Translating C to Rust: Better, cheaper, faster

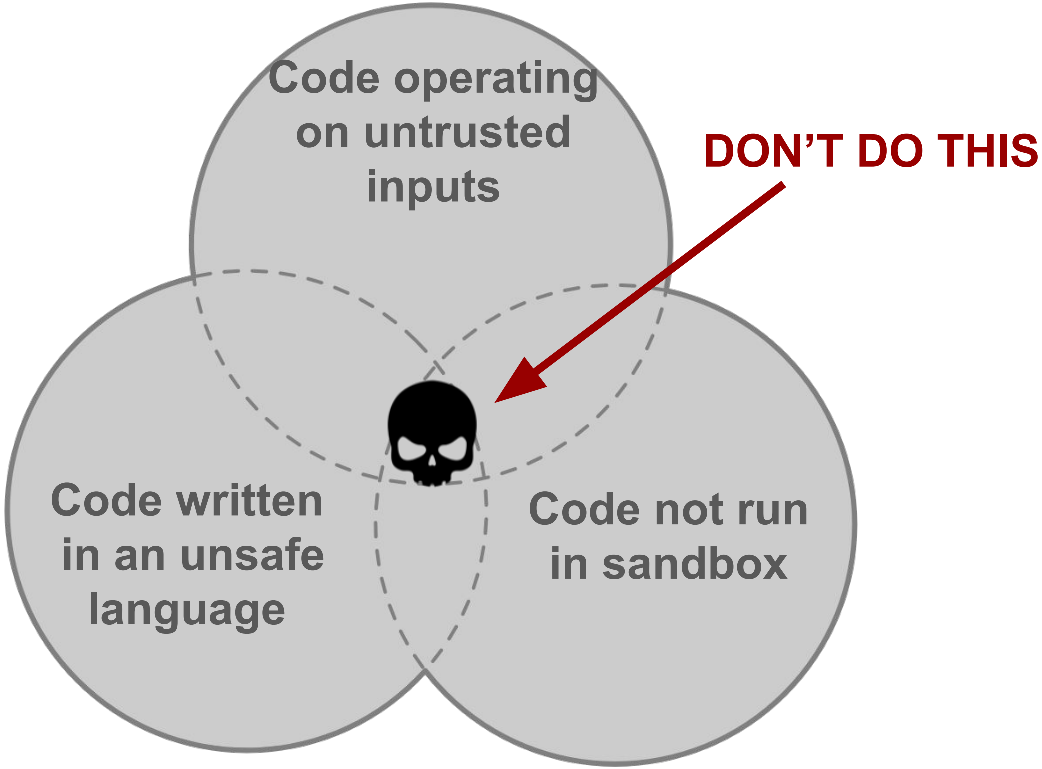
Per Larsen, Immunant, Inc.

HCSS: AI and Models in the Software Development Lifecycle

May 14, 2025, Annapolis, Maryland

immunant x galois

Chromium's Rule of Two


Code operating on untrusted inputs
DON'T DO THIS
Code written in an unsafe language
Code not run in sandbox

https://chromium.googlesource.com/chromium/src/+/master/docs/security/rule-of-2.md

