Translating C To Rust: Lessons from a User Study

Ruishi Li*
National University of Singapore
Singapore
liruishi@u.nus.edu

Bo Wang*
National University of Singapore
Singapore
bo_wang@u.nus.edu

Tianyu Li
National University of Singapore
Singapore
tianyuli@u.nus.edu

Prateek Saxena
National University of Singapore
Singapore
prateeks@comp.nus.edu.sg

Ashish Kundu
Cisco Research
San Jose, CA, USA
ashkundu@cisco.com

***Abstract*—Rust aims to offer full memory safety for programs, a guarantee that untamed C programs do not enjoy. How difficult is it to translate existing C code to Rust? To get a complementary view from that of automatic C to Rust translators, we report on a user study asking humans to translate real-world C programs to Rust. Our participants are able to produce safe Rust translations, whereas state-of-the-art automatic tools are not able to do so. Our analysis highlights that the high-level strategy taken by users departs significantly from those of automatic tools we study. We also find that users often choose zero-cost (static) abstractions for temporal safety, which addresses a predominant component of runtime costs in other full memory safety defenses. User-provided translations showcase a rich landscape of specialized strategies to translate the same C program in different ways to safe Rust, which future automatic translators can consider.**

I. Introduction

Memory safety vulnerabilities in C programs are still a prominent category of CVEs reported in commodity software, and number in thousands each year [29]. Several approaches to secure such unsafe code are being investigated.

Full memory safety through runtime checks inserted by compiler instrumentation is achievable. However, it incurs high performance overhead ($\geq$ 50%) and is often not deployed in production [44], [45]. Instead, partial memory safety defenses which have overheads below 20% have found deployment [54]. Emerging hardware features can reduce performance overheads of partial safety techniques, but even then, faulty programs that will often ungracefully abort in production systems have limited appeal. Similarly, automatic patching techniques that can localize bugs and suggest fixes for them are being developed [28], [32], [50], [51]. But these approaches do not aim to rule out the existence of memory safety bugs altogether.

A different approach has been to *write secure code* which is free of memory safety bugs from the ground up. The idea is to have a stricter language and compiler that forces developers to rewrite unsafe code to use safe patterns everywhere. The advantage is that full memory safety can be achieved *mostly* statically, while a few runtime checks incur low overheads.

Designing safe C dialects with that goal has a long history. These dialects largely aim to retain low-level features of C for compatibility, making it difficult to move away from unsafe C patterns entirely while keeping overheads low [34], [46]. More recent designs aim for better compatibility by allowing mixed (statically) safe and unsafe C [22], but when temporal memory safety is desired in them, overheads can exceed 30% [41].

Rust is an alternative mainstream language that offers low-level control over memory, while providing full memory safety. It has a growing ecosystem and increasing support from commodity OSes. It departs from the conventional approaches to safe C dialects in that it largely abandons the programming abstractions in C, such as the use of unchecked raw pointers and unsafe type casts. It is natural to ask: How difficult is it to translate existing C code to Rust then?

In the last few years, automatic techniques to translate C code to Rust have started to emerge. There are two main approaches, one based on compiler-based analysis [24], [33] and another based on large language models (LLMs) [25], [59]. But both approaches have had very limited effectiveness so far, even on small programs of about 100 lines of code. For example, Emre et. al. report that only about 11% of raw C pointers can be converted to safe Rust references through static analysis [24]. Similarly, recent work on Flourine [25] reports that less than half of the small C programs they consider can be auto-translated to Rust using LLM-based repair.

It is unclear what strategies, if any, enable a successful search for C to Rust translations. In order to gain a complementary perspective, in this paper, we study how *human users approach the C to Rust translation task*. We conducted a user study in which we asked undergraduate students taking a course in computer security to translate a given set of real-world C programs to safe Rust. Our participants are familiar with C and memory safety issues, but have minimal prior exposure to Rust. To the best of our knowledge, this is the first such analysis of a user study on C to Rust translation.

33 of our user study participants consented to their translated programs being analyzed and reported on. Most of the

*Contributed equally to this work.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241407
www.ndss-symposium.org

users succeed in providing reasonable translations of our C benchmark to Rust, whereas state-of-the-art automatic C to Rust translation tools are ineffective. Analyzing how users succeed is our main goal. We highlight several high-level principles and strategies that are common across the user translations, which could be useful for future automatic translators.

First, we find that all user-provided Rust translations *semantically lift* from the low-level abstractions used in the C code to re-express the logic in Rust, rather than trying to mimic the original program and data-flow structure too closely. This approach helps break free of the low-level constraints present in the original C code, which would violate the strict rules in Rust if preserved. Rust enforces stricter rules on pointer aliasing than C. Users decomposed the object lifetimes in different ways that are *specialized to access patterns* used for the object to satisfy Rust rules. This shows that there are multiple strategies to translate the same C program to safe Rust, with room for context-specific translation strategies.

