Multi-step Translation from C to Rust using Static Analysis

Tianyang Zhou
University of Illinois Urbana-Champaign
May 05, 2025

About Me

- 2nd-year PhD student, CS @ UIUC
  - Advisers: Varun Chandrasekaran & Kirill Levchenko
- Research: System security → ML for software engineering → LLM-based C to Rust translation
- Security background: memory-safety exploits, fuzzing, WebAssembly, container isolation

Why Translate C to Rust

Home / Tech / Security

Microsoft: 70 percent of all security bugs are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

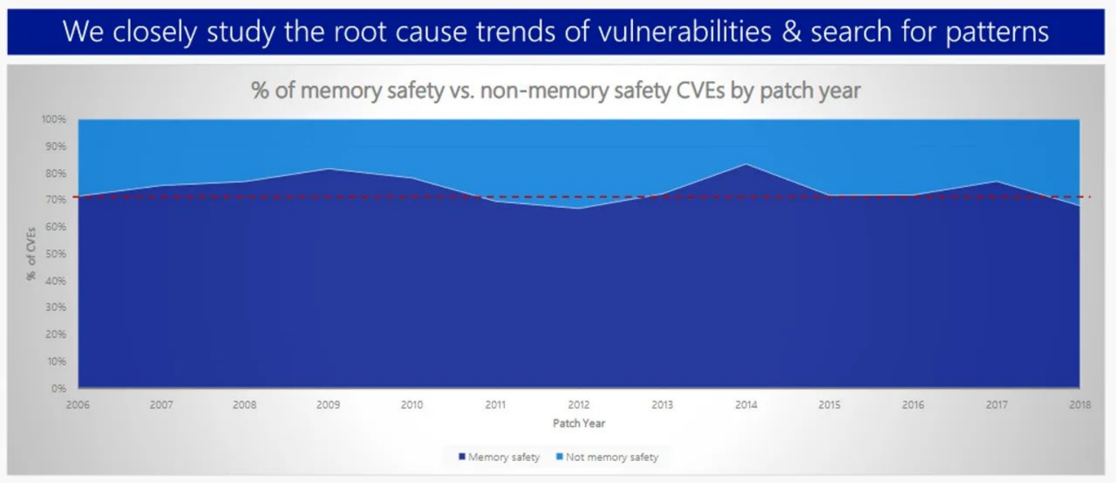
Written by **Catalin Cimpanu,** Contributor

Feb. 11, 2019 at 7:48 a.m. PT

Why Translate C to Rust

Home / Tech / Security

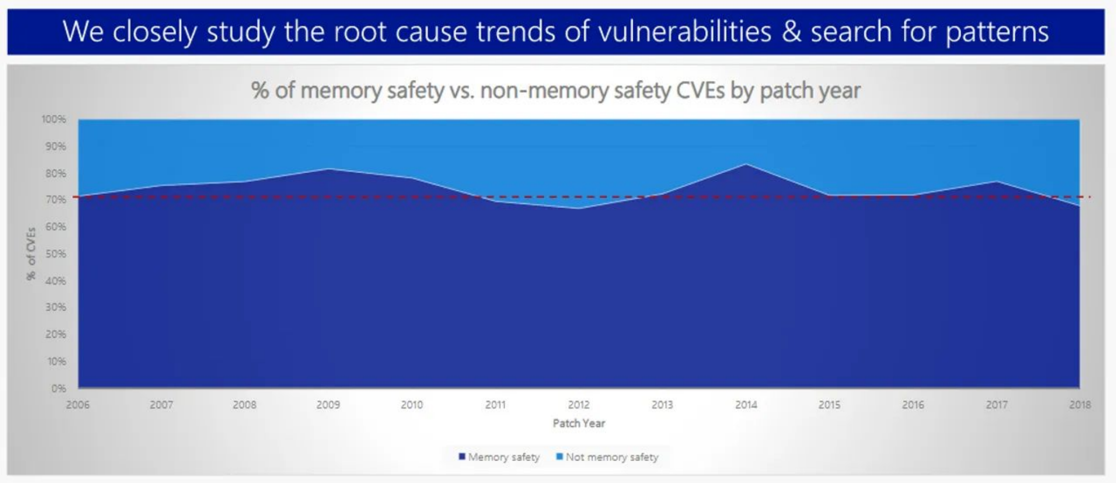
Microsoft: 70 percent of all security bugs are memory safety issues



