PROGnosticator:

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

Yeaseen Arafat
University of Utah

Stefan Nagy
University of Utah

FUTURES^3
LAB FUTURE TECHNOLOGY FOR USABLE, RELIABLE, & EFFICIENT SECURITY OF SOFTWARE & SYSTEMS

Transpilers: Automated Cross-language Code Translation

- **High-level goal:** convert code from one programming language to another

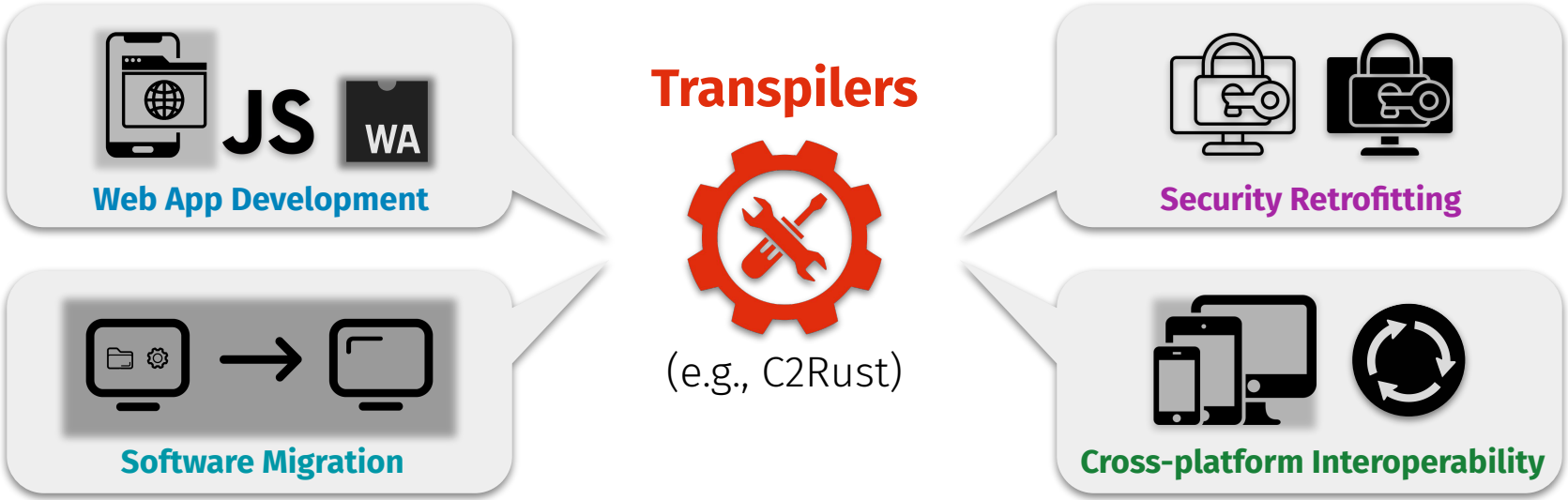
```c
int func(){
    int x = 0, y = 0;
    if (x <= 3) {
        goto WORK; }
    if (y >= 10) {
        goto EXIT;}
    WORK:
        x = x+y;
    EXIT:
        return x;
}
```

(e.g., C2Rust)

```rust
pub unsafe extern "C" fn func()
-> libc::c_int {
  let mut x: libc::c_int = 0;
  let mut y: libc::c_int = 0;
  let mut cur_block: i32;
   if x <= 3
       { cur_blk = 1;  }
   else if y >= 10
       { cur_blk = 2;  }
   else
       { cur_blk = 1;  }
   match cur_blk {
       1 => { x = x+y; }
       _ => {}
   }
   x }
```

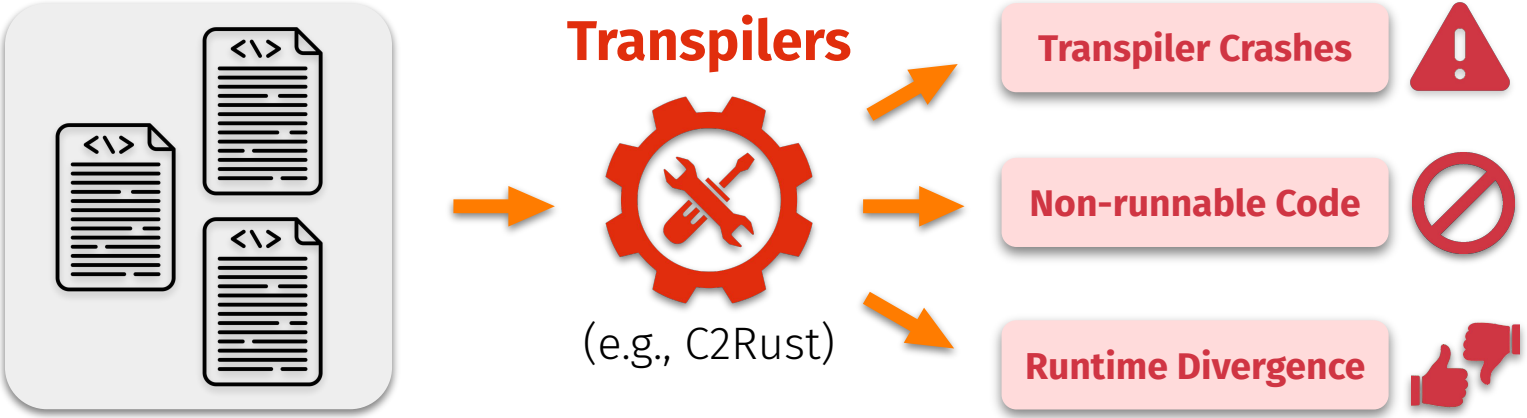
Transpilers: Automated Cross-language Code Translation

