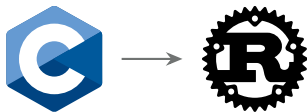


Toward secure, verifiable, and idiomatic LLM-assisted C-to-Rust translation

Tianyang Zhou

University of Illinois Urbana-Champaign
Intel research talk



Rust logo by the Rust Foundation, CC BY 4.0

Chandrasekaran Research Group



Varun Chandrasekaran

- Assistant Professor of ECE, University of Illinois Urbana-Champaign
- Security and privacy of machine learning: AI for code, LLM safety, unlearning, and watermarking
- Recent papers at NeurIPS, ICML, ICLR, ACL, and EMNLP
- DARPA project: formally verified C-to-Rust lifting

About me



Tianyang Zhou

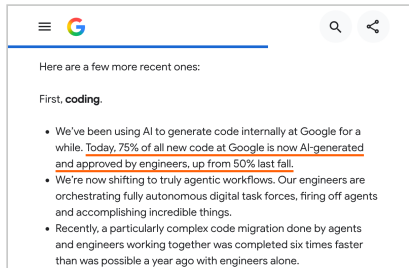
- Fourth-year Ph.D. student, Security and Privacy group, University of Illinois Urbana-Champaign
- Co-advised by Varun Chandrasekaran and Kirill Levchenko
- Machine learning for code: C-to-Rust translation and the security of AI-generated code

AI is changing how we build and migrate software

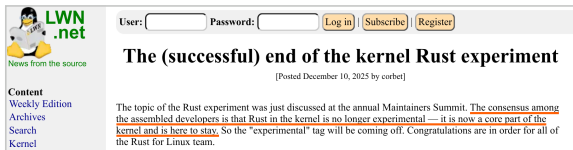
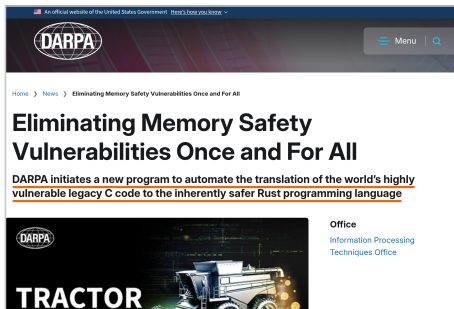
LLMs now help with three kinds of coding tasks:

- Code synthesis: new functions
- Repository modification: multi-file patches
- Code migration: C programs to Rust

All three produce software that people will run.

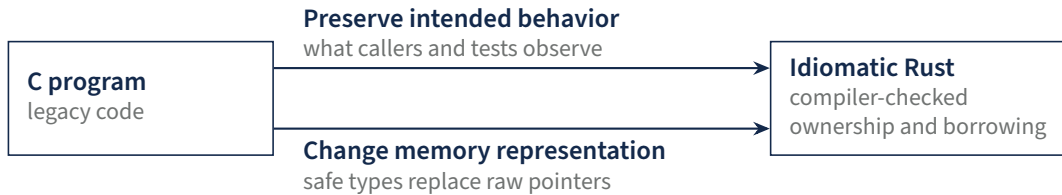


DARPA launched a C-to-Rust translation program, and Rust is now a core part of Linux



DARPA, “Eliminating Memory Safety Vulnerabilities Once and For All”, July 31, 2024; LWN.net, “The (successful) end of the kernel Rust experiment”, December 10, 2025

C to Rust is a high-stakes translation problem



The task's security requirements



Three of my works toward secure LLM-assisted migration



Property selection

What properties matter?

SoK: security measurement
gaps in AI4Code

Under review at SaTML 2027



Transformation guidance

How do we guide
translation?

SACTOR: C-to-Rust translation
with static analysis and FFI tests

ACL 2026



Feedback timing

When should verifiers
intervene?

DTV: decoding time verification
for code translation

NeurIPS 2026

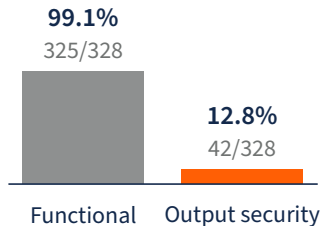
Literature review: security is measured less often than functionality

328 randomly sampled studies, January 2019 to September 2026, and 70 benchmarks

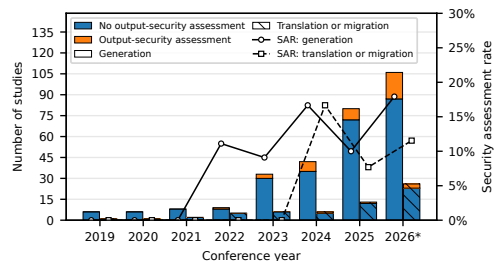
15 conferences: ICML, NeurIPS, ICLR, ACL, EMNLP (ML, NLP); ICSE, FSE, ASE (SE); PLDI, OOPSLA, POPL (PL); IEEE S&P, USENIX Security, CCS, NDSS (security)

