Re-Testing LLM-Based C-to-Rust Translations Against Their Original C

Muhammad Hassnain
University of California, Davis
mhassnain@ucdavis.edu

***Abstract*—Most C-to-Rust translators use large language models (LLMs), and each reports a success rate from its own correctness check. We survey 34 C-to-Rust translators that use LLMs. Only one reports testing its check, and eleven release the original C and their Rust in runnable form. We re-test all eleven. We run the C and the Rust on generated inputs, compare every output, and count a divergence only where the C has no undefined behavior. All eleven release defective translations: we find 160 defects, 93 of them outright errors on ordinary inputs. Their own checks accepted 86 defective translations. For 81 of them, the released check never compared the programs on an input that exposes the defect. For three of the other five, it ran one but compared only part of the output. A success rate is only as good as its check. Before counting a translation as a success, authors should compare every output with the C's on inputs the repair loop never saw, check that a verifier reaches its assertion, release exactly what the check accepted, and report how they tested the check.**

***Index Terms*—C-to-Rust translation, differential testing, test oracles, translation validation, LLM code translation**

I. Introduction

Memory-safety errors are a leading cause of security vulnerabilities in C and C++ software, and the U.S. government has urged a move to memory-safe languages such as Rust [1]. Rewriting existing C by hand is costly, so DARPA's TRACTOR program [2] and a growing number of research systems translate C to Rust automatically, most with large language models (LLMs). An LLM can produce Rust that compiles yet behaves differently from the C. Each system therefore checks its translations with a fuzzer, a model checker, symbolic execution or tests, and reports the share that pass as its success rate.

Memory safety alone does not make code trustworthy, even in safe Rust [3]; trust needs evidence, such as test and verification results [4]. A success rate is only as good as the check behind it, yet few papers examine their checks. We survey 34 C-to-Rust systems (Section II). Of the 25 whose repair loop runs inputs, six report results on inputs the loop never used. Only one tests its check on known-wrong translations. Eleven release the original C and their own Rust in runnable form.

We re-test the released translations of all eleven (Fig. 1). The original C is the *oracle*: the translation claims to preserve its behavior. For each system, a test *harness* runs the C and the Rust on generated inputs and compares every output; a divergence counts only on inputs where we find no undefined behavior (UB) in the C. Reading the Rust source, we group divergences into *defects*, one per faulty mechanism. From each artifact we then measure whether the system's own check *accepted* each defective translation and ran an exposing input. We ask:

- RQ1: Do released translations preserve the behavior of their original C on inputs where the C is well-defined?
- RQ2: What did each system's own check record for the translations that do not, and did it run an input that exposes the difference?
- RQ3: How robust are these findings to undefined behavior and preconditions in the C, to the toolchain and build profile, and to the power of our harnesses?

All eleven release defective translations (RQ1): 160 defects, of which 93 are *outright errors*, wrong outputs or crashes on ordinary inputs. Their checks accepted 86 defective translations, 56 with an outright error (RQ2). For 81 of the 86, the released check never compared the programs on an exposing input. For three of the other five, it ran one but compared only part of the output. FLOURINE's own check rejected 20 files that its artifact ships as successes; six are defective. Mutation analysis measures our harnesses' power (RQ3): they catch from 20% of planted bugs (C2SaferRust) to 91% (PtrTrans).

We contribute a survey of 34 systems (Section II), a method for re-testing released translations against their original C (Section III), and a re-test of all eleven with measured harness power (Section IV).

II. Selection of Translators

We include every C-to-Rust system that uses an LLM to write or repair the Rust and was first public by 31 October 2026. Its paper must contribute a translation pipeline, not a benchmark or study, and report a success rate from an automated correctness check. We searched arXiv and Crossref and followed the related work of each system we found. Of 69 C-to-Rust candidates, 34 qualify. From each paper we record whether the evaluation is *held out*, using inputs that the repair loop (where the LLM revises the Rust) never saw. We also record whether the authors *mutation-test* the check by feeding it known-wrong translations.

Six of the 34 report a held-out result [5]–[10], two of them for a subset only. Nineteen do not. In eight, the repair loop runs no inputs, so every evaluation input is new to it. Heimdall's check is symbolic and runs no inputs at all; it is the one system that mutation-tests its check [11]. No system does both.