Second, we find that resulting Rust translations often contain zero-cost statically-checked safety abstractions. Temporal memory safety, which is a source of significant runtime overhead in many prior defenses [41], [45], is achieved mostly statically. The resulting Rust programs have comparable performance to C, and rarely more than 20% slower than the corresponding C code, even though our participants are not explicitly asked to optimize for performance. At the same time, known spatial and temporal vulnerabilities in our C benchmarks are eliminated in the translated Rust code. This suggests that C to Rust translation is worth it, as the trade-off between performance and security we observe in translations by non-expert Rust users is desirable.

Third, nearly all Rust translations have functional discrepancies compared with the corresponding C code. These discrepancies are easy to find with fuzzing and Rust translations of the same C program have correlated failures, i.e., they often fail the same tests. Some of these discrepancies are expected since undefined behavior present in C programs are handled differently by different users. But, several others are logical errors, suggesting a "last mile" phenomenon at play: While users find it feasible to convert most C functionality to its Rust counterpart, making a 100% correct translation is tedious.

Lastly, in order to examine the broader validity of our observations, we perform a post-hoc investigation of a mature open-source Rust project that mirrors the functionality of a set of popular C programs. Several findings from our user study, which involves non-experts, are also seen in the open-source project maintained by more experienced Rust developers.

Contribution. We report on the lessons garnered from the first user study on translating real-world C programs to Rust by non-expert Rust users. Our work sheds light on strategies and policy decisions that users have to make, which are complementary to automatic mechanisms. We highlight the end gains in user-provided translations, such as the ubiquitous use of zero-cost temporal safety abstractions, elimination of unsafe code patterns, and the trade-offs with functional correctness.

II. Background

Rust allows programs to have fine-grained control over memory but with spatial and temporal memory safety.

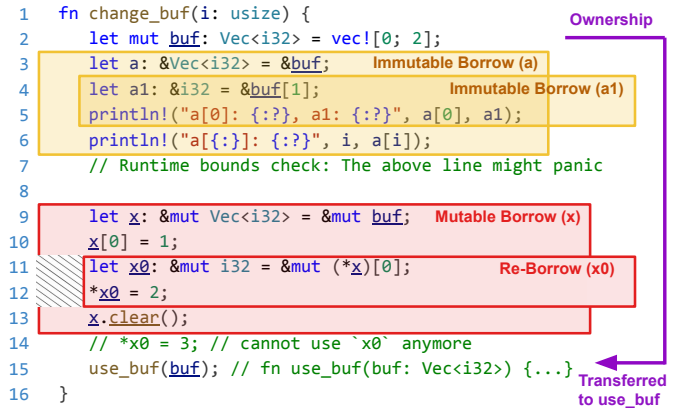


Fig. 1: A Rust code example showing the concepts of ownership transfer, mut./immut. borrowing, and reference lifetime.

A. A Tour of Safe Rust

We briefly explain Rust design principles for achieving full memory safety with a running example shown in Figure 1.

Ownership. Rust ensures that each data object has an exclusive *owner* at any given program point [18]. When the variable owning an object goes out of scope, the compiler automatically deallocates the object. The single-owner principle ensures that the object is deallocated only once, avoiding double-free bugs prevalent in C programs. The compiler knows the scope of all objects in the program. Stack variables have statically determined block/function scope, and so do globals. Objects can be owners of other objects, creating a transitive chain of ownership. The chain starts with a stack/global variable, so the compiler can track ownership of all objects.

Ownership can not be duplicated but only transferred between variables through assignment or parameter passing. The Rust compiler, therefore, tracks the unique owner of every object at every program point. For example, the `buf` variable in the example of Figure 1 is on the stack and encapsulates a (smart) pointer to `Vec` object allocated on the heap. The object allocated on Line 2 in Figure 1 has a unique owner `buf` from Line 2-14, after which the ownership transfers to the first parameter of `use_buf()`. The object will be deallocated when the new owner goes out of scope, i.e., at the end of the function scope of `use_buf()`.

Borrowing. Exclusive ownership, by itself, disallows all aliasing altogether, which is too restrictive. Rust relaxes this restriction by introducing a *borrowing* mechanism. It allows creating *references* (also called *borrows*) to temporarily access the data value without transferring the ownership. References `a`, `a1`, and `x` borrow from the owner `buf`, whereas `x0` *reborrows* from `x` in the example. This allows limited forms of aliasing.

Lifetime. With borrowed references, it becomes essential to ensure that a reference never points to an object that has been deallocated. The Rust compiler statically tracks the *lifetime* of variables: the region of code that the variable must be valid for. Rust enforces that the lifetime of all references is a strict subset of the lifetime of the owner statically. Since the object will be deallocated only when the lifetime of the owner expires, all live references point to valid objects, eliminating use-after-

free vulnerabilities that arise in C programs. The example in Figure 1 creates multiple references to an array object. The variable `buf` gets ownership of the heap-allocated array on Line 2. There are four borrowed references created directly or indirectly from `buf` on lines 3, 4, 9, and 11. Their lifetimes are enforced to be smaller than that of `buf` and are highlighted. The object is automatically deallocated only when the lifetime of the owner `buf` ends, thereby ensuring that all references point to valid memory throughout their lifetime.