- **High-level goal:** convert code from one programming language to another
  - A growing foundation for modern **translation-driven** development tasks



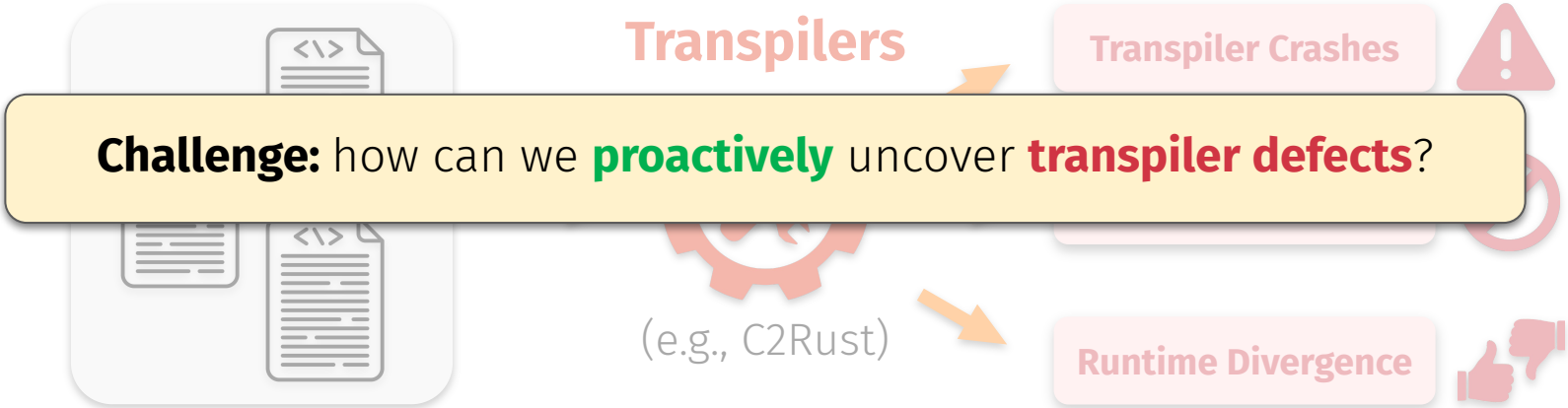
Transpilers: Automated Cross-language Code Translation

- **High-level goal:** convert code from one programming language to another
  - A growing foundation for modern **translation-driven** development tasks
  - **Problem:** mistakes during code translation result in **downstream failures**



Transpilers: Automated Cross-language Code Translation

- **High-level goal:** convert code from one programming language to another
  - A growing foundation for modern **translation-driven** development tasks
  - **Problem:** mistakes during code translation result in **downstream failures**



Our Prior Work: Automated Transpiler Testing

- First study on testing transpilers by repurposing **compiler fuzzing techniques**
  - **Idea:** stress-test transpilers' logic by randomly generating **large, logically dense programs**

```
static int32_t g_1 = 1;
static uint32_t g_2 = 7;
static struct S0 g_3 = { 5 };
struct S0 { unsigned f0 : 3; };
```

(Not shown: over 1100 more lines of code…)

```
static int32_t func_1(int32_t p_1) {
  int32_t l_4 = 0;
  for (int i = 0; i < 5; i++)
    l_4 ^= (((g_3.f0 + i) * 3) ^ (g_2 & 7)) - (p_1 | i),
      (l_4 & 1)
        ? (g_1 += ((g_3.f0 << 1) | i) - (l_4 % 3),
           g_2 ^= g_1 + (l_4 >> 1))
        : (g_1 -= ((i * 2) + (g_2 % 5)) ^ g_3.f0,
           g_2 += (g_3.f0 ^ l_4) & 31);
    return ((p_1 + l_4 + g_1) ^ (g_2 & 15)) + (g_3.f0 << 2);
}
int main(void) {
    int32_t result = func_1(g_1);
    if (result > 10) g_3.f0 = result & 7;
    return g_3.f0;
}
```

Lines of code in bug-triggering input: 1200

c2rust-transpile: bug about aligned struct with bitfield #887

Open

Labels: mistranslation

zone-hai opened on Apr 11, 2023 — Last edited by zone-hai

c2rust v0.17.0
Ubuntu 20.04.2

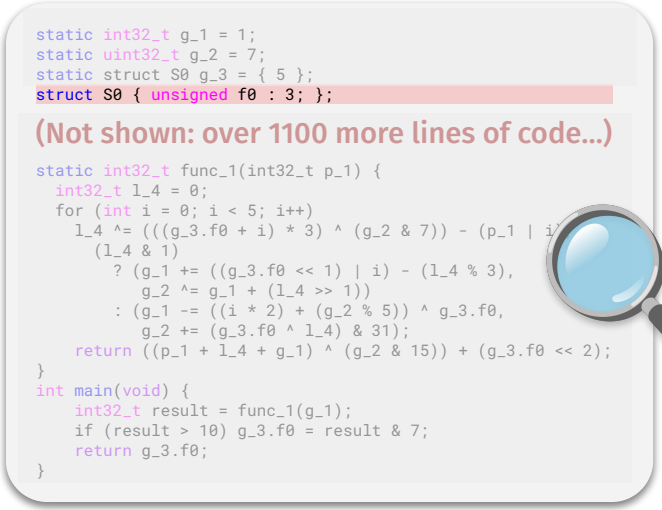
Consider the following C programs

```
#include<stdio.h>
//   struct S4 actually takes up 8 bytes of memory
//   4B:  f0(1B), f1(1B), PADDING(1B), PADING(1B)
//   4B:  f2(4B)
struct S4 {
   const unsigned f0 : 4;
   signed f1 : 4;
   unsigned f2 : 31;
};
struct S4 g_210 = {0,1,2};
int main(){
   printf("%d\n", g_210.f1);
}
```

github.com/immunant/c2rust/issues/887

Our Prior Work: Automated Transpiler Testing

- First study on testing transpilers by repurposing **compiler fuzzing techniques**
  - **Idea:** stress-test transpilers' logic by randomly generating **large, logically dense programs**