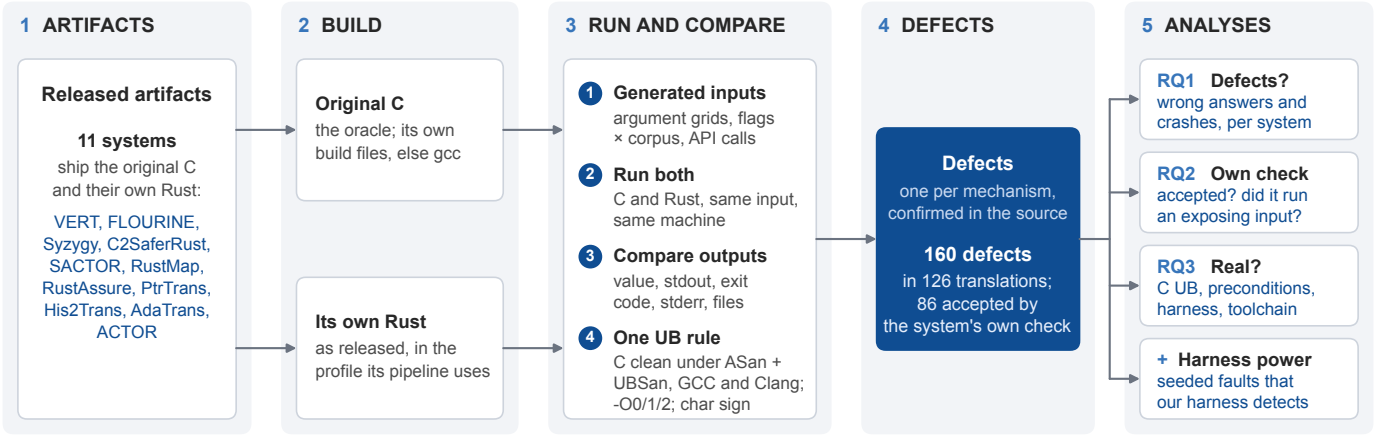


Fig. 1. Re-testing released translations. For every system whose artifact ships the original C and its own Rust (1), we build both (2), run them on generated inputs, compare every output, and keep a divergence only on inputs where the C passes our UB rule (3). We group the divergences into defects, one per mechanism, and confirm each in the translated source (4). The analyses (5) answer RQ1-RQ3 and measure our harnesses' power by planting bugs.

TABLE I
THE SURVEYED SYSTEMS WHOSE ARTIFACTS MEET OUR RE-TEST RULE (ALL: EVERY SHIPPED TRANSLATION QUALIFIES; PART: SOME DO). HELD-OUT: EVALUATED ON INPUTS ITS REPAIR LOOP NEVER USED (-: THE LOOP RUNS NO INPUTS). MUT.: CHECK MUTATION-TESTED.

System	Own check	Held-out	Mut.	Art.
VERT [13]	fuzzer, model checker vs. a Rust copy of the C	No‡	No	all
FLOURINE [14]	cross-language fuzzing	No	No	all
Syzygy [15]	equivalence tests vs. the C	No	No	all
C2SaferRust [16]	coreutils end-to-end tests	No	No	all
SACTOR [17]	end-to-end tests via FFI	No	No	part
RustMap [18]	functional tests; log diff	No	No	all
RustAssure [19]	KLEE symbolic testing	-	No	part
PtrTrans [20]	equivalence unit tests	-	No	part
His2Trans [21]	project tests	-	No	part
AdaTrans [7]	fuzzing oracle	Yes§	No	all
ACTOR [8]	benchmark hidden tests	Yes	No	all

‡For the fuzzer; its model checker runs no inputs. §Per its Sec. 5.1; its Sec. 3.4 says the repair loop also runs a fuzzing suite.

We re-test every artifact that ships the original C and the system's own Rust, both runnable in user space. Eleven systems qualify (Table I). Heimdall's programs are eBPF kernel hooks, not user-space programs, and we could not build the C that Rustine [12] ships.

III. METHOD

What we count. A system releases *translations*: functions, libraries or programs. One run of the C and the Rust on the same input is a *case*; the case *diverges* if their outputs differ. For each divergence we find the cause in the Rust source. A *defect* is one such cause: a mechanism that makes the translation give a wrong output or crash on an input where the C is well-defined. All cases with the same cause are one defect. We report defects and the translations that contain them; case counts only show how often a defect appears. A divergence is a *wrong output* (a different return value, exit code or primary output), a *crash* (a Rust panic or segmentation fault where the C completes), a *stderr-only difference*, or a *build failure or hang*; only the first two make defects.

What we do not count. We do not count a divergence that occurs only (i) where the C has undefined behavior, (ii) on inputs that break a precondition the original states in its name, comments or documentation, (iii) in a value the C standard leaves unspecified, (iv) in a callback that frees memory, or (v) when the C passes a negative value to a `scanf %u` or `printf %x/%u` conversion, which the standard does not define. We added rules (iii)-(v) during triage, after seeing the cases they cover. Of 180 candidate defects, 12 fall to (i) and (ii) and 3 to (iii)-(v); 4 more are hangs or build failures, and 1 has no cause we could confirm. An LLM-based assistant grouped cases into defects and assigned their classes and exclusions. An author, blind to these labels, labeled a stratified sample of 12 cases (9 defects, 3 exclusions). The two agreed on the cause for all 9 and the class for 8, and differed on 2 count decisions: a precondition call, and an ACTOR parsing defect our rule (v) had dropped, which a re-run confirmed and we now count.