Aliasing Xor Mutability (AXM). Aliasing references can update the object storage, possibly moving them, and associated allocation metadata (e.g. size, internal pointers) can become inconsistent. When multiple references are pointing to the same object (e.g. a `vector` type) to insert/delete elements or change its capacity, the location of its internal buffer and size metadata might change. Consequently, this will invalidate other references to the same object, including iterators and references to its elements. Memory-unsafe languages like C leave careful management of aliased pointers to the programmer, which has been a source of mistakes. To avoid this, Rust proposes *Aliasing Xor Mutability (AXM)* principle. First, references are divided into two types: mutable (`&mut`) and immutable (`&`). Mutable references can read and write the referent object, while immutable ones have read-only access. Second, the Rust compiler enforces that either only one mutable reference or multiple (aliasing) immutable references are active at a program point. Mutable aliased references can not be active at the same program point. The AXM principle eliminates the need for careful pointer invalidation by the programmer and helps memory safety in concurrent code as well.

Specifically in the example of Figure 1, Lines 3-6 have multiple immutable references to `buf`, and Lines 9-13 have one mutable reference at each point, created using `&mut` type declaration. The compiler can statically infer the lifetime of `x` and other mutable references that may alias it, such as `x0`. It automatically splits their range of legal use such that there will be a single mutable reference to the object active in scope at each statement, i.e, `x` is permitted on Line $(9-10, 13)$ and `x0` on Lines 11-12. The compiler would forbid using `x0` on or after line 13 as it would create two mutable references to the vector elements, highlighted by commented code on Lines 14 for illustration. In the equivalent C code, dereferencing `x0` on Line 14 would be legal and lead to undefined behavior, since `x0` refers to a vector whose underlying elements were cleared (possibly deallocated) via `x` on Line 13. This can result in out-of-bound access due to a dangling dereference. Such memory errors are disallowed by the AXM principle in Rust.

Runtime AXM and Thread Safety. The AXM principle can be too restrictive in certain circumstances. When multiple mutable references are necessary, Rust programs can resort to runtime borrow checks and reference counting for memory safety, through the use of data types such as `RefCell<T>` and `Rc<T>` for single-thread synchronization [10] and `Mutex<T>` and `Arc<T>` for multi-thread synchronization. These generic types are implemented in the Rust standard library with internal use of unsafe Rust and have been proven sound [36]. Mutable pointers are particularly common in multi-threaded code, and in fact, concurrent C code is notoriously difficult to analyze for memory safety bugs. The Rust compiler enforces that concurrent access or data copying across threads will never result in memory safety violations. Thus, when Rust allows relaxations to the AXM principle, runtime checks are used for memory safety. We will refer to all such references that carry dynamic checks as *dynamic references* to avoid confusion with regular Rust references that obey the AXM principle.

Spatial Bounds Checking. The Rust compiler aims to ensure that the accesses by owners or references to the referent are within spatial bounds. For several types of objects, the compiler can do so statically, but when it cannot, it adds runtime bounds checks. In line 6 of Figure 1, `a[i]` operation is checked to see if `i` is within the boundary of `buf`. If passing 3 to `i`, the Rust code would abort the execution of the program.

Type Safety. Rust is a type-safe language and supports safe type casting among data types using APIs provided in the standard library. All type casts guarantee that the converted values are within the legal range of the target type. Safe Rust disallows direct reinterpretation of memory to an unsafe type. For example, the conversion between pointers (`void*`), non-primitives (`struct`, `union`, etc.) and primitives (`bool`, `enum`, etc.) can cause memory errors in C, but safe Rust forbids them.

B. Existing Automatic C to Rust Translation

There have been 2 main approaches to automatic C-to-Rust translation: One based on compiler or static analysis, while the other using large language models (LLMs) for code synthesis.

Compiler-based Approach. Most of the work in this direction decomposes the code translation problem into two stages: (1) C to unsafe Rust, and (2) unsafe Rust to safe Rust [23], [24], [61]. A mature tool called `c2rust` performs a robust line-by-line syntactic translation from C to corresponding `unsafe` Rust code blocks [33]. The use of `unsafe{...}` keyword allows bypassing the Rust safety checks mentioned above in the enclosed code block. The resulting Rust translation can thus rely on raw pointers (`mut*`), C-compatible data types, and foreign function calls to C libraries (`extern "C" fn`). The second stage of the translation is where the bulk of the challenges lie. The goal of prior automatic tools is to refactor the unsafe Rust code into safe Rust and reduce unsafe code. Laertes proposed a trial-and-error approach leveraging Rust compiler feedback to lift a certain subset of raw pointers into safer Rust references [24]. However, a later work reports that the majority of raw pointers (79% in their benchmarks) cannot be translated to safe references via such techniques [23]. Crown proposed improvements using ownership analysis based on constraint generation and SAT-solving, for a subset of pointer types such as `Box<T>` [61]. We present an evaluation of these state-of-the-art tools later and find that a majority of C pointers are not lifted into safe Rust abstractions by present tools.

