Translating C To Rust: Lessons from a User Study

Ruishi Li*, **Bo Wang***, **Tianyu Li** , **Prateek Saxena**,

Ashish Kundu†

National University of Singapore
Cisco Research†

*Equal Contribution

A “dream problem”: Translating C to Rust for memory safety

Memory unsafe C software

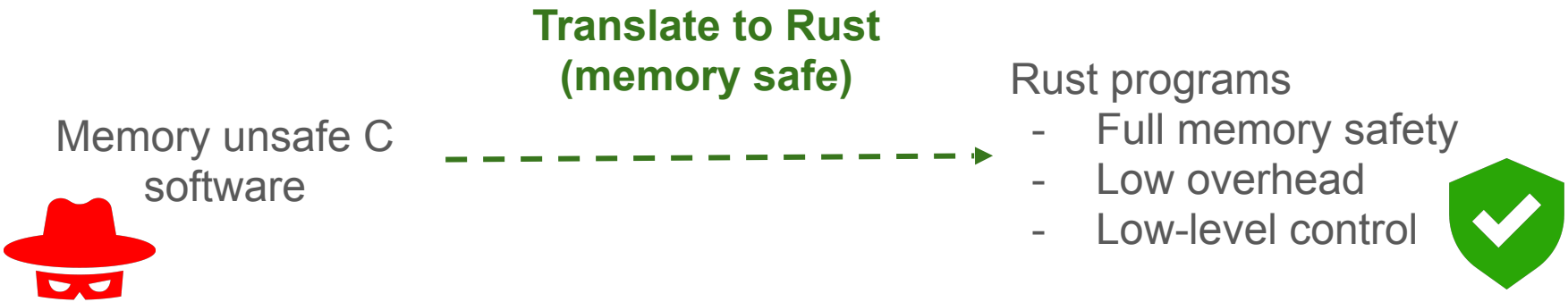
[1] “Translate all C to Rust” by DARPA. USA. 2024

[2] Rust for Linux Github organization: https://github.com/Rust-for-Linux by Linux. 2024

[3] Rust for Windows: https://github.com/microsoft/windows-rs by Microsoft. 2024

[4] Bare-metal Rust in Android: https://security.googleblog.com/search/label/rust by Google. 2024

A “dream problem”: Translating C to Rust for memory safety



[1] “Translate all C to Rust” by DARPA. USA. 2024
[2] Rust for Linux Github organization: https://github.com/Rust-for-Linux by Linux. 2024
[3] Rust for Windows: https://github.com/microsoft/windows-rs by Microsoft. 2024
[4] Bare-metal Rust in Android: https://security.googleblog.com/search/label/rust by Google. 2024

A “dream problem”: Translating C to Rust for memory safety

Memory unsafe C software

Translate to Rust (memory safe)

Rust programs
- Full memory safety
- Low overhead
- Low-level control

DARPA

Google

[1] “Translate all C to Rust” by DARPA. USA. 2024
[2] Rust for Linux Github organization: https://github.com/Rust-for-Linux by Linux. 2024
[3] Rust for Windows: https://github.com/microsoft/windows-rs by Microsoft. 2024
[4] Bare-metal Rust in Android: https://security.googleblog.com/search/label/rust by Google. 2024

Limitations of existing work

Compiler-based translators
(Laertes[1], Crown[2])

LLMs-based translators
(Flourine[3], Vert[4])

[1] Emre, Mehmet, et al. "Translating C to safer Rust." Proceedings of the ACM on Programming Languages 5.OOPSLA (2021): 1-29.
[2] Zhang, Hanliang, et al. "Ownership guided C to Rust translation." International Conference on Computer Aided Verification. Cham: Springer Nature Switzerland, 2023.
[3] Eniser, Hasan Ferit, et al. "Towards translating real-world code with LLMs: A study of translating to Rust." arXiv preprint arXiv:2405.11514 (2024).
[4] Yang, Aidan ZH, et al. "Vert: Verified equivalent rust transpilation with few-shot learning." arXiv preprint arXiv:2404.18852 (2024).

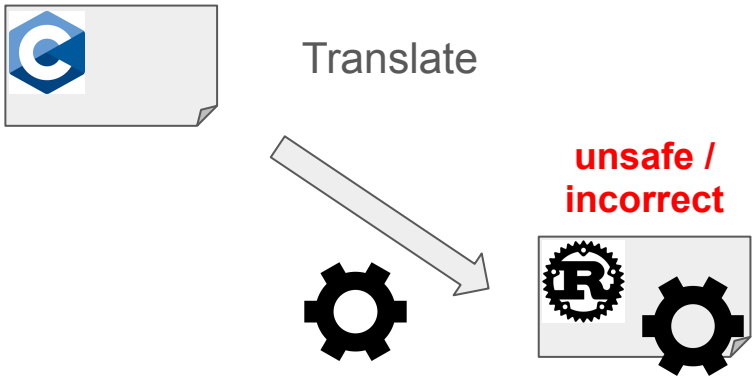
Limitations of existing work

	Translations	
Compiler-based translators (Laertes[1], Crown[2])	**unsafe**	**mostly correct**
LLMs-based translators (Flourine[3], Vert[4])	**mostly safe**	**incorrect**

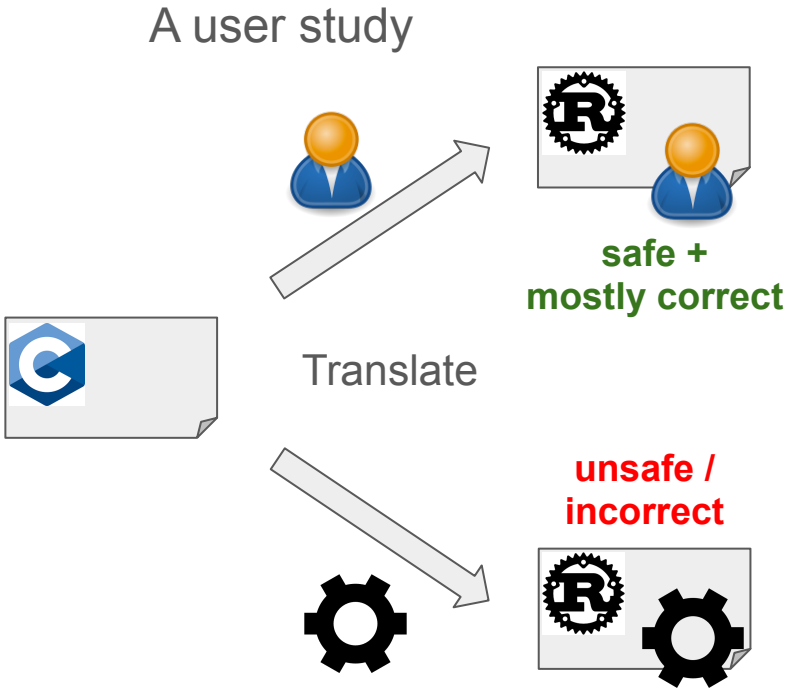
* Correct under tests.