What the systems' checks recorded. For each defective translation we look up what the system's own check recorded. It is *accepted* if the artifact records a pass and *rejected* if it records a failure; for a check with tiers, a pass at any tier the system counts as success is a pass. If the artifact records nothing, we re-run the documented check on the system's own inputs; if we cannot, the translation has *no record*. An input is *held out* if the system's check (in the survey: its repair loop) never ran it. We compare against the inputs the artifact records or, for VERT, those its released commands run; otherwise held-out status is *not measurable*.

How we run and compare. We build the original C and the system's released Rust, run both on the same inputs, and compare every observable output: a function's return value (in four systems, floats within 10^{-6}: relative, or absolute below magnitude 1); a program's standard output, exit code, standard

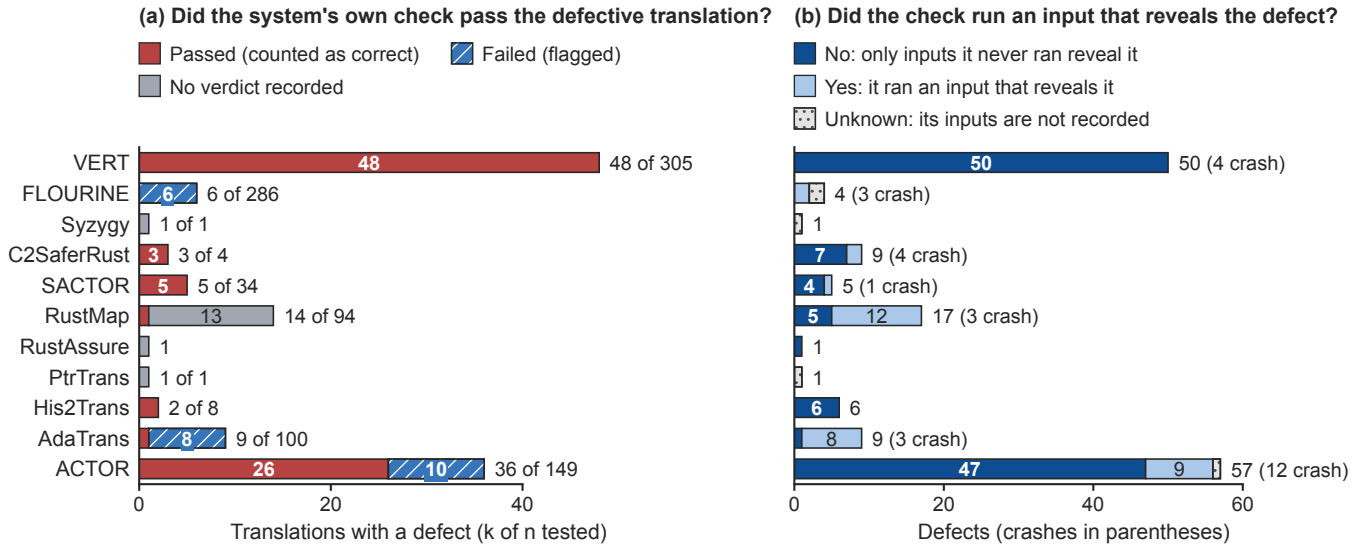


Fig. 2. Re-test outcomes per system. (a) Translations with a defect, by what the system's own check recorded: passed (*accepted*), failed (*rejected*), or no verdict; "k of n": defective of tested translations. (b) Defects, crashes in parentheses, by whether the check ran an exposing input: no (*held out*), yes, or unknown (check inputs not recorded).

error and written files; and a library's sequence of results and callbacks. Where a system's check compares an output, we never compare it more strictly (for RustMap's Rosetta programs we normalize whitespace, case and float precision). Where our compiler differs from the one an artifact pins or documents (VERT, RustAssure, SACTOR, ACTOR), rebuilding with that compiler changes no verdict. We build each system's Rust in the profile its own pipeline uses (His2Trans: debug and release; FLOURINE: release with overflow checks). We re-run every counted crash and the 58 wrong outputs in integer code in the other profile: 86 of 88 still diverge from the C; the other two are FLOURINE overflow panics.

Undefined behavior in the C. Our oracle is the C that GCC and Clang build for LP64 Linux with glibc. We take its documented implementation-defined choices (e.g. arithmetic right shift, 64-bit `long`) as part of the C. On an input where the C has UB, any Rust result is consistent with the C. A divergence therefore counts only on an input where the C (i) passes the address and UB sanitizers (ASan, UBSan) under both GCC and Clang (including float-to-integer overflow); (ii) gives identical results at `-O0`, `-O1` and `-O2`, which can reveal uninitialized reads that ASan does not detect; and (iii) gives identical results with signed and unsigned `char`. Per-system scripts apply this rule; an input whose sanitizer, optimization-level or `char` run fails or times out does not count. For VERT the full rule re-checks every exposing input. We exclude by hand two ACTOR defects: the C of `underhanded-c-luggage` reads stale stack memory, which the sanitizers miss, and that of `strcmp` passes an out-of-range value to `atoi`.

