Theme Article: Language Translation for Security

Agentic C to Rust translation

Benedikt Schesch, *Amazon, London, UK*

Michael D. Ernst, *U. of Washington, Seattle, WA, USA*

Abstract—

The Rust programming language provides stronger guarantees about memory safety than C. Therefore, translating C to Rust is one way to reduce security vulnerabilities. One common translation strategy uses LLM queries. We present an alternative agentic approach that gives an LLM freedom and guardrails, and a publicly available implementation named ACTOR that outperforms all previous tools.

When programming in C, a programmer must manually manage memory and other resources, such as by calling `free` after each call to `malloc` or by calling `fclose` after each call to `fopen`. It is easy to violate these rules, leading to memory corruption, resource exhaustion, and security vulnerabilities. This is a serious problem, because much of the world's critical software infrastructure is written in C.

The systems programming language Rust is a promising replacement for C. Rust prevents many errors via compile-time analyses, such as for type-checking and ownership. There is a growing movement to replace software written in C by safer versions written in Rust. To date, that replacement has required manual rewriting, which is time-consuming, costly, and unscalable. An alternative approach is to *automatically* translate C to Rust.

Previous C-to-Rust translation tools fall into three categories. *Rule-based transpilers* like C2Rust [3], Crown [14], and Laertes [2] use language-specific AST transformations and are pinned to specific compiler versions. They break when the input does not match their expectations and cannot be adapted to other language pairs without a complete rewrite. Since they largely target a Rust-specific problem, `unsafe` blocks, their ideas may not even generalize to other target languages. *LLM-assisted translators* like SmartC2Rust [13] and C2SaferRust [6] use LLMs for code generation but wrap them in substantial language-specific infrastructure: AST parsers, macro transformers, error-code classifiers, and language-specific segmentation (splitting the C program into small pieces before translating, using knowledge of C's structure) — 38,000 lines of Python for SmartC2Rust. Furthermore, the interaction between the translation system and the LLM, such as the order in which LLM queries are issued and what information is provided to the LLM, has been tuned to the specific task. Adapting these to a different language pair would require revising the translation strategy and reimplementing the infrastructure.

This article proposes a third category: *agentic translation*. Our pipeline contains no AST parsing, no language-specific segmentation, and no compiler version pinning. All language knowledge comes from an off-the-shelf LLM and the tools it runs. The agent chooses its own translation strategy, which may differ depending on the program being translated.

Our methodology consists of prompts plus a translate-then-validate pipeline, which we have run with three distinct off-the-shelf agentic harnesses (Kiro, Claude Code, and Codex) backed by three different LLM providers. The methodology is therefore not tied to a single agentic framework.

In experiments, our agentic approach outperforms the other approaches.

LLMS AND AGENTS

Three ways to use an LLM are via queries, conversations, and agents.

To *query* an LLM, the user provides a prompt (typically a question or an imperative command), and the LLM returns a response. Query-based translation of C to Rust would provide a prompt such as "Translate the following C code into safe, idiomatic Rust whose behavior is identical to the behavior of the C code."

A *conversation* is a sequence of queries. Each prompt can refer to previous user prompts or previous LLM responses. In conversational translation of C to Rust, the client provides a sequence of prompts. The first prompt might choose a Rust signature for each

Digital Object Identifier 10.1109/XXX.0000.0000000

function. Next, the client would supply one prompt to translate each function. The client would then compile the resulting Rust program and run tests. If compilation or testing fails, the client would continue the conversation by providing the error message and asking the LLM to adjust the translation. The client can be a human or a program that provides prompts to the LLM and reads its output.

A generative AI *agent* is an LLM that achieves its goals by planning and performing multi-step operations. It is analogous to the client of a conversational LLM. An agent has access to *tools* that manipulate its environment, such as by running programs and editing files. The user prompts the agent with a goal and a way to measure success. Then the agent iterates, choosing and performing actions, until it achieves its goal. Whereas a scripted conversational client directs the LLM in the same way for each translation task, an agent has autonomy to plan and to dynamically change its plan.

An agent can recursively invoke subagents, delegating subtasks to specialized instances that work independently and report back. Subagents avoid polluting the main agent's context with irrelevant details: each subagent starts with a fresh context focused on its specific subtask. This sidesteps the degraded performance that LLMs exhibit when their context window fills with long, loosely related information.

Queries and conversations give the user more control of the high-level task, but they require more effort from the user. When using an agent, the user trusts the LLM to make its own decisions more effectively than the user could. (This may be discouraging to people who believe that their skills and perspective are uniquely necessary to complete the task.) This paper shows the effectiveness of the fully agentic approach.

AGENTIC TRANSLATION

Our system, ACTOR (for Agentic C to Rust translator) [10], converts an entire C project to a complete Rust project. Unlike function-by-function approaches, the agent can see all source files and can make holistic decisions, such as replacing C idioms with Rust equivalents (e.g., linked lists with `Vec`), restructuring modules, and choosing appropriate abstractions.

ACTOR performs two steps: translation and validation. Each has its own prompt. The complete prompts are publicly available [10].

The translation prompt

The translation prompt is 116, 166 (fig. 1), or 536 words, depending on whether ACTOR determines that it is being asked to translate a program, a library, or a project that produces both a shared library and a binary executable.