LLM-based Approach. Another line of work uses LLMs to translate C programs to Rust [25], [59]. The LLM-guided approach tends to produce code that is much more readable and idiomatic. Modern LLMs can suggest safe Rust data types, APIs, and coding conventions to use. However, LLM-based translation is often difficult to control as LLMs can make semantic mistakes [15]. One of the most recent tools called Flourine, reports that with current LLMs (GPT-4, claude3, and so on), less than 20% for C programs longer than 150 lines of code can be satisfactorily translated to Rust [25]. We evaluate Flourine as well, and we have similar findings.

In summary, existing automatic techniques are insufficient to translate even small C programs of about 200 LoC to Rust. Our observations from the user study hope to offer a holistic perspective on how users get past the inherent challenges.

III. User Study

Prior literature has taken a *bottom-up* approach to the problem of C to Rust translation, aiming to show how automatic tools can address particular sub-problems encountered in translation. Our study offers a complementary perspective on how human users approach the same task. This gives us a *top-down* view of the problem: We can see what common strategies do users use and what challenges remain thereafter.

Our participants are undergraduate students enrolled in a course on computer security. All participants are familiar with memory safety errors in C/C++ programs and were given an introduction to Rust[1]. The task, which is part of a graded course project, is to translate 8 real-world C programs we collected into safe Rust. Each participant is randomly assigned one out of 8 such C programs and is asked to provide a translated program in safe Rust within 20 days[2]. The C programs are taken from GitHub and are small due to the 20-day time limit, with lengths varying from 322 to 536 LoC. The participants were permitted to consult the web freely and use any existing tools, for example, `c2rust` and LLMs.

Ethical Concerns. This study has been granted an IRB exemption from the NUS School of Computing DERC (Department Ethics Review Committee). We followed the procedure advised by the IRB to protect the privacy of the participants and avoid bias. 73 undergraduate students who undertook the study were initially invited to give consent towards the use of their submissions. 33 participants gave their consent and only their data is included in this study. The participant data was anonymized and kept confidentially on our research infrastructure, without being hosted on third-party cloud services. Aggregate statistics are reported here as far as possible, and wherever code snippets are shown for illustration of a concept, they are constructed synthetically by retaining the high-level patterns seen in user submissions (not replicated). Analysis of the participants' data was conducted only after grades were finalized to avoid any influence resulting from the study on the grades.

TABLE I: C Benchmarks for the User Study.

Prog. Name	LoC	Description
`csplit (bsd`[3]`)`	322	Split files based on patterns
`expr (bsd)`	451	Evaluate expressions
`fmt (bsd)`	415	Format text files
`join (bsd)`	472	Join lines of two files
`printf (bsd)`	375	Print formatted text
`test (bsd)`	536	Check file attributes and values
`shoco`	388	String Data Compression
`urlparser`	437	Parse URLs

Benchmarks. Table I shows the C programs we collected for the user study. These programs are representative of C programs that implement lower-level functionality and self-manage memory where memory safety errors arise. Such functionalities are often implemented in C and used by higher-level software systems. We also considered the translation difficulty and chose programs between 300 and 600 LoC, considering the time and effort of the participants. Thus, we chose 6 C programs[3] from the `BSDCoreUtils` [42] collection of system utilities and 2 libraries, i.e., `shoco` and `urlparser`.

All of the chosen programs (Table I) are self-contained, requiring only the C standard library as a dependency. Four programs—`csplit`, `fmt`, `join`, and `test`—perform file processing. The `expr` and `printf` are pure computation utilities for strings and numbers. The `shoco` library is for data compression, and the `urlparser` library parses URL strings.

Task Requirements. Each individual participant is tasked to translate the assigned C program into safe Rust, satisfying two requirements, i.e., *safety* and *functional correctness*.

1) **Safety.** The translated program must be written in safe Rust only. The use of keyword `unsafe` is strictly forbidden. For dependencies, only the Rust standard library should be used by default. When that is insufficient to implement certain functionalities, additional third-party dependencies can be used if they are well-maintained.
2) **Functional correctness.** The translated Rust program should have the equivalent external behavior to the C source program. Since most of the chosen C programs do not come with high-quality unit tests by default, we asked the participants to write their own tests with line coverage aiming for at least 85% on the original C program to test correctness. If the source program and the translation behave the same (i.e., output, return code, effects on the file system, etc.) on those tests, we say that the translated program is *correct* if it passes all test cases created by users. Besides the final version of their translated program, we also asked the participants to submit an *initial version* of the translation that could compile before they developed tests, so we could analyze the changes.