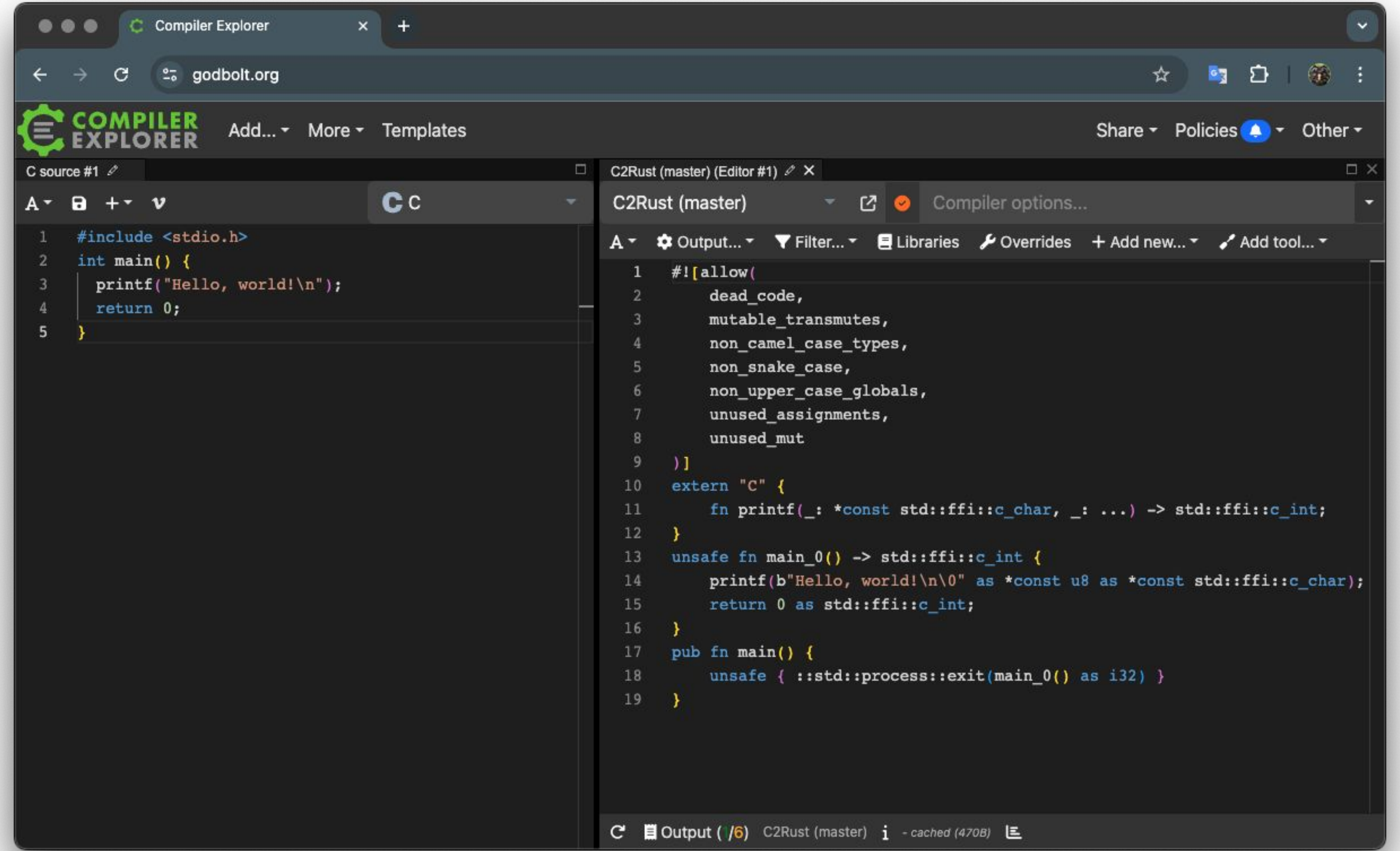
The Multi-Billion Line Problem - Why Bother?

- Legacy **C code is everywhere**:
  - Critical infrastructure, operating systems, embedded systems, web browsers, vehicles, etc.
  - Persistently plagued by memory vulnerabilities (buffer overflows, use-after-free, etc.)
- Rust offers safety by design w/o performance sacrifices
  - Rewriting by hand? Very expensive & slow. Not scalable.
- How to migrate critical systems efficiently and safely?

Our Starting Point - `c2rust`

- Core capability (`c2rust-transpile`):
  - Automatically translate C to unsafe, C-like Rust
  - Structurally equivalent, functional parity
- Real-world uses:
  - `serde_yaml` wraps `c2rust-transpiled` code
  - `rav1d` port of high-performance `dav1d` AV1 decoder
- Inspired 10+ academic papers
- Only the first step:
  - Have: unsafe, unidiomatic starting point. All tests pass.
  - Want: safe, idiomatic Rust largely free of unsafe blocks.

