CRAM: C++ to Rust Assisted Migration

High-Confidence Software and Systems

Nathaniel Berch, Paul Rodriguez, Thomas Wahl

May 9, 2023

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001123C0079. The views, opinions, and/or findings expressed in this document are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

The Story

C++: perhaps *the* language for efficient systems programming

- Inherits (from C) low-level memory manipulation capabilities
- Can cause memory-related crashes and vulnerabilities

Safer efficient languages do exist today: Rust!

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Intermezzo: Rust as the Language of Choice?

- **memory safety:** access via strict interface: *ownership*
- **efficiency:** no need for garbage collection (cmp. C#, Go, Java)
- **coolness factor:** modern language, supportive build system, active community

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

The Story (continued)

C++: perhaps *the* language for systems programming.

- inherits (from C) low-level memory manipulation capabilities
- can cause memory-related crashes and vulnerabilities

Safer efficient languages do exist today: Rust!

But what about legacy code??

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Lifting Legacy Code to Safer Languages (LiLaC-SL)

Goal: semi-automatic migration of legacy C/C++ code

Target: (your favorite) *safe* programming language

May: assume well-designed C/C++ code

Must: take advantage of target's idiomatic features

Must: deliver assurance of correctness and safety

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

The Story (continued)

C++: perhaps *the* language for systems programming.

- inherits (from C) low-level memory manipulation capabilities
- can cause memory-related crashes and vulnerabilities

Safer efficient languages do exist today: Rust!

But what about legacy code??

CRAM: semi-automated migration from C++ to Rust

GRAMMATECH

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Overview of CRAM

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Vision: From C++ to Rust

> *"Rust is like C++, except the [language semantics] forces programmers to do what they should be doing anyway."*
>
> -- Peter Aldous, GT

Implementing this vision:

Stage 1: *Refactor* the C++ program, to make it "Rust compliant":

- enforce ownership rules
- reduce mutability

Stage 2: Migrate the "safer" C++ to Rust, by lifting language idioms

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Technical Approach

"Migrating C++ to Rust from the inside out:"

Legacy C++

- ✓ efficient
- ⚠ potentially unsafe (crashes, exploits)

Refactoring →

Refactored C++

- ✓ efficient
- ✓ safer, hardened
- ⚠ still unsafe for future development

Migration →

Generated Rust

- ✓ efficient
- ✓ provably safe
- ✓ idiomatic, human-maintainable
- ✓ "modern"

GRAMMATECH

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Does the 2-Stage approach work wonders?

Nah, cross-language migration is hard even for “similar” languages.

→ assume source *is well-designed*:

- ~~pointer-implemented data structures~~ (use STL)
- ~~low-level pointer manipulation, p-arithmetic~~
- ~~abuse of overloading, obscure code~~ (e.g., `operator=`)

→ afford partial user assistance

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

CRAM-Style Refactoring and Migration

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Stage-1 Example: `Const` Hardening

C++: variables are by default *mutable* (assignable).

Rust: `let mut d: f64 = 0.0`

In C++, declare vars as `const` whenever possible:

- ✓ prevents some programming errors
- ✓ facilitates aliasing

Refactoring

```
      double segdist = p1->Distance(*p2);
```

```
const double segdist = p1->Distance(*p2);
```

GRAMMATECH

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Stage-1 Example: Breaking Up Alias Nests

Refactoring multiple non-const references to same memory cell:

```
void multiple_borrows() {

  Point p;
  Point* p0 = &p;
  Point& p1 = *p0;

  f(p1);
  g(p);
}
```

```
void single_borrow() {

  Point p;
  Point* p0 = &p;
  // Point& p1 = *p0;

  f(*p0);
  g(*p0);
}
```

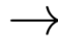
General principle: "safer in C++, ***required*** in Rust."

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Stage 2: Rust Code Generation

Vision: after refactoring, migration mostly involves porting idiomatic language constructs:

```
switch (v) {
  case a:   <code_a>; break;
  case b:   <code_b>; break;
  default: <code_cde>; }
```



```
match v {
      a => { <code_a>   },
      b => { <code_b>   },
      _ => { <code_cde> } }
```

GRAMMATECH

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Stage-2 Example: Container Traversal

```
for (std::vector<Point>::iterator
        p1 = pts.begin(), p2 = std::next(pts.begin());
        p2 != pts.end(); ++p1, ++p2)       { ... }
```

What defines a *container traversal idiom*?