[1] Emre, Mehmet, et al. "Translating C to safer Rust." Proceedings of the ACM on Programming Languages 5.OOPSLA (2021): 1-29.
[2] Zhang, Hanliang, et al. "Ownership guided C to Rust translation." International Conference on Computer Aided Verification. Cham: Springer Nature Switzerland, 2023.
[3] Eniser, Hasan Ferit, et al. "Towards translating real-world code with LLMs: A study of translating to Rust." arXiv preprint arXiv:2405.11514 (2024).
[4] Yang, Aidan ZH, et al. "Vert: Verified equivalent rust transpilation with few-shot learning." arXiv preprint arXiv:2404.18852 (2024).

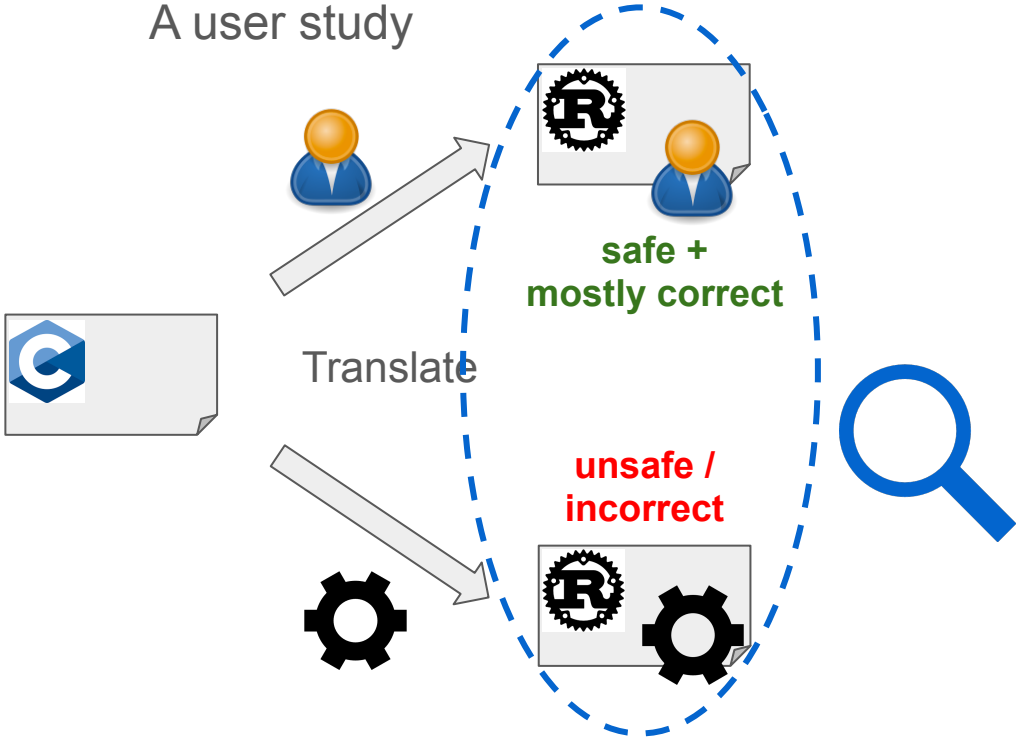
Contributions



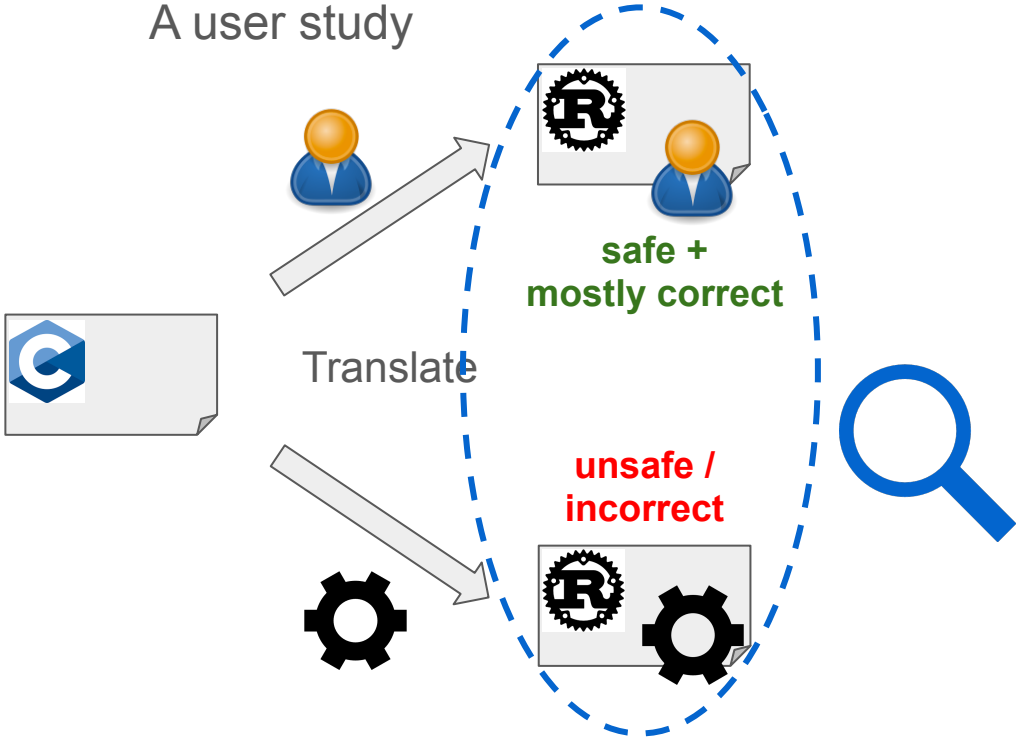
Contributions



Contributions



Contributions

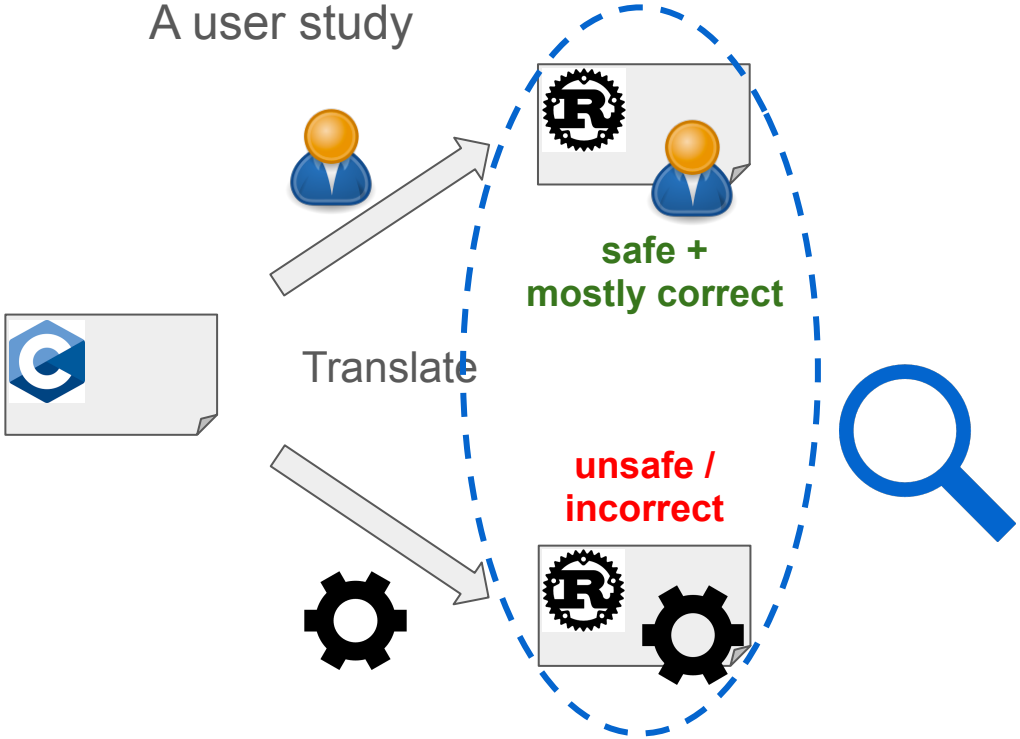


Differences between their translations

↓