*Translate the C code in `c_src/` to Rust that produces **byte-identical output** for the same inputs. Write `Cargo.toml` and `src/` files in the current directory (NOT in `c_src/`).*
This is a LIBRARY. Requirements:

- *Cargo.toml must have `crate-type = ["cdylib"]` under `[lib]`*
- *All public C functions must use `#[unsafe(no_mangle)]` and `extern "C"`*
- *Pay attention to C preprocessor macros that RENAME functions (e.g., `#define foo NAMESPACE(foo)` makes the linker symbol `PREFIX_foo`, not `foo`). The Rust `#[no_mangle]` name must match the FINAL linker symbol, not the source-level name. Check header files for namespace macros.*
- *Preserve the exact C function signatures (use `*const c_char`, `c_int`, etc. from `std::ffi`)*
- *Do NOT fix bugs in the original C code — if the C has incorrect behavior, reproduce it exactly*
- *Preserve the exact order of error checks and validation*
- *Use safe Rust internally where possible*

Run `cargo build --release` and fix any errors until it compiles. Do NOT modify anything in `c_src/`.

FIGURE 1. ACTOR's library translation prompt (166 words).

The variant for translating a program simply omits the four cdylib/FFI bullets. The shared-library variant targets projects that compile to many configurations from one source tree. One of our case study programs, SPHINCS+, uses CMake variables (hash backend, security parameter, etc.) to build 128 distinct binaries from the same C source. Rust expresses the same idea with Cargo features and `#[cfg(...)]`; the shared prompt tells the agent to mirror each CMake variable as a Cargo feature and gate modules with `#[cfg(...)]` so all configurations build from one Rust crate. The shared prompt also tells the agent to break the work into subtasks for large multi-file projects.

We arrived at the prompts iteratively. We added the library prompt's namespace-macro warning after observing the agent emit `#[no_mangle] fn crypto_sign_keypair` for a SPHINCS+ build whose actual linker symbol was prefixed (`SPX_crypto_sign_keypair`). We added the shared prompt's configuration section after seeing the agent hardcode a single hash backend on the first SPHINCS+ attempt, breaking the other 127 configurations. The ablations in section "Ablations" quantify the contribution of each section.

For many projects, using ACTOR requires no user effort. In some cases, the user must tune the prompt, which is a straightforward, though heuristic, process. For example, for the CRUST-Bench benchmark (described below), we slightly edited the prompts (translation

prompt is 264 words, validation prompt is 642 words) to communicate its idiosyncratic requirements regarding linkage and test file format.

A strength of our agentic approach is that it is easy to adapt to different translation contexts and needs. For example, sometimes byte-for-byte identical output is essential, but in other situations it would be desirable to fix defects (bugs) during transpilation. The user can also easily augment the prompt to give ACTOR hints about what architecture, tools, etc. to use.

For C programs with a `main` function, the Rust code must produce byte-identical `stdout` output. For library translations, the Rust code must export the same C-compatible symbols as the original, so that the Rust library is a drop-in replacement. For configurable projects with compile-time options, each configuration must map to an equivalent Rust build configuration, and all combinations must compile.

The prompt instructs the agent to decompose large or configurable projects into subtasks and delegate each to a subagent. The main translation agent decides how to break down the work and what context to provide to each subagent, such as which C source files to translate and which Rust modules to produce. Each subagent translates its assigned subset of the code and verifies that its work compiles before handing off to the next. This hierarchical delegation mirrors how a human team lead would divide a large translation effort among developers.

The validation prompt

After translation, a second agentic step validates correctness using the C implementation as an oracle. In this step, ACTOR provides the translated Rust code alongside the original C source. The validation prompt is structurally similar to the translation prompt but longer; it instructs the agent to test the translation against the C implementation. This process typically involves the agent writing tests, but the agent has full freedom in how to do so.

The prompt tells the agent that the C code is always the ground truth: if the C code does something unexpected, the Rust code must replicate that behavior. For example, when the C code builds a shared library, the agent writes Rust integration tests that load both the C and Rust libraries via dynamic linking, call the same functions with the same inputs, and assert that the outputs match. The prompt directs the agent to start with low-level utility functions and work upward in the call graph. For projects with multiple build configurations, the agent must repeat this process for every feature combination.

Within these constraints, the validation agent has complete autonomy. It decides which functions to prioritize, what test inputs to generate, how to structure the test code, and how to diagnose and fix any mismatches it discovers. This catches semantic bugs that compilation alone cannot detect, without requiring human-written test cases.

Underlying agent and LLM

ACTOR is a methodology, consisting of prompts plus a translate-then-validate pipeline, rather than a single implementation tied to a particular agentic harness. We have run ACTOR using three off-the-shelf agentic harnesses: Kiro [5], Claude Code [1], and Codex [8] (running GPT-5.4 on Amazon Bedrock). All three use general-purpose tools (file editing, shell commands, subagent delegation, iterative execution). On the TRACTOR benchmark (see below), Kiro and Claude Code agree within 2 percentage points (325/338 vs. 319/338); the Codex harness running GPT-5.4 reaches 244/338, with the gap concentrated on the SPHINCS+ battery; this shows that the harness does matter, even though ACTOR has no harness-specific behavior. This portability across harnesses is a contribution: ACTOR's effectiveness comes from the prompt structure and validation methodology, not from harness-specific machinery. Adapting ACTOR to a different agentic harness requires only configuring how the harness is invoked.

EVALUATION METHODOLOGY

The agent works in an isolated directory containing only the C source code (including build files), with no access to the benchmark's test cases or expected outputs.