Why Translate C to Rust

Home / Tech / Security

Microsoft: 70 percent of all security bugs are memory safety issues



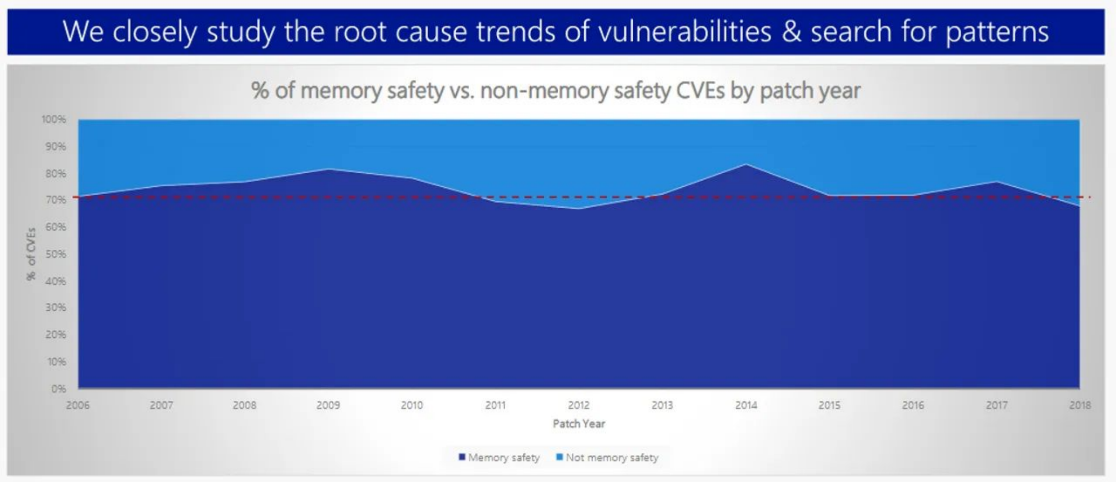
Zero-cost memory safety

- Ownership, Borrow-Checker
- No perf loss

Why Translate C to Rust

Home / Tech / Security

Microsoft: 70 percent of all security bugs are memory safety issues

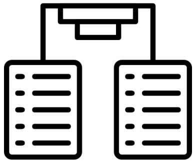


Zero-cost memory safety

- Ownership, Borrow-Checker
- No perf loss

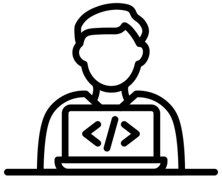
Manual write is hard → Need automation

Translation Challenges



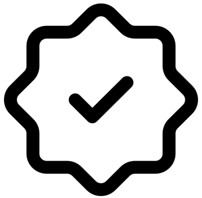
Semantics

- Pointer vs Reference
- Lifetime
- Ownership …



Idiomaticity

- C types to Rust types
- Unsafe remove
- Write in the “Rust way”



Verification

- Hard to prove
- Too strict or too relax

C vs Rust: Key Differences

	C	Rust
Memory Mgmt.	`malloc/free`	Ownership + Borrow checker
Null safety	`NULL`, unchecked	`Option<T>`
Pointers	Raw pointers only	Safe refs, `Box`, smart ptrs; raw pointer only under `unsafe` gates
Concurrency	Data-race prone	Compile-time race freedom, no data-race possible in safe Rust
Error handling	Return error code	`Result<T,E>` pattern matching
Type conversion	Implicit + explicit type conversions	Explicit only

Idiomatic Rust: Why It's Important

- What is idiomatic Rust
  - Idiomatic Rust ≠ Compiles-in-Rust
  - Reads like native Rust: expressive types, pattern matching, iterators
  - `unsafe` kept *minimal & audited* (ideally 0%)
  - Follows community lints (rust-clippy) and module conventions
- Why it's important:
  - Memory safety are only **guaranteed** by compiler in **unsafe-free** blocks
  - Critical for long-term maintainability, contributor onboarding

Code Examples: Unidiomatic vs Idiomatic Rust

```rust
// Unidiomatic
unsafe fn strlen(s: *const u8) -> usize {
    let mut len = 0;
    while *s.add(len) != 0 {
        len += 1;
    }
    len
}

// Idiomatic
fn strlen(s: &str) -> usize {
    s.len()
}
```

Unidiomatic Rust can not ensure the memory safety, but idiomatic Rust can!