How we generate inputs. Fixed scripts generate every input except three that an author wrote for one ACTOR defect; we use no language model. The scripts follow the shape of the code under test. A function's integer arguments take the ends of their range, powers of two ± 1 and values in between, fewer per argument at higher arity, and its array arguments take fixed patterns. A program runs with single command-line flags and pairs of them, up to 3000 cases per utility. It reads a fixed set of files (empty, text, binary, multibyte, NUL-delimited, large) from standard input, a file, a missing file or a directory. A library runs fixed sequences of API calls, and a compressor a fixed set of byte strings, one of them incompressible.

How we test our harnesses. A harness that misses bugs would make wrong translations look correct, so we measure how many planted bugs ours catch, by mutation analysis [22], [23]. We plant one bug at a time in a translation that agrees with the C: we replace a relational or arithmetic operator, shift a constant by one, or, for VERT, compute an integer expression in `f32`. We then re-run the harness. It *catches* the planted bug, or *mutant*, if it reports a wrong output or a crash on one of our inputs where the C has no UB. A difference only in standard error, a hang, or a failure of the system's own tests does not count. We report the share of compiled mutants caught per system. This share is a lower bound where some mutants hang (VERT, AdaTrans, ACTOR) and an upper bound where the records do not separate standard-error and timeout catches from the rest (SACTOR).

IV. Results

Every system releases defective translations, and in seven of them the system's own check accepted some (Fig. 2a). Most defects, 121 of 160, appear only on inputs the system's check never ran (Fig. 2b). Figures 4 to 7 show four defects as released. All runs use Docker images built from digest-pinned bases, with the C and the Rust built for the same architecture.

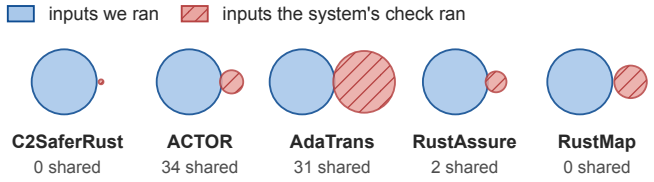


Fig. 3. Inputs we ran and inputs the system's own check ran, where its artifact records them; areas and overlaps are to scale within each pair. We ran 8574, 9130, 8310, 360 and 20 inputs; the checks ran 47, 1164, 7477, 37 and 5, respectively (ACTOR: executables; RustMap: bzip2; RustAssure: 37 listed, 9 reached).

Overlap with the checks' inputs. Our inputs rarely coincide with those the systems' checks ran (Fig. 3), which is why most defects are held out. None of our FLOURINE inputs is among its logged examples. For VERT we take the inputs its released commands run: none for the fuzzer, which ships no corpus for these functions, and $n \in \{0, 1\}$, which Kani draws for the Rust copy of the C only. Syzygy, PtrTrans, the ACTOR libraries and RustAssure's per-function check record no check inputs; Fig. 3 shows RustAssure's libcsv composition test.

A. VERT

VERT [13] accepts a translation that passes two checks against an *rWasm copy* of the C, the C re-expressed as Rust via WebAssembly: the fuzzing front-end *bolero* [24] and, where it can, the bounded model checker *Kani* [25]. We test 189 integer functions (arguments in $[0, 512]$, 80 to 100 inputs each) and 116 array and float functions. Of these, 48 contain 50 defects (46 wrong outputs, 4 crashes), and VERT's records mark all 48 as passing bolero and 21 also bounded Kani (Fig. 4 shows one). In 8 of them the C returns a 64-bit integer: in 2 the Rust computes a wrong value, and in 6 it returns a 32-bit one that overflows. The checks pass these translations because, as released, neither compares the programs on an exposing input. The Kani harness evaluates the Rust at the fixed value 12 before the rWasm copy draws a symbolic $n \in \{0, 1\}$, so the equivalence assertion is unreachable in all 21 defective passes. The bolero command only replays a saved corpus, which 2 of 506 benchmarks ship. Fuzzing each for 30 s instead reports a failure for 24 of the 36 defective scalar functions, only 6 of them real divergences where the C has no UB. VERT's records do not say which mode produced their passes. Our harnesses catch at least 77.7% (scalar) and 59.7% (array) of planted bugs. We do not count 6 further functions that diverge only on negative or larger inputs, outside the domain we test.

