Theme Article: Language Translation for Security

Agentic translation of C to Rust

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 to the Rust programming language is one way to reduce security vulnerabilities. One common translation strategy is using LLM queries. We present an alternate agentic approach that gives an LLM freedom and guardrails.

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.

Rust prevent many errors via compile-time analyses, such as for type-checking and ownership. Rust programs do not suffer slowdowns from the runtime system, such as garbage collection, making it a promising replacement for C. There is a growing movement to replace software written in C by safer versions written in Rust. To date, that replacement has required manual rewrites, a time-consuming and costly approach that does not scale. An alternate approach is to *automatically* translate C to Rust.

The best-known translator is the C2Rust transpiler (https://tools.galois.com/c2rust). It converts C line-by-line into unsafe Rust code, after which an expert team must fix up the unsafe Rust code, converting it to safe, idiomatic Rust. Many researchers are working to automate the fixup step, typically using LLMs.

Another popular approach is to convert one C function at a time, which is more likely to produce good-quality Rust code.[Hong and Ryu(2025)], [Shiraishi et al.(2026)] These systems fix the signature (parameter and return types) of each function, prompt an LLM to translate each function, then assemble the result into a Rust program.

This article proposes a different approach: *agentic translation* of the entire program. Our experiments show that it outperforms other approaches.

Digital Object Identifier 10.1109/XXX.0000.0000000

LLMS AND AGENTS

A large language model (LLM) is a statistical model of human language text. Given some text, it predicts what may follow. A simple statistical model of English might be able to predict that the next word after "to be or not to" is most likely "be". An LLM can contain trillions of decision-making units (neurons or parameters), enabling it to predict, given a prompt such as "Summarize the following 5 pages of text in one paragraph: . . . ", a paragraph that is likely to answer the question.

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 provides 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 information that appeared in previous user prompts or previous LLM responses. Conversational translation of C to Rust would provide a sequence of prompts. The first prompt might choose a Rust signature for each function, and then there would be one prompt to translate each function. The client of the LLM 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.

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. Crucially, 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 verify 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. We chose the agentic approach for our C-to-Rust translator.

AGENTIC TRANSLATION

Our system takes an entire C project as input and produces a complete Rust project. Our implementation uses the Kiro CLI agent as the underlying agentic framework. Unlike function-by-function approaches, the agent sees all source files and can make holistic decisions, replacing C idioms with Rust equivalents (e.g., linked lists with `Vec`), restructuring modules, and choosing appropriate abstractions. The agent works in an isolated directory containing only the C source code, with no access to test cases or expected outputs. Our pipeline has three phases: translation, verification, and testing.

Translation

The agent receives a natural-language prompt describing the translation task and its constraints. The prompt specifies that the output must be behaviorally identical to the C code, that the agent must produce a compilable Cargo project, that safe Rust should be used wherever possible, and that the agent must not "fix" bugs in the C code but reproduce them exactly. The agent must iteratively compile and fix errors until the build succeeds.

The prompt varies by case type but the core constraints are the same. For executable translations, the Rust binary 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.

For large or configurable projects, the agent decomposes the task into subtasks and delegates 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.

Verification

After translation, a second agentic pass verifies correctness using the C implementation as an oracle. The agent receives the translated Rust code alongside the original C source and a prompt that instructs it to test the translation, but gives it 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 must replicate that behavior, not "fix" it. The agent is instructed to build the C code as a shared library, write 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 to higher-level ones, following the call hierarchy. For projects with multiple build configurations, the agent must repeat this process for every feature combination.

Within these constraints, the 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 hand-written test cases.

Testing

Finally, we run each benchmark's test suite against the translated Rust code. The testing methodology differs by benchmark (see below). The full prompts and pipeline implementation are available in our reproduction package.[1]

EVALUATION METHODOLOGY

Goals

To obtain the greatest benefits of translation from C to Rust, the Rust output should be correct, safe, and idiomatic (in that order). The Rust program is correct if it has the same behavior as the C program. The Rust program is safe if it obeys Rust's type and ownership discipline; it has no `unsafe` blocks that bypass the compiler's guarantees. The Rust program is idiomatic if it is written in good Rust style; for example, a C linked list of the form `struct Node { ...; struct Node *next; };` should be translated into a `vec`, not Rust code that manipulates linked list nodes.

We evaluated each of these desiderata.

[1] https://github.com/UW-HARVEST/harvest-agentic

Comparison systems

We compared our system to other C-to-Rust translation systems. We chose C2Rust (https://tools.galois.com/c2rust) as the best deterministic, symbolic approach. We chose SmartC2Rust[Shiraishi et al.(2026)] as the best conversational LLM approach.

Benchmarks

Different papers use different benchmarks to evaluate the quality of their translation. This makes it difficult to compare tools to one another. To enable comparing our work to more translators, our experiments use three different benchmark sets.