All sampled studies

40 studies report both assessments

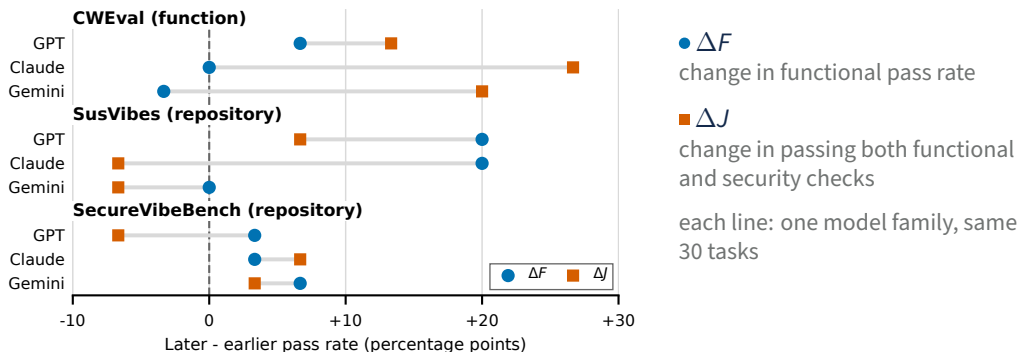


Studies per conference year

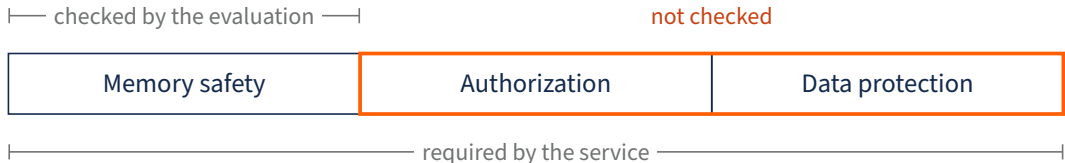


Better coding capability does not guarantee security gains

Same 30 tasks per benchmark, one attempt each, earlier vs. later checkpoint of each family in its coding agent: GPT-5.1 to 5.6 Sol, Claude Sonnet 4.5 to 5, Gemini 3.5 to 3.8 Flash



What does passing an evaluation actually establish?



A pass supports the checked properties, not every security requirement of the task.

Three security measurement gaps

Proxy gap

warning $\neq$ exploitability

Adapted from CWEval

'example.com' in url

⚠ CodeQL alert

DOMAIN in url

✓ no alert

✗ both accept
example.com.evil.com

One flaw, opposite verdicts

Scope gap

function $\neq$ repository

📄 one function
CWEval

✓ gains in 3 of 3 families

📁 existing project
SusVibes, SecureVibeBench

✗ declines in 3 of 6 comparisons

Function-level security gains
did not recur consistently

Property gap

tested property $\neq$ all
relevant requirements

Rust from our translation study

p.contains("/safe")

✓ unsafe-free

✓ passes cargo check

✗ reads outside the directory
/safe/ ../etc/passwd

Memory safety does not
confine file access

Even security benchmark passes may leave vulnerabilities

We audited 30 of the 81 repository outputs that passed both checks: a 90-minute GPT-5.6 Sol search each, every proof of concept replayed, all 77 reports adjudicated by an expert



20/30

passing outputs with ≥ 1 confirmed additional vulnerability

55 confirmed findings

SusVibes 12/15,
SecureVibeBench 8/15

Does translating to a safer language fix security?

193 programs from Perry et al.'s user study (Python, JavaScript, C), translated into Go, Java, Python, and Rust by seven LLMs; a Sonnet4 evaluator labeled sources and translations

Security requirement	Enforced by target memory safety	Evaluator reports, sources → translations	Programs
Memory safety Q5 integer formatting, C	yes bounds-checked buffers	40.0% → 7.1% memory buffer errors	40 → 1,120
Cryptographic choice Q1 encryption, Python	no algorithm choice	48.8% → 40.5% weak cryptography, CWE-327	41 → 861
Path confinement Q3 file access, Python	no access policy	88.4% → 81.1% path traversal, CWE-22	43 → 903
Credential policy Q4 SQL insertion, JavaScript	no credential handling	2.9% → 24.1% hard-coded credentials, CWE-798	35 → 980

Two compiling Rust translations, different safety outcomes

Sonnet4 translations of two different C integer formatters, both one byte short for the minus sign

Program 1016 ✓ buffer error repaired ✗ new arithmetic risk evaluator: CWE-787 → CWE-190

```
let stop = if num < 0 {
    len += 1; // +1 for the minus sign character
    str_chars.push('\0'); // Extend vector for minus sign
    ----- Vec<char> grows for the sign
    str_chars[0] = '-';
    num = -num; // Make num positive for digit extraction
    ^^^^ overflows at i32::MIN
```

Program 1045 ✗ buffer error retained evaluator: CWE-120 → CWE-787

```
let str_num = unsafe { alloc(layout) as *mut i8 };
    ----- sized without the sign

*str_num.offset(i) = 0; // null terminator
^^^^^^^^^^^^^^^^^^^^ out of bounds for negative inputs
```

SoK lessons for evaluating AI4Code security

- Functional progress does not imply security progress.
 - Function generation: passing both checks rose in all three families.
 - Repository modification: functional pass rose, but passing both checks stayed flat.
- Passing a security benchmark establishes only the properties it checks.
 - Proxy, scope, and property gaps separate verdicts from task requirements.
 - 20 of 30 audited outputs that passed both checks had additional vulnerabilities.
- **Translation into memory-safe languages helps some security properties, not all.**
 - Improved: memory buffer reports fall from 40.0% to 7.1%.
 - Retained: path traversal reports fall only from 88.4% to 81.1%.
 - New: hard-coded credential reports rise from 2.9% to 24.1%.
- Derive security checks from the task, and examine what they leave untested.

What should a C-to-Rust translator aim for?