How human users approach this task?

Contributions



Differences between their translations

↓

How human users approach this task?

Challenges to address for future automatic translators.

How we conducted our user study

Undergraduates in a computer security course.

Know memory safety and Rust principles.

33 out of 71 students participated.

How we conducted our user study

Undergraduates in a computer security course.

Know memory safety and Rust principles.

33 out of 71 students participated.

C program + tests

Rust translation

Has **safe** Rust translations.

How we conducted our user study

Undergraduates in a computer security course.

Know memory safety and Rust principles.

33 out of 71 students participated.

C program + tests

Rust translation

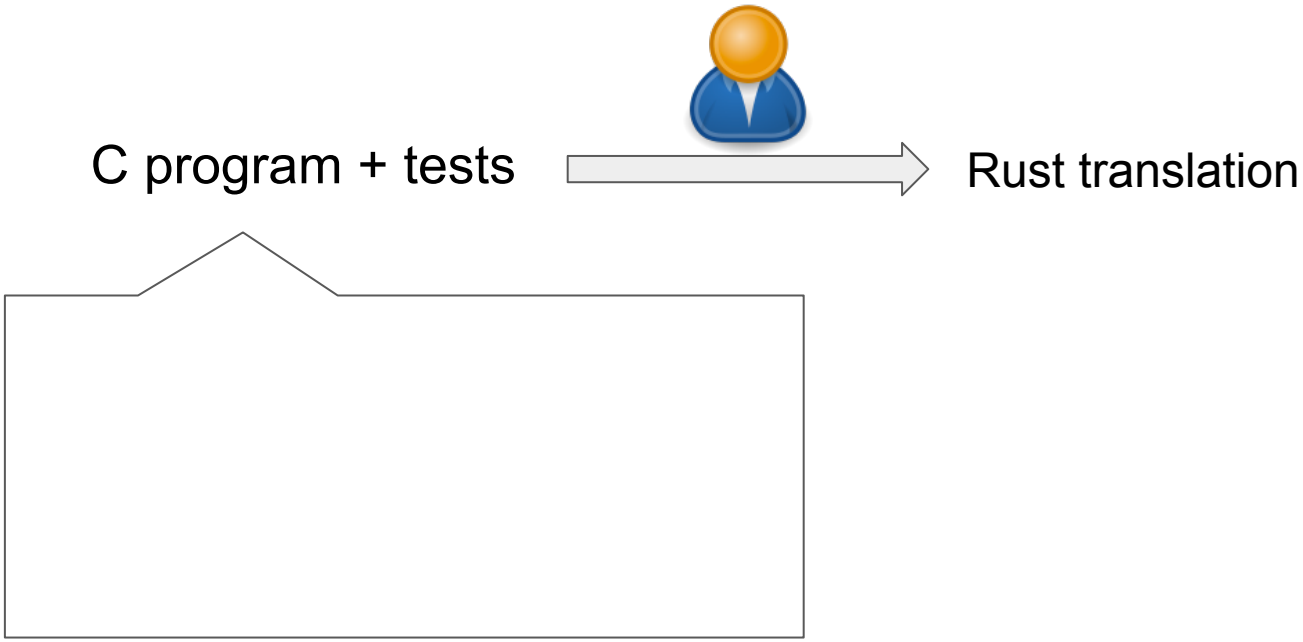
Has **safe** Rust translations.