Non-LLM Baseline: C2Rust

- Pros:
    - Robust AST-level converter; handles full C99
    - Always compiles
    - Functional equivalent by design
- Cons:
    - Produces verbose, unreadable code littered with `unsafe` (≈ 100 % unsafe-token fraction)
        - Not enough memory safety provided
    - Strips comments/macros: unmaintainable
- C2Rust is designed as the starting point for manual code translation

Code Example of C2Rust Output

```c
void to_uppercase(char *s) {
    for (int i = 0; s[i] != '\0'; ++i) {
        if (s[i] >= 'a' && s[i] <= 'z') {
            s[i] = s[i] - 'a' + 'A';
        }
    }
}
```

C code

```rust
#[no_mangle]
pub unsafe extern "C" fn to_uppercase(mut s: *mut libc::c_char) {
    let mut i: libc::c_int = 0 as libc::c_int;
    while *s.offset(i as isize) as libc::c_int != '\0' as i32 {
        if *s.offset(i as isize) as libc::c_int >= 'a' as i32
            && *s.offset(i as isize) as libc::c_int <= 'z' as i32
        {
            *s
                .offset(
                    i as isize,
                ) = (*s.offset(i as isize) as libc::c_int - 'a' as i32 + 'A' as i32)
                as libc::c_char;
        }
        i += 1;
        i;
    }
}
```

C2Rust Translated code

Other Non-LLM Approaches

- **Baseline:** Most build on C2Rust for initial translation
- **Crown** (ICCAV ’23): Ownership analysis to reduce unsafe pointer use
- **Rule/Heuristic tools** (Hong & Ryu ’24; Ling et al. ’22): Target specific idioms (e.g., improve output params, handle null ptr)
- All of these approaches have significant defects:
  - Still ≈ 100 % unsafe-token fraction – No safety guarantee
  - Code still mostly unreadable

LLM Approaches

Approach	Idea/Pros	Cons
C2SaferRust ('25)	LLM-polishes C2Rust output; Significantly reduce `unsafe`	Still tied to C2Rust (unmaintainable); moderate unsafe remains;
Syzygy ('24)	Uses runtime traces to analyze pointers + guide LLM	Dynamic analysis can be inaccurate for low-coverage tests
Fluorine ('24)	Iterative prompt-repair with compile errors; Verify by fuzzing and data type serialization;	Requires per-function fuzzing specs; 50% failures due to serialization/type mismatch;
Vert ('24)	Fuzz + SymEx for equivalence	Scales to small programs only; Unable to support complex code features

Consolidate All of The Cons From Prior Work

- Heavy reliance on `unsafe` ⇒ safety not guaranteed
- Loss of readability/idiomaticity
- Limited support for complex code features
  - e.g. function pointers, complex data structures
- Verification unscalable
  - Fuzzing testing: requires hand-written input specs per function
  - Symbolic execution: too strict, may introduce false positives (functional equivalence but semantic mismatches)

SACTOR (Structure-Aware C-to-Rust Translator)

Key ideas (high-level):

- Static-analysis-guided prompts
  - Prompt the LLM with concrete pointer + dependency information
- Syntactic-first (unidiomatic translation), semantic-second (idiomatic translation)
  - Unidiomatic translation: preserve behavior
  - Idiomatic translation: strip `unsafe` for idioms
- Verification by end-to-end tests
  - Link translated back into C tests via FFI harnesses
- Result: Keeps the high correctness with better idiomaticity and better adaptability

SACTOR Methodology

1. **Task division** (libclang): dependency-ordered fragments
    - Obtain dependency graph of functions and data structures
2. **Step 1 – Unidiomatic translation**
    - LLMs translates with C-like semantics (allows `unsafe`, allows `libc` functions)
3. **Step 2 – Idiomatic refinement**
    - Use `Crown` as pointer analyzer, extract pointer metadata (fatness, ownership, mutability)
    - Prompt LLM with pointer metadata
4. **Verification loop by end-to-end tests**
    - Compile + FFI end-to-end tests until pass (≤ 6 tries)
    - Feedback with compilation errors or test error information (input+output)

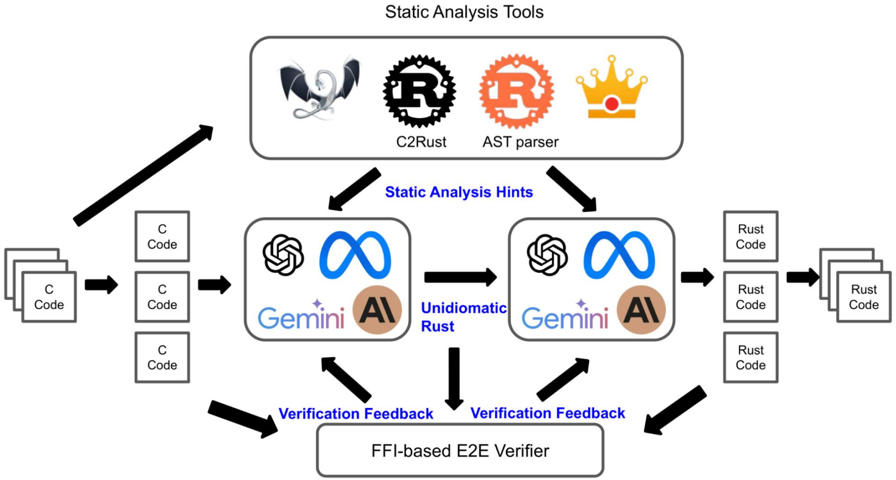


Figure 2: Overview of the SACTOR methodology.