Collected Translations. The 33 participants provided 31 *final* translations that can compile and achieved 70% to 98% line coverage on the C program under their respective tests. 26 of the 31 translations pass tests that reach 85% coverage and nearly all of them[4] have coverage above 80%. All of the 31 final translations are written in pure *safe* Rust, without the use of `unsafe`. They form the main target of our analysis. 17 participants also submitted their compilable *initial* versions.

Our Goal. We analyze these 31 final translations to gain insights into the following research questions:
RQ1. Is there a set of common strategies that users used for successful translation?
RQ2. How is the security-performance trade-off in the Rust translation?
RQ3. What are the common errors and correctness gaps in translated Rust programs?
RQ4. How do the state-of-the-art automatic C to Rust translation tools perform on the same task?

IV. High-Level Approach Taken by Users

[1]The content described in Section II-A with example exercises constitutes the introduction given, after the participants are familiar with C memory errors.
[2]The participants conducted their work in April 2024.
[3]Obtained from the `BSDCoreUtils` [42] code repository (version d2b28e0).
[4]Except for 2 translations with coverage less than 80%.

```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);
    } ...
}
```

Example C Program (above) → c2rust Translation (below)

```rust
unsafe extern "C" fn center_stream(
  mut stream: *mut FILE, mut name: *const libc::c_char)
{
    let mut p1: *mut libc::c_char = 0 as *mut libc::c_char;
    let mut p2: *mut libc::c_char = 0 as *mut libc::c_char;
    let mut wc: wchar_t = 0; // ...
    loop {
        p1 = get_line(stream);
        if p1.is_null() { break; }
        len = 0 as libc::c_int as size_t;
        p2 = p1;
        while *p2 as libc::c_int != '\0' as i32 {
            if *p2 as libc::c_int == '\t' as i32 {
                *p2 = ' ' as i32 as libc::c_char; }
            // ...
            if len == 0 ... && iswspace(wc as wint_t) != 0 {
                p1 = p1.offset(w2 as isize);
            } else {  len = ...;  }
            p2 = p2.offset(w2 as isize);
        }
        while len < goal_length {
            putchar(' ' as i32);
            len = (len as libc::c_ulong).wrapping_add(2) ...;
        }
        puts(p1);
    } ...
}
```

Fig. 2: (Top) An example of C program with aliasing pointers, i.e., multiple pointers pointing to the same region of memory. (Bottom) A line-by-line translation by `c2rust`. Uses of unsafe raw pointers and `libc` APIs/types are highlighted in red.

Fig. 2 shows an example of C code from the `fmt` benchmark and the translation produced by the `c2rust` tool. The latter is created by preserving the structure of the original code, line-by-line and variable-to-variable, but contains unsafe Rust code blocks. The main challenge is to refactor the unsafe code to safe Rust. A straightforward removal of the keyword `unsafe` does not work as there are two immediate challenges:

a) the unsafe Rust code, much like the original C code, uses low-level raw pointers and C library calls that are disallowed in safe Rust; and
b) even if there exists a way to replace all pointers with references, the resulting code will violate safe Rust rules.

In fact, for our shown example, there is no translation that keeps the original control-flow structure as-is and maps all the original C pointers to Rust references. This is because preserving the original lifetime and read/write semantics of the C variables in the Rust code will always violate the AXM principle, since `p2` needs to be a mutable reference and remain alive from Lines 8-14, and that interferes with the use of `p1`.

Despite this apparent challenge, there are ways out of the quandary, and our participants translated such programs correctly. We explain what strategies they used and how often.

```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());
    } ...
```

Fig. 3: A safe Rust translation of the C program (Version A)

TABLE II: Dissimilarities between source C programs and Rust translations. Numbers are averaged over all translations.

Benchmark	#user-translations	#function (kept/rm./add.)	#signature (sim./dissim.)	#pointer decl. (kept/rm.)
csplit	5	7 / 2 / 3	5 / 2	54% / 46%
expr	2	8 / 10 / 8	4 / 4	61% / 39%
fmt	4	10 / 2 / 2	6 / 5	82% / 18%
join	2	6 / 6 / 11	4 / 2	70% / 30%
printf	3	11 / 2 / 4	5 / 6	71% / 29%
test	4	16 / 2 / 3	10 / 6	89% / 11%
shoco	6	6 / 0 / 2	2 / 5	72% / 28%
urlparser	5	18 / 3 / 3	3 / 15	79% / 21%

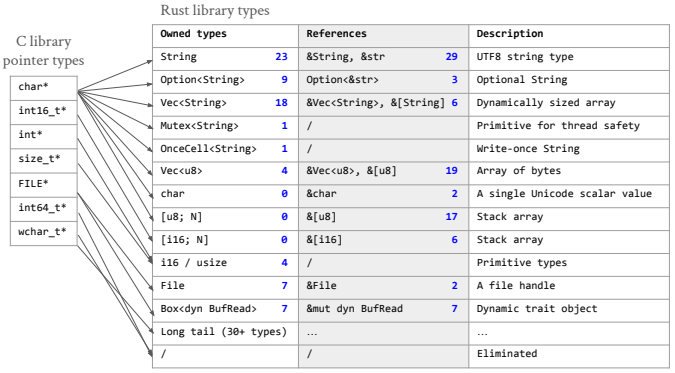


Fig. 4: Raw C pointers to Rust data types lifted in translations and the number of programs using each Rust type (in blue).