https://crates.io/crates/serde_yaml
https://github.com/memorysafety/rav1d



```c
#include <stdio.h>
int main() {
  printf("Hello, world!\n");
  return 0;
}
```

```rust
#![allow(
    dead_code,
    mutable_transmutes,
    non_camel_case_types,
    non_snake_case,
    non_upper_case_globals,
    unused_assignments,
    unused_mut
)]
extern "C" {
    fn printf(_: *const std::ffi::c_char, _: ...) -> std::ffi::c_int;
}
unsafe fn main_0() -> std::ffi::c_int {
    printf(b"Hello, world!\n\0" as *const u8 as *const std::ffi::c_char);
    return 0 as std::ffi::c_int;
}
pub fn main() {
    unsafe { ::std::process::exit(main_0() as i32) }
}
```

Next Steps: From Unsafe to Safe Rust

- Goal: extend the `c2rust` pipeline to automate more
- Key technical challenges
  - C is implicit where Rust is explicit
    - C pointers (`void*`, `char*`) hide a lot.
    - Rust is explicit about ownership, borrowing, lifetimes, nullability.
  - Idiomatic gaps:
    - C does not limit mutation and aliasing
    - Rust enforces the “aliasing XOR mutability” rule
    - (also…memory management, concurrency, macros, etc.)

Our pre-AI Approach: Analysis & Rewriting

- Strategy: Combine static & dynamic analysis to inform automated code rewriting
- Static Analysis:
  - Infer ptr permissions (read, write, free), nullability, uniqueness (for `&mut T`), etc.
- Dynamic Analysis:
  - Instrument code to observe pointers at runtime (is `ptr` ever `NULL` in practice?)
- Code Rewriter:
  - Consumes analysis results and transforms raw pointers (`*mut T`) into safe Rust types (`&T`, `&mut T` , `Box<T>`).
  - Replaces unsafe `libc` calls (`malloc`, `memcpy`) with safe Rust equivalents

Analysis & Rewriting in Action: Promising but …

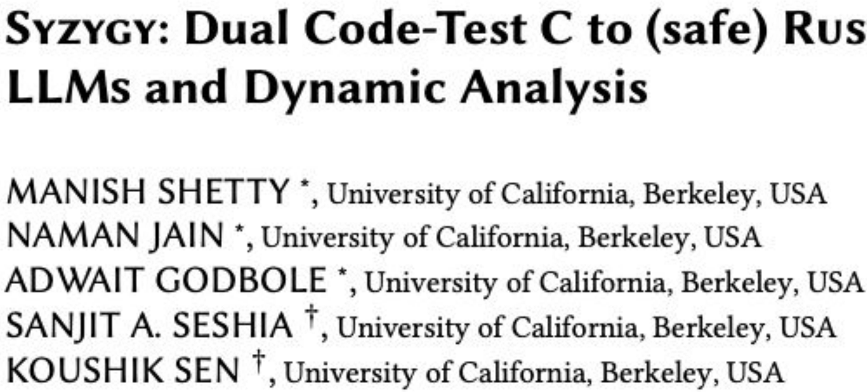
- Success:
  - `lighttpd algo_md5` module (fully safe),
  - `lighttpd buffer` module (partially safe)
- Sobering results:
  - Low conversion rate for large projects such as `lighttpd`
  - Symbolic/rule-based approaches (even with dynamic analysis hints) hit a complexity ceiling
  - Adding rules doesn’t scale for “long tail” of C idioms

Why AI is the Missing Piece

- The bottleneck: Traditional approach struggles with non-structural transformations, intent, and idioms.
- LLMs:
  - Excellent at pattern recognition, context, and code generation
  - Prone to “hallucinate” which can introduce subtle bugs
- Our experiments:
  - LLMs can do complex refactors (remove `c2rust` state machines)
  - Open AI o1 can do rewrites that take hours by hand
  - … not a magic bullet; verification & guidance still needed.

