/blog/porting-c-to-rust-for-av1/.
[36] DARPA. 2024. TRACTOR: Translating All C to Rust. https://www.darpa.mil/research/programs/translating-all-c-to-rust.
[37] Folkert de Vries. 2025. Translating Bzip2 with C2Rust. https://trifectatech.org/blog/translating-bzip2-with-c2rust/.
[38] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*.
[39] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[40] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*.
[41] Alex Groce, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. 2022. Making No-Fuss Compiler Fuzzing Effective. In *ACM SIGPLAN International Conference on Compiler Construction (CC)*.
[42] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm Testing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.
[43] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*.
[44] William Hatch, Pierce Darragh, Sorawee Porncharoenwase, Guy Watson, and Eric Eide. 2023. Generating Conforming Programs with Xsmith. In *ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*.
[45] Dong Huang, Jie M Zhang, Qingwen Bu, Xiaofei Xie, Junjie Chen, and Heming Cui. 2024. Bias Testing and Mitigation in LLM-based Code Generation. *ACM Transactions on Software Engineering and Methodology* 35, 1 (2024).
[46] Immunant. 2024. C2Rust: CSmith Testing Scripts. https://github.com/immunant/c2rust/blob/master/scripts/csmith.py.
[47] Immunant. 2025. C2Rust: Migrate C Code to Rust. https://github.com/immunant/c2rust.
[48] Davide Italiano and Chris Cummins. 2025. Finding Missed Code Size Optimizations in Compilers using Large Language Models. In *ACM SIGPLAN International Conference on Compiler Construction (CC)*.
[49] Jarble. 2024. A List of Source-to-Source Compilers for Various Languages. https://github.com/jarble/list-of-transpilers.
[50] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A Survey on Large Language Models for Code Generation. *ACM Transactions on Software Engineering and Methodology* 35, 2 (2026).
[51] Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. 2021. Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
[52] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. In *USENIX Security Symposium (SEC)*.
[53] Andrew Kelley. 2025. Zig Translate-C: Automatic Translation from C Source Code. https://zig.guide/working-with-c/translate-c/.
[54] Hyungseok Kim, Soomin Kim, Jungwoo Lee, and Sang Kil Cha. 2024. AsFuzzer: Differential Testing of Assemblers with Error-Driven Grammar Inference. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.
[55] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[56] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.
[57] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.
[58] Zhibo Liu and Shuai Wang. 2020. How Far We Have Come: Testing Decompilation Correctness of C Decompilers. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.
[59] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.
[60] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt Fuzzing for Fuzz Driver Generation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[61] Haoyang Ma, Wuqi Zhang, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2024. Towards Understanding the Bugs in Solidity Compiler. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.
[62] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. 2023. Recent Advances in Natural Language Processing via Large Pre-Trained Language Models: A Survey. *Comput. Surveys* 56, 2 (2023).
[63] Charalambos Mitropoulos, Thodoris Sotiropoulos, Sotiris Ioannidis, and Dimitris Mitropoulos. 2023. Syntax-Aware Mutation for Testing the Solidity Compiler. In *European Symposium on Research in Computer Security (ESORICS)*.
[64] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2025. A Comprehensive Overview of Large Language Models. *ACM Transactions on Intelligent Systems and Technology* 16, 5 (2025).
[65] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced

by Large Language Models while Translating Code. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.
[66] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. DIE-corpus/jsc/JSTests. https://github.com/sslab-gatech/DIE-corpus/tree/master/jsc/JSTests.
[67] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *IEEE Symposium on Security and Privacy (Oakland)*.
[68] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[69] Charles Saternos. 2022. Making of Tom's Rock. https://c.har.li/e/2022/05/28/Making-of-Toms-Rock.html.
[70] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2023).
[71] Joschua Schilling, Andreas Wendler, Philipp Görz, Nils Bars, Moritz Schloegel, and Thorsten Holz. 2024. A Binary-level Thread Sanitizer or Why Sanitizing on the Binary Level is Hard. In *USENIX Security Symposium (SEC)*.
[72] Vidush Singhal, Akul Abhilash Pillai, Charitha Saumya, Milind Kulkarni, and Aravind Machiry. 2022. Cornucopia: A Framework for Feedback Guided Generation of Binaries. In *IEEE/ACM International Conference on Automated Software Engineering (ICSE)*.
[73] Smirnov, Denys. 2025. CxGo: Tool for Transpiling C to Go. https://github.com/gotranspile/cxgo.
[74] Elliott Stoneham. 2025. Go2Hx: Go to Haxe Source-to-Source Compiler. https://github.com/go2hx/go2hx.
[75] Simon Ask Ulsnes. 2024. Porting LibYAML to Safe Rust: Some Thoughts. https://simonask.github.io/libyaml-safer/.
[76] Vasudev Vikram, Rohan Padhye, and Koushik Sen. 2021. Growing a Test Corpus with Bonsai Fuzzing. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.
[77] Dmitry Vyukov. 2014. GoSmith-generated Test Case for LLGo Issue #175. https://gist.github.com/dvyukov/4230b576d02be4803d99.
[78] Dmitry Vyukov. 2025. GoSmith. https://github.com/dvyukov/gosmith.
[79] William Woodruff, Niki Carroll, and Sebastiaan Peters. 2021. Differential Analysis of x86-64 Instruction Decoders. In *IEEE Security and Privacy Workshops (SPW)*.
[80] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.
[81] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[82] Zao Yang and Stefan Nagy. 2025. Bin2Wrong: A Unified Fuzzing Framework for Uncovering Semantic Errors in Binary-to-C Decompilers. In *USENIX Annual Technical Conference*.
[83] Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA)*.
[84] Michal Zalewski. 2017. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.
[85] Chibin Zhang, Gwangmu Lee, Qiang Liu, and Mathias Payer. 2025. REFLECTA: Reflection-Based Scalable and Semantic Scripting Language Fuzzing. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*.
[86] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.
[87] Arkaitz Zubiaga. 2024. Natural Language Processing in the Era of Large Language Models. *Frontiers in Artificial Intelligence* 6 (2024).

Received 2025-09-10; accepted 2025-12-22