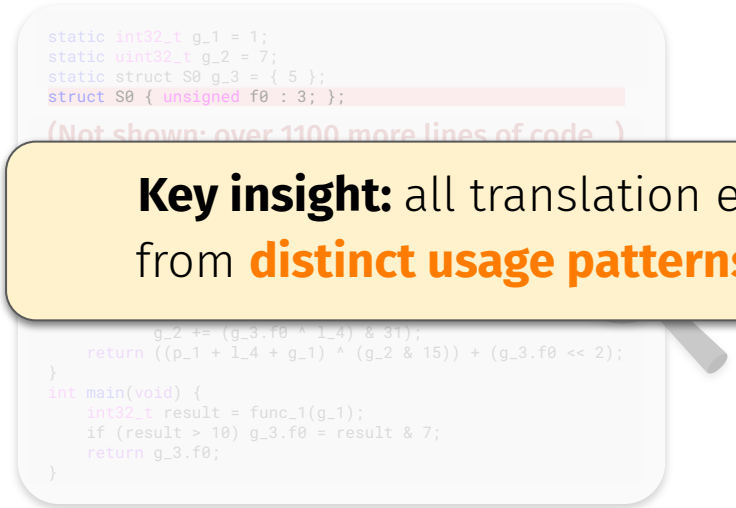


Lines of code in bug-triggering input: 1200

Actual trigger: struct with unsigned bit-field

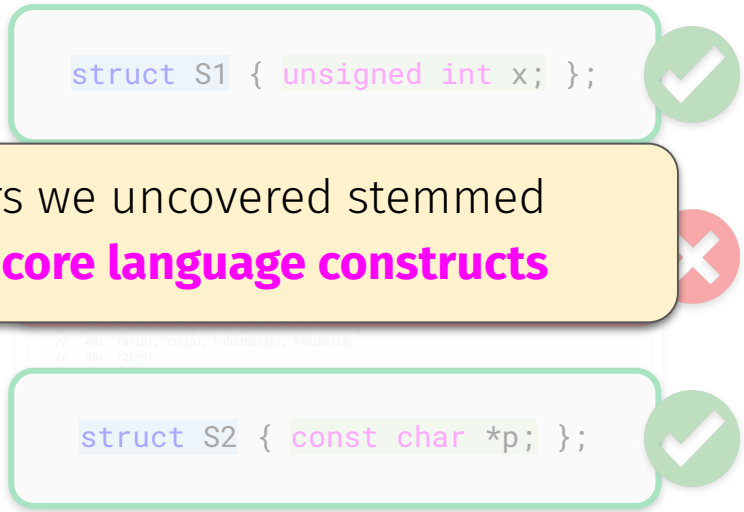
Our Prior Work: Automated Transpiler Testing

- First study on testing transpilers by repurposing **compiler fuzzing techniques**
  - **Idea:** stress-test transpilers' logic by randomly generating **large, logically dense programs**



Lines of code in bug-triggering input: 1200

Key insight: all translation errors we uncovered stemmed from **distinct usage patterns** of **core language constructs**



Actual trigger: struct with unsigned bit-field

Motivation: Real-world Transpiler Bugs

- Scaled-up analysis to all of C2Rust's previously-reported translation bugs
- **Outcome:** over 73% of translation bugs from **core language construct patterns**

Origins of Prior C2Rust Issues	Total
Platform-specific Constructs (e.g., SIMD intrinsics, kernel APIs)	21
Core C Language Constructs (e.g., packed structs, function-like macros, type mismatches, strings)	57

Motivation: Real-world Transpiler Bugs

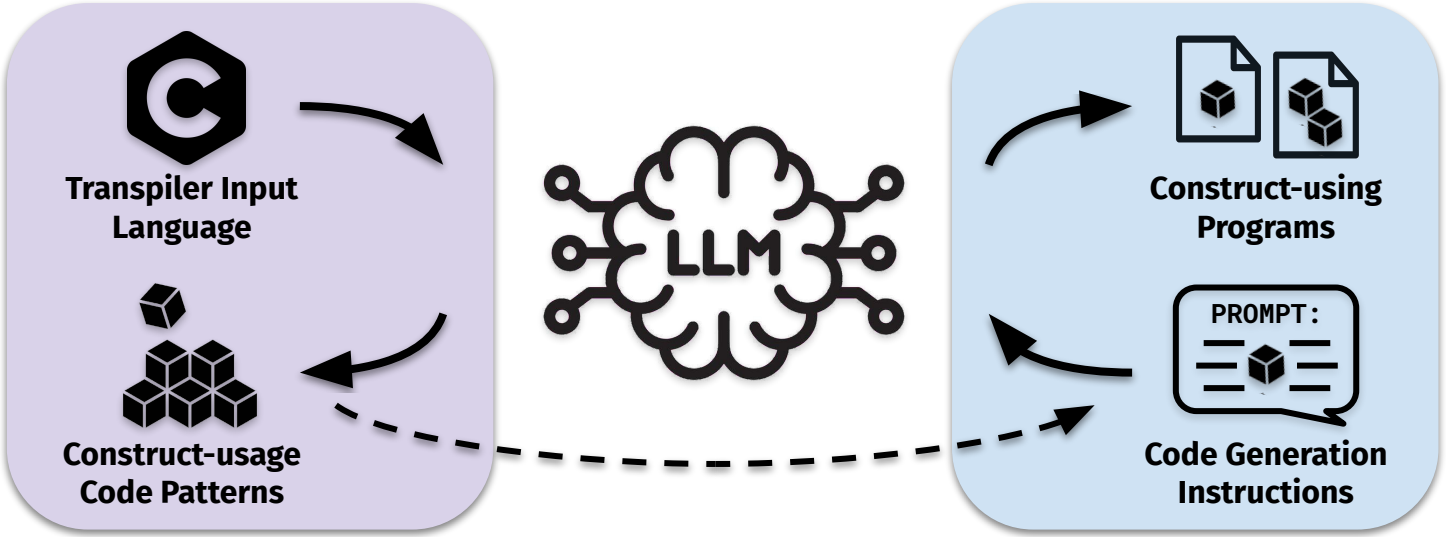
- Scaled-up analysis to all of C2Rust's previously-reported translation bugs
- **Outcome:** over 73% of translation bugs from **core language construct patterns**
- **Problem:** existing fuzzers struggle at **systematically exploring constructs**