(1) CRUST-Bench[Khatry et al.(2025)] contains 100 C repositories averaging 958 LOC each, each paired with manually crafted Rust interfaces and test cases. The agent receives a Rust scaffold with function signatures and test binaries; its task is to implement each function body in idiomatic Rust without using FFI. (2) The benchmarks of SmartC2Rust [Shiraishi et al.(2026)] contains 21 programs averaging 984 LOC). The tests are end-to-end tests, rather than requiring specific structure for the generated Rust programs. (3) The benchmarks of the DARPA TRACTOR program (https://www.darpa.mil/research/programs/translating-all-c-to-rust) contain 338 test cases averaging 172 LOC (largest 8,513 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.

Each benchmark comes with test cases for the C programs.

Metrics

To evaluate correctness, we ran each Rust program upon the test cases supplied with the benchmark. We counted the translation of program *P* as correct if the Rust version of *P* passed every test case. We counted the translation as incorrect if it failed any test case.

To evaluate safety, we counted the number of unsafe blocks in each translated program and the number of lines of code within unsafe blocks.

To evaluate idiomicity, we asked experienced Rust programmers to manually assess compiled code and rank its idiomicity on a Likert scale. To reduce human effort, we only did this evaluation for Rust programs that are correct.

TABLE 1. DARPA TRACTOR results. syn = synthetic, org = organic. Builds = programs that compile successfully. Tests = test vectors passed. Rust LOC = total lines of generated Rust (excluding verification tests). Unsafe LOC = lines inside `unsafe` blocks (% of total); this benchmark requires C-compatible FFI exports, which inherently need `unsafe`.

Battery	System	Builds	Tests	LOC	Unsafe
B01-syn	Ours	85/85	85/85	6k	1.2k (22%)
	C2Rust	85/85	85/85	7k	3k (44%)
B01-org	Ours	38/38	38/38	3k	1.0k (37%)
	C2Rust	38/38	38/38	10k	3k (28%)
B02-syn	Ours	42/42	31/42	15k	5k (34%)
	C2Rust	42/42	38/42	23k	17k (74%)
B02-org	Ours	44/44	42/44	25k	17k (66%)
	C2Rust	44/44	42/44	46k	35k (76%)
P00	Ours	1/1	1/1	0.6k	0 (0%)
	C2Rust	1/1	1/1	1.7k	0.5k (30%)
P01	Ours	128/128	80/128	613k	83k (14%)
	C2Rust	128/128	0/128	633k	347k (55%)

RESULTS

DARPA TRACTOR

Table 1 shows results on the DARPA TRACTOR benchmarks, where the translated Rust must be a drop-in shared-library replacement for the original C.

C2Rust fails all 128 test cases on P01_sphincs_plus despite compiling successfully. The C codebase declares the function `randombytes` with conflicting return types across different files (`void` in one, `int` in another). C2Rust transpiles both declarations literally, producing code that compiles but behaves incorrectly at runtime. An agentic translator can detect and resolve such inconsistencies.

CRUST-Bench

Table 2 shows results on CRUST-Bench, where the agent implements Rust function bodies from a scaffold and correctness is evaluated by the benchmark's own Rust test suite.

We used a slightly modified version of CRUST-bench, because we found errors in its tests and translations. We reported these problems to the authors of CRUST-bench, who acknowledged them and fixed the benchmark. For examples, see issues 37, 39, 40, and 41 at https://github.com/anirudhkhatry/CRUST-bench/issues.

We discovered these errors because our system generated Rust code that failed either the CRUST-bench tests or tests that our system our generated. When we examined the errors, we discovered that our system was correct and the benchmark was incorrect.

The best previous system, as reported by the CRUST-bench leaderboard, was GPT-5 (high) + test repair, which correctly translated 70% of the programs in CRUST-bench.

We report results on 95 of the 100 projects, ex-

TABLE 2. CRUST-Bench results. Builds = projects that compile successfully. Tests = projects passing all test cases. Unsafe LOC = lines inside `unsafe` blocks (% of total). Prior results from the CRUST-Bench leaderboard [Khatry et al.(2025)]; translated code is not public, so unsafe LOC is unavailable for prior systems. We skip 5 of the 100 projects due to unsatisfiable tests.

System	Setting	Builds	Tests	Unsafe
Ours	self-verified	95/95	80/95	2k (2%)
Ours	benchmark tests	**95/95**	**86/95**	2k (2%)
Prior results from CRUST-Bench leaderboard				
GPT-5 (high)	compiler repair	92/100	43/100	–
	test repair	85/100	70/100	–
Gemini 3 Pro	compiler repair	88/100	41/100	–
	test repair	83/100	66/100	–
OpenAI o3	compiler repair	68/100	31/100	–
	test repair	63/100	48/100	–
SWE-agent	agentic	41/100	32/100	–

TABLE 3. SmartC2Rust benchmark results. Tests = percentage of test cases passed. Unsafe LOC = lines inside `unsafe` blocks. Prior results from Shiraishi et al. [Shiraishi et al.(2026)].