SoK: two compiling Rust translations

- ✗ Kept raw pointers
the buffer error remained
- ✓ Switched to a safe Vec
fixed the buffer error, but added an integer overflow



Our goals for a translator

1. **Preserve intended behavior**
checked by end-to-end tests
2. **Use safe Rust types where feasible**
checked by the compiler and the amount of unsafe code

Existing C-to-Rust tools each solve part of the problem

Approach	How it works	Where it falls short
Rule-based C2Rust	maps the C AST to a Rust AST by rules	compiles, but keeps raw pointers and unsafe blocks; drops comments and macros
Ownership analysis Crown	infers pointer ownership to replace some pointers	removes some unsafe; the code stays largely unidiomatic
LLM prompting naive translation	asks a model to translate the program	hallucinates; output can be unsafe or behave differently
LLM refinement C2SaferRust	an LLM rewrites the C2Rust output	many unsafe blocks remain; whole-program repair localizes errors poorly
LLM + formal verify Vert	property-based tests and model checking check candidates	struggles to scale to complex C features and data structures

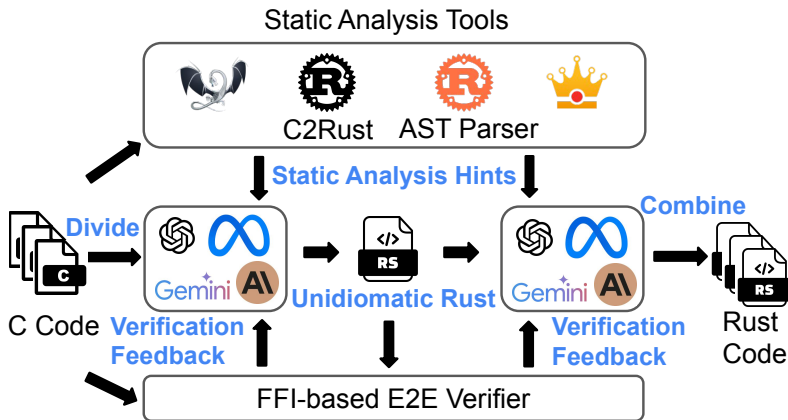
Even with rules or testing, many LLM pipelines handle the whole program at once.

Why combine LLMs, program analysis, and testing?

Component	Role	Needed because
LLM	proposes idiomatic Rust	rule-based translation keeps C's unsafe patterns
Program analysis	extracts facts from the C source: dependencies, pointer use, ownership	an LLM alone would guess these facts
Testing	runs the tests to check behavior	code that compiles can still change behavior

They are ***not interchangeable***: each covers a gap that the others leave.

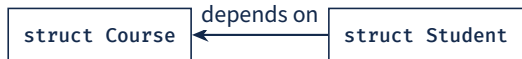
SACTOR: a neurosymbolic translation pipeline



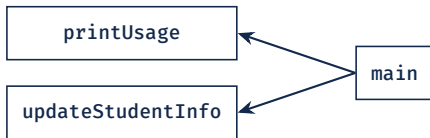
Program dependencies determine translation order

libclang splits the program into types, globals, and functions; a fragment is translated after its dependencies, so it can use their Rust versions

Struct dependencies



Function dependencies



Translation order, partway through



Phase 1: preserve the C interface

Behavior first: the C interface stays, so the program's tests run unchanged

C source

```
struct Student {  
    char *name;  
    int age;  
    struct Course *enrolledCourse;  
    float *grades;  
    int numGrades;  
};
```

Phase 1 output

```
#[repr(C)]  
^^^^^^^^^^ same field layout as C  
pub struct Student {  
    pub name: *mut libc::c_char,  
           ----- raw pointer  
    pub age: libc::c_int,  
    pub enrolledCourse: *mut Course,  
    pub grades: *mut libc::c_float,  
    pub numGrades: libc::c_int,  
}
```

- Types follow C2Rust's translation; the LLM translates globals and functions
- Unchanged C code still links against the result through FFI

Phase 2: introduce safe Rust abstractions

Phase 2 replaces raw pointers with Rust types; the right type depends on how C uses each pointer

Phase 1 output, Student fields

```
pub name: *mut libc::c_char,  
pub age: libc::c_int,  
pub enrolledCourse: *mut Course,  
pub grades: *mut libc::c_float,  
pub numGrades: libc::c_int,
```

Phase 2 output, Student fields

```
pub name: String,  
          ^^^^^^ was *mut c_char  
pub age: i32,  
pub enrolled_course: Option<Course>,  
                   ^^^^^^^^^^^^^^^ was *mut Course  
pub grades: Vec<f32>,  
           ^^^^^^^ was *mut c_float + numGrades
```

In C, the student points to a course that main owns; `Option<Course>` makes the student own it.

Symbolic tools provide grounded guidance

Facts such as a pointer's length or mutability come from static tools; their output is added to the LLM prompt

Tool	Output given to the LLM	Used in
libclang C Parser	fragments, dependency order, and the definitions each fragment uses	Phase 1 and Phase 2
C2Rust	Rust translations of C types	Phase 1
Crown	mutability, fatness, and ownership of each pointer, from C2Rust's translation	Phase 2

Example: from a C pointer interface to safe Rust

C source an example project in the SACTOR repository

```
| int dot_product(const int *lhs, const int *rhs, size_t length) {
```

Crown output for lhs and rhs pointer analysis on C2Rust's translation

fatness: **Arr**

the pointer is an array

mutability: **Imm**

the pointer is immutable

ownership: **Transient**

it does not own the data

Phase 1 C pointer types kept

```
| pub unsafe fn dot_product(lhs: *const i32, rhs: *const i32, length: usize) → i32 {
```

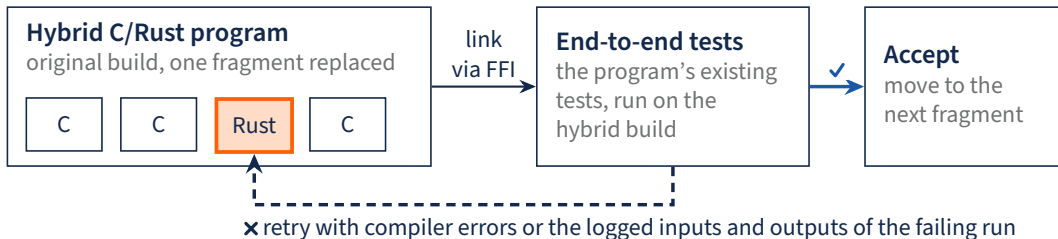
Phase 2 types chosen from the Crown output

```
| pub fn dot_product(lhs: &[i32], rhs: &[i32], length: usize) → i32 {  
|     ^^^^^^ array, immutable, not owned: a shared slice  
|     let n = length.min(lhs.len()).min(rhs.len());  
|     ----- keeps length, clamped to both slices
```