Comparison systems

We compared ACTOR to other C-to-Rust translation systems. All comparison runs, the translated outputs, and our harness for driving each tool are publicly available in the ACTOR repository [10].

C2Rust [3] is a well-known rule-based transpiler. It uses Clang's AST to translate C into semantically equivalent unsafe Rust, producing compilable code that mirrors the original C line by line. C2Rust is deterministic and fast. The output is `unsafe` and non-idiomatic: raw pointers, manual memory management, and C library calls are preserved verbatim. The resulting Rust code requires extensive manual refactoring to become safe Rust.

Laertes [2] post-processes and refactors C2Rust output. It applies a sequence of transformations: replacing raw pointers with safe references, converting `unsafe`

FIGURE 2. Benchmark statistics for TRACTOR (top) and CRUST-Bench. LOC counts non-comment, non-blank lines in C source files, excluding test files. P01 has 128 test vectors but one shared C codebase.

		Non-comment, non-blank lines of code			
Dataset	Cases	Total	Mean	Median	Max
B01-synthetic	85	2,597	31	26	114
B01-organic	38	2,455	65	38	376
B02-synthetic	40	9,657	241	129	873
B02-organic	44	23,072	524	485	2,853
P00 (Perlin)	1	380	380	380	380
P01 (SPHINCS+)	1	6,611	6,611	6,611	6,611
CRUST-Bench	87	80,958	931	449	25,436

blocks to safe code, and resolving imports. Because it operates on C2Rust output, it inherits C2Rust's limitations and is pinned to Rust nightly-2020-10-15.

Crown [14], like Laertes, post-processes C2Rust output and is pinned to Rust nightly-2023-01-26. It performs static ownership analysis to infer which raw pointers can be safely replaced with Rust references, `Box`, or other owned types.

C2SaferRust [6] post-processes C2Rust output with an LLM, rewriting each "translation unit" (<150 lines of C) to Rust. Running it required forking the tool to repair a never-exercised model backend.

SmartC2Rust [13] translates C to Rust by segmenting the program, translating each segment, and repairing against compilation and test feedback; it produces the safest output of any system we measured (around 2% unsafe in the "Total" section of fig. 3). However, it requires a *runnable* program with a test harness; library functions must be handled by manually writing a C driver per function that "calls the (library) function and displays the result". Running it required forking three of its packages, repairing its unexercised Bedrock backend, and building two custom-patched LLVM toolchains from source — the kind of language- and version-specific machinery that the agentic approach avoids.

As LLM baselines, we evaluated Kimi K2.5, GPT-5.4, and Gemini 3.1 Pro. Each model receives system prompts similar to ACTOR's (slightly different due to query-based vs. agentic translation). The model translates the entire C project in a single API call with no external iteration.

We were unable to compare against Syzygy [12], which uses LLMs augmented with dynamic analysis. Its translation tool is not publicly available (only the translated output for a single program, Zopfli, is released).

Benchmarks

Our experiments use two benchmarks (fig. 2).

(1) The TRACTOR benchmark [7] contains 338 test cases averaging 132 LOC (largest 6,611 LOC). The translated Rust must export the same C-compatible interface as the original, so that it serves as a drop-in shared-library replacement. B* are microbenchmarks. P00 Perlin is a texture generator for computer graphics. P01 SPHINCS+ computes hash-based post-quantum digital signatures.

(2) CRUST-Bench [4] contains 100 C repositories averaging 924 LOC, each paired with manually crafted Rust interfaces and test cases. The agent receives a Rust scaffold with function signatures; its task is to implement each function body in idiomatic Rust without using FFI.

We used a subset of CRUST-Bench because 13 of its 100 test suites contain errors. We noticed them when ACTOR's translation failed either the CRUST-Bench tests or ACTOR's own tests; on investigation, ACTOR was correct and CRUST-Bench was wrong. We reported these problems to the authors of CRUST-Bench, who acknowledged them and fixed the benchmark [11]. Our experiments predate those fixes.

The metric of correctness is behavioral equivalence to C on the benchmark's hidden tests. This is an imperfect proxy for correctness, which is unknowable because it depends on the intentions of the original programmers.

Human effectiveness in C-to-Rust translation

CRUST-Bench's tests provide an important datapoint about the effectiveness of humans doing C-to-Rust translation. When creating CRUST-Bench, "annotators ... construct Rust test files by adapting existing C tests" [4] — in other words, the human annotators translated C tests to Rust.

The C tests are relatively small and simple, but the per-suite correctness rate for translating them was only 87%. This strongly motivates the use of more reliable technology, such as ACTOR, for C-to-Rust transpilation.

RESULTS

TRACTOR

Figure 3 shows results on the TRACTOR benchmark. Every system is scored over all 338 cases. ACTOR has the best overall correctness and is best on all batteries except B02-synthetic, where it trails slightly. When other approaches have similar correctness, ACTOR produces safer code.

B01 contains small, single-file programs where literal transpilation works well. C2Rust passes all tests, as does ACTOR.

FIGURE 3. Results for the TRACTOR benchmark. This benchmark inherently requires `unsafe`.