Requirements:
1. Safety: no **unsafe**
2. Correct under tests

Benchmarks & Collected translations

C program + tests  Rust translation

Benchmarks & Collected translations



Benchmarks & Collected translations

- 8 self-contained programs.

Benchmarks & Collected translations

- 8 self-contained programs.
- 300 – 600 LoC.

Benchmarks & Collected translations

C program + tests → Rust translation

- 8 self-contained programs.
- 300 – 600 LoC.
- Functionality: file/data processing, strings/numbers computation, and parsing.

Benchmarks & Collected translations

C program + tests → Rust translation

- 8 self-contained programs.
- 300 – 600 LoC.
- Functionality: file/data processing, strings/numbers computation, and parsing.

Collected 31 final translations:
Safe and (mostly) correct.

C to safe Rust is difficult!
Translating line-by-line doesn’t work

C to safe Rust is difficult!
Translating line-by-line doesn’t work

Snippet of C benchmark

```c
char *p1, *p2; // p1 = ...
p2 = p1;
while (*p2 != '\0') {
 if (condition 1) { *p2 = ' '; }
 if (condition 2) { p1 += 1; }
 p2++;
}
puts(p1);
```

C to safe Rust is difficult!
Translating line-by-line doesn't work

Snippet of C benchmark

```c
char *p1, *p2; // p1 = ...
p2 = p1;
while (*p2 != '\0') {
 if (condition 1) { *p2 = ' '; }
 if (condition 2) { p1 += 1; }
 p2++;
}
puts(p1);
```

A handcrafted line-by-line translation example

```rust
let mut p1: Vec<char>; // p1 = ...
let p2 = &mut p1;
for c in p2 {
 if condition 1 { *c = ' '; }
 if condition 2 {p1… }
}
println!("{:?}", p1…);
```

Violate Rust borrow rules

Compiler-based translation vs. User translation

Compiler-based translation vs. User translation

Snippet of C benchmark

```c
char *p1, *p2; // p1 = ...
p2 = p1;
while (*p2 != '\0') {
 if (*p2 == '\t') { *p2 = ' '; }
 if (leading space) { p1 += 1; }
 p2++;
}
puts(p1);
```

Compiler-based translation vs. User translation

Snippet of C benchmark

```c
char *p1, *p2; // p1 = ...
p2 = p1;
while (*p2 != '\0') {
 if (*p2 == '\t') { *p2 = ' '; }
 if (leading space) { p1 += 1; }
 p2++;
}
puts(p1);
```

Compiler-based translation by Laertes

```rust
unsafe {
 let mut p1: *mut std::os::raw::c_char;
 let mut p2: *mut std::os::raw::c_char; // p1 = ...
 p2 = p1;
 while *p2 != '\0' as i8 {
   if *p2 == '\t' as i8 { *p2 = ' ' as i8; }
   if leading space { p1 = p1.offset(1); }
   p2 = p2.offset(1);
 }
 puts(p1); }
```

Compiler-based translation vs. User translation

Snippet of C benchmark

```c
char *p1, *p2; // p1 = ...
p2 = p1;
while (*p2 != '\0') {
 if (*p2 == '\t') { *p2 = ' '; }
 if (leading space) { p1 += 1; }
 p2++;
}
puts(p1);
```

Compiler-based translation by Laertes

```rust
unsafe {
 let mut p1: *mut std::os::raw::c_char;
 let mut p2: *mut std::os::raw::c_char; // p1 = ...
 p2 = p1;
 while *p2 != '\0' as i8 {
   if *p2 == '\t' as i8 { *p2 = ' ' as i8; }
   if leading space { p1 = p1.offset(1); }
   p2 = p2.offset(1);
 }
 puts(p1); }
```

```rust
let mut p1: String; // p1 = ...
p1 = p1.replace('\t', " ");
p1 = p1.trim_start().to_string();
println!("{}", p1);
```

User translation

Compiler-based translation vs. User translation

Snippet of C benchmark

```c
char *p1, *p2; // p1 = ...
p2 = p1;
while (*p2 != '\0') {
 if (*p2 == '\t') { *p2 = ' '; }
 if (leading space) { p1 += 1; }
 p2++;
}
puts(p1);
```

Compiler-based translation by Laertes

```
unsafe {
 let mut p1: *mut std::os::raw::c_char;
 let mut p2: *mut std::os::raw::c_char; // p1 = ...
 p2 = p1;
 while *p2 != '\0' as i8 {
   if *p2 == '
   if leading
   p2 = p2.offset(1);
 }
 puts(p1); }
```

unsafe Rust.
Line-by-line and pointer-by-pointer

```
let mut p1: String; // p1 = ...
p1 = p
p1 = p
printl
```

safe Rust.
Abstract the C code.

User translation

Difference 1: translation of aliasing pointers

Compiler-based approaches:
Line-by-line and pointer-by-pointer

Difference 1: translation of aliasing pointers

Compiler-based approaches:
Line-by-line and pointer-by-pointer

Strategy (a): **Elide** aliasing pointers with Rust methods.

```
let mut p1: String; // p1 = ...
p1 = p1.replace('\t', " ");
p1 = p1.trim_start().to_string();
println!("{}", p1);
```

Difference 1: translation of aliasing pointers

Compiler-based approaches:
Line-by-line and pointer-by-pointer

Strategy (a): **Elide** aliasing pointers with Rust methods.

```
let mut p1: String; // p1 = ...
p1 = p1.replace('\t', " ");
p1 = p1.trim_start().to_string();
println!("{}", p1);
```

Strategy (b): **Clone** the object to separate r/w access.

```
let p1: String; // p1 = ...
let mut str: String = String::new();
for c in p1.chars() {
 if c == '\t' { str.push(' '); }
 else if not leading space { str.push(c); }
}
println!("{}", str);
```

Difference 2: translation of C pointers and C API calls

Snippet of C benchmark

```
char *p1;
…
puts(p1);
```

Compiler-based translation by Laertes

```
unsafe {
  let p1: *mut std::os::raw::c_char;
  …
  puts(p1);
}
```

User translation

```
let p1: String;
…
println!("{}", p1);
```

Difference 2: translation of C pointers and C API calls

Snippet of C benchmark

```
char *p1;
…
puts(p1);
```

Compiler-based translation by Laertes

```
unsafe {
  let p1: *mut std::os::raw::c_char;
  …
  puts(p1);
}
```