Why AI is the Missing Piece (cont’d) - Syzygy

- Observation: LLMs struggle to infer semantic information directly from source code
  - Idea: mine nullability, aliasing, sizes, types, etc. via dynamic analysis
  - Idea: generate tests to detect incorrect LLM translations



SYZYGY: Dual Code-Test C to (safe) RUST Translation using LLMs and Dynamic Analysis

MANISH SHETTY [*], University of California, Berkeley, USA
NAMAN JAIN [*], University of California, Berkeley, USA
ADWAIT GODBOLE [*], University of California, Berkeley, USA
SANJIT A. SESHIA [†], University of California, Berkeley, USA
KOUSHIK SEN [†], University of California, Berkeley, USA

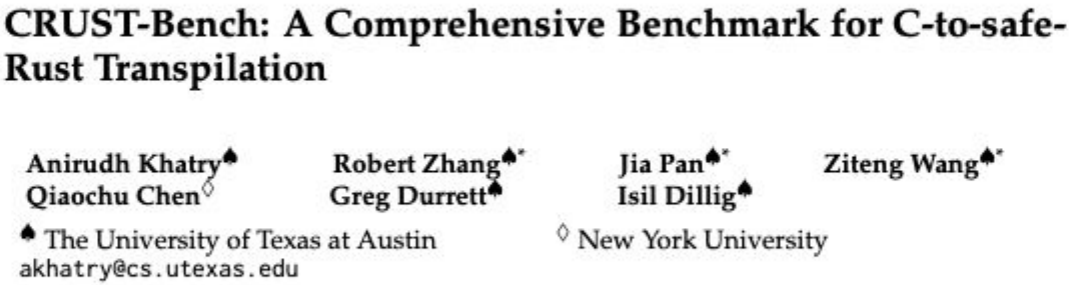
https://arxiv.org/pdf/2412.14234
https://syzygy-project.github.io/

Why AI is the Missing Piece (cont’d) - Syzygy

- Observation: LLMs struggle to infer semantic information directly from source code
  - Idea: mine nullability, aliasing, sizes, types, etc. via dynamic analysis
  - Idea: generate tests to detect incorrect LLM translations
- Results:
  - Translated Zopfli (a ~3000 LoC C compression library) to ~7000 LoC safe, test-validated Rust.
  - Inference took ~15 hours and cost ~$2500.
  - Rust code runs substantially slower than C (1.47-3.67x).

New Benchmarks Needed

- “You can’t improve what you can’t measure”
- CRUST-Bench
  - 100 C repos with manually-written safe Rust interfaces & tests
  - SOTA LLM (o1) solves 15% tasks w/o repair
  - SOTA LLM (o1) solves 37% tasks w/repair



CRUST-Bench: A Comprehensive Benchmark for C-to-safe-Rust Transpilation

Anirudh Khatry[♠] Robert Zhang[♠*] Jia Pan[♠*] Ziteng Wang[♠*]
Qiaochu Chen[◊] Greg Durrett[♠] Isil Dillig[♠]
[♠] The University of Texas at Austin [◊] New York University
akhatry@cs.utexas.edu

https://arxiv.org/abs/2504.15254

New Benchmarks Needed

- “You can’t improve what you can’t measure”
- CRUST-Bench
  - 100 C repos with manually-written safe Rust interfaces & tests
  - SOTA LLM (o1) solves 15% tasks w/o repair
  - SOTA LLM (o1) solves 37% tasks w/repair
- LLM code often fails to compile due to typing errors
  - “These errors suggest that models often struggle to reason precisely about lifetimes, mutability, and type compatibility”

The Vision: A Hybrid Approach is Key

- `c2rust` provides **baseline** to test rewrites (automatic or manual) against
- **symbolic analysis** surfaces knowledge about pointer usage, control flow, etc.
- **dynamic analysis** surfaces additional program properties at runtime
- **LLMs** suggests complex refactorings
  - to remove unsafety
  - to bridge semantic gaps
  - to make the code idiomatic
  - … using analysis results to guide the generated code
  - … subject to testing and formal verification to counter hallucinations
- rigorous **benchmarking** to measure progress and detect shortcomings

Conclusions

- We have to follow the rule of two; migration key part of the solution.
- Migration only feasible if we can increase efficiency by an order of magnitude.
- Program analysis and rewriting approaches show limited scalability
- **Hybrid approaches (LLM + program analysis) show a lot of promise!**

Thank you for listening!

Get in touch: perl@immunant.com & miked@galois.com
Code: github.com/immunant/c2rust
Try: c2rust.com & godbolt.org