How Static Analysis Helps Translation

- C parser (based on libclang): extract types, globals, function signatures, dependency graph
    - Extract translation order
    - Adding dependency into prompt as extra information
- Crown: pointer ownership/fatness: suggest `&T`, `Box<T>` vs raw pointers
    - Provide pointer analyze information to LLM
- Rust procedure macro: Collect input/output of target function when test failed
    - Inject debug code into target translated function
    - Collect debug information + valgrind (memory checker) output

Verification Strategy: How We Use FFI to Run Tests

- Unidiomatic phase:
  - Compile Rust function as a shared library
  - Directly link back into C via shared library, reuse existing end-to-end tests
- Idiomatic phase:
  - Use LLM to generate test harness: Converts C ↔ Rust structs, function signatures
  - Link translated function together with harness back into C
- Feedback:
  - compiler errors & logged I/O collected from injected debugging code fed into next LLM attempt

```
fn concat_str_idiomatic(orig: &str, num: i32) -> String {
    format!("{}{}", orig, num)
}

fn concat_str(orig: *const c_char, num: c_int) -> *const c_char {
    // convert input
    let orig_str = CStr::from_ptr(newName)
        .to_str()
        .expect("Invalid UTF-8 string");
    // call target function
    let out = concat_str_idiomatic(orig_str, num as i32);
    // convert output
    let out_str = CString::new(out).unwrap();
    // `into_raw` transfers ownership to the caller
    out_str.into_raw()
}
```

Figure 3: Example of the test harness function.

Example Code Translation Process: atoi

```c
#include <stdio.h>

int atoi(char *str) {
    int result = 0;
    int sign = 1;

    while (*str == ' ' || *str == '\t' || *str == '\n' ||
           *str == '\r' || *str == '\v' || *str == '\f') {
        str++;
    }

    if (*str == '+' || *str == '-') {
        if (*str == '-') {
            sign = -1;
        }
        str++;
    }

    while (*str >= '0' && *str <= '9') {
        result = result * 10 + (*str - '0');
        str++;
    }

    return sign * result;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number>\n", argv[0]);
        return 1;
    }

    int value = atoi(argv[1]);
    printf("Parsed integer: %d\n", value);
    return 0;
}
```

(a) C implementation of `atoi`

```rust
use libc::c_char;
use std::env;
use std::ffi::CString;
use std::process;
pub unsafe fn atoi(str: *const c_char) -> i32 {
    let mut result: i32 = 0;
    let mut sign: i32 = 1;
    let mut ptr = str;
    while *ptr == ' ' as c_char
        || *ptr == '\t' as c_char
        || *ptr == '\n' as c_char
        || *ptr == '\r' as c_char
        || *ptr == '\x0B' as c_char
        || *ptr == '\x0C' as c_char
    {
        ptr = ptr.add(1);
    }
    if *ptr == '+' as c_char || *ptr == '-' as c_char {
        if *ptr == '-' as c_char {
            sign = -1;
        }
        ptr = ptr.add(1);
    }
    while *ptr >= '0' as c_char && *ptr <= '9' as c_char {
        let digit = (*ptr - '0' as c_char) as i32;
        if let Some(new_result) = result.checked_mul(10).and_then(
            |r| r.checked_add(digit),
        ) {
            result = new_result;
        } else {
            return if sign == 1 { i32::MAX } else { i32::MIN };
        }
        ptr = ptr.add(1);
    }
    sign * result
}
pub fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() != 2 {
        println!("Usage: {} <number>", args[0]);
        process::exit(1);
    }
    let c_str = match CString::new(args[1].as_str()) {
        Ok(cstring) => cstring,
        Err(_) => {
            eprintln!("Failed to create CString from input");
            process::exit(1);
        }
    };
    let value = unsafe { atoi(c_str.as_ptr() as *const c_char) };
    println!("Parsed integer: {}", value);
}
```