User translation

```
let p1: String;
…
println!("{}", p1);
```

- Limited Rust types or **unsafe** raw pointers.
- **unsafe** C API calls

Difference 2: translation of C pointers and C API calls

Snippet of C benchmark

```
char *p1;
…
puts(p1);
```

Compiler-based translation by Laertes

```
unsafe {
  let p1: *mut std::os::raw::c_char;
  …
  puts(p1);
}
```

- Limited Rust types or **unsafe** raw pointers.
- **unsafe** C API calls

User translation

```
let p1: String;
…
println!("{}", p1);
```

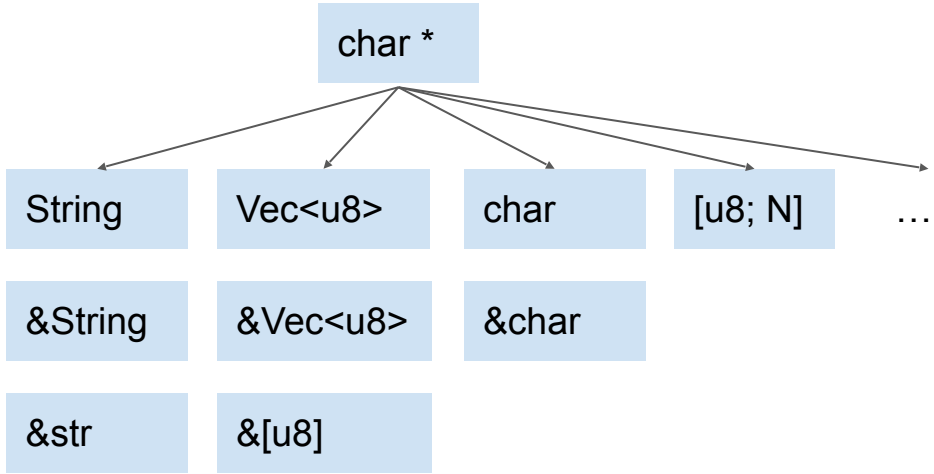
- **safe** Rust types
- **safe** Rust methods.

Users semantically lift low-level pointers and API calls

Users semantically lift low-level pointers and API calls

User translations:

- Various Rust types.
- No raw pointers.



Users often use zero-cost abstractions

String	Vec<u8>	char	[u8; N]
&String	&Vec<u8>	&char	…
&str	&[u8]		

Zero-cost abstractions

Statically guaranteed

Temporal memory safety

Spatial memory safety

Mostly dynamically guaranteed

Users semantically lift low-level pointers and API calls

Compiler-based work:

- Call **unsafe** C APIs

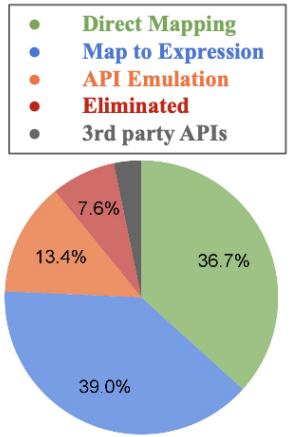
Users:

- Find **safe** Rust alternatives.
- Or emulate them.

Direct Mapping (36.7%):
`isdigit`
`is_ascii_digit`
	Map to Expr (39%):
	`fopen(file, "w+")`
	`File::options()`
	`.read(true)`
	`.write(true)`
	`.create(true)`
	`.truncate(true)`
	`.open(file)?;`

API Emulation (13.4%):
`sscanf`
`sscanf_before, sscanf_after`	

Eliminated (7.6%):
`malloc, free, fclose`
	3rd party APIs (3.3%):
	`geteuid`
	`nix::unistd::get euid`



Difference 3: translation of unsafe C features

C features

Mutable globals

C unions

Difference 3: translation of unsafe C features

C features

Compiler-based translation

C features	Compiler-based translation
Mutable globals	**unsafe** plain globals
C unions	**unsafe** Rust unions

Difference 3: translation of unsafe C features

 C features	 Compiler-based translation	 User translation
Mutable globals	**unsafe** plain globals	Locals Dynamic references with runtime overhead. Atomic types (lock-free)
C unions	**unsafe** Rust unions	

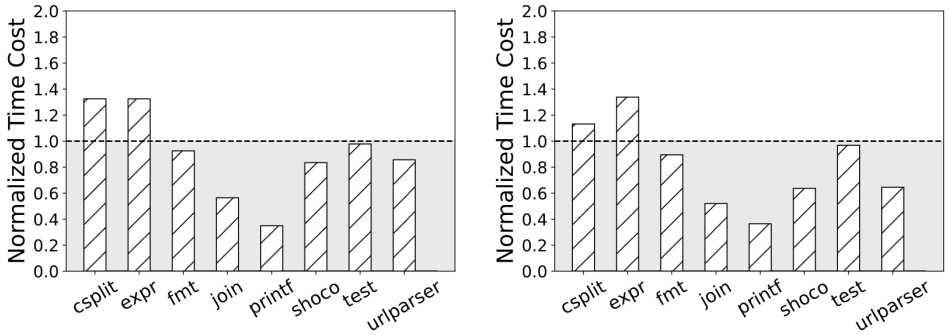
Difference 3: translation of unsafe C features

 C features	 Compiler-based translation	 User translation
Mutable globals	**unsafe** plain globals	Locals Dynamic references with runtime overhead. Atomic types (lock-free)
C unions	**unsafe** Rust unions	Use safe Rust APIs Rust enum

User translations: memory-safe with good performance

User translations: memory-safe with good performance

The execution overhead is mostly within 20%.



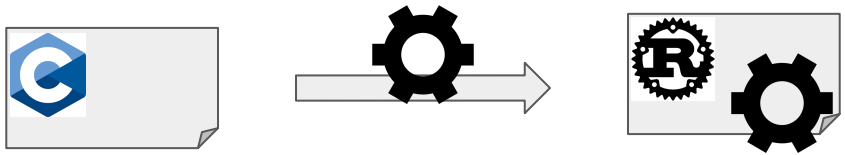
(a) Compiled with `-O2`

(b) Compiled with `-O3`

Dashed line: the execution time for C programs.

