C2Rust
Migrating Legacy Code to Rust
Acknowledgements & Disclaimer

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA).

The views, opinions and/or findings expressed 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 is unlimited.
Who am I?

- Per Larsen
- Co-Founder at Immunant, Inc.
- From Denmark / Located in Irvine, CA
- Background in C/C++ exploit mitigation
Who?

Eric Mertens
Galois

Alec Theriault
Galois

Ryan Wright
Galois

Andrei Homescu
Immunant

Daniel Kolsoi
Immunant

Stephen Crane
Immunant
Why?

C/C++ mitigations are far from perfect.

Rust is an attractive migration target. Can we make migration easier?

1. reduce the tedium of initial translation
2. help catch errors during refactoring
Transpiling

Design Goals:

● Robust C and C++ parsing
● Preserve functionality of input code
● Generate output fit for human consumption
● Write back end in Rust; reuse Rust internals
Other efforts

- Corrode
  - uses Haskell C parsing library
  - [https://github.com/jameysharp/corrode](https://github.com/jameysharp/corrode)
- Citrus-rs
  - uses clang for parsing
  - “transforms C syntax to Rust syntax, but ignores C semantics”
  - [https://gitlab.com/citrus-rs/citrus](https://gitlab.com/citrus-rs/citrus)
Transpiler
Transpiler

- .json
  Compile Commands
- .c
  C sources
- .rs
  Rust sources
- transpile.py

clang → .cbor
ast-exporter

rustc
ast-importer
syntax
AST importer

- .cbor
- clang AST
- importer AST
- rustc AST
- prune decls
- reloop loops

.rs
Preprocessor directives

- Translating after preprocessing
Preprocessor directives

- Translating after preprocessing
- How was compiler invoked?
  - compile_commands.json
- Recording compile commands
  - Use **cmake** 2.8.5 or later
  - Use **intercept-build** for makefile projects
  - ... or **bear** (Linux only)

https://github.com/rizsotto/bear
Simple loops
Simple loops

\[
\begin{align*}
\text{int } x &= 0; \\
\text{while } (x < 42) \{ x++; \} \\
\text{for } (x = 0; x < 42; x++) \{ \} \\
\end{align*}
\]

\[
\begin{align*}
\text{let mut } x &\text{: libc::c_int = 0i32; } \\
\text{while } x < 42i32 \{ x += 1 \} \\
x &= 0i32; \\
\text{while } x < 42i32 \{ x += 1 \};
\end{align*}
\]
```c
int sum(int count) {  
goto a;

b:  
    --count;
    goto d;

a:;  
int x = 0;
    goto d;

c:  
    return x;

d:  
    if(count <= 0)  
        goto c;  
        goto e;

e:  
    x += count;
    goto b;
}
```
Translator limitations

Unimplemented

- variadic function definitions (Rust RFCs blocking issue #2137)
- bitfields (Rust RFCs blocking issue #314)
- long double and _Complex types (Rust libc blocking issue #355)
- macros

Likely won’t ever support

- longjmp and setjmp
- jumps in and out of GNU C statement expressions
C source code

```c
void insertion_sort(int const n, int * const p) {
    for (int i = 1; i < n; i++) {
        int const tmp = p[i];
        int j = i;
        while (j > 0 && p[j-1] > tmp) {
            p[j] = p[j-1];
            j--;
        }
        p[j] = tmp;
    }
}
```

Generated Rust source code

```rust
#[allow(dead_code, mutable_transmutes, non_camel_case_types, non_snake_case, non_upper_case_globals, unused_mut)]
#[feature(libc)]
extern crate libc;

pub unsafe extern "C" fn insertion_sort(n: libc::c_int, p: *mut libc::c_int) -> () {
    let mut i: libc::c_int = 1i32;
    while i < n {
        let tmp: libc::c_int = *p.offset(i as isize);
        let mut j: libc::c_int = i;
        while j > 0i32 && *p.offset((j - 1i32) as isize) >
            *p.offset(j as isize) - *p.offset((j - 1i32) as isize) as libc::c_int + 1
            j = 1
        }
        *p.offset(j as isize) = tmp;
        i += 1
    }
    
    Download output.rs
```
Running locally

$ git clone git@github.com:immunant/c2rust

$ c2rust/scripts/docker_build.sh

$ cd c2rust/vagrant && vagrant up
Building

$ c2rust/scripts/build_translator.py --with-clang

$ c2rust/scripts/build_cross_checks.py

$ cd c2rust/rust-refactor && cargo build
Transpiling

$ git clone git@github.com:immunant/buffer.git

$ cd buffer && bear make

✓ ok

$ path/to/transpile.py -m=test compile_commands.json

$ cd c2rust-build && RUSTFLAGS=-A warnings cargo run

✓ ok
/*
 * Allocate a new buffer with `n` bytes.
 */

buffer_t *
buffer_new_with_size(size_t n) {
    buffer_t *self = malloc(sizeof(buffer_t));
    if (!self) return NULL;
    self->len = n;
    self->data = self->alloc = calloc(n + 1, 1);
    return self;
}
Example Rust output

```rust
#[no_mangle]
pub unsafe extern "C" fn buffer_new_with_size(mut n: size_t) -> *mut buffer_t {
    let mut self_0: *mut buffer_t =
        malloc(::std::mem::size_of::<buffer_t>() as libc::c_ulong) as *mut buffer_t;
    if self_0.is_null() {
        return 0 as *mut buffer_t
    } else {
        (*self_0).len = n;
        (*self_0).alloc =
            calloc(n.wrapping_add(1i32 as libc::c_ulong),
                   1i32 as libc::c_ulong) as *mut libc::c_char;
        (*self_0).data = (*self_0).alloc;
        return self_0
    }
}
```
#[no_mangle]

pub extern "C" fn buffer_new_with_size(mut n: size_t) -> *mut buffer_t {

    let mut v = vec![0; n + 1];
    let mut b = Box::new(buffer_t {
        len: n,
        data: v.as_mut_ptr(),
        alloc: v,
    });
    Box::into_raw(b)
}
Cross checking
Cross checking