Fuzzer	Seed Agnostic	Code Diversity	Program Validity	Broad Support
AFL	✘	✘	✘	✔
Polyglot	✘	✘	✘	✘
TransFuzz	✘	✘	✔	✘
YARPGen	✔	✘	✔	✘
CSmith	✔	✘	✔	✘
GoSmith	✔	✘	✔	✘

Motivation: effective transpiler testing demands an approach targeting **core construct-usage code patterns** without **language-specific tailoring**

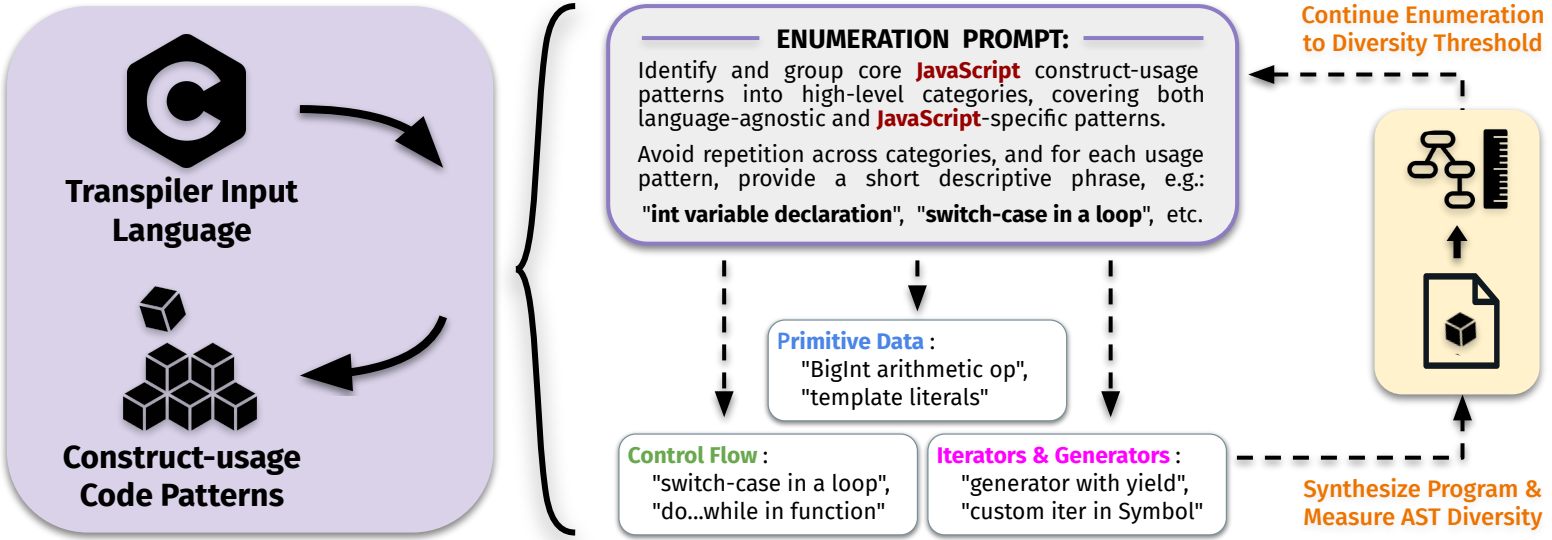
Our Solution: Construct-Oriented Transpiler Fuzzing

- **Idea:** harness LLMs' capabilities for **code reasoning** and **code generation**
  - **Construct-usage Enumeration:** broadly capture how a language's core constructs **are used**
  - **Construct-oriented Generation:** create programs **exercising and combining** these patterns