A. *Semantic Data Type Lifting*

Several users recognize that the low-level `char*` pointers used in the example C code are semantically operating on a higher-level data type and they *lift* the object's type to a relevant Rust abstract data type. Figure 3 shows one such successful translation that lifts the `char*` buffer in C to `String` type. The C code with library calls is replaced with invocations to the Rust `String` methods. For example, the C code in Lines 6-20, which is responsible for counting the whitespace characters and center-justifying the string, is implemented with safe Rust `String` methods in the translation. The result of

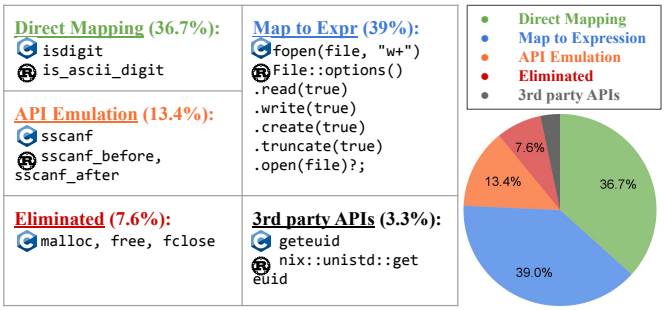


Fig. 5: Breakdown of C library API translations with examples.

```rust
fn center_stream<R: BufRead>(mut stream: R, name: &str) {
    // ...
    while let p1: Vec<char> = get_line(stream) {
        for c in p1 {  // implicit mutable borrow by the loop
            if c == '\t' {
                c = ' ';
            } ...
            if (l == 0 && isWhiteSpace(wc))
                p1.drain(..wcl); // AXM violation (FAIL!)
            // ...
        } ...
        println!("{:?}", p1);
    } ...
```

Fig. 6: A Line-by-line Rust translation that fails to compile.

such type lifting is that the Rust code is less similar to the original C code in the control-flow structure and variables used. Table II summarizes the amount of dissimilarity we observe in 31 final translations and their corresponding C code.

Multiple choices for the Rust data types exist here. For example, `char*` may be lifted to Rust `String`, `Vec<u8>`, `Box<[u8]>`, and so on. Figure 4 shows the top Rust data types that our participants lifted from C pointers across 8 benchmarks in their final 31 translations. Note that there are subtle differences between the original C type and the corresponding lifted data type that the user chose. In Version A, the code uses a `String` type, which does not support strings with invalid UTF-8 characters (unlike the original C code). The corresponding Rust translation elides that particular behavior present in the original C code. Section VI quantitatively analyzes such semantic discrepancies in more detail.

A different sub-challenge with type lifting is how to translate operations on the original data type, which is often implemented with standard C library API, to corresponding Rust data type methods. Foreign function interface (or FFI) calls to C libraries are disallowed in safe Rust. User translations used the strategies below to translate C library API calls:

(a) *Direct Mapping to Safe Rust APIs.* 36.7% of the API calls are translated to single safe Rust method calls.
(b) *Translating to expressions.* 39% of the API calls are translated into expressions that make use of multiple Rust methods and operators to achieve similar effects.
(c) *API Emulation.* 13.4% of the API calls are translated into calls to user-implemented functions that emulate them.
(d) *Elimination.* 7.6% of the calls for manual memory and resource management are eliminated since Rust data types can automatically achieve similar effects.
(e) *Third-party Libraries.* 3.3% of the calls are translated to API calls provided by third-party libraries (crates).

The breakdown of these choices with examples is summarized in Figure 5. As a result of the above challenges, we observed that the initial guess of the lifted data type chosen by a user was often not optimal. Of the 17 participants who shared their initial version of Rust code, 10 refined or changed their initial types chosen to other ones in the final version.

User-provided translations break away from the low-level structure of the original C code by semantically mapping C data types and APIs to similar ones in Rust. They often refine their initial guess of such C to Rust mappings.

B. Dealing with Aliasing

Fig. 6 shows another example of translation wherein, after data types have been lifted, the Rust compiler checks are violated. This example illustrates a separate challenge: Users have to decide how each one of the original C pointers should be mapped to Rust references while satisfying Rust borrowing rules in the presence of aliases (borrows). Rust provides the option to use reference-counted dynamic references, which have accompanying runtime costs, but we see that users did not use them for heap and stack data references. Instead, users found 2 strategies, specialized to the access patterns used by the C program, that satisfy static checks of the Rust compiler.