(b) Unidiomatic Rust translation from C

Example Code Translation Process: atoi

```rust
use libc::c_char;
use std::env;
use std::ffi::CString;
use std::process;
pub unsafe fn atoi(str: *const c_char) -> i32 {
    let mut result: i32 = 0;
    let mut sign: i32 = 1;
    let mut ptr = str;
    while *ptr == ' ' as c_char
        || *ptr == '\t' as c_char
        || *ptr == '\n' as c_char
        || *ptr == '\r' as c_char
        || *ptr == '\x0B' as c_char
        || *ptr == '\x0C' as c_char
    {
        ptr = ptr.add(1);
    }
    if *ptr == '+' as c_char || *ptr == '-' as c_char {
        if *ptr == '-' as c_char {
            sign = -1;
        }
        ptr = ptr.add(1);
    }
    while *ptr >= '0' as c_char && *ptr <= '9' as c_char {
        let digit = (*ptr - '0' as c_char) as i32;
        if let Some(new_result) = result.checked_mul(10).and_then(
            |r| r.checked_add(digit),
        ) {
            result = new_result;
        } else {
            return if sign == 1 { i32::MAX } else { i32::MIN };
        }
        ptr = ptr.add(1);
    }
    sign * result
}
pub fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() != 2 {
        println!("Usage: {} <number>", args[0]);
        process::exit(1);
    }
    let c_str = match CString::new(args[1].as_str()) {
        Ok(cstring) => cstring,
        Err(_) => {
            eprintln!("Failed to create CString from input");
            process::exit(1);
        }
    };
    let value = unsafe { atoi(c_str.as_ptr() as *const c_char) };
    println!("Parsed integer: {}", value);
}
```

(b) Unidiomatic Rust translation from C

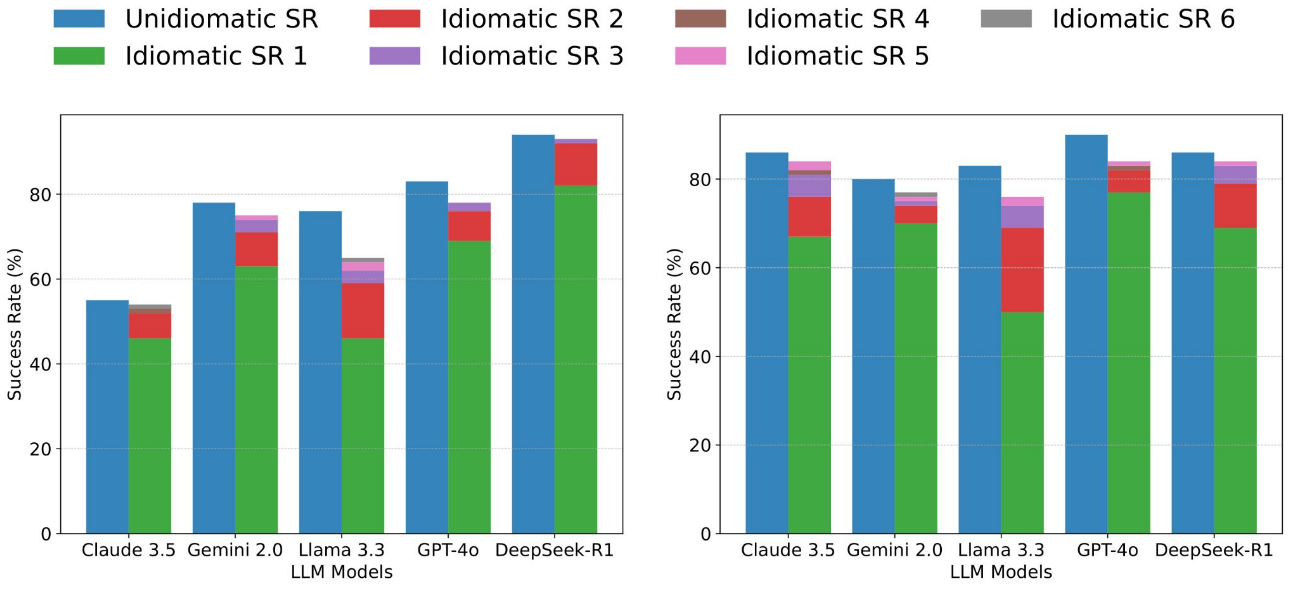
```rust
use std::env;
use std::process;
pub fn atoi(input: &str) -> i32 {
    let mut result: i32 = 0;
    let mut sign: i32 = 1;
    let mut chars = input.chars().peekable();
    while let Some(&c) = chars.peek() {
        if c.is_whitespace() {
            chars.next();
        } else {
            break;
        }
    }
    if let Some(&c) = chars.peek() {
        if c == '+' || c == '-' {
            if c == '-' {
                sign = -1;
            }
            chars.next();
        }
    }
    for c in chars {
        if let Some(digit) = c.to_digit(10) {
            if let Some(new_result) = result
                .checked_mul(10)
                .and_then(|r| r.checked_add(digit as i32))
            {
                result = new_result;
            } else {
                return if sign == 1 { i32::MAX } else { i32::MIN };
            }
        } else {
            break;
        }
    }
    sign * result
}
pub fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() != 2 {
        println!("Usage: {} <number>", args[0]);
        process::exit(1);
    }
    let input = &args[1];
    let value = atoi(input);
    println!("Parsed integer: {}", value);
}
```