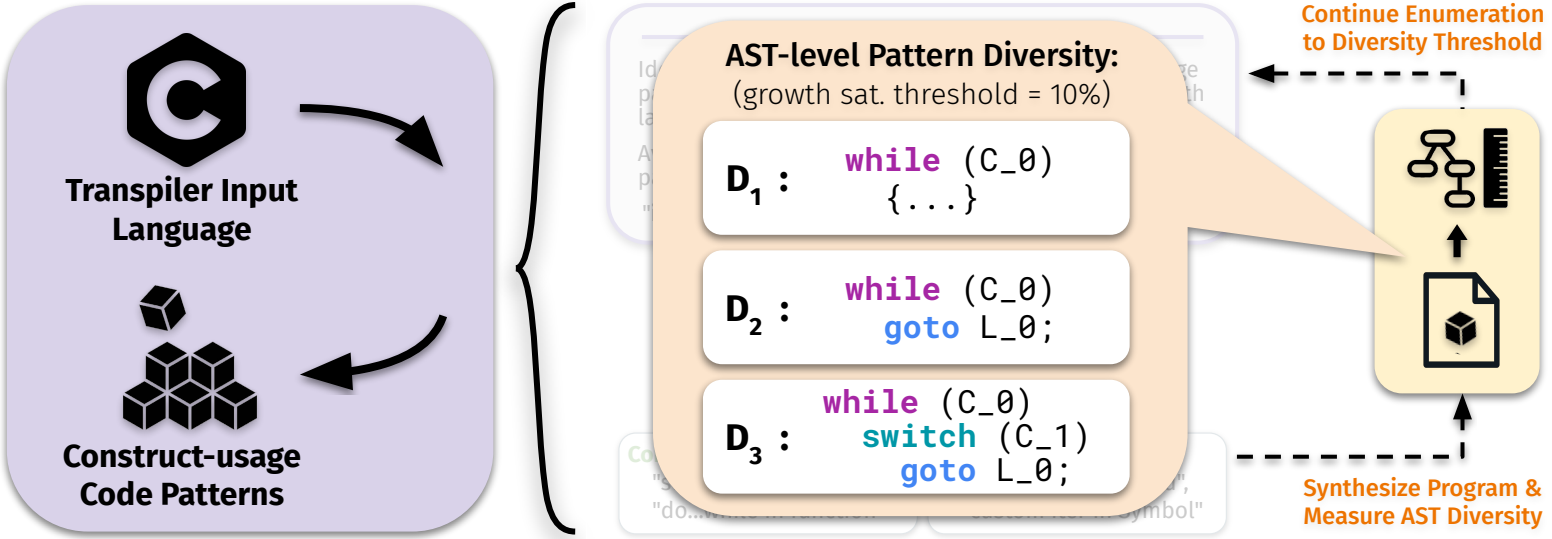
Stage 1: Construct Usage Enumeration

- **Goal:** mine **construct-usage patterns** to seed subsequent program generation
  - **Approach:** continuously prompt for novel code patterns **until pattern diversity saturates**



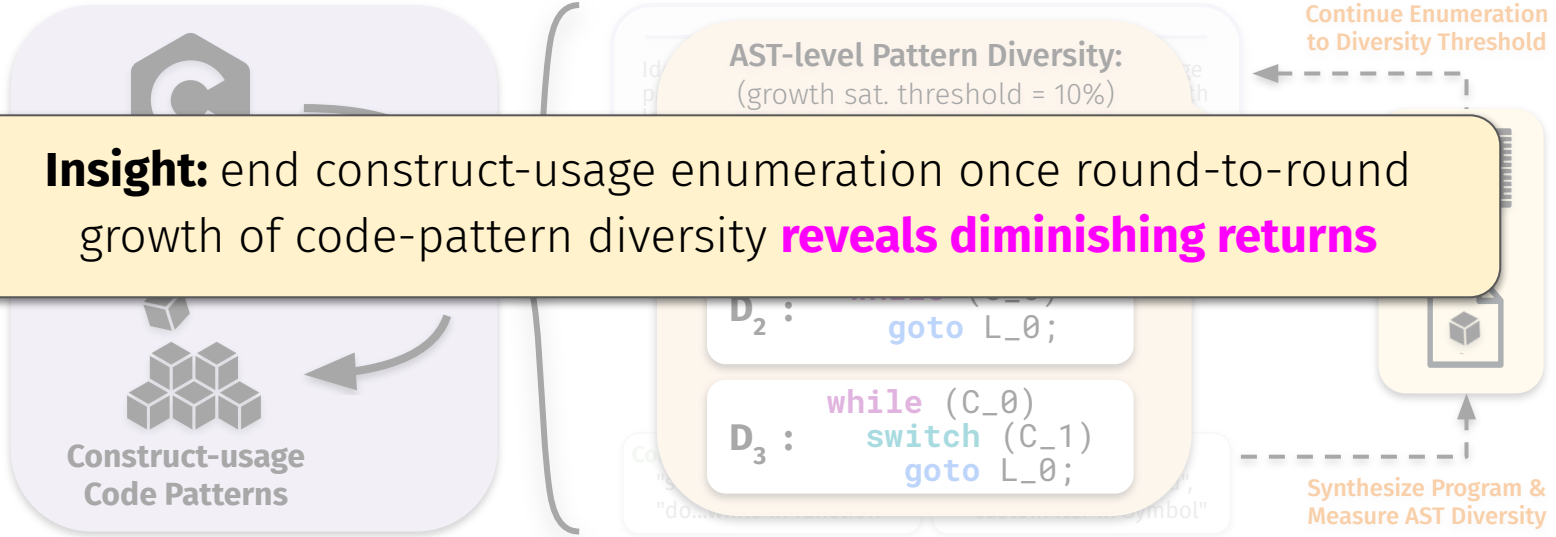
Stage 1: Construct Usage Enumeration

- **Goal:** mine **construct-usage patterns** to seed subsequent program generation
  - **Approach:** continuously prompt for novel code patterns **until pattern diversity saturates**
  - Calculated via **AST-level construct depth:** solo constructs (D_1), tuples (D_2), and up to six (D_6)