Safe Rust translation: memory safety and good performance is achievable.

Challenging subproblems to address for future automatic translators



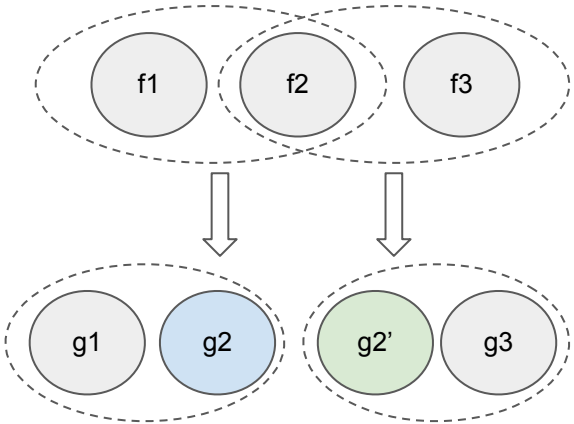
Challenge 1: the decomposition problem

LLMs-based tools: decompose the code based on function dependencies.

Challenge 1: the decomposition problem

LLMs-based tools: decompose the code based on function dependencies.

- ➢ Inconsistency issue

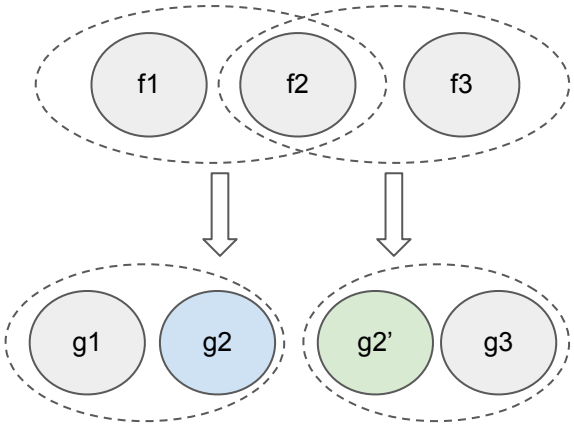


g2 and g2’ are inconsistent!

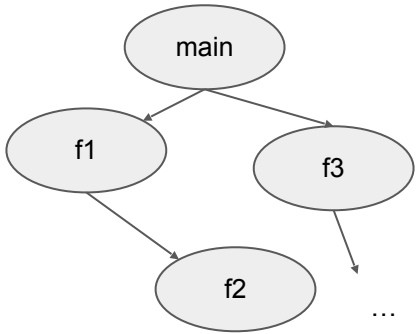
Challenge 1: the decomposition problem

LLMs-based tools: decompose the code based on function dependencies.

- ➢ Inconsistency issue
- ➢ Complicated dependencies



g2 and g2’ are inconsistent!

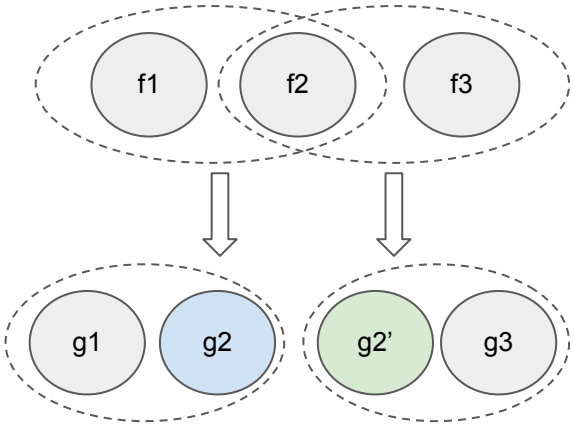


The main of a binary depends on all functions!

Challenge 1: the decomposition problem

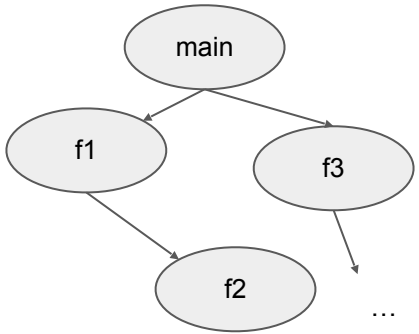
LLMs-based tools: decompose the code based on function dependencies.

- ➢ Inconsistency issue
- ➢ Complicated dependencies



g2 and g2’ are inconsistent!

A better decomposition strategy?



The main of a binary depends on all functions!

Challenge 2: the correctness gap problem

For example,

A user translation of fmt

```rust
fn center_stream<R: BufRead>(mut stream: R, _name: &str, config:
      &Config) {
    let mut p1 = String::new();
    while let Ok(bytes_read) = get_line(stream, &mut p1) {
        if bytes_read == 0 { break; }
        let len: usize = p1.trim().chars().map(|c| if c == '\t'
     { ' ' } else { c }).map(char::len_utf8).sum();
        let padding = (config.goal_length - len) / 2; //
     Calculate padding to center the line
        for _ in 0..padding { print!("␣"); }
        println!("{}", p1.trim());
    } ...
```

The original C fmt

```c
static void center_stream(FILE *stream, const char *name)
{
    char *p1, *p2; // aliased pointers
    size_t len; int w1, w2; wchar_t wc;
    // ...
    while ((p1 = get_line(stream)) != NULL) {
        len = 0;
        for (p2 = p1; *p2 != '\0'; p2 += w2) {
            if (*p2 == '\t')
                *p2 = '␣';
            // ... skipped
            if (len == 0 && iswspace(wc))  p1 += w2;
            else  len += w1;
        }
        while (l < goal_length) {
            putchar('␣');
            len += 2;
        }
        puts(p1);
    } ...
}
```

Challenge 2: the correctness gap problem

For example,

A user translation of fmt

```rust
fn center_stream<R: BufRead>(mut stream: R, _name: &str, config:
      &Config) {
    let mut p1 = String::new();
    while let Ok(bytes_read) = get_line(stream, &mut p1) {
        if bytes_read == 0 { break; }
        let len: usize = p1.trim().chars().map(|c| if c == '\t'
     { ' ' } else { c }).map(char::len_utf8).sum();
        let padding = (config.goal_length - len) / 2; //
     Calculate padding to center the line
        for _ in 0..padding { print!("␣"); }
        println!("{}", p1.trim());
    } ...
```