(c) Idiomatic Rust translation from unidiomatic Rust

Experiment Setup

- Datasets
  - 100 TransCoder-IR programs
  - 100 CodeNet programs
  - 2 real projects (AVL tree, urlparser)
- LLMs
  - GPT-4o, Claude 3.5, Gemini 2.0, Llama 3.3-70B, DeepSeek-R1
  - Evaluated on 01/2025
  - Gemini 2.5 pro
- Metrics
  - Success Rate (code can compile and pass tests)
  - Idiomaticity
    - Clippy alerts by `cargo clippy` (Rust linter)
    - `unsafe` fraction
  - tokens & queries cost

Experiment Result: Success Rate



(a) TransCoder-IR SR

(b) CodeNet SR

- *TransCoder-IR*: DeepSeek-R1 **93 %**, GPT-4o 78%, Gemini 2.0 75%, Llama 64%, Claude 54%, Gemini 2.5 pro 84%
- *CodeNet*: GPT-4o / Claude / DeepSeek-R1 **84%**, Gemini 2.0 77%, Llama 76%, Gemini 2.5 pro 81%

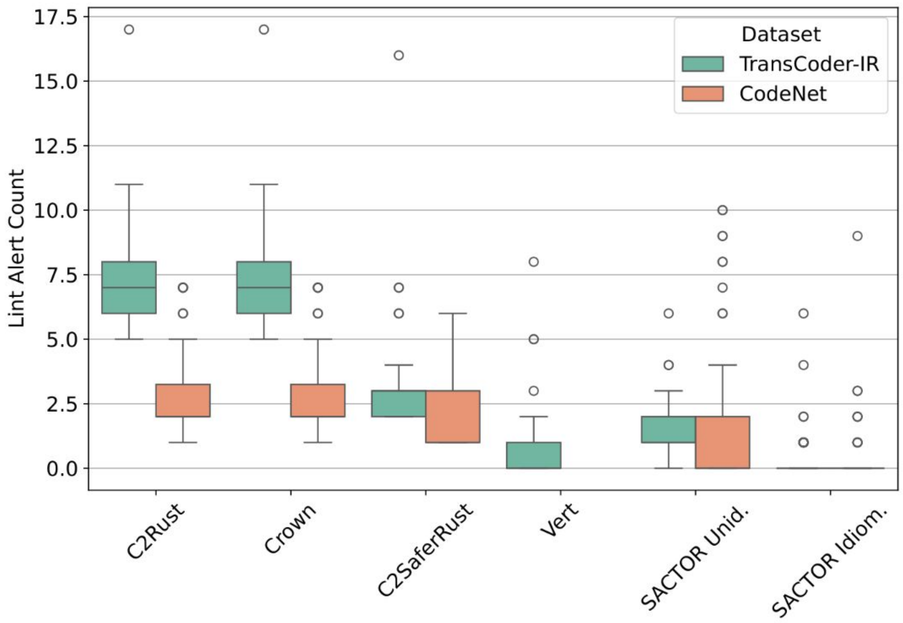
Experiment Result: Cost Across Different LLMs

Table 4: Average Cost Comparison of Different LLMs Across Two Datasets. The color intensity represents the relative cost of each metric for each dataset.

LLM	Dataset	Tokens	Avg. Queries
Claude 3.5	TransCoder-IR	4595.33	5.15
	CodeNet	3080.28	3.15
Gemini 2.0	TransCoder-IR	3343.12	4.24
	CodeNet	2209.38	2.39
Llama 3.3	TransCoder-IR	6265.13	6.13
	CodeNet	3035.43	3.06
GPT-4o	TransCoder-IR	2651.21	4.24
	CodeNet	2565.36	2.95
DeepSeek-R1	TransCoder-IR	17895.52	4.77
	CodeNet	13592.61	3.11

- **GPT-4o & Gemini 2.0** are most efficient: ~2.3–2.7 k tokens & 2–4 queries per program.
- **DeepSeek-R1** although the best, reasoning at a 5-7x token overhead.
- **Gemini 2.5 pro** consumes **10797.34, 5877.65** tokens per dataset; **5.45, 3.05** queries per dataset.
    - Around 2-3x overhead than other models.