Stage 1: Construct Usage Enumeration

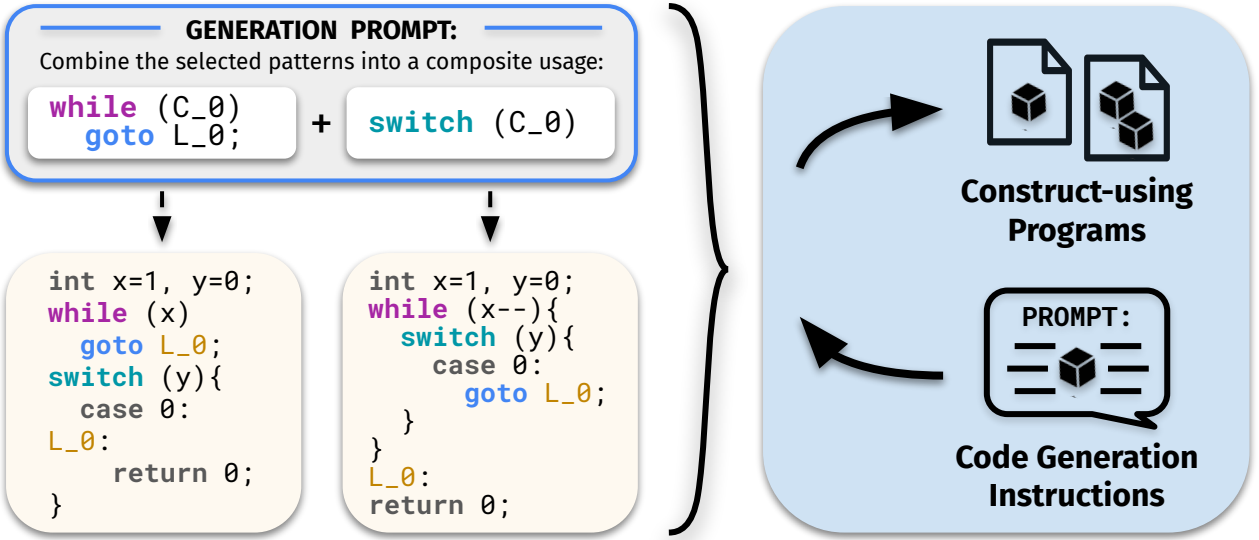
- **Goal:** mine **construct-usage patterns** to seed subsequent program generation
  - **Approach:** continuously prompt for novel code patterns **until pattern diversity saturates**
  - Calculated via **AST-level construct depth:** solo constructs ($\mathbf{D_1}$), tuples ($\mathbf{D_2}$), and up to six ($\mathbf{D_6}$)



Insight: end construct-usage enumeration once round-to-round growth of code-pattern diversity **reveals diminishing returns**

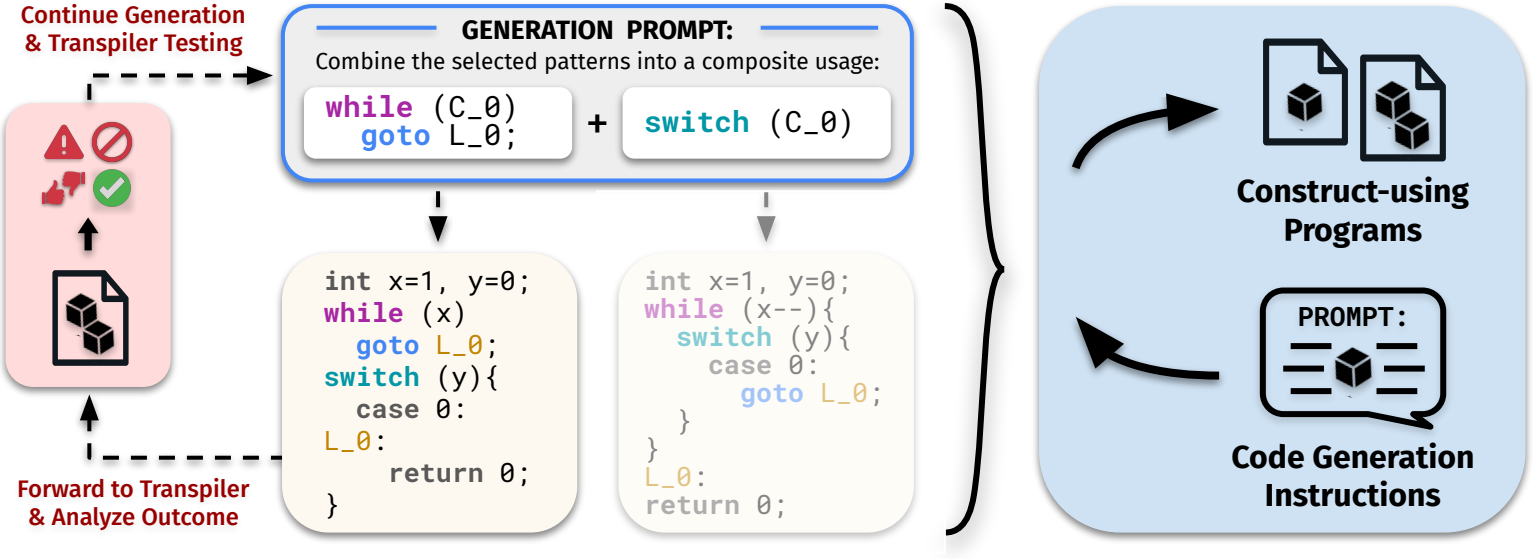
Stage 2: Construct-Oriented Program Generation

- **Goal:** create programs **containing and combining** construct-usage patterns