Strategy (a): Elision. When objects are lifted from C to Rust, many of the original C pointers do not need to be mapped to Rust references and can be elided in the translated code. For example, code in Version A (Figure 3) elides the C pointer `p2` because its functionality is encapsulated by Rust `String` methods. Version A satisfies the Rust compiler checks.

Strategy (b): Cloning. The translation shown in Version B (Figure 7) embodies a different strategy. It satisfies the Rust borrow checking rules by separating the read and write accesses of the original object into two separate objects. In Figure 6, the `p1` reference and the iterator are simultaneously trying to modify the object, hence violating the AXM principle. In Version B, however, a new copy of the original `String` object is made. The `p1` reference can thus remain immutable and refer to the original object, while all writes are made to the copy by the mutable reference `ans`. Later on, only the writable copy of the object is used. Such a rewrite mechanism satisfies the borrow rules in safe Rust and compiles successfully.

Two pointers to a read-only object can both access the object without any restriction in C. But in Rust, if one reference is (re)borrowed from the other, it should have a statically determined shorter lifetime than that of the other, which is more restrictive than C. The cloning strategy is thus also useful when translating C code with multiple immutable pointers. It disentangles the lifetime of two immutable pointers referring to an object by making a copy of it with the same value.

How often are the above strategies used? *Elision* is the most frequently used specialization strategy when translating code fragments involving aliasing references. We find that it is used in 25 final translations. *Cloning* is used in 13 translations.

```rust
fn center_stream<T: BufRead>(&mut self, stream: T, name: &str) {
    for line in stream.lines() {
        let p1 = match line {
            Ok(line) => line,  Err(e) => { ... }
        };
        let mut ans = String::new();
        let mut len = 0;
        for c in p1.chars() {
            // ... if c is a space, skip
            if c == '\t' {
                ans.push(' ');
            } else {
                ans.push(c);
            }
            len += c.width().unwrap_or(1);
        }
        println!("{:>width$}", ans, width = ...);
    ...
```

Fig. 7: Possible translations of the C program (Version B)

TABLE III: Temporal and spatial memory safety of data references used in Rust translations.

References	Fraction	Temporal Safety		Spatial Safety	
		static	dynamic	static	dynamic
Owning	**49.2%**	**95.6%**	**4.4%**	**1.6%**	**98.4%**
- stack	14.2%	100.0%	0.0%	12.7%	87.3%
- heap	31.4%	100.0%	0.0%	0.0%	100.0%
- global	4.4%	45.9%	54.1%	0.0%	100.0%
Borrowing	**50.8%**	**99.7%**	**0.3%**	**9.1%**	**90.9%**
- mut.	11.6%	98.6%	1.4%	21.2%	78.8%
- immut.	39.2%	100.0%	0.0%	5.5%	94.5%
Nullable	**8.3%**	**100.0%**	**0.0%**	**10.8%**	**89.2%**
DST	**79.7%**	**99.6%**	**0.4%**	**0.0%**	**100.0%**
- string	62.5%	99.7%	0.3%	0.0%	100.0%
- buffer	16.4%	99.0%	1.0%	0.0%	100.0%
- poly.	0.8%	100.0%	0.0%	0.0%	100.0%

Rust translations provided by users choose specialized strategies to satisfy static Rust safety rules, rather than resort to dynamic references (ref-counted), to handle aliasing.

V. Security and Performance of User Translations

One of the most important motivations for translating C to Rust is to guarantee full memory safety without sacrificing performance. Some Rust safety abstractions are completely static, thereby having zero runtime costs, while others employ runtime checks. We analyze the usage of Rust abstractions in the translations obtained in our study to understand how often zero-cost safety abstractions are used. We then measure end-to-end performance of the Rust translations. We also highlight prominent examples of code patterns known to be dangerous in C. These are forbidden in safe Rust, and we explain how they were translated to safe Rust code by our participants.

A. Breakdown of Safe Abstractions Used

Rust offers smart references (pointers) and safe data type abstractions which can replace raw pointers in C. We analyze (1) how often the Rust translations use those data type abstractions that encapsulate pointers, and (2) whether the memory safety properties on those types are enforced at compile time (statically) or run time (dynamically).

Breakdown of Different Types of References. Table III summarizes the different types of smart references and data types that our users used, along with their frequency of usage. There are 1261 explicitly declared reference-like variables in the 31 final Rust translations in total. Those variables can be classified into either owning references (49.2%)[5] or borrowing references (50.8%). The owning references consist of references of stack data (14.2%), heap data (31.4%), and global data (4.4%). Among borrowed references, 11.6% are mutable references, and 39.2% are immutable references.