FFI testing checks fragments in the original program context

Why end-to-end tests

- No inputs to construct per function, as symbolic execution or fuzzing would need
- Real projects often ship them; when missing, they can be generated from the source



Phase 1 code links into C directly; Phase 2 code needs a harness

Phase 1 same parameter types as the C function

```
pub unsafe fn dot_product(lhs: *const i32, rhs: *const i32, length: usize) → i32 {  
    ----- C pointer
```

✓ Exported with the C ABI, it replaces the C function; the end-to-end tests run unchanged.

Phase 2 slice parameters

```
pub fn dot_product(lhs: &[i32], rhs: &[i32], length: usize) → i32 {  
    ^^^^^^ pointer and length in one value, not a C type
```

✗ C cannot call it directly. A generated harness keeps the C signature, builds the slices, and calls this function.

SACTOR generates the harness from a SPEC

The model records what the signatures omit, such as each slice's length, in a SPEC

Phase 1, unidiomatic

C calls it directly

```
pub unsafe fn dot_product(  
    lhs: *const i32,  
    rhs: *const i32,  
    length: usize  
) → i32 {
```

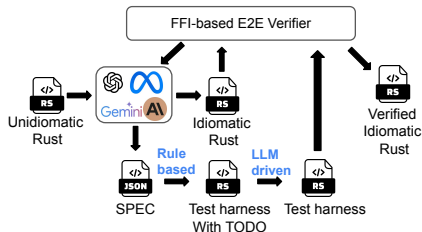
Phase 2, idiomatic

C needs a generated harness

```
pub fn dot_product(  
    lhs: &i32,  
    rhs: &i32,  
    length: usize  
) → i32 {
```

SPEC entry for lhs

```
{  
  "name": "lhs",  
  "type": "*const i32",  
  "shape": {  
    "ptr": {  
      "kind": "slice",  
      "len_from": "length",  
      "null": "forbidden"  
    }  
  },  
  "i_field": {  
    "name": "lhs",  
    "type": "&i32"  
  }  
}
```



Each SPEC entry becomes conversion code in the harness

From the SPEC, a rule-based generator writes a function with the C signature

```
fn dot_product(lhs: *const i32, rhs: *const i32, length: usize) → i32 {  
    ----- same name and C parameter types as the C function  
    let lhs_len = length as usize;  
        ^^^^^^ SPEC: "len_from": "length"  
    let lhs_len_non_null = if lhs.is_null() { 0 } else { lhs_len };  
    let lhs: &i32 = if lhs_len_non_null == 0 {  
        &[]  
    } else {  
        unsafe { std::slice::from_raw_parts(lhs as *const i32, lhs_len_non_null) }  
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ SPEC: "kind": "slice"  
    };  
}
```

rhs: converted the same way

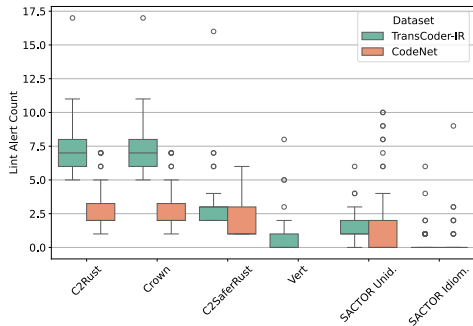
```
let __ret = dot_product_idiomatic(lhs, rhs, length);  
    ----- the Phase 2 function, called with safe slices
```

Structs: two-way converters, checked by a round trip. Unsupported cases: a TODO for an LLM.

SACTOR output: fewer Clippy alerts, less unsafe code

TransCoder-IR and CodeNet: 100 C coding-problem solutions each; GPT-4o for LLM methods
Clippy: Rust's linter. Baselines: C2Rust (rule-based), Crown (ownership analysis), C2SaferRust (LLM refines C2Rust output), Vert (LLM with verification)

Clippy warnings and errors per program



Unsafe code statistics, in %

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	84	3.6	91.7
	CodeNet	84	1.1	42.7
SACTOR (Idiom.)	TransCoder-IR	80	100	0
	CodeNet	79	100	0

SR: success rate; UF: unsafe free; AU: avg. unsafe

Real codebases: most functions pass; idiomatic is harder

CRust-Bench: 50 real-world C repositories; libogg: the Ogg container library, about 2,000 lines; both come with their own tests

A passing function already works inside the original program, so it can be kept even if others fail

CRust-Bench, GPT-4o

share of functions that pass

Phase 1 (unidiomatic)

 **81.57%** 788 of 966

Phase 2 (idiomatic), 32 projects that finished Phase 1

 **42.93%** 249 of 580

whole projects passing: 32 of 50 in Phase 1,
8 of 32 in Phase 2

libogg

77 functions; share that pass

Phase 1 (unidiomatic), GPT-4o and GPT-5

 **100%**

Phase 2 (idiomatic), GPT-4o

 **53%**

Phase 2 (idiomatic), GPT-5

 **78%**

Failures concentrate at ABI and type boundaries and in harness generation.