Experiment Result: Idiomaticity (Clippy Alerts)



- Only evaluated on GPT-4o
- SACTOR cuts warnings by 90 % relative to C2Rust; beats Vert
- SACTOR unidiomatic beats C2SaferRust

Experiment Result: Idiomaticity (Unsafe Fraction)

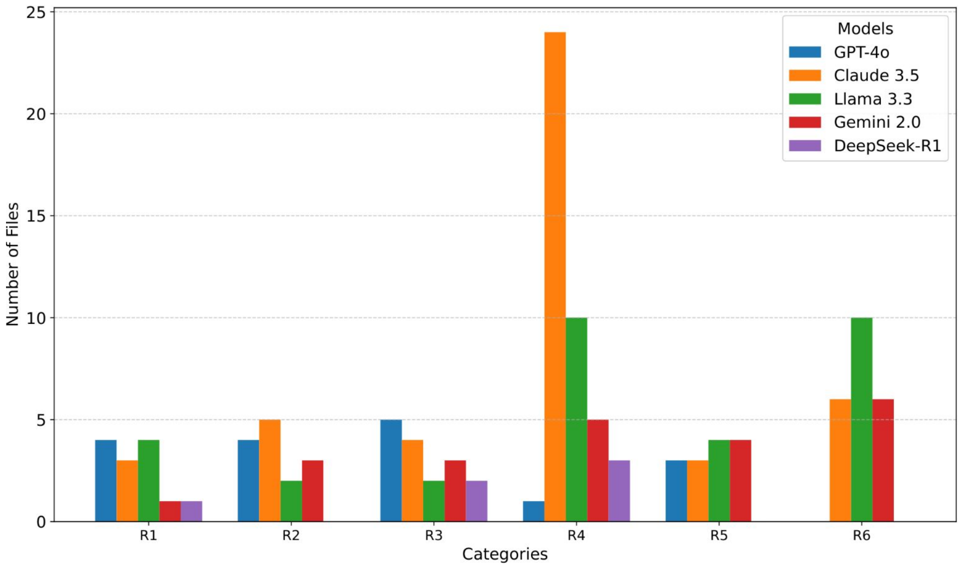
Table 5: Unsafe Code Statistics. UF denotes Unsafe Free and AU denotes Avg. Unsafe

METHOD	DATASET	SR (%)	UF (%)	AU (%)
C2Rust	TransCoder-IR	100	0	100
	CodeNet	100	0	75.9
Crown	TransCoder-IR	100	0	100
	CodeNet	100	0	75.9
C2SaferRust	TransCoder-IR	90	45.6	10.8
	CodeNet	93	0	75.8
Vert	TransCoder-IR	92	95.7	1.6
SACTOR (Unid.)	TransCoder-IR	83	3.6	91.7
	CodeNet	90	1.1	42.7
SACTOR (Idiom.)	TransCoder-IR	78	**100**	**0**
	CodeNet	84	**100**	**0**

- UF: How many fraction of programs are free of `unsafe`; AU: Average `unsafe` fraction across all programs
- `unsafe` fraction: SACTOR **0 %** vs C2Rust 100 %, Crown 100 %, C2SaferRust 11 %

Failure Analysis: TransCoder-IR

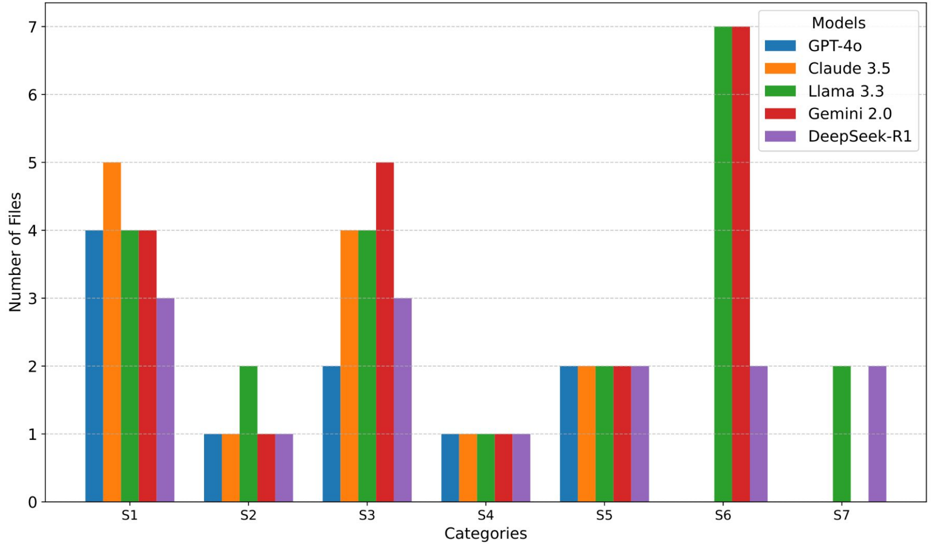
CATEGORY	DESCRIPTION
R1	Memory safety violations in array operations due to improper bounds checking
R2	Mismatched data type translations
R3	Incorrect array sizing and memory layout translations
R4	Incorrect string representation conversion between C and Rust
R5	Failure to handle C's undefined behavior with Rust's safety mechanisms
R6	Use of C-specific functions in Rust without proper Rust unsafe wrapper



- Main failures: string conversion (Claude 3.5 most), array layout, unsafe C calls.
- DeepSeek-R1 reduces errors via reasoning before code.

