
[def/acc] GUARDIAN: Guarded Universal Architecture for Defensive Interpretation And traNslation

Aditya Thalang
Independent

Josh Brown
Roadfire Software

With
Apart Research

Abstract

GUARDIAN is a multi-stage, LLM-driven system to automate the translation of C codebases to memory-safe Rust. GUARDIAN promotes defense acceleration at-scale by guiding an LLM transpiler with dependency graph strongly-connected-components, static-analysis-guided rule hints, examples from the demonstration corpora and iterative, compiler-guided refinement.

In evaluation on 27 C functions, including 20 with adversarial vulnerabilities, GUARDIAN achieves 100% compilation success and 92.6% fully safe outputs, outperforming a baseline LLM by 22.2pp. GUARDIAN demonstrates that safety-oriented constraints can significantly improve automated translation quality at scale. Limitations include evaluation on a small test set and a lack of functional-equivalence guarantees; future work will target repository-scale evaluation, expanding the classes of vulnerabilities covered by static analyses, adding functional-equivalence guarantees and robust evaluation sandboxing.

1. Introduction

Memory-unsafe operations remain a dominant source of high-severity vulnerabilities in C and other memory-unsafe languages. (CWE, 2024) At the same time, adversaries are beginning to use large language models (LLMs) to automate the discovery and exploitation of such vulnerabilities, increasing the speed and scale of attacks that previously required humans in the loop.

Industry and government bodies have called for a broad shift toward memory-safe languages such as Rust for critical codebases.(MSRC, 2019; Office of the National Cyber Director, 2024) However, large-scale migration from C to Rust remains slow and labour-intensive. Existing tools like c2rust can generate semantically equivalent Rust codebases, but the result typically contains large unsafe regions that require substantial manual refactoring to achieve idiomatic, memory-safe Rust. (*Immunant/C2rust*, 2018/2025) These unsafe regions can contain vulnerabilities not caught by Rust’s compiler or sanitisers. (Wu et al., 2025)

LLMs have shown significant promise at the task of code translation. However, unguided LLMs have struggled with full-repository translation from C to Rust. (Khatry et al., 2025; Ou et al., 2025) A number of distinct LLM-driven pipelines have been proposed to translate from C to Rust. (Zhou et al., 2025; Nitin et al., 2025; Luo et al., 2025)

We present GUARDIAN, a multi-stage LLM-driven translation tool from C to Rust which translates whole repositories by operating on the strongly-connected-components (SCC) of the dependency graph of structs, typedefs, enums and functions. The code generation is augmented with rule hints based on static analyses as well as a summary of the SCCs to implement and the signatures of immediate dependencies.

2. Methods

GUARDIAN implements a multi-stage pipeline:

Key Components

Static Analyser: The static analyser checks for patterns of four categories, following Luo et al., 2025:

- I/O operations (`scanf` ==> `read_line`)
- pointers (`malloc` ==> `Vec/Box`)
- arrays (int index ==> `as usize`)
- type mixing (==> explicit casts)

Example Corpus: 15 hand-crafted C to Rust pairs showing safe idioms, retrieved by BM25 matching detected pattern categories

Translator: DSPy chain-of-thought module receives hints + examples + structured summary

Compiler Verification: `rustc` catches type errors, borrow checker violations, lifetime issues and provides automatic safety validation

Bounded Refinement: Fixes `rustc` errors with context, with a max of 3 iterations