The original C fmt

```c
static void center_stream(FILE *stream, const char *name)
{
    char *p1, *p2; // aliased pointers
    size_t len; int w1, w2; wchar_t wc;
    // ...
    while ((p1 = get_line(stream)) != NULL) {
        len = 0;
        for (p2 = p1; *p2 != '\0'; p2 += w2) {
            if (*p2 == '\t')
                *p2 = '␣';
            // ... skipped
            if (len == 0 && iswspace(wc))  p1 += w2;
            else  len += w1;
        }
        while (l < goal_length) {
            putchar('␣');
            len += 2;
        }
        puts(p1);
    } ...
}
```

Fuzz test

CMD: ./fmt -w 10 -c
Stdin: \x7a\xc3

Challenge 2: the correctness gap problem

For example,

A user translation of fmt

```rust
fn center_stream<R: BufRead>(mut stream: R, _name: &str, config:
      &Config) {
    let mut p1 = String::new();
    while let Ok(bytes_read) = get_line(stream, &mut p1) {
        if bytes_read == 0 { break; }
        let len: usize = p1.trim().chars().map(|c| if c == '\t'
     { ' ' } else { c }).map(char::len_utf8).sum();
        let padding = (config.goal_length - len) / 2; //
     Calculate padding to center the line
        for _ in 0..padding { print!("␣"); }
        println!("{}", p1.trim());
    } ...
```

The original C fmt

```c
static void center_stream(FILE *stream, const char *name)
{
    char *p1, *p2; // aliased pointers
    size_t len; int w1, w2; wchar_t wc;
    // ...
    while ((p1 = get_line(stream)) != NULL) {
        len = 0;
        for (p2 = p1; *p2 != '\0'; p2 += w2) {
            if (*p2 == '\t')
                *p2 = '␣';
            // ... skipped
            if (len == 0 && iswspace(wc))  p1 += w2;
            else  len += w1;
        }
        while (l < goal_length) {
            putchar('␣');
            len += 2;
        }
        puts(p1);
    } ...
}
```

Fuzz test

CMD: ./fmt -w 10 -c
Stdin: \x7a\xc3

Stdout of Rust:

(empty)

❌ Different behavior!

Stdout of C:

␣␣␣␣␣z?

Challenge 2: the correctness gap problem

For example,

A user translation of fmt

```rust
fn center_stream<R: BufRead>(mut stream: R, _name: &str, config:
      &Config) {
    let mut p1 = String::new();
    wh                                                    {
                                                          == '\t'
    }
    Calculate padding to center the line
       for _ in 0..padding { print!("␣"); }
       println!("{}", p1.trim());
    } ...
```

Logical difference!

The original C fmt

```c
static void center_stream(FILE *stream, const char *name)
{
    char *p1, *p2; // aliased pointers
    size_t len; int w1, w2; wchar_t wc;
    // ...
    while ((p1 = get_line(stream)) != NULL) {
        len = 0;
        for (p2 = p1; *p2 != '\0'; p2 += w2) {
            if (*p2 == '\t')
                *p2 = '␣';
            // ... skipped
            if (len == 0 && iswspace(wc))  p1 += w2;
            else  len += w1;
        }
        while (l < goal_length) {
            putchar('␣');
            len += 2;
        }
        puts(p1);
    } ...
}
```

Fuzz test

CMD: ./fmt -w 10 -c
Stdin: \x7a\xc3

Stdout of Rust:

(empty)

❌ Different behavior!

Stdout of C:

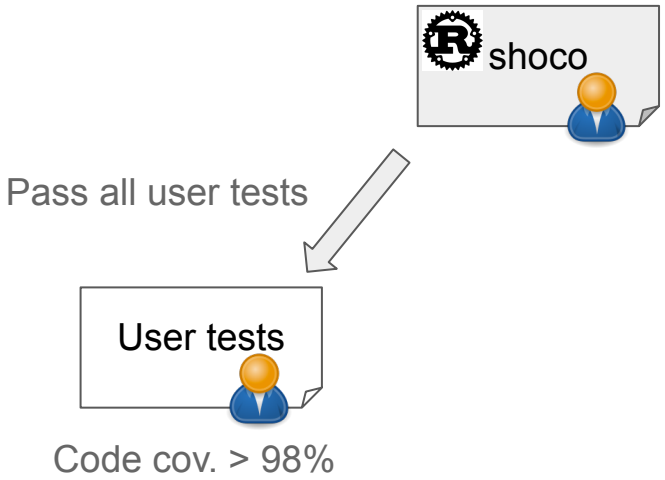
␣␣␣␣␣␣␣␣␣z?

“Correct” user translations are not fully equivalent to C

No user translation is fully equivalent.

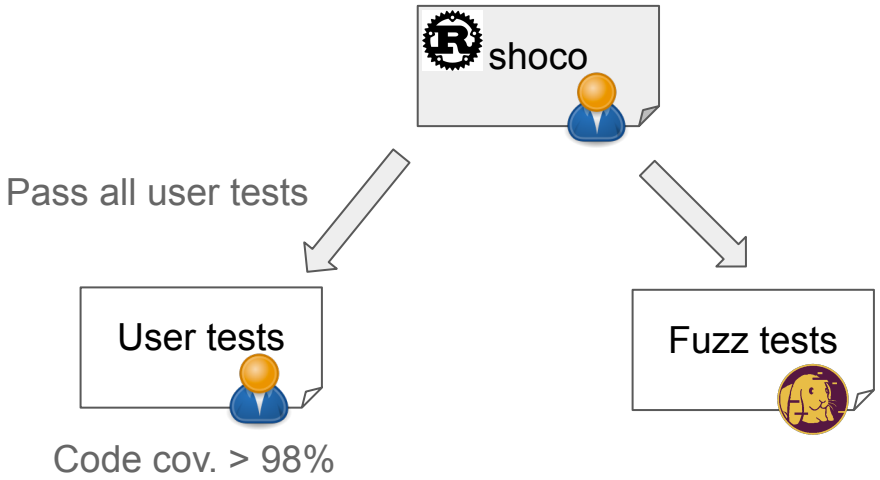
“Correct” user translations are not fully equivalent to C

No user translation is fully equivalent.