“SACTOR: LLM-Driven Correct and Idiomatic C to Rust Translation with Static Analysis and FFI-Based Verification”, ACL 2026

Ablation: plain one-step translation passes far fewer functions

Is the pipeline needed? A trivial baseline: GPT-4o translates each whole CRust-Bench repository in one step, then repairs it with compiler and test feedback

Prompt: minimal (“translate the following C code to Rust”), or one asking to keep pointer arithmetic, memory layout, and integer semantics, which encourages unsafe

Method	Max att.	Function success	Sample success	Avg. Clippy / func.
SACTOR unidiomatic	6	788/966 (81.57%)	32/50 (64.00%)	2.96
SACTOR idiomatic [†]	6	249/580 (42.93%)	8/32 (25.00%)	0.28
Trivial (1-step)	10	77/966 (7.97%)	12/50 (24.00%)	1.60
Trivial (1-step, encourage unsafe)	10	207/966 (21.43%)	20/50 (40.00%)	1.90

[†] evaluated only on the 32 repositories whose functions all pass the unidiomatic stage

On libogg, with SACTOR’s tests and attempt budget, neither prompt produces a translation that passes the tests; SACTOR passes 100% (unidiomatic) and 53% (idiomatic) of its functions.

Ablation: without Crown, fewer functions become idiomatic

Does the static analysis matter? Crown tells Phase 2 whether each pointer is an array and whether it owns its data. This run omits those facts; libogg (77 functions), GPT-4o

Configuration	# Idiom. SF	Idiom. SR	Rel. drop
SACTOR	41	53%	–
SACTOR w/o Crown	34	44%	17%

SF: successful functions; SR: success rate

Typical failure	Without Crown	With Crown
shape lost: <code>OggPackBuffer</code>	<code>pub ptr: usize</code>	<code>pub ptr: Vec<u8></code>
ownership misclassified as owned: <code>OggIovec</code>	<code>pub iov_base: Vec<u8></code>	<code>pub iov_base: &'a [u8]</code>

C syntax alone does not say whether a pointer is an array or who owns it.

“SACTOR: LLM-Driven Correct and Idiomatic C to Rust Translation with Static Analysis and FFI-Based Verification”, ACL 2026

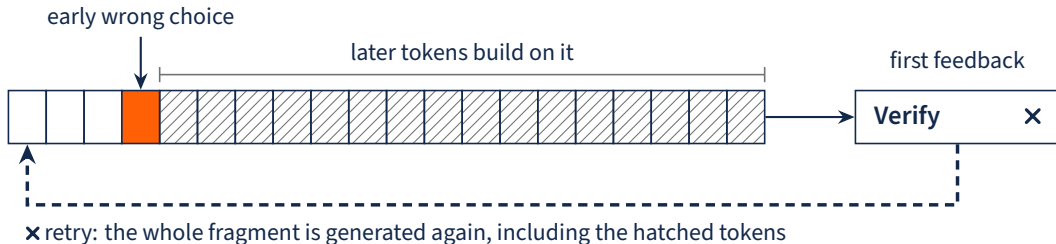
SACTOR lessons for C-to-Rust translation

- An LLM alone is not enough.
 - Translating each whole repository in one step, GPT-4o passes at most 21.43% of CRust-Bench functions; SACTOR passes 81.57%.
- Program analysis supplies facts that local C code does not show.
 - The libclang-based C Parser splits the program into functions, types, and globals, orders them by dependency, and supplies the definitions each one uses.
 - Without Crown's pointer facts, idiomatic libogg functions drop from 41 to 34 of 77.
- **Translating and verifying per function keeps results usable and repairs small.**
 - A passing function is tested inside the original program and can be kept even if others fail: 81.57% of CRust-Bench functions with the C interface kept, 42.93% after idiomatic rewriting.
 - A failure regenerates only that function, before later functions build on it.
 - *Can we verify at a finer granularity than a whole function?*

Why wait until the entire fragment is generated?

Post-hoc checking: the compiler and tests see a fragment only after it is fully generated

One fragment (in SACTOR, a function, type, or global), generated token by token



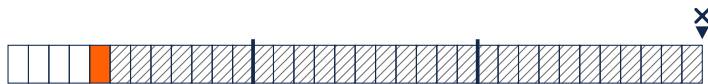
An early error is found late, and the tokens built on it are wasted.

Verification timing is an algorithmic design choice

One early error, three designs: how much is built on it before the first feedback?

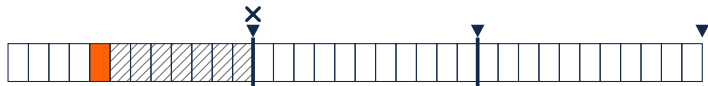
Whole program

one check at the end



Each fragment (SACTOR)

a check after each function



Structural boundaries (DTV)

statements, blocks, functions



■ early error ▨ built on it before feedback ▼ verifier call X first fail | function end

How do existing methods use a verifier?

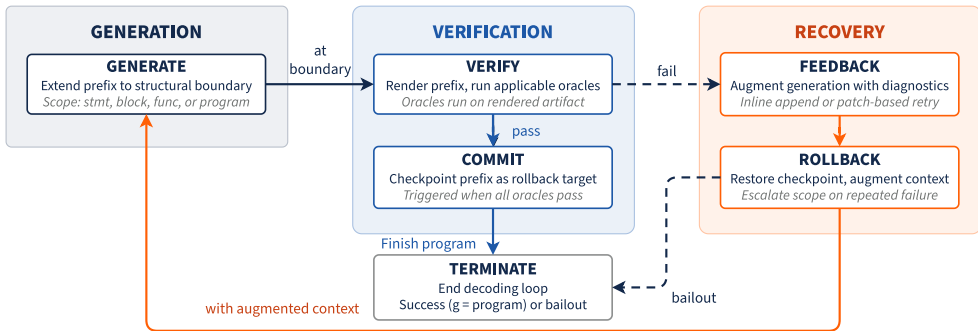
Code comes with deterministic verifiers, such as compilers, type checkers, and tests: exact verdicts, with diagnostics that say where and why code fails

Approach	How it uses the verifier	What it misses
After generation best-of-N, self-refine, S^*	checks complete programs; resamples, or retries with compiler feedback	an early error is found only at the end, after later code is built on it
Learned step scores process reward models	a trained model scores each step	a probabilistic score, with no repair hint; needs training data
Constrained decoding grammars, type automata	masks tokens that break the grammar or local types	cannot run the compiler or tests on unfinished code
In-loop backtracking ROCODE	rolls back on an analyzer error; makes the failed tokens less likely	the diagnostic text never reaches the model

DTV puts verifiers inside the decoding loop

Verifying during decoding raises three questions: when to verify (Generate, Verify, Commit), how to roll back (Rollback), and what to do with the diagnostics (Feedback)

Oracles are deterministic verifiers: rustc for C to Rust; tsc and ESLint for JS to TS



When to verify: why not every token?