Battery	System	Compiles	Passes tests	LOC	Unsafe
B01-syn	ACTOR (Kiro)	85/85	**85/85**	5k	15%
	ACTOR (Claude Code)	85/85	**85/85**	6k	18%
	ACTOR (Codex)	85/85	84/85	4k	30%
	ACTOR (Kiro, no validate)	85/85	83/85	3k	17%
	C2Rust	85/85	**85/85**	5k	57%
	Laertes	83/85	83/85	6k	50%
	C2SaferRust	85/85	68/85	5k	20%
	SmartC2Rust	41/85	38/85	2k	2%
	Kimi K2.5 (query)	75/85	56/85	2k	8%
	GPT-5.4 (query)	82/85	71/85	3k	2%
	Gemini 3.1 Pro (query)	82/85	75/85	2k	5%
B01-org	ACTOR (Kiro)	38/38	**38/38**	3k	38%
	ACTOR (Claude Code)	38/38	**38/38**	3k	44%
	ACTOR (Codex)	38/38	**38/38**	2k	61%
	ACTOR (Kiro, no validate)	38/38	**38/38**	3k	38%
	C2Rust	38/38	**38/38**	9k	33%
	Laertes	37/38	37/38	10k	27%
	C2SaferRust	38/38	25/38	7k	6%
	SmartC2Rust	0/38	0/38	–	–
	Kimi K2.5 (query)	34/38	30/38	2k	30%
	GPT-5.4 (query)	36/38	32/38	3k	20%
	Gemini 3.1 Pro (query)	37/38	35/38	2k	23%
B02-syn	ACTOR (Kiro)	42/42	31/42	12k	31%
	ACTOR (Claude Code)	42/42	**36/42**	15k	41%
	ACTOR (Codex)	42/42	**36/42**	11k	32%
	ACTOR (Kiro, no validate)	42/42	31/42	8k	24%
	C2Rust	39/42	38/42	21k	76%
	Laertes	40/42	39/42	21k	73%
	C2SaferRust	38/42	32/42	20k	65%
	SmartC2Rust	6/42	2/42	4k	1%
	Kimi K2.5 (query)	25/42	16/42	9k	10%
	GPT-5.4 (query)	38/42	26/42	9k	2%
	Gemini 3.1 Pro (query)	37/42	24/42	8k	5%
B02-org	ACTOR (Kiro)	44/44	42/44	22k	67%
	ACTOR (Claude Code)	44/44	**43/44**	24k	78%
	ACTOR (Codex)	43/44	41/44	16k	48%
	ACTOR (Kiro, no validate)	43/44	41/44	18k	66%
	C2Rust	42/44	42/44	45k	76%
	Laertes	41/44	41/44	45k	73%
	C2SaferRust	31/44	28/44	43k	68%
	SmartC2Rust	0/44	0/44	–	–
	Kimi K2.5 (query)	22/44	16/44	11k	23%
	GPT-5.4 (query)	32/44	25/44	13k	16%
	Gemini 3.1 Pro (query)	29/44	22/44	10k	26%
P00 (Perlin)	ACTOR (Kiro)	1/1	**1/1**	556	0%
	ACTOR (Claude Code)	1/1	**1/1**	450	1%
	ACTOR (Codex)	1/1	**1/1**	394	5%
	ACTOR (Kiro, no validate)	1/1	**1/1**	315	0%
	C2Rust	1/1	**1/1**	1.6k	31%
	Laertes	1/1	**1/1**	1.6k	31%
	C2SaferRust	1/1	**1/1**	1.6k	27%
	SmartC2Rust	1/1	0/1	330	0%
	Kimi K2.5 (query)	1/1	0/1	443	0%
	GPT-5.4 (query)	1/1	0/1	355	0%
	Gemini 3.1 Pro (query)	1/1	0/1	271	0%
P01 (SPHINCS+)	ACTOR (Kiro)	128/128	**128/128**	5k	64%
	ACTOR (Claude Code)	128/128	116/128	5k	25%
	ACTOR (Codex)	128/128	44/128	2k	6%
	ACTOR (Kiro, no validate)	128/128	36/128	5k	65%
	C2Rust	0/128	0/128	5k	73%
	Laertes	0/128	0/128	5k	73%
	C2SaferRust	0/128	0/128	5k	73%
	SmartC2Rust	0/128	0/128	–	–
	Kimi K2.5 (query)	0/128	0/128	–	–
	GPT-5.4 (query)	0/128	0/128	816	0%
	Gemini 3.1 Pro (query)	0/128	0/128	1.2k	3%
Total	ACTOR (Kiro)	338/338	**325/338**	47k	50%
	ACTOR (Claude Code)	338/338	319/338	53k	53%
	ACTOR (Codex)	337/338	244/338	36k	39%
	ACTOR (Kiro, no validate)	337/338	230/338	37k	50%
	C2Rust	205/338	204/338	87k	70%
	Laertes	202/338	201/338	88k	66%
	C2SaferRust	193/338	154/338	82k	59%
	SmartC2Rust	48/338	40/338	7k	2%
	Kimi K2.5 (query)	157/338	118/338	25k	17%
	GPT-5.4 (query)	189/338	154/338	28k	10%
	Gemini 3.1 Pro (query)	186/338	156/338	24k	15%

For P00 (Perlin), ACTOR produced a translation that is both much smaller and much safer than C2Rust.

SPHINCS+ has extensive, complex pointer arithmetic that must produce bit-identical hash outputs. This might seem difficult to reproduce in Rust. However, the need for bit-identical outputs is helpful to the agent. Even simple tests will not pass unless the Rust code is algorithmically identical to the C code. Such tests are powerful guardrails for an LLM. This supports the folk wisdom among vibe coders that the most important part of LLM prompting is creating automated validation conditions that constrain the LLM by rejecting incorrect code.