B. FLOURINE

FLOURINE [14] accepts a translation that its differential fuzzer finds equivalent. Of the 286 shipped `success` files we test (101 scalar functions, tested on up to 2000 signed inputs each with $|x| \leq 2^{20}$, and 185 struct accessors), 6 contain one of 4 defects. A byte packing became a sum because the Rust shifts a `u8` (Fig. 5). In two functions a mask `(1<<len)-1` became `(1>>len)-1`, which underflows, and in a third a multiplication overflows where the C's unsigned arithmetic

VERT · HEXAGONAL_NUMBER source ↗

n = 3: C returns 15, the Rust 9 (it computes n^2)

C HEXAGONAL_NUMBER.c:21–23 ↗

```c
21 int f_gold ( int n ) {
22   return n * ( 2 * n - 1 );
23 }
```

Rust main.rs:388–396 ↗

```rust
388 fn f_gold(n: i32) -> i32 {
    . . .
392     result = 0;
393     for i in 1..=n {
394         result += i * 2 - 1;
395     }
396     result
```

Fig. 4. VERT's `HEXAGONAL_NUMBER`: the Rust sums a series that equals n^2, where the C computes $n(2n-1)$. VERT's records mark it as passing bolero and bounded Kani.

FLOURINE · opl_emu_registers_operator_list source ↗

shifting a u8 by 8, 16, 24 wraps to a no-op: (24,211,56,46) gives 337, not 775,475,992

C opl_emu_registers_operator_list.c:16–19 ↗

```c
16 uint32_t opl_emu_registers_operator_list(uint8_t o1, uint8_t o2, . . .
17 {
18     return o1 | (o2 << 8) | (o3 << 16) | (o4 << 24);
19 }
```

Rust opl_emu_registers_operator_list.rs:4–10 ↗

```rust
4 fn opl_emu_registers_operator_list(o1: u8, o2: u8, o3: u8, o4: u8. . .
5     let o1 = o1 as u32;
6     let o2 = o2.wrapping_shl(8) as u32;
7     let o3 = o3.wrapping_shl(16) as u32;
8     let o4 = o4.wrapping_shl(24) as u32;
9     o1.wrapping_add(o2).wrapping_add(o3).wrapping_add(o4)
10 }
```

Fig. 5. FLOURINE's byte packing: shifting a `u8` by 8, 16 or 24 wraps to a no-op, so the Rust adds the bytes. FLOURINE's own log rejects this file, which its artifact ships as a success.

wraps; all three panic in FLOURINE's build. FLOURINE's own logs reject all six: each is an experiment's first candidate, logged as not equivalent; FLOURINE does not ship the accepted one. This mismatch affects 20 of the 286 files. We do not count divergences on ranges whose minimum exceeds the maximum, though FLOURINE's fuzzer does. Our scalar harness catches 88% of 632 planted bugs.

C. RustAssure

RustAssure [19] checks each translated function by differential symbolic execution under KLEE [26]. Of the 151 GPT-4o functions in its archived run, 2 run in isolation (both agree), so we also test the two libraries it ships in runnable form. In libcsv with `CSV_EMPTY_IS_NULL` set, the C passes the field callback a NULL pointer for an empty unquoted field. GPT-4o's `csv_parse` instead made the parameter a non-nullable slice, so `a,,b` and `a,"",b` become indistinguishable (12 of 360 cases). RustAssure records no verdict for `csv_parse`. In its artifact, the per-function check also replaces calls through function pointers with stubs, so it cannot observe a callback. In u8c, a decoding divergence that RustAssure's own bug list records depends on the signedness of `char` in the C, so we do not count it.

C2SaferRust · `tail -c / -n +N` source ↗

the C sets the global count_lines; the Rust parses into a new local that only picks an error message, so count_lines keeps the true that main sets and tail -c 3 prints 3 lines, not 3 bytes

C tail.c:169–2213 ↗

```
169 static bool count_lines;
    . . .
2211 case 'c':
2212 case 'n':
2213   count_lines = (c == 'n');
```

Rust tail.rs:835–4667 ↗

```
835 static mut count_lines: bool = false;
    . . .
4649     99 | 110 => {
4650         let is_count_lines = c == 'n' as i32;
    . . .
4667     if is_count_lines {
```

Fig. 6. C2SaferRust's `tail`: the Rust parses `-c` into a new local instead of the global `count_lines`, so `tail -c 3` prints three lines, not three bytes. The log records this rewrite as passing.

His2Trans · `buffer_append_n` source ↗

the length argument is ignored: append_n("hello", 1) appends "h" in C, "hello" in Rust

C buffer.c:195–201 ↗

```
195 buffer_append_n(buffer_t *self, const char *str, size_t len) {
196   size_t prev = strlen(self->data);
197   size_t needed = len + prev;
    . . .
200   if (self->len > needed) {
201     strncat(self->data, str, len);
```

Rust src_buffer.rs:154–159 ↗

```
154 pub fn buffer_append_n(
155     b: &mut buffer_t,
156     data: &[u8],
157     _len: size_t,
158 ) -> c_int {
159     let len = data.len();
```

Fig. 7. His2Trans's `buffer_append_n` ignores its length argument, so appending one byte of `"hello"` appends all five. His2Trans's results and our re-run of its tests pass this library.