1. Instrument original C and translated Rust
2. Run programs with identical inputs
3. **Optional:** Configure cross checking

```c
int id(int x) {
    return x;
}
```

```rust
pub fn id(x: libc::c_int) -> libc::c_int {
    return x;
}
```
Cross checking options

1. **Online**
   - Using **ReMon** MVEE*
   - + no log files
   - + replicates program input
   - - limited compatibility

2. **Offline**
   - + broad compatibility
   - - user must ensure identical inputs
   - - log files grow quickly
Multi-variant execution environment

https://github.com/stijn-volckaert/ReMon
Cross checking instrumentation

1. **clang** plug-in for C code
2. **rustc** plug-in for Rust code
3. Cross-checking runtimes  
   a. MVEE-based  
   b. log-based
4. Zeroing **malloc** replacement
5. **ptrace**-based segfault handler
Cross-checking a library (1/2)

$ cd buffer && bear make
✓ ok

$ path/to/transpile.py -x -u -m=test compile_commands.json

$ cd c2rust-build && RUSTFLAGS=-Awarnings cargo build
Cross-checking a library (2/2)

$ export LD_LIBRARY_PATH=path/to/libfakechecks.so

$ cargo run --quiet 2> ../buffer.rust.xchecks

$ cd .. && make test_xcheck 2> buffer.c.xchecks

$ diff buffer.rust.xchecks buffer.c.xchecks || echo "fail"
Refactoring
fn main() {
    let mut i;

    i = 0;
    'a: while (i < 10) {
        println!("{}", i);
        i = i + 1;
    }

    i = 0;
    'a: while (i < 10) {
        println!("{}", i);
        i = i + 2;
    }
}
fn main() {
    let mut i;

    'a: for i in 0..10 {
        println!("{}");
    }

    'a: for i in (0..10).step_by(2) {
        println!("{}");
    }
}
Major enhancements

- Automate safety transformations
- C++ subset support
- Translation of macros/preprocessor directives
Automate safety transformations

- 100% automation not possible
- Challenges
  - Lack of domain knowledge
  - Differences in type systems
    - Ownership
    - Mutability
  - Differences between C preprocessor macros and Rust macros.
Refactoring quicksort (1/4)

```rust
code
pub unsafe extern "C" fn swap(mut a: *mut libc::c_int,
                            mut b: *mut libc::c_int) -> () {
    let mut t: libc::c_int = *a;
    *a = *b;
    *b = t;
}

code
pub unsafe extern "C" fn partition(mut arr: *mut libc::c_int,
                                  mut low: libc::c_int,
                                  mut high: libc::c_int) -> libc::c_int {
    // elided
    swap(&mut *arr.offset(i as isize) as *mut libc::c_int,
         &mut *arr.offset(j as isize) as *mut libc::c_int);
    // elided
}
```
Refactoring quicksort (2/4)

```rust
pub extern "C" fn swap(mut a: &mut libc::c_int,
                        mut b: &mut libc::c_int) -> () {
    let t: libc::c_int = *a;
    *a = *b;
    *b = t;
}

pub unsafe extern "C" fn partition(mut arr: &mut [libc::c_int],
                                    mut low: libc::c_int,
                                    mut high: libc::c_int) -> libc::c_int {
    // elided
    // requires two mutable borrows, won't compile
    swap(&mut arr[i as usize],
         &mut arr[j as usize]);
    // elided
}
```
Refactoring quicksort (3/4)

```rust
pub extern "C" fn swap(mut a: &mut libc::c_int,
                        mut b: &mut libc::c_int) -> () {
    let t: libc::c_int = *a;
    *a = *b;
    *b = t;
}

pub unsafe extern "C" fn partition(mut arr: &mut [libc::c_int],
                                    mut low: libc::c_int,
                                    mut high: libc::c_int) -> libc::c_int {
    // elided
    // the idiomatic solution; requires human insight
    arr.swap(i as usize, j as usize);
    // elided
}
```
Refactoring quicksort (4/4)

```rust
pub extern "C" fn swap(mut a: &mut libc::c_int,
    mut b: &mut libc::c_int) -> () {
    let t: libc::c_int = *a;
    *a = *b;
    *b = t;
}

pub unsafe extern "C" fn partition(mut arr: &mut [libc::c_int],
    mut low: libc::c_int,
    mut high: libc::c_int) -> libc::c_int {
    // elided
    let mut a = mem::replace(&mut arr[i as usize], 0);
    let mut b = mem::replace(&mut arr[j as usize], 0);
    swap(&mut a, &mut b);
    mem::replace(&mut arr[i as usize], a);
    mem::replace(&mut arr[j as usize], b);
    // elided
}
```
Let’s translate your code to Rust