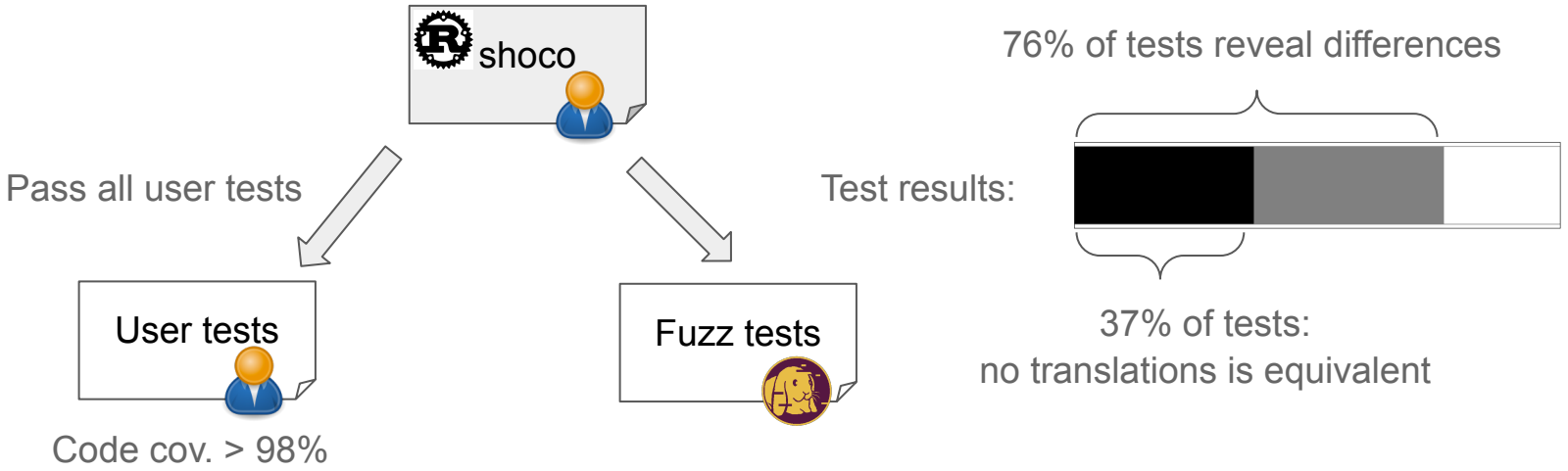
“Correct” user translations are not fully equivalent to C

No user translation is fully equivalent.



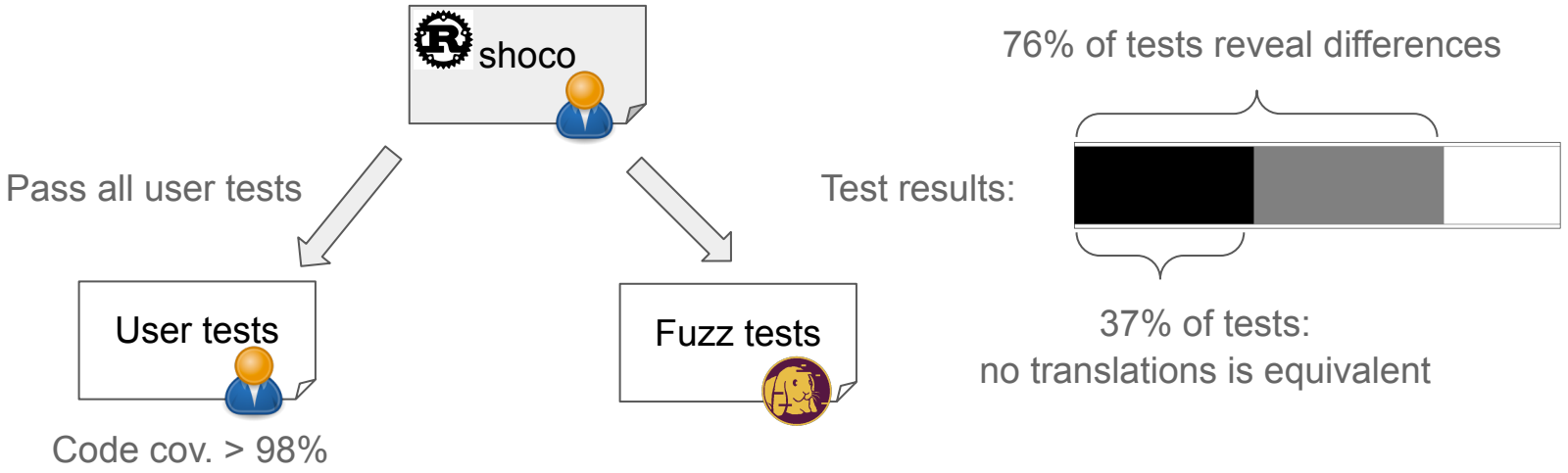
“Correct” user translations are not fully equivalent to C

No user translation is fully equivalent.



“Correct” user translations are not fully equivalent to C

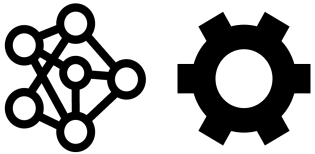
No user translation is fully equivalent.



It’s necessary to
➢ Specify what behaviors to keep across languages.

We are launching an automatic translation service!

Basic Plan:
5 business days
$100



Premium Plan:
1 business day
$200

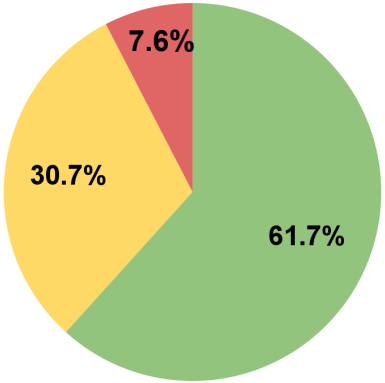
Scan to learn more!

Automated Program Translation
KISP Lab @NUS

* This slide is for entertainment purposes only.

Breakdown of behavioral differences found by fuzz tests

The best
shoco translation



Equivalent behaviors

I/O encoding errors

Runtime safety aborts

Logical differences

Averaged on translations
of all programs