We ran ACTOR using three off-the-shelf agentic harnesses, with no methodology changes. For ACTOR (Codex), 84 of the 128 configurations fail on SPHINCS+ because Codex with GPT-5.4 reproduces only one hash backend correctly. ACTOR's portability across harnesses demonstrates that its effectiveness comes from its prompt structure and translate-then-validate methodology, not from harness-specific machinery.

The "ACTOR (no validate)" row runs only ACTOR's translation step, skipping its validation step; section "Ablations" discusses how that affects ACTOR's success rate.

C2Rust 0.22.1 fails all 128 tests for P01. SPHINCS+ declares two functions named `randombytes` in different files, with conflicting return types (`void` vs. `int`). C2Rust transpiles both declarations literally, producing code that compiles but behaves incorrectly at run time. Our agentic translator ACTOR detects and resolves the inconsistency.

The C2Rust postprocessors start from C2Rust's defective transpilation. Laertes breaks 5 compilations that C2Rust had working (while fixing 2) and fails all 128 configurations of P01. C2SaferRust also passes fewer tests than the C2Rust output it starts from. C2Rust emits an `unsafe extern "C"` function whose signature matches the C ABI. C2SaferRust's goal is to make that function idiomatic, and in doing so it rewrites the signature (e.g., `*mut c_char` to `&[u8]`), which breaks the C ABI. The resulting library is safer but is no longer a drop-in replacement, so it fails to link against the harness. In other words, C2SaferRust trades correctness for safety.

SmartC2Rust requires a runnable program, so it fails on libraries and P01.

Figure 3 omits Crown. Crown produced no changes to C2Rust output on all B* projects because the `extern "C"` FFI signatures require raw pointer semantics that cannot be safely converted to references. On P01 (SPHINCS+), Crown's ownership analysis crashes: its struct depth computation underflows when analyzing

FIGURE 4. Root causes for ACTOR translation failures.

Root cause	count
Undefined behavior	3
Macros	3
Configuration	1
Input processing	9
Underspecified behavior	2
Truncated output	12

cross-module struct pointers with nested pointer fields, triggering a panic in the precision calculation. This is a real bug in Crown, illustrating how rule-based refactoring tools struggle with the complexity of real-world C codebases.

In a further test of ACTOR, the DARPA TRACTOR team ran 6 new C-to-Rust translators on a program of their choosing (libgit2). They reported that "only one out of six teams produced something even close to correct"; the successful translator was ACTOR. ACTOR's output was also the most idiomatic (score 4.2/5; the next best score was 3.6/5).

It is tempting to re-use existing tools like C2Rust. However, we found that it is more effective to start from scratch rather than to try to fix an unsafe Rust program. This is true even when C2Rust produces a program with the same external behavior as the original C program.

ACTOR's failures of translation Figure 4 categorizes the root causes when ACTOR's translation did not pass all tests. In some cases a failing translation had multiple problems. Most of the failures were on synthetic torture-test programs.

Undefined behavior
One C program passed `unsigned int` to `scanf` (which is a type error), another called `atoi` on out-of-range input, and a third wrote over the end of an array. When a C program's behavior is undefined, there is no Rust program that is guaranteed to reproduce its actual behavior, because that behavior may depend on the compiler, libraries, operating system, and CPU. A workaround would be to delegate to the original C library via FFI, but this compromises safety.

Macros
In two cases, Rust translated C macros to Cargo features (which is reasonable) but set the wrong default value in the Cargo files. In another case, all operations were within `assert()`, which compiles to nothing in the usual case (production, not debugging, compilation), so ACTOR's translation was empty. Hayroll [9] would solve these problems, which arise from the C preprocessor rather than the C language proper.

Configuration
In one case, there was a hyphen in a name in the generated `Cargo.toml`, which is illegal and led to a compilation failure.

Input processing
In 3 cases, the Rust version processes the entire input but the C version truncates starting at the first NUL (0) character in the input. In 2 cases, the C version splits long input lines, incorrectly processing the second part as a new nonsensical command, whereas the Rust version processes the entire input without issuing errors. In 2 cases, the C version silently truncates long input, whereas the Rust version processes the entire input. In 2 cases, the Rust version issues an "illegal input" error for nonsense input, but C's `sscanf` succeeds so long as a prefix of the input is a legal number. In all of these cases, the Rust output is arguably *better* than the C output. However, the test harness counted ACTOR's output as wrong, because, when run, its output was not byte-identical to the C output.

Underspecified behavior
In one case, when a search pattern is empty, the C version prints the entire input, but the Rust version reads another pattern. In another case, the C version permits a city name to be empty (zero-length string), but the Rust version prohibits that. Again, these are arguably better behavior, but the C programmer did not specify whether the implemented behavior was essential (part of the specification) or accidental (the input would not occur in the field).

Truncated output
In 12 cases in P01 SPHINCS+, ACTOR (Claude Code)'s Rust program copied data incompletely (e.g., only first 16 out of 32 bytes). The Rust program output by ACTOR (Kiro) passed a raw pointer, so all bytes were available and the output matched that of the C program.

Unsafe code in ACTOR translations The translated Rust code makes heavy use of unsafe constructs: one per four lines of code is inside an `unsafe` blocks or function. Well-written Rust code contains little, if any, unsafe code.