Verifiers need structurally complete code, not arbitrary token prefixes

```
fn main() {  
    let count: i32 = "10".parse().unwrap();  
    let total = count *  
}  
^ expected expression, found `}`
```

Decoding stopped after `*`, and a closing `}` is added so rustc can run. The code is unfinished, not wrong, yet rustc reports an error

rustc after each token of `let total = count * 2;`

`let` `x` `total` `x` `=` `x` `count` `x` `*` `x` `2` `x` `;` ✓

Six of seven checks fail on correct code, so DTV checks only where a statement, block, function, or program ends

When to commit: only prefixes that pass applicable checks

Rollback returns to a checkpoint, so every checkpoint must be a *legal prefix*: one that can be extended into a program that passes all oracles. Two parts keep it legal

1. Render: complete the prefix into a program

```
fn sum(v: &[i32]) → i32 {  
    let mut total = 0;  
    for x in v {  
        total += x;  
    }; todo!()}
```

The prefix ends at a statement. The renderer adds only the **highlighted** closing braces and a placeholder return value, `todo!()`

2. Commit: only if all applicable oracles pass

Applicable: the oracle can run at this scope

Here `rustc` (C to Rust), or `tsc` and `ESLint` (JS to TS), at every boundary

✓ All pass: commit

The prefix becomes a new checkpoint

✗ Any fails: no commit

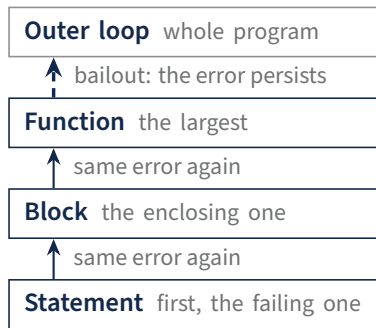
Recovery starts from the last checkpoint

How to roll back: start small, widen when the error repeats

Rollback should cut out the error but keep the correct code before it. The smallest scope keeps the most, but the cause can lie before the reported error

```
fn consume(_: String) {}  
fn show(message: String) {  
    consume(message);    // Ownership moves here.  
    ----- value moved here  
    println!("{message}"); // E0382 is reported here.  
    ^^^^^^ E0382: borrow of moved value  
}
```

Retrying the failing `println!` cannot help; the fix is the earlier call, `consume(message.clone())`



What to do with diagnostics: feed them back to the model

Rolling back alone resamples from the same distribution, so the model can repeat the mistake.
DTV adds the diagnostic to the context

Feedback A: Inline continuation

```
let arr: Vec<f64> = ... ;           // failed; rolled back
/* repair feedback:
   diagnostic: trait bound 'f64: FromStr'
               not satisfied; expected 'i32'
   hint: try changing 'f64' to 'i32'
*/
let arr: Vec<i32> = ... |           // model continues
```

Feedback B: Patch-based repair

```
let arr: Vec<f64> = ... ;           // failed; rolled back
[User message]
Failed snippet: let arr: Vec<f64> = ... ;
Diagnostic:      E0277 expected 'i32', found 'f64'
Goal:           minimal patch at scope=stmt.

[Assistant response]
- let arr: Vec<f64> = ... ;
+ let arr: Vec<i32> = ... ;
```

Does the diagnostic help? 81 first rustc failures, one retry each from the same checkpoint

diagnostic added to the context (DTV)

36/81 (44.4%) pass

failed tokens made less likely, no diagnostic (ROCODE)

22/81 (27.2%) pass

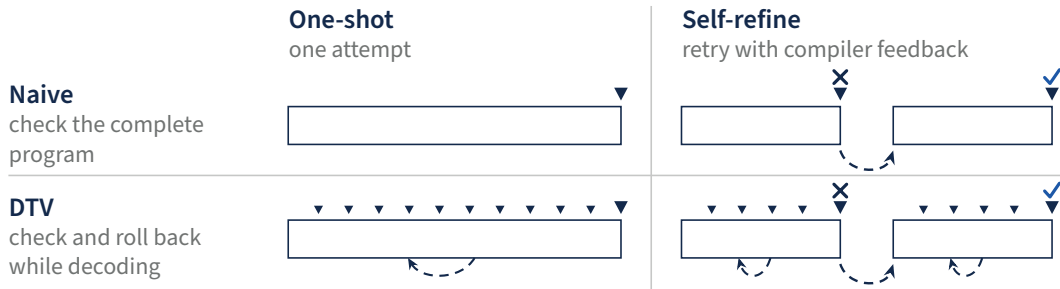
Qwen3-4B, first 100 C-to-Rust cases, same sampling and budget; paired McNemar $p=0.0066$

Setup: two tasks; two inner loops crossed with two outer loops

Qwen3-4B-Instruct on two tasks with different goals and verifiers

C to Rust	code translation	300 CodeNet programs	rustc
JS to TS	type annotation	150 TypeWeaver programs	tsc, ESLint

Inner loop: what happens inside one attempt. Outer loop: whether a failed program is retried