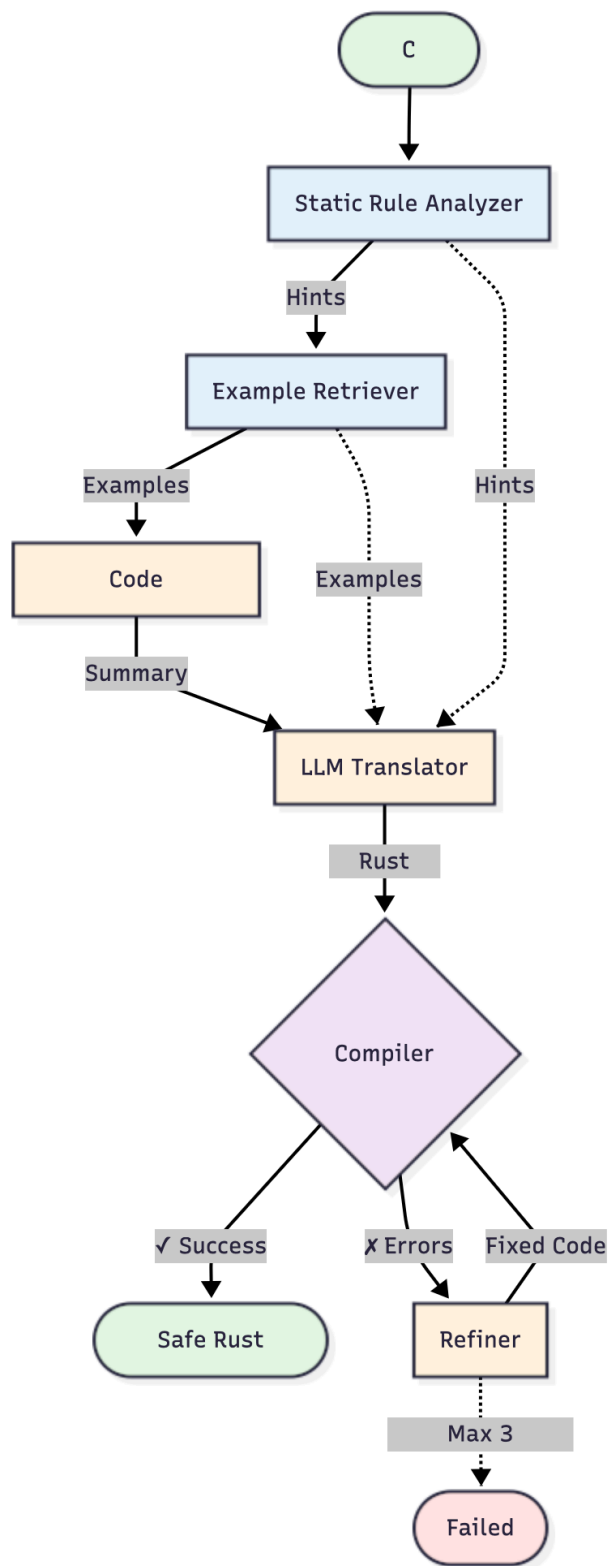


Figure 1 – System Architecture

Evaluation

Test suite: 27 C programs

- 7 basic: patterns such as scanf, malloc, arrays, casts
- 20 adversarial: buffer overflow, use-after-free, double free, null deref, integer overflow, and more

Baseline: Vanilla LLM with simple prompt. No analysis, examples, or refinement.

Metrics:

- Compilation success (**rustc** passes)
- Safe translations (zero **unsafe** blocks)
- Unsafe block count

Implementation: Python, DSPy, libclang, rustc, LLM

Repo: <https://github.com/athalang/deface>

3. Results

Quantitative Results

Basic cases (7 samples): Both approaches achieve 100% compilation, 100% safe results.

Adversarial cases (20 security vulnerabilities):

Metric	Vanilla LLM	GUARDIAN	Improvement
Compilation Success	13/20 (65%)	20/20 (100%)	+ 35%
Safe (0 unsafe blocks)	12/20 (60%)	18/20 (90%)	+ 30%
Unsafe Blocks	9 total	8 total	- 1

Overall Comparison (27 samples):

Metric	Vanilla LLM	GUARDIAN	Improvement
Compilation Success	21/27 (77.8%)	27/27 (100%)	+ 22.2%
Safe (0 unsafe blocks)	19/27 (70.4%)	25/27 (92.6%)	+ 22.2%
Unsafe Blocks	9 total	8 total	- 1

Qualitative Results

Array Index Out of Bounds

Original C program

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int idx = 10; // Out of bounds!
    printf("%d\n", arr[idx]); // Accessing invalid memory
    return 0;
}
```

Vanilla LLM: Compilation error

```
fn main() {
    let arr = [1, 2, 3, 4, 5];
    let idx = 10;
    if idx < arr.len() {
        println!("{}", arr[idx]);
    } else {
        eprintln!("Index out of bounds");
    }
}
```

GUARDIAN: Compilation success