Figure 5 lists the root causes for unsafe Rust constructs. The driver of most of the unsafe Rust is that the test harness requires an ABI-compatible translation that can be linked against C tests. This *directly* caused the first section of the table: nearly 2000 symbols declared on one side of the C–Rust boundary in the

FIGURE 5. Unsafe Rust constructs in ACTOR output, summed across 382 files (56,258 LOC, plus 1070 lines of comments and 7020 blank lines).

Root cause	Count
C ABI preservation (drop-in `.so`/symbol compat)	1648
External FFI calls (libc, inter-module thash)	208
Function-pointer dispatch / extern callbacks	36
Mutable global state (FFI for static C globals)	116
Bridging raw C pointers into safe crates	60
C-string / pointer conversions	2194
`ptr::read/write/copy` intrinsics	527
Uninitialized / zero-init structs	70
Raw-pointer types in signatures/casts	5957
Pointer arithmetic	3931
Others	44
Total	**14791**

FIGURE 6. Results for CRUST-Bench. CRUST-Bench leaderboard translations are not public, so we cannot compute an unsafe percentage. We report out of 87 projects, excluding 13 with unsatisfiable tests. It is likely that the leaderboard marks those wrong for every translator.

System	Setting	Builds	Tests	LOC	Unsafe
ACTOR (Kiro)		82/87	56/87	43k	1%
C2Rust		0/87	0/87	–	–
Laertes		0/87	0/87	–	–
C2SaferRust		0/87	0/87	–	–
SmartC2Rust		0/87	0/87	–	–
GPT-5.4		82/87	50/87	57k	0%
Kimi K2.5		46/87	31/87	28k	0%
Gemini 3.1 Pro		11/87	8/87	15k	0%
Ablations					
ACTOR (Kiro)	test repair	87/87	82/87	52k	1%
ACTOR (Claude)	test repair	85/87	75/87	45k	0%
ACTOR (Codex)	test repair	87/87	81/87	48k	1%
GPT-5.4	test repair	79/87	64/87	58k	0%
Kimi K2.5	test repair	45/87	39/87	28k	0%
Gemini 3.1 Pro	test repair	8/87	7/87	15k	0%
Prior results from CRUST-Bench leaderboard [4]					
GPT-5		92/100	43/100	–	–
Gemini 3		88/100	41/100	–	–
o3		68/100	31/100	–	–
SWE-agent	test repair	41/100	32/100	–	–
GPT-5	test repair	85/100	70/100	–	–
Gemini 3	test repair	83/100	66/100	–	–
o3	test repair	63/100	48/100	–	–

test harness that need to be used on the other side. It *indirectly* caused most of the rest of the table at uses of those symbols.

Future work should investigate the extent to which well-written C code, with no undefined behavior and good specifications, can be translated into safe Rust.

CRUST-Bench

Figure 6 shows results on CRUST-Bench. On the CRUST-Bench leaderboard, the non-agentic approaches (GPT-5, Gemini, o3) have a fixed budget of three rounds and can only react to errors. An agent can plan and iterate as many times as needed.

SWE-agent, which is also agentic, only passes 32 projects. This suggests that being agentic alone is not sufficient: ACTOR's prompt design and task decomposition matter.

Gemini 3.1 Pro frequently failed to follow the required output format, omitting file markers or placing them where the parser could not extract them, so most responses could not be split into compilable files.

ACTOR's success on both TRACTOR and CRUST-Bench illustrates its generality, which is a key advantage of the agentic approach. The TRACTOR benchmark requires FFI-compatible exports. By contrast, CRUST-Bench requires implementing Rust function bodies from a scaffold with no FFI. These are fundamentally different testing paradigms. ACTOR handles both with only a prompt change, because the agent reads the project structure, understands the testing requirements, and adapts its translation strategy accordingly.

Ablations

We performed multiple ablation studies, which determine the relative importance of parts of ACTOR.

Validation step.
One ablation removed the validation step, leaving only ACTOR's translation step. The result is indicated by the difference between the "ACTOR (Kiro)" and "ACTOR (Kiro, no validate)" rows of fig. 3. Validation improved the number of test successes for B01-syn, B02-org, and P01, and it did not harm the other three test batteries. ACTOR's validation step has the largest impact on P01 (SPHINCS+), improving test pass rate from 36/128 without validation to 128/128 with it. For the artificial tests of B02-synthetic, the validation step caused the addition of a large amount of code, much of it unsafe, but had no effect on success rates. An earlier section explained features of B02 that foiled ACTOR.

The tests encode expectations that the C does not.
A translator that works from the C source alone can only reproduce what the C does. Some of CRUST-Bench's ground-truth tests, however, expect a value that the reference C never produces, so a faithful translation is scored as a failure. This differs from the 13 suites we excluded as unsatisfiable, where the test demands behavior that safe Rust cannot express and no translation can pass. The 21 projects discussed here are the opposite: they are perfectly translatable, and ACTOR reproduces the C's behavior exactly. The defect is that the test's expected value diverges from the C, and that expectation cannot be recovered from the C.