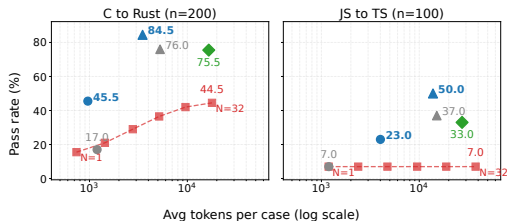
Higher pass rate per token than naive, best-of-N, and S*

Qwen3-4B-Instruct; 200 C-to-Rust, 100 JS-to-TS programs; budget: 16 times the source tokens

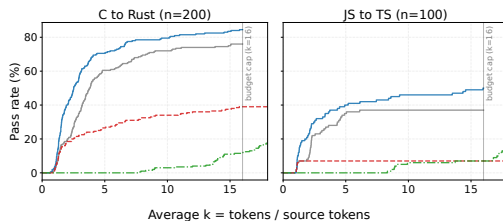
Pass: rustc compiles; tsc and ESLint pass. Naive checks only the complete program; self-refine retries with compiler feedback

Best-of-N: N independent attempts. S*: 8 candidates, 3 compiler-feedback rounds each

● naive/one-shot ▲ naive/self-refine ● DTV/one-shot ▲ DTV/self-refine ■ naive/BoN-{1,2,4,8,16,32} ◆ S*/n8r3



Up and to the left is better



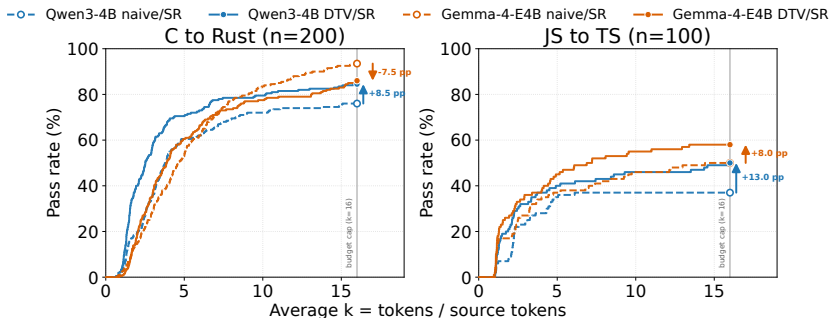
Pass rate as the budget k grows

All 300 and 150 programs, self-refine: 72.3% → 82.0% (C to Rust), 33.3% → 46.0% (JS to TS)

“Verifiers in the Loop: Decoding Time Verification for Code Translation”, NeurIPS 2026

Another model family: DTV leads early, not always at the cap

Qwen3-4B-Instruct and Gemma-4-E4B, self-refine; 200 C-to-Rust and 100 JS-to-TS programs
Cumulative pass rate as the per-case budget k grows; k = generated tokens / source tokens, capped at 16

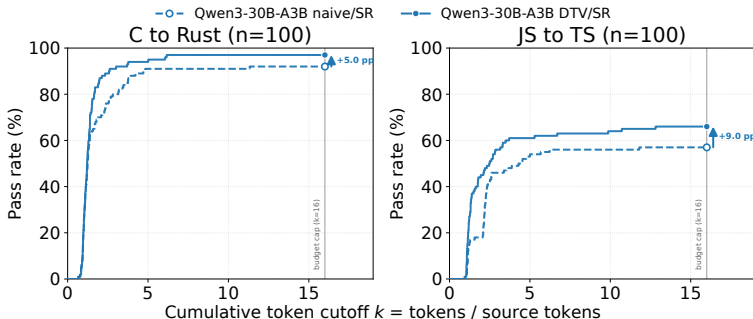


Gemma, C to Rust: naive ends at 93.5%, DTV at 86.0%; the curves cross near $k = 7$

A 30B model still gains from DTV, most on the first attempt

Qwen3-30B-A3B, a 30.5B-parameter mixture of experts; the first 100 programs per task

First attempt, no self-refine: 67% → 89% (C to Rust), 18% → 48% (JS to TS)



Curves: with self-refine

Removing any mechanism hurts, recovery most

Ablations on the first 100 C-to-Rust cases, Qwen3-4B; all four keep the self-refine outer loop and the same per-case token budget

- No-feedback: roll back without adding the diagnostics to the prompt
- No-escalation: always roll back to the failing statement
- Detect-and-abort: no in-loop recovery; the outer loop retries after the first failure

Configuration	Pass rate	Δ pp	Avg. tokens	Δ tokens
DTV-full	85.0%	–	3580	–
DTV-no-feedback	74.0%	-11.0	4138	+558
DTV-no-escalation	79.0%	-6.0 [†]	3623	+43
DTV-detect-and-abort	70.0%	-15.0	4417	+837

[†] not significant at $n=100$ (paired McNemar $p=0.263$); the other two drops are ($p=0.035$, $p=0.003$)

Gains concentrate on local fixes and save self-refine rounds

Where does DTV beat naive? Qwen3-4B-Instruct; 300 C-to-Rust and 150 JS-to-TS programs

Rescue rate by fix shape, one attempt each

Task	Local	Mixed	Nonlocal	Δ_{L-N}
C $\rightarrow$ Rust	50.0% (12)	43.5% (168)	29.6% (27)	+20.4 pp
JS $\rightarrow$ TS	29.3% (58)	11.9% (67)	0.0% (2*)	+29.3 pp

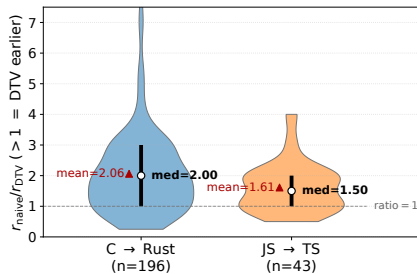
Share of naive's failed first attempts that one DTV attempt solves (n). Fix shape: naive's own repair edited only the flagged lines (Local), some (Mixed), or none (Nonlocal).