D. C2SaferRust

C2SaferRust [16] rewrites C2Rust's rule-based translation of coreutils [27] with an LLM. Its artifact keeps a rewrite only if the test scripts that passed on the starting translation still pass. We test `cat`, `head`, `tail` and `uniq` (four of its seven utilities) on 8574 cases (5942 distinct), including the file `uniq` writes when given an output operand. We find 9 defects in three, all from rewrites the log records as passing (the C2Rust starting point agrees with the C). `tail` binds its parsed `-c`/`+N` mode to locals and never stores them, so `-c N` runs as `-n N` (Fig. 6). `head` drops a final partial buffer when it reads a pipe, and on x86-64 its inherited C2Rust code reads regular files that way too. It loses 4,095 bytes of its own acceptance test's file; the test misses this because it sends the output to `/dev/full`. Four defects crash. In our re-run, the shipped translation fails 2 of the 27 tests its loop selects across its seven utilities, one because of the `tail` defect. `uniq` also segfaults, but the C performs `NULL+0` pointer arithmetic (UB in C17) on every input `uniq` reads, so we do not count it. Our harness catches 20% of 1892 bugs we plant in the C2Rust starting point (19.2% of 1074 in C2SaferRust's output). Of the 935 survivors we could not show equivalent, 619 lie in code our inputs never execute.

E. Syzygy

Syzygy's [15] translated Zopfli compressor emits nothing on empty input, where the C's `do-while` always emits a final block. Its paper reports that it passed 10^6 random inputs of 10 to 10^7 characters, a range that excludes the empty input. It ships neither a verdict nor those inputs.

F. RustMap

RustMap [18] ships a translated bzip2 and 124 Rosetta Code programs, of which 93 build in both languages; with bzip2 we test 94. Its bzip2 check salvages the Rust's output with `bzip2recover`, decompresses it with the C, and passes on RustMap's test files. The translation has four defects. Input bytes $\geq$`0x80` become `0xFF`, the Rust encodes runs of identical bytes wrongly, and a 1-byte input panics. Every output also starts with 5000 zero bytes, which plain `bzip2 -d` rejects, also on RustMap's own files. The Rosetta programs read no input; RustMap's own run never reaches 2 of them. Its paper reports that all 125 give the same output as the C. Its shipped whole-log diff, however, differs on every program, so it records no per-program verdict. After normalizing, the outputs differ for the 13 programs we count (e.g. a Roman-numeral decoder reads `MCmxC` as 1000; two knapsack solvers keep 1 of 22 items and 1 of 9). Six more implement a different task, which we set aside. We run these on x86-64.

G. His2Trans

His2Trans [21] reports how many of a translated library's tests pass. We test the 8 libraries of its open-source benchmark function by function and find 6 defects in 2. Its results and our re-run of its tests pass all 8. `buffer` has five (e.g. `buffer_append_n` ignores its length argument, Fig. 7) and `rgba` one. For `buffer`, His2Trans runs 14 tests of its own instead of the project's, none of which calls `buffer_append_n`; the project's own test for it fails on the shipped Rust. One `buffer` defect makes the Rust correct where the C His2Trans ships is not; it counts because the C is the oracle. We do not count `genann`'s replaced `rand()`, whose sequence ISO C leaves unspecified. Our harness catches 53% of 113 planted bugs.

H. SACTOR

SACTOR [17] accepts a translation if the C project's end-to-end tests pass against the Rust through the C interface. SACTOR does not ship its headline translations; we test the 34 shipped ones that are complete, keep the C interface (its unidiomatic stage) and build. Five contain a defect, and SACTOR's check accepts all five. In Holdem-Odds, a nested stub returns `true` for every card; SACTOR's own test prints the failure but exits 0, and the check reads only the exit code. Two other defects corrupt a buffer or crash at the C interface.

Four of the five diverge only on inputs SACTOR's tests do not run. Our harnesses catch at most 63% of 109 planted bugs.

I. PtrTrans

PtrTrans [20] ships its own Rust for one of its 16 projects, `quadtree`, whose search never descends below the root. Its paper reports `quadtree` as 87.5% equivalent; it ships no per-function verdict or test inputs. Our harness catches 91% of 64 bugs we plant in the crate after repairing its search.

J. ACTOR

ACTOR [8] translates a DARPA test corpus. It counts a translation as correct if it passes the corpus's hidden test vectors. We test the 149 of its 211 translations (all 61 executables and 88 libraries) whose interfaces take only scalars or strings and that have a linkable C oracle. Of these, 36 contain 57 defects; ACTOR counted 26 of the 36 correct. About half the defects read numbers differently: Rust's `str::parse` rejects inputs that `scanf` or `strtol` accept, such as `" 12"` or `3.14` for an integer. Of the 26, 14 differ only in parsing numbers, 9 crash on non-UTF-8 input or missing tokens, and 3 differ in NaN spelling, long lines, or a right shift of a negative value. Our harness, run on ACTOR's 1129 replayable recorded inputs, finds defects only in translations ACTOR rejected. Our harness catches at least 42.2% of 393 bugs we plant in 84 of its 94 agreeing translations.