For example, the `Simple_Config` library prints a trailing newline after each entry, and the C's own test accepts it by comparing only a prefix. The benchmark's Rust test instead requires exact equality against a constant that has no trailing newline, so ACTOR emits the C's real output and is marked wrong. In `Math_Library_in_C`, the C meets the test's numerical tolerance only because it accumulates in extended (80-bit) precision. The benchmark's interface uses standard 64-bit floats, which places the target below what that precision can represent, so ACTOR's faithful 64-bit translation cannot satisfy it, and the C's own suite would not either at that width. In neither case is the expected value a property of the C source: a translator that has not seen the test cannot infer it, and reproducing it would require departing from the C.

We reached this conclusion by inspecting all 26 projects that ACTOR passes with the benchmark's tests but not with its own, comparing each failing test to the original C and, in several cases, compiling and running the C. In 21 of them the expected value is not derivable from the source, as above. The remaining 5 are genuine translation bugs, where ACTOR's output diverges from the C, and these are the failures that test feedback legitimately repairs.

This places a ceiling on the setting in which the agent has no access to the tests. Because these expectations are invisible to any system that sees only the C, no system we evaluate passes more than 56/87 of the 87 projects without the tests; the next best is 50/87. Supplying the ground-truth tests removes the ceiling, raising ACTOR (Kiro) from 56/87 to 82/87, because the tests provide the specification that the C omits: the model can read the implicit assumptions and conform to them even where they depart from the C. Test access therefore acts as an implicit specification for these benchmarks. The effect applies equally to every system we evaluate, and it reflects a limitation of some benchmark tests in the version we evaluated rather than a flaw in the harness.

Prompt sensitivity.

To determine the value of each part of ACTOR's prompts, we performed five further ablations on the translation prompts, each removing one section in isolation while keeping all other guidance intact, and ran every ablation on TRACTOR, CRUST-Bench (test-repair setting), and CRUST-Bench (self-generated tests setting; the same benchmarks used for the main results).

Figure 7 shows that two prompt sections are most important. The first is the iteration loop: the explicit instruction to compile, read the error messages, and fix them. Without iteration prompting, the agent declares victory too early on hard cases.

FIGURE 7. Prompt-sensitivity ablations on all benchmarks. Each variant removes one section of the engineered prompt, keeping all others. Variants: *no-subtask* removes "decompose into subtasks"; *no-iterate* removes "compile, fix errors, iterate"; *no-features* removes multi-configuration build guidance; *minimal* removes all engineered guidance.

		CRUST-Bench	
	TRACTOR	no tests given	test repair
ACTOR (Claude Code)	319/338	56/87	75/87
no-subtask	313/338	56/87	79/87
no-iterate	249/338	31/87	73/87
no-features	204/338	55/87	76/87
minimal	171/338	41/87	75/87
Cross-prompt swap on TRACTOR B0X (n = 210)			
lib prompt on execs	59/61		
exec prompt on libs	4/149		

The second concerns build-time configurability. Some C projects compile to many distinct binaries from a single source. The prompt tells the agent to mirror each CMake variable as a Cargo feature so all configurations build from one Rust crate. Without that guidance, the agent hardcodes a single backend for SPHINCS+ and 127 of the 128 configurations fail. CRUST-Bench contains no multi-configuration projects.

Removing the subtask-decomposition block ("create a TODO list, work through subtasks one at a time") costs 6 cases on TRACTOR and gains 4 on CRUST-Bench (test repair), within run-to-run noise; the agent decomposes the project on its own.

A separate ablation swapped the executable and library prompts: each library was translated using the executable prompt and vice versa. Recall from fig. 3 that ACTOR is perfect on B01. The library prompt is essentially harmless on executables (59/61 cases pass), while the executable prompt is catastrophic on libraries (4/149 cases pass): without an explicit instruction to emit a `cdylib` with `#[unsafe(no_mangle)]` exports, the agent writes idiomatic Rust that compiles but produces a shared library with no callable C-ABI symbols.

These results suggest a refinement of the prompts for future work. The instructions that tell the agent something it cannot see in the C source, namely that the project should be compiled as a shared library with C-ABI exports or that one C source produces many binary configurations, are essential. Process-level scaffolding such as "decompose into subtasks" is largely redundant with what a competent agent already does, and a single library-style prompt subsumes the executable prompt.

Cost

On Kiro, ACTOR's translation and validation cost $93 for the entire TRACTOR benchmark and $67 for CRUST-Bench. The largest project, P01 (SPHINCS+, 6,611 LOC), cost $3.61 and took 76 minutes to translate and validate. Per kilo-LOC of translated output, this is $1.97 and 34 minutes for TRACTOR, and $1.57 and 19 minutes for CRUST-Bench.

On Claude Code, costs are roughly six times higher (about $570 for the full TRACTOR benchmark) due to a different pricing model, although test pass rates agree within 2 percentage points. The prompt-sensitivity ablations each cost a similar amount on Claude Code; the entire ablation study cost approximately $2,900.

The validation step accounts for about 58% of the cost and time on either harness, but it yields substantial improvements in correctness.

LIMITATIONS AND FUTURE WORK

LLM outputs are nondeterministic, so results may vary between runs. We have observed this in practice.

On B02-synthetic, C2Rust outperforms ACTOR on test pass rate, showing that literal transpilation can be more reliable for programs with complex macro patterns. On the negative side, C2Rust runs the C preprocessor once, losing all configurability and portability. Peng [9] shows a way to post-process multiple C2Rust runs (with different macros defined) in order to produce a configurable Rust program.