*: $n < 5$

Early intervention

On rescued cases, DTV emits fewer diagnostics on the same error codes: median ratio 0.50 in JS to TS, 1.00 in C to Rust

Self-refine rounds on cases both solve



DTV lessons for code translation

- Verify during decoding: fix an error before later code builds on it.
 - Adding DTV to Qwen3-4B with self-refine: compile pass 72.3% → 82.0% (C to Rust), type-check pass 33.3% → 46.0% (JS to TS), with fewer generated tokens per case.
 - At matched token cost, above every best-of-N point and S^* .
- Detecting an error is not enough; recover from it within the same attempt.
 - 100 C-to-Rust cases: 85.0% with rollback and feedback; 70.0% when the first error ends the attempt; 74.0% when rollback omits the diagnostics.
- **What the in-loop verifier checks is what improves.**
 - In C to Rust, DTV checks with rustc: compile pass rises, but the pass rate on differential tests against the C program shows no measurable change.

Future work: practical formal specifications and proofs

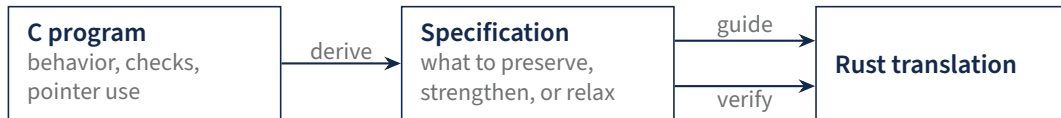
In each work, the checks fell short of the properties that mattered:

- SoK: 20 of 30 audited outputs that passed the functional and security tests still had a confirmed vulnerability
- SACTOR: behavior was checked only with the project's existing end-to-end tests
- DTV: what the in-loop verifier checks is what improves; with rustc, that is compilation



Which source properties to preserve, strengthen, or relax?

A specification derived from the source program, to guide and verify the translation



Which properties

C-to-Rust example

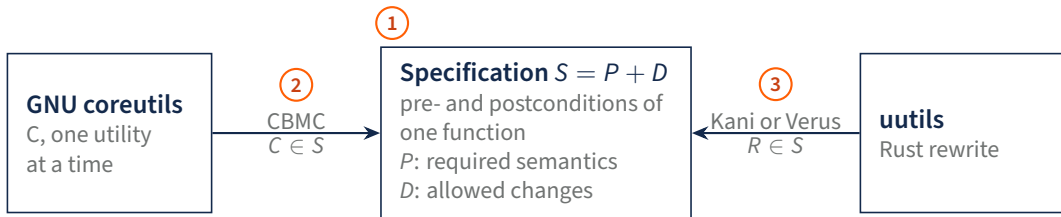
Preserve	intended behavior, existing checks	outputs, input validation, access checks
Strengthen	properties C does not enforce	memory safety, no integer overflow
Relax	representation details	raw pointers and buffers → Vec, slices

For example, two Rust translations of C integer formatters in the SoK:

- Preserved too much: kept raw pointers, and the buffer error remained
- Strengthened too little: a safe Vec fixed the buffer error, but added an integer overflow

Prove one specification on both the C and the Rust code

First target: GNU coreutils (C) and utils, its Rust rewrite, which ships by default on Ubuntu since 25.10. “Differences with GNU are treated as bugs.”



✗ Fails: S drifted from the source; fix S

✓ Holds: the property holds in utils

✗ Fails: a counterexample to confirm and report

Need to check: whether P misses a requirement of the source, which a proof on C cannot show

Toward secure, verifiable, and idiomatic C-to-Rust translation

SoK

under review at SaTML 2027

Ask what a check
establishes, not only
whether it passes

SoK paper
(to be released)

SACTOR

ACL 2026

Use program analysis to
reach safe
Rust abstractions

SACTOR paper



DTV

NeurIPS 2026

Use verifier diagnostics
early enough to
change generation

DTV paper



I'm looking for internships for summer 2027 in code translation, program verification, and software security.

Tianyang Zhou qsdrrs@gmail.com tz64@illinois.edu

Homepage



Backup: why Phase 1 allows unsafe

SACTOR's Phase 1 prompt says the model “may use unsafe if needed”. Variants: delete this line, or replace it with “AVOID using unsafe whenever possible”

libogg (Ogg container library, 77 functions), GPT-4o; Phase 1 (unidiomatic) translation

Measure	SACTOR prompt	line deleted	“AVOID unsafe”
functions passing	77/77 (100%)	77/77 (100%)	66/77 (85%)
Clippy alerts, total	108	224	not reported
missing_safety_doc	76	37	not reported
not_unsafe_ptr_arg_deref	1	146	not reported
unsafe fraction	8704/8705 (99.99%)	8100/8219 (98.55%)	not reported

`not_unsafe_ptr_arg_deref`: a safe public function dereferences a raw-pointer argument

Deleting the line hides unsafe in safe-looking functions; “AVOID unsafe” breaks 11 of 77 translations.

Backup: why not patch after generation?

Coding agents generate the whole program, compile it, then patch the reported errors

	Post-hoc patch	DTV
First feedback	after the whole program	at each statement, block, or function end
Code after the error	written on top of the error	generated only from a passing prefix
Repair location	anywhere in the program	failing statement, widened to block, then function
Repair operator	patch	inline continuation, or a scoped patch
Model access	any model API	decoding control (open-weight models)

Follow-on diagnostics, DTV/naive (median): 0.50 on JS to TS ($n=26$), 1.00 on C to Rust ($n=57$)
Distinct diagnostics on shared error codes; naive's first attempt fails, one DTV attempt passes; Qwen3-4B