System	Builds	Tests	LOC	Unsafe
Ours	21/21	95%	5k	0
Prior results				
SmartC2Rust	21/21	100%	–	8
C2Rust	21/21	80%	–	1,550
C2SaferRust	19/21	80%	–	871
Crown	11/21	35%	–	1,553
Laertes	19/21	80%	–	954

cluding 5 with unsatisfiable tests. Prior systems on the leaderboard report out of 100. Even counting our 5 excluded projects as failures, our score of 86/100 (86%) still exceeds the best prior result of 70/100 (70%).

SmartC2Rust Benchmarks

Table 3 shows results on the 21-program benchmark of SmartC2Rust [Shiraishi et al.(2026)]. Prior results are from the SmartC2Rust paper.

Discussion

On CRUST-Bench, our agentic approach passes 86 of 95 projects (91%), compared to 70 of 100 (70%) for the best prior system (GPT-5 with test repair). Repair-loop approaches have a fixed budget of three rounds and can only react to errors. Our agent can plan, restructure code, and iterate as many times as needed. SWE-agent, which is also agentic, only passes 32 of 100 projects, suggesting that being agentic alone is not sufficient; the prompt design and task decomposition matter.

On safety, our CRUST-Bench translations contain only 2% unsafe code, while C2Rust on the TRACTOR benchmarks produces 28–76% unsafe code. On the same TRACTOR benchmarks, our translations contain 14–66% unsafe code, still high but substantially less than C2Rust. The remaining unsafe in our TRACTOR translations is largely due to the benchmark's requirement that translated libraries export C-compatible symbols via FFI. On CRUST-Bench, where no FFI is required, our unsafe rate drops to 2%. A mechanical transpiler like C2Rust converts pointer operations literally, preserving every `unsafe` pattern from the original C. An agentic translator can instead choose safe Rust idioms from scratch, replacing linked lists with `Vec`, raw pointers with references, and manual memory management with ownership.

On the DARPA TRACTOR benchmarks, our system achieves 100% build success across all batteries and passes all tests on B01. On B02 and P01, some test failures remain: 31 of 42 on B02-synthetic and 80 of 128 on P01-sphincs. C2Rust achieves near-perfect test pass rates on B01 and B02 because its literal transpilation preserves behavior by construction. However, it fails all 128 test cases on P01_sphincs_plus, where conflicting function declarations across files cause incorrect runtime behavior. C2Rust cannot reason about such inconsistencies; it can only mechanically convert what it sees. An agentic translator detects the conflict and resolves it.

The self-verified setting, where the agent generates its own tests without access to the benchmark's test suite, is the more realistic scenario: in practice, a translation tool does not receive a ground-truth test suite. If self-verified results approach benchmark-test results, that validates the verification step as an effective substitute for hand-written tests.

To evaluate idiomaticity, two experienced Rust programmers independently rated a sample of correct translations on a scale of 1 (not idiomatic) to 5 (fully idiomatic). Our system scored an average of 3.0, compared to 1.0 for C2Rust.

Our approach has limitations. Agentic translation is slower and more expensive than C2Rust or single-shot LLM queries. LLM outputs are nondeterministic, so results may vary between runs. We have not yet evaluated on codebases exceeding 10,000 lines of code.

CONCLUSION

We have introduced a new, agentic approach to translating C to Rust with an LLM. Our system outperforms other symbolic and neural approaches on metrics of correctness, safety, and idiomicity. On CRUST-Bench, our system passes 86 of 95 projects, outperforming the best prior system's 70 of 100. On the DARPA TRACTOR benchmarks, it achieves 100% build suc-

cess across all batteries and passes 80 of 128 test cases on the largest project (sphincs_plus), where C2Rust passes none. A key enabler is the verification step, where the agent uses the original C code as an oracle to test its own translation without requiring hand-written test cases. This success suggests that agentic approaches should be applied more broadly to other topics in computing, even ones where correctness is a requirement. Future work includes scaling to larger codebases, reducing translation cost, and extending the approach to other unsafe-to-safe language pairs.

ACKNOWLEDGMENTS

We thank the UW HARVEST team for their assistance. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590132.

REFERENCES

Hong and Ryu(2025). Jaemin Hong and Sukyoung Ryu. 2025. Type-migrating C-to-Rust translation using a large language model. *Empirical Softw. Engg.* 30, 3 (2025).

Khatry et al.(2025). Anirudh Khatry, Robert Zhang, Jia Pan, Ziteng Wang, Qiaochu Chen, Greg Durrett, and Isil Dillig. 2025. CRUST-Bench: A Comprehensive Benchmark for C-to-safe-Rust Transpilation. arXiv:2504.15254 [cs.SE] https://arxiv.org/abs/2504.15254

Shiraishi et al.(2026). Momoko Shiraishi, Yinzhi Cao, and Takahiro Shinagawa. 2026. 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.

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.