Stage 2: Construct-Oriented Program Generation

- **Goal:** create programs **containing and combining** construct-usage patterns
  - Following generation, finalize test case program and **forward to the targeted transpiler**
  - **Outcomes:** **crashes transpiler**, **non-runnable code**, **runtime divergence**, or **passes**



Our Solution: Construct-Oriented Transpiler Fuzzing

- **Idea:** harness LLMs' capabilities for **code reasoning** and **code generation**
  - **Construct-usage Enumeration:** broadly capture how a language's core constructs **are used**
  - **Construct-oriented Generation:** create programs **exercising and combining** these patterns

Fuzzer	Seed Agnostic	Code Diversity	Program Validity	Broad Support
AFL	✘	✘	✘	✔
Polyglot	✘	✘	✘	✘
TransFuzz	✘	✘	✔	✘
YARPGen	✔	✘	✔	✘
CSmith	✔	✘	✔	✘
GoSmith	✔	✘	✔	✘
Our Approach	✔	✔	✔	✔

Outcome: broad exploration of diverse construct-usage code patterns **without any language-specific tailoring**.

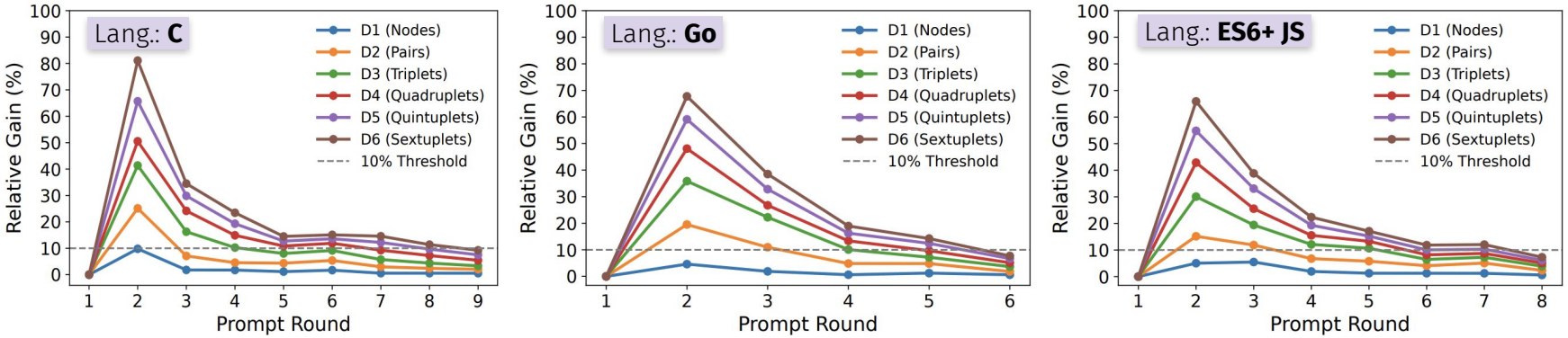
Evaluation: Overview

- **Implementation: PROGnosticator** (GPT-4)
- **Fundamental questions:**
  1. Can PROGnosticator capture **diverse code patterns**?
  2. Does PROGnosticator yield **higher quality** programs?
  3. Can PROGnosticator uncover **more translation bugs**
- Competing fuzzing approaches:
  - **CSmith** (C compiler fuzzer)
  - **TransFuzz** (JS transpiler fuzzer)
  - **Polyglot, AFL-Compiler-Fuzzer** (multi-language compiler fuzzers)
- Benchmarked on **seven transpilers** spanning six distinct language pairings

Transpiler	In Lang.	Out Lang.
C2Rust	C	Rust
CxGo	C	Go
Zig Translate-C	C	Zig
Go2Hx	Go	Haxe
TinyGo	Go	Wasm
Babel, SWC	JS ES6+	JS ES5

Evaluation: Code Pattern Diversity

- Measured **relative gain of AST-level diversity** across enumeration rounds
  - Configured generation of **90 new construct-usage patterns** per prompting round



- **Takeaways:** **peak construct diversity** captured in **earlier enumeration rounds**
  - With 10% growth threshold, diversity saturates at: **9 rounds** for C, **6** for Go, and **8** for JavaScript

Evaluation: Generated Program Diversity

- Compared **relative program diversity** across all fuzzers' final test programs
  - Competitors run for five 24-hour campaigns; PROGnosticator capped at 10K progs./campaign
  - Measured cumulative diversity as the sum of all D_1–D_6 **AST-level diversity** per program corpus

Metric	PROGnosticator vs. CSmith	PROGnosticator vs. TransFuzz	PROGnosticator vs. Polyglot	PROGnosticator vs. AFL-Comp-Fuzz
Rel. Mean Cumul. Diversity	**+ 4.23×**	**+ 3.85×**	**+ 4.15×**	**+ 23.26×**

- **Takeaways:** construct-oriented fuzzing sees **highest code pattern diversity**
  - Prior fuzzers limited to constructs derived from **seed inputs and/or hardcoded grammars**

Evaluation: Generated Program Validity

- Compared **relative program validity** across all fuzzers' final test programs
  - Competitors run for five 24-hour campaigns; PROGnosticator capped at 10K progs./campaign
  - Measured proportion of generated programs that **compile and execute** pre-transpilation

Metric	PROGnosticator vs. CSmith	PROGnosticator vs. TransFuzz	PROGnosticator vs. Polyglot	PROGnosticator vs. AFL-Comp-Fuzz
Rel. Mean Validity	**- 0.22×**	**- 0.08×**	**+ 2.43×**	**+ 55.16×**

- **Takeaways:** construct-oriented fuzzing **upholds high validity on all languages**
  - Trades-off negligible language-specific precision for **broader language support**