K. AdaTrans

AdaTrans [7] counts a translated LeetCode-style program as solved if it passes its repair loop's example tests and then a fuzzing oracle. The loop compares the Rust with per-problem expected outputs. For 7 of the 100 problems whose translation builds, the C does not reproduce these outputs. Of the 100, 9 contain a defect, and AdaTrans counts one of them as solved, though it panics at $n = 1$, a value no fuzzing input reaches. Three of the nine follow the recorded outputs, not the C. Our harness catches at least 75% of 410 planted bugs.

V. Discussion

Findings

RQ1. *Do translations preserve the C's behavior?* Not always. All eleven systems release defective translations: 160 defects in 126 of the translations we test. Of the defects, 93 are outright errors. Another 54 are *edge-input* defects, on malformed or extreme inputs on which the C is defined; in 12 the Rust follows a bug fix or a recorded expected output instead of the C, and 1 right-shifts a negative value.

RQ2. *What did the checks record, and did they run an exposing input?* Their own checks accepted 86 of these translations, in seven systems; 56 contain an outright error. For 81 of the 86, the released check never compared the programs on an exposing input. For three of the other five, it ran one but compared only part of the result: the exit code of a test suite that exits 0 even when its tests fail, the exit status and error message only, or an output that a recovery tool repaired.

RQ3. *How robust are these findings?* Every counted defect has an exposing input where the C passes our UB rule, and we confirmed each in the source. All but two of the 30 crashes and 58 wrong outputs in integer code also diverge in the other build profile. Where our compiler differs from the one an artifact pins, rebuilding with that compiler changes no verdict. Our harnesses catch from 20% of planted bugs (C2SaferRust) to 91% (PtrTrans).

What the checks miss. Where artifacts record inputs, 105 of the 109 defects in accepted translations lie on inputs the check never ran. Of the other four, three pass the partial comparisons in RQ2 and one (C2SaferRust's `tail`) fails our re-run of the check. Where a check compared fully, it worked: ACTOR's inputs expose defects only in translations it rejected, and FLOURINE's fuzzer rejected every defective file we found.

Recommendations. Four practices address the problems we report: comparing every output on held-out inputs, checking that a verifier reaches its assertion, releasing exactly what the check accepted, and reporting tests of the check.

VI. Related work

We use differential testing [28], [29] with the original program as the oracle [30]. Sanitizers [31] and mutation analysis [22], [23] validate our oracle and harnesses. None of VERT's 90 recorded bounded-Kani passes reaches the assertion; 81 are *vacuous* [32], [33]. EvalPlus [34] showed that weak tests overstate the correctness of LLM-generated code. Pan et al. [35] catalogued the bugs LLMs introduce when translating code. Among C-to-Rust systems, ACToR [5] (not the ACTOR we test) searches adversarially for inputs where C and Rust disagree, and Heimdall [11] tests its symbolic check on known-wrong translations.

VII. Availability

The artifact, with all files and results, is at https://github.com/muhammad-hassnain/retest-c-to-rust.

Use of AI. Claude Code (Claude Opus 5.5) helped write code, triage divergences, label defects and draft this paper.

VIII. Conclusion

We re-tested eleven C-to-Rust systems against their original C: all release defective translations, and their checks accepted 86 of them. A success rate earns trust when authors report how they tested the check and release exactly what it accepted.

References

[1] Office of the National Cyber Director, "Back to the building blocks: A path toward secure and measurable software," The White House, https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf, 2024.

[2] DARPA, "Translating all C to Rust (TRACTOR)," https://www.darpa.mil/research/programs/translating-all-c-to-rust, 2024.

[3] M. Hassnain and C. Stanford, "Counterexamples in safe Rust," in *Proc. 39th IEEE/ACM Int. Conf. on Automated Software Engineering Workshops (ASEW)*, 2024.

[4] M. Hassnain, A. Basu, E. Ng, and C. Stanford, "Cargo Sherlock: An SMT-based checker for software trust costs," in *Proc. IEEE/ACM 14th Int. Conf. on Formal Methods in Software Engineering (FormaliSE)*, 2026, pp. 1–12.

[5] T. Li, R. Li, B. Wang, B. Paulsen, U. Mathur, and P. Saxena, "Adversarial agent collaboration for correctness improvements of C to safe Rust translation," arXiv:2510.03879, 2025.

[6] B. Wang, B. Paulsen, J. Dodds, D. Kroening, U. Mathur, and P. Saxena, "Mostly automatic translation of language interpreters from C to safe Rust," arXiv:2606.27122, 2026.

[7] X. Liu, Z. Li, Z. Zhao, Z. Shuai, Y. Yang, Q. Xin, and J. Xuan, "AdaTrans: Automated C to Rust transformation via error-adaptive repair," arXiv:2606.31706, 2026.