ACTOR's validation step relies on the agent generating tests, and those tests typically cover the code paths the agent already understands from the C source. They may miss subtle edge cases (unusual sign-extension, null-terminated buffer conventions, implicit type promotions) that the benchmark's tests exercise. The gap between ACTOR tests (56/87 on CRUST-Bench) and ACTOR with benchmark tests (82/87) quantifies this: better automated test generation by LLMs is an important future research direction.

We tuned ACTOR's prompts on the same programs used for evaluation. Our ablations, and especially the independent DARPA TRACTOR team evaluation on a program of their choosing, mitigate this threat.

Our prompt-sensitivity ablations apply only to C-to-Rust translation on the benchmarks we evaluated; we have not tested other language pairs. Adapting ACTOR to a different source/target language pair requires writing new prompts, which may require knowledge of the target language's idioms (e.g., FFI conventions, build-system features).

Future work includes comparing to conversational translation approaches; scaling to larger codebases; automatically identifying when a difference between C and Rust behavior indicates a problem in the original C code rather than the translated Rust code; and experimenting with other language pairs.

CONCLUSION

We have introduced ACTOR, the first agentic methodology for translating C to Rust. ACTOR is publicly available, and it substantially outperforms other symbolic and neural approaches on metrics of correctness and safety. Its key enablers are the prompting strategy and the validation step that tests the agent's own translation against the original C as an oracle. ACTOR is harness-portable: it runs on three distinct off-the-shelf agentic frameworks (Kiro, Claude Code, and Codex), demonstrating that the contribution lies in the methodology rather than any single framework.

ACTOR is more effective than competing approaches that use heavyweight program analysis. Our success suggests that fully agentic approaches should be applied more broadly to other topics in computing, even ones where correctness is a requirement.

ACKNOWLEDGMENTS

We thank the UW HARVEST team and the DARPA TRACTOR team for their assistance. Haoran Peng and James Yoo provided comments on an earlier draft of this article. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590132.

REFERENCES

1. Anthropic. Claude Code: Agentic coding in your terminal. https://www.anthropic.com/claude-code, February 2026.
2. Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Translating C to safer Rust. *Proc. ACM Program. Lang.*, 5(OOPSLA), 2021.
3. Immunant. C2Rust. https://github.com/immunant/c2rust, 2022.
4. Anirudh Khatry, Robert Zhang, Jia Pan, Ziteng Wang, Qiaochu Chen, Greg Durrett, and Isil Dillig. CRUST-Bench: A comprehensive benchmark for C-to-safe-Rust transpilation, 2025.
5. Kiro. Prompt to code to deployment in your terminal. https://kiro.dev/cli/, April 2026.
6. Vikram Nitin, Rahul Krishna, Luiz Lemos do Valle, and Baishakhi Ray. C2SaferRust: Transforming C projects into safer Rust with neurosymbolic techniques. *IEEE Transactions on Software Engineering*,

52(2):618–630, 2026. https://github.com/vikramnitin9/c2saferrust.
7. Brandt Ogden. DARPA TRACTOR public test corpus. https://github.com/DARPA-TRACTOR-Program/PUBLIC-Test-Corpus. Part of this benchmark is not yet public; we expect it to be by the time the article is published., February 2026.
8. OpenAI. OpenAI Codex CLI: Agentic coding tool that runs in your terminal. https://developers.openai.com/codex/cli, February 2026.
9. Haoran Peng, Baris Kasikci, Gilbert Bernstein, and Michael D. Ernst. Hayroll: A modular wrapper for translating C macros and conditional compilation to Rust. In *PLDI 2026: Proceedings of the ACM SIGPLAN 2026 Conference on Programming Language Design and Implementation*, Boulder, CO, USA, June 2026.
10. Benedikt Schesch. ACTOR: Agentic C to Rust translation. https://github.com/UW-HARVEST/ACTOR, April 2026. Note to reviewers: some of the submodules are private pending legal approval. We uploaded them in full with this submission. We will make the repositories public after acceptance.
11. Benedikt Schesch. CRUST-Bench test errors. https://github.com/anirudhkhatry/CRUST-bench/issues. For examples, see issues 37, 39, 40, and 41., April 2026.
12. Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A. Seshia, and Koushik Sen. Syzygy: Dual code-test C to (safe) Rust translation using LLMs and dynamic analysis. In *LLM4Code Workshop at ICSE*, 2025.
13. Momoko Shiraishi, Yinzhi Cao, and Takahiro Shinagawa. SmartC2Rust: Iterative, feedback-driven C-to-Rust translation via large language models for safety and equivalence. In *ICSE 2026, Proceedings of the 48th International Conference on Software Engineering*, Rio de Janeiro, Brazil, Apr. 2026. https://github.com/momo-trip/SmartC2Rust.
14. Hanwen Zhang, Yizhuo Luo, Yuxuan Xue, and Zhiyun Qian. Ownership guided C to Rust translation. In *Computer Aided Verification (CAV)*, 2023.

Benedikt Schesch is an Applied Scientist at Amazon. His research focuses on leveraging LLMs and programming tools to produce large-scale, production-ready, safe, and performant software. He received an M.S. in Computer Science from ETH Zurich. Contact him at b.schesch@gmail.com.

Michael D. Ernst is a professor of Computer Science & Engineering at the University of Washington. His research in programmer productivity aims to make software more reliable, more secure, and easier (and more fun!) to produce. His homepage is https://homes.cs.washington.edu/~mernst/. Contact him at mernst@uw.edu.