1. **direction:** left-to-right vs. right-to-left traversal
2. **mutation:** destructive or non-destructive
3. **indexing:** 1, 2, 3 iterators

finite abstract "idiom space"

CRAM:
1. *abstract* traversal to idiom level, using static analysis + user
2. *retarget* to idiomatic Rust (using Rust idiom library)
3. *complete* construct by recursively calling CRAM on loop body

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Current Results

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Overview

Project duration so far: ≈ 12 months

Experimental use case: Valhalla routing library[1]

1. Valhalla coverage: ≈33% (growing)
2. Degree of automation: high
   → user assistance for disambiguation (multiple-choice)
3. Performance: (see next)
4. Code usability: very readable and idiomatic Rust (see next)
   → thanks to idiom library

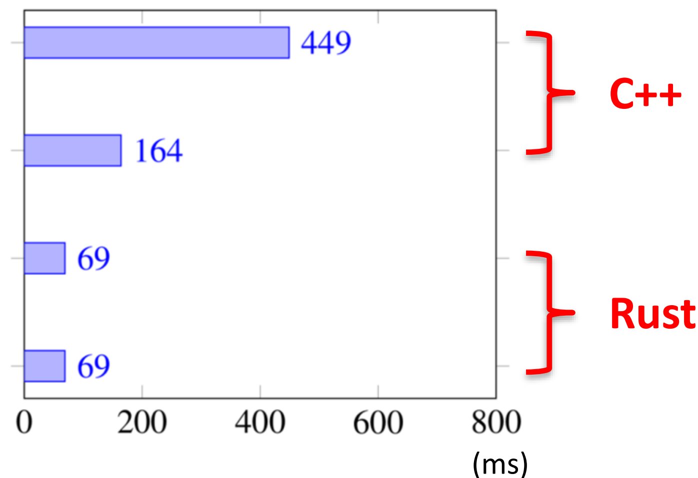
[1] https://github.com/valhalla/valhalla

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Performance

Example: runtime comparisons for a traversal of a (large) `list<Point>`:

- original
- after hardening refactorings
- after translation to Rust by an expert
- after translation to Rust using CRAM (resulting in rather different code)



Disclaimer: Evidence anecdotal at this time!

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Code Usability

```cpp
std::list<Point> trim_front(std::list<Point>& pts, float dist) {
  if (pts.size() < 2)
    return {};

  std::list<Point> result;
  result.push_back(pts.front());
  double d = 0.0f;
  for (auto p1 = pts.begin(), p2 = std::next(pts.begin()); p2 != pts.end(); ++p1, ++p2) {
    Point& next_point = *p2;
    double segdist = p1->Distance(next_point);
    if ((d + segdist) > dist) {
      double frac = (dist - d) / segdist;
      auto midpoint = p1->PointAlongSegment(next_point, frac);
      result.push_back(midpoint);

      pts.erase(pts.begin(), p1);
      pts.front() = midpoint;
      return result;
    } else {
      d += segdist;
      result.push_back(*p2);
    }
  }

  pts.clear();
  return result;
}
```

C++

Rust

```rust
fn trim_front(pts: &mut Vec<Point>, dist: f32) -> Vec<Point> {
    if pts.len() < 2 {
        return vec![];
    };
    let mut result: Vec<Point> = Vec::new();
    result.push(pts[0].clone());
    let mut d: f64 = 0.0f32 as f64;
    for i in 0..(pts.len() - 1) {
        let next_point: &Point = &pts[i + 1];
        let segdist: f64 = pts[i].distance(&next_point);
        if (d + segdist) > (dist as f64) {
            let frac: f64 = ((dist as f64) - d) / segdist;
            let midpoint: Point = pts[i].point_along_segment(&next_point, frac);
            result.push(midpoint.clone());
            pts.drain(0..i);
            pts[0] = midpoint;
            return result;
        } else {
            d += segdist;
            result.push(pts[i + 1].clone());
        };
    }
    pts.clear();
    result
}
```


DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Outlook & Wrapping Up

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

Outlook

Future will bring *deep-dives* into advanced C++ idioms:

- C preprocessor: `#include`s, cond'l compilation
- OOP: inheritance, exception handling
- Concurrency: portable threads (`std::thread`, not `pthread`)
- C++11 and beyond: smart pointers, concepts and modules
- General: Advanced Assurance Techniques

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

CRAM: Capabilities & Benefits

Capabilities:

- ingest well-designed C++
- eliminate hazardous coding patterns in C++
- generate idiomatic, efficient (so far), and human-maintainable Rust code

Benefits:

- remove crashes and vulnerabilities due to common memory access errors
- improve development experience (via the Rust build system)
- modernize your code

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.

How to Obtain & Usage

Distribution:

- website with demo video: https://cpp-rust-assisted-migration.gitlab.io
- fully open-source: https://gitlab.com/cpp-rust-assisted-migration/code
- easiest to obtain via docker image
- dependencies: VS Code, Mnemosyne (both freely available)

Sponsorship:

- DARPA program: LiLaC-SL (Lifting Legacy Code to Safer Languages)
- Program Manager: Dr. Sergey Bratus, SETA: Jorge Buenfil (I2O)

DISTRIBUTION STATEMENT A. Approved for Public Release. Distribution Unlimited.