[8] B. Schesch and M. D. Ernst, "Agentic C to Rust translation," *IEEE Security & Privacy*, vol. 24, no. 5, pp. 18–27, 2026.

[9] M. Farrukh, B. Coskun, T. Palit, and M. Polychronakis, "ORBIT: Guided agentic orchestration for autonomous C-to-Rust transpilation," arXiv:2604.12048, 2026.

[10] Y. Gao, C. Wang, P. Huang, X. Liu, M. Zheng, and X. Zhang, "Raw pointer rewriting with LLMs for translating C to safer Rust," in *Findings of the Association for Computational Linguistics: ACL 2026*, 2026, pp. 35 360–35 373.

[11] V. A. Dasu, M. Santra, M. R. U. Rashid, A. Kumar, S. Tizpaz-Niari, and G. Tan, "Heimdall: Formally verified automated migration of legacy eBPF programs to Rust," arXiv:2605.25411, 2026.

[12] S. Dehghan, T. Sun, T. Wu, Z. Li, and R. Jabbarvand, "Translating large-scale C repositories to idiomatic Rust," arXiv:2511.20617, 2025.

[13] A. Z. H. Yang, Y. Takashima, B. Paulsen, J. Dodds, and D. Kroening, "VERT: Polyglot verified equivalent Rust transpilation with large language models," in *Proc. 40th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, 2025, pp. 1453–1463, preprint and artifact: arXiv:2404.18852 (2024), Zenodo 10927704.

[14] H. F. Eniser, H. Zhang, C. David, M. Wang, M. Christakis, B. Paulsen, J. Dodds, and D. Kroening, "Towards translating real-world code with LLMs: A study of translating to Rust," arXiv:2405.11514, 2024.

[15] M. Shetty, N. Jain, A. Godbole, S. A. Seshia, and K. Sen, "Syzygy: Dual code-test C to (safe) Rust translation using LLMs and dynamic analysis," arXiv:2412.14234, preliminary version at LLM4Code 2025, 2024.

[16] V. Nitin, R. Krishna, L. Lemos do Valle, and B. Ray, "C2SaferRust: Transforming C projects into safer Rust with neurosymbolic techniques," *IEEE Transactions on Software Engineering*, vol. 52, no. 2, pp. 618–630, 2026, arXiv:2501.14257.

[17] T. Zhou, Z. Zhang, H. Lin, S. Jha, M. Christodorescu, K. Levchenko, and V. Chandrasekaran, "SACTOR: LLM-driven correct and idiomatic C to Rust translation with static analysis and FFI-based verification," in *Proc. 64th Annual Meeting of the Association for Computational Linguistics (ACL), Vol. 1: Long Papers*, 2026, pp. 646–673, arXiv:2503.12511.

[18] X. Cai, J. Liu, X. Huang, Y. Yu, H. Wu, C. Li, B. Wang, I. N. B. Yusuf, and L. Jiang, "RustMap: Towards project-scale C-to-Rust migration via program analysis and LLM," in *Engineering of Complex Computer Systems (ICECCS 2025)*, ser. LNCS, 2025, pp. 283–302, arXiv:2503.17741.

[19] Y. Bai and T. Palit, "RustAssure: Differential symbolic testing for LLM-transpiled C-to-Rust code," in *Proc. 40th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, 2025, pp. 534–546, arXiv:2510.07604.

[20] Z. Yuan, W. Mao, Z. Chen, X. Shang, C. Wang, Y. Lou, and X. Peng, "Project-level C-to-Rust translation via pointer knowledge graphs," *Proceedings of the ACM on Software Engineering*, vol. 3, no. FSE, pp. 3675–3698, 2026.

[21] S. Wang, M. Liu, G. Ou, Y. Chen, Z. Li, Y. Wang, and Z. Zheng, "His2Trans: A knowledge-guided agentic framework for project-level C-to-Rust migration," arXiv:2603.02617, 2026.

[22] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[23] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[24] bolero Contributors, "bolero: Fuzz and property testing front-end framework for Rust," https://github.com/camshaft/bolero, 2024.

[25] Kani Contributors, "The Kani Rust verifier," https://github.com/model-checking/kani, 2024.

[26] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. OSDI*, 2008, pp. 209–224.

[27] GNU Project, "GNU coreutils," https://www.gnu.org/software/coreutils/, 2024.

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

[29] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proc. PLDI*, 2011, pp. 283–294.

[30] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[31] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," in *Proc. USENIX ATC*, 2012, pp. 309–318.

[32] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in ACTL formulas," in *Proc. CAV*, ser. LNCS, vol. 1254, 1997, pp. 279–290.

[33] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," in *Proc. CHARME*, ser. LNCS, vol. 1703, 1999, pp. 82–98.

[34] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation," in *Proc. NeurIPS*, 2023, arXiv:2305.01210.

[35] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *Proc. ICSE*, 2024.