```
fn main() {
    let arr = [1, 2, 3, 4, 5];
    let idx = 10;
    match arr.get(idx as usize) {
        Some(value) => println!("{}", value),
        None => println!("Index out of bounds"),
    }
}
```

Mechanism: Compiler caught index out of bounds error $\Rightarrow$ refiner restructured $\Rightarrow$ index verified

4. Discussion and Conclusion

GUARDIAN produces measurably safer translations than the baseline LLM in evaluation. The pipeline successfully compiled all 27 programs and increased fully safe translations from 70.4% to 92.6%, with the largest gains occurring in adversarial cases containing memory-unsafe patterns. A key outcome is that GUARDIAN consistently converts dangerous constructs such as unchecked pointer arithmetic and out-of-bounds indexing into compiler-verified Rust equivalents. The qualitative case study illustrates how the refinement stage uses compiler diagnostics to revise translations into safe, idiomatic forms. This behavior supports the broader objective of defense acceleration: enforcing safety properties during translation reduces the manual burden of eliminating vulnerabilities in legacy C code and shrinks the window in which AI-enabled offensive tools can exploit them.

However, the results also highlight several constraints. The test suite contains small, self-contained programs, and the system’s correctness beyond compilation was not evaluated. Two translations retained minimal unsafe blocks, indicating that the current analyses and refinement loop do not guarantee complete elimination of unsafe constructs.

For GUARDIAN to be used in production, we would have to expand testing to full repositories with realistic dependency structures, evaluate semantic equivalence through unit-test generation or specification alignment, and broaden static analyses to detect additional classes of vulnerabilities. Performance comparisons between translated Rust and original C code, multi-model experiments, and adversarial red-team evaluations will further clarify GUARDIAN’s reliability and defensive value at scale.

5. References

CWE. (2024). *2024 CWE Top 25 Most Dangerous Software Weaknesses*.

https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

Immunant/c2rust. (2025). [Rust]. Immunant.

<https://github.com/immunant/c2rust> (Original work published 2018)

Khatry, A., Zhang, R., Pan, J., Wang, Z., Chen, Q., Durrett, G., & Dillig, I.

(2025). *CRUST-Bench: A Comprehensive Benchmark for C-to-safe-Rust*

Transpilation (No. arXiv:2504.15254). arXiv.

<https://doi.org/10.48550/arXiv.2504.15254>

MSRC. (2019, July 16). *A proactive approach to more secure code*.

<https://www.microsoft.com/en-us/msrc/blog/2019/07/a-proactive-approach-to-more-secure-code>

Office of the National Cyber Director. (2024). *BACK TO THE BUILDING BLOCKS: A PATH TOWARD SECURE AND MEASURABLE SOFTWARE*.

<https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf>

Ou, G., Liu, M., Chen, Y., Wang, Y., Peng, X., & Zheng, Z. (2025).

RustRepoTrans: Repository-level Code Translation Benchmark Targeting Rust (No. arXiv:2411.13990). arXiv.

<https://doi.org/10.48550/arXiv.2411.13990>

Wu, S., Yang, H., & Hua, B. (2025). Security Risks of Transpiling C Programs to Rust. *2025 IEEE 24th International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom 2025)*.

Feng Luo et al., “Integrating Rules and Semantics for LLM-Based C-to-Rust Translation,” arXiv:2508.06926, preprint, arXiv, August 9, 2025, <https://doi.org/10.48550/arXiv.2508.06926>.

Nitin, Vikram, Rahul Krishna, Luiz Lemos do Valle, and Baishakhi Ray. “C2SaferRust: Transforming C Projects into Safer Rust with NeuroSymbolic Techniques.” arXiv:2501.14257. Preprint, arXiv, January 24, 2025. <https://doi.org/10.48550/arXiv.2501.14257>.

Zhou, Tianyang, Haowen Lin, Somesh Jha, Mihai Christodorescu, Kirill Levchenko, and Varun Chandrasekaran. “LLM-Driven Multi-Step Translation from C to Rust Using Static Analysis.” arXiv:2503.12511. Preprint, arXiv, March 18, 2025. <https://doi.org/10.48550/arXiv.2503.12511>.

6. Appendix: Security Considerations

Potential Limitations:

1. **Limited Coverage:** GUARDIAN was tested on self-contained programs; large C projects introduce complex dependency structures, build configurations, and platform-specific behavior that may expose failure modes not observed in the current test suite.
2. **Semantic Gaps:** Compilation success and the absence of unsafe blocks do not ensure semantic equivalence with the original C program. Logic errors, incorrect edge-case handling, or subtle behavioral deviations may persist even when the translated Rust code is memory-safe.
3. **Adversarial Robustness:** Unknown if crafted C code could bypass defenses. Needs red-team testing.
4. **Residual Unsafe Blocks:** A small percentage of cases contain unsafe blocks. Static analysis needs enhancement.
5. **Performance Unknown:** Our design effort focused on safety which may introduce performance overheads or non-idiomatic patterns in some cases. The impact on runtime efficiency and integration with performance-critical systems remains untested.

Future Improvements:

- Enhanced pattern detection (race conditions, crypto weaknesses)
- Formal verification integration (Kani, Creusot)
- Property-based test generation for semantic correctness
- Performance optimization phase
- Human-in-the-loop review for critical code