Evaluation: Discovered Transpiler Bugs

- Compared **total translation bugs** across all transpiler fuzzing campaigns
  - **Bug categories:** **transpiler crashes**, **non-runnable code**, and **runtime divergences**
  - All bugs manually deduplicated and reported to their respective developers

Metric	CSmith	TransFuzz	Polyglot	AFL-Comp-Fuzz	PROGnosticator
Total Bugs Found	6	0	0	0	77
Newly-found Bugs	1	0	0	0	64

- **Takeaways:** construct-oriented fuzzing **surfaces the most transpiler defects**

Evaluation: Discovered Transpiler Bugs

- Compared **total translation bugs** across all transpiler fuzzing campaigns
  - **Bug categories:** **transpiler crashes**, **non-runnable code**, and **runtime divergences**
  - All bugs manually deduplicated and reported to their respective developers

Transpiler	New Bugs	Confirmed	Fixed
C2Rust	**15**	**14**	**11**
Zig Translate-C	**9**	**9**	**0**
CxGo	**11**	**11**	**0**
TinyGo	**2**	**2**	**2**
Go2Hx	**15**	**15**	**10**
Babel	**2**	**2**	**1**
SWC	**10**	**10**	**9**

- **Takeaways:** construct-oriented fuzzing **surfaces the most transpiler defects**
  - Of PROGnosticator's 64 newly-found bugs, **63 are confirmed**, with **33 fixed post-reporting**

Case Studies: PROGnosticator-found Bugs

- **C2Rust #1165:** transpiler crash on **incrementing within a compound literal**
  - C2Rust successfully parses compound literals containing side-effect-free field initializers
  - Yet, it crashes when encountering an initializer containing a **post-increment expression**

```
struct s { int i; };

int j = 42 ;

struct s *p = &((struct s){ j });

struct s *p = &((struct s){ j++ });
```

C2Rust outcome:

error: Failed to translate: Expected no statements in field expression.

- **Takeaways:** construct combinations often expose **internal heuristic errors**
  - C2Rust erroneously treated compound-literal fields as strictly side-effect-free

Case Studies: PROGnosticator-found Bugs

- **Zig #23987:** invalid code caused by **bitwise negation on boolean results**
  - Zig Translate-C uses functions (e.g., `intFromBool`) to resolve Zig's lack of implicit casts
  - While it handles boolean-to-integer conversions, it misses them inside **bitwise negation**

```
int x = (5 > 3);

int x = ~(5 > 3);
```

Input C code

```
var x: c_int = intFromBool(as(c_int,5)
                  > as(c_int, 3));

var x: c_int = ~(as(c_int, 5)
                  > as(c_int, 3));
```

Outcome: **uncompilable Zig code**

- **Takeaways:** many subtle bugs arise from **context-sensitive type semantics**
  - Zig Translate-C misses necessary type conversions under nested contexts

Case Studies: PROGnosticator-found Bugs

- **Go2Hx #256:** runtime divergence on **blank-identifier tuple unpacking**
  - Go supports multiple assignments with "_", preserving the remaining value order
  - Go2Hx mishandles "_" during tuple unpacking, binding `res` **to the wrong field**

```
type S struct{ a, b, c int }
p := S{1, 2, 3}
_, res, _ = p.a, p.b, p.c
print(res)
```

Go program output: 2

```
var p = S{1, 2, 3};
var _0 = p.a, _1 = p.b, _2 = p.c;
var _, _, res = _0, _1, _2;
print(res);
```

Translated Haxe output: 3

- **Takeaways:** correct translation requires **preserving value-flow semantics**
  - Go2Hx failed to match value positions across discarded assignment slots

Case Studies: PROGnosticator-found Bugs

- **TinyGo #4786:** transpiler crash from `len()` on a nil array pointer
  - Go returns the static array size for `len(*p)`, even when p is a nil pointer
  - TinyGo instead incorrectly dereferences p, **causing a nil-pointer panic**

```
func foo() *[4]byte {
      return nil
}

func main() {
      println(len(*foo()))
}
```

Go program output: 4

TinyGo outcome:

panic: runtime error at 0x203697:
nil pointer dereference

Go-produced SSA:

```
t0 = foo() // nil
t1 = *t0   // bug
println(4)
```

- **Takeaways:** transpiler bugs can also stem **from upstream tooling errors**
  - TinyGo inherited the dereference **from Go's own SSA**, yet Go prunes it in its own compilation

Conclusion: Why Construct-Oriented Fuzzing?

- Transpiler errors **impede translation-driven tasks**
  - Hence, transpiler correctness demands thorough testing
- Existing fuzzers remain **ineffective on transpilers**
  - Most transpiler bugs stem from core language constructs
  - Yet, current fuzzers **struggle to systematically cover them**
- **Our solution: Construct-Oriented Fuzzing**
  - Utilize LLMs' knowledge of language syntax & semantics to explore diverse patterns of core language constructs
  - Further harness LLMs to generate programs embodying and combining retrieved construct-usage code patterns
  - **Outcome:** broad, language-agnostic transpiler fuzzing culminating in **the most transpiler bugs found to date**

Key Results:

2–55× higher code validity,

4–23× higher code diversity,

64 new bugs (**63** confirmed)

Thank you!

github.com/FuturesLab/PROGnosticator

Contact:
y.arafat@utah.edu
@yeaseen_
@yeaseen.bsky.social

FUTURES^3
LAB FUTURE TECHNOLOGY FOR USABLE, RELIABLE, & EFFICIENT SECURITY OF SOFTWARE & SYSTEMS