Failure Analysis: CodeNet

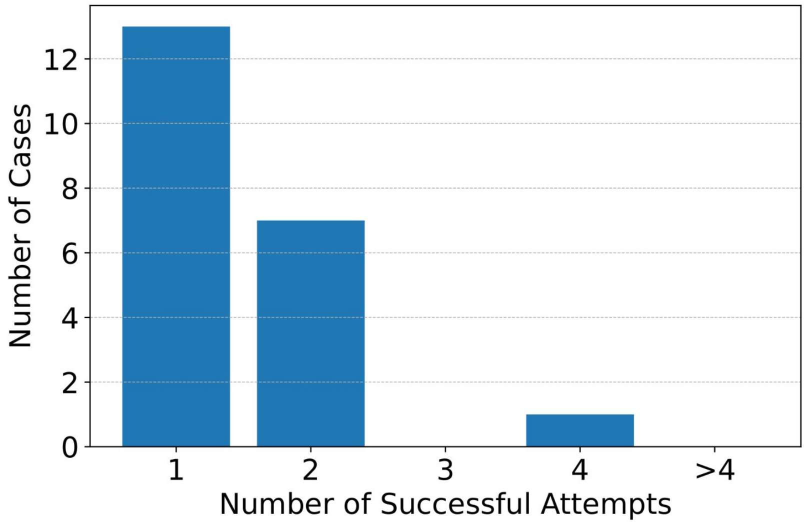
CATEGORY	DESCRIPTION
S1	Improper translation of command-line argument handling or try to fix wrong command-line argument handling
S2	Function naming mismatches between C and Rust
S3	Format string directive mistranslation causing output inconsistencies
S4	Original code contains random number generation
S5	SACTOR unable to translate mutable global state variables
S6	Mismatched data type translations
S7	Incorrect control flow or loop boundary condition translations



- Common issues: format strings, CLI parsing, type mismatches.
- All models struggle with precise C I/O semantics.

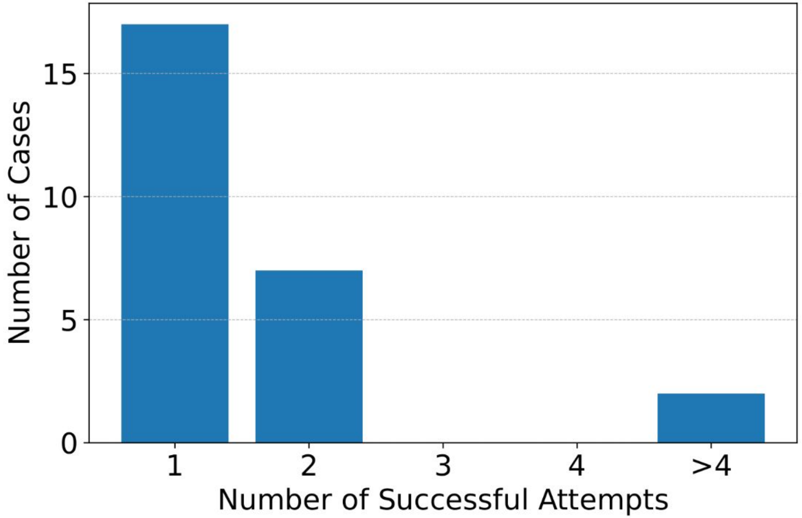
Case study 1: avl_tree

- A C implementation of AVL tree
- SACTOR can obtain complete unidiomatic Rust translation
- Failed to get idiomatic translation
  - function pointer not supported



Case study 2: URL parser

- SACTOR can obtain complete unidiomatic Rust translation
- Obainted 10/23 total idiomatic functions translation



Conclusion & Future work

- SACTOR:
  - Static-analysis + two-phase prompting → 78–93 % correct, better idiomaticity
- **Key takeaways:**
  - External analysis → better capability
  - Re-using test suites via FFI → better adaptability
  - Two phase translation: Decouples syntax vs semantics → extra flexibility
- **Next**:
  - Support richer code features
  - Improve e2e test coverage
  - Cost-efficient prompting under test-time scaling
  - Broader evaluation

Tianyang Zhou, tz64@illinois.edu/qsdrqs@gmail.com, Website: qsdrqs.github.io