Spatial and Temporal Safety of References. All the references in safe Rust are strongly typed so that the Rust compiler can enforce certain safety invariants when using those types. For temporal safety guarantees, most of the owning references (95.6%) and borrowing references (99.7%) are compile-time checked, which are *statically* proven to be free of temporal memory errors. This includes almost all of the stack and heap data references. The remaining are dynamic references, which involve either partial or full dynamic checks. Typical types in the translations include `OnceCell<T>` (runtime check on the first write) and `Mutex<T>` (check on every code region of access). Most of the dynamic references are for global variables. For spatial safety, most references used (90.9%) may require runtime checks on access.

Two sub-categories of references, i.e., nullable references (8.3%) and references to dynamically-sized types (79.7%), affect abort handling and performance and are worth mentioning.

Nullable References. Nullable references are typically represented by `Option<T>`, which piggybacks on static type safety to separate the case where a reference is `NULL` from when it is not. It is often a zero-cost abstraction when `T` is a non-null reference, while preventing ungraceful aborts. Usage of nullable references forces the developer to specify how the code should handle null pointer deference, preventing the software from aborting ungracefully when memory safety is violated. If the user does not want to specify how such exceptions should be handled, Rust gives a default way in which the compiler inserts null checks. They result in runtime panic on safety-violating inputs. It is explicitly reflected in the syntax (e.g., `unwrap(..)`). In the user translations, 63.1% of the accesses use such default null checks that may cause runtime panic on null pointer access, while the other 37% use of `Option<T>` are *panic-free* (not raising runtime aborts).

Dynamically Sized Types. Dynamically sized types (DSTs) are useful to support strings, buffers, as well as runtime polymorphism (e.g., `Box<dyn T>`) [4]. References of DSTs are typically "fat" pointers that store additional information to facilitates dynamic checks for spatial safety. In our Rust translations, 79.7% of the variables are DST references.[6]

Temporal safety is achieved mostly statically (95.6%), whereas spatial safety is mostly through runtime checks.

Case Studies: Known Vulnerabilities Eliminated. It is evident, even in our small-scale user study, that C to Rust translation directly addresses the root cause of memory safety vulnerabilities, namely insecure coding practices. 2 out of 8

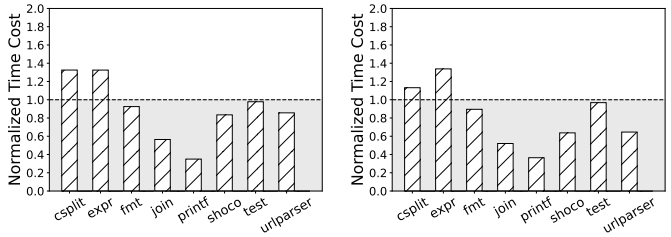
[5]We consider smart pointers (e.g., `String` and `Vec`) as owning references.

[6]We count `String` and `Vec` as "references" to dynamically-sized types as well, even though those smart pointers themselves have fixed size.

of our benchmarks, `shoco` and `urlparser`, have 3 known vulnerabilities in our chosen versions. The data compression library `shoco` has one spatial memory vulnerability (CVE-2017-11367) on the access of an array called `packs`. All users who translated this benchmark eliminated the spatial error and the out-of-bound is caught at runtime. Similarly, a heap-buffer overflow on a string buffer in `urlparser` is caught at runtime as well. The C `char*` pointers pointing to the string are lifted into `String`, `&str`, or `&String` with spatial safety guarantees. For temporal safety, there is a use-after-free (UAF) vulnerability [7] in the `urlparser` C program when the input string to the parser does not live long enough before calling certain library APIs such as `url_data_inspect`. This bug is statically eliminated as the lifetime and borrowing rules in Rust forbid such code patterns. All participants created a copy of the borrowed input string that needed to live longer than the original, thereby eliminating the vulnerable pattern in the C code. More details on the case studies are in the Appendix A.

All the known memory safety vulnerabilities in C programs are eliminated in each one of the Rust translations.

B. Performance Comparison



(a) Compiled with `-O2` (b) Compiled with `-O3`

Fig. 8: Running time of the most similar Rust translation compared to the original C at different optimization levels. The time for the baseline C (dashed line) is normalized to 1.

We now turn to a comparison of performance overheads introduced when one migrates the same C code to Rust. An issue here is that our user-provided translations do not have the exact same behavior as the original C code on all inputs. Furthermore, there are multiple candidate translations for each of 8 C benchmarks to compare with, each with differences from the original C code. To deal with these, we choose the Rust program which is most similar to the corresponding C program on tests. Specifically, we select the Rust translation with the highest test coverage as the main comparison baseline, tie-breaking in favor of the shorter translation. All the tests provided by all the participants for the C benchmarks are considered when computing the (line) coverage. This gives us 8 Rust translations, one for each C benchmark.

We report performance only on test cases for which the Rust code and corresponding C have the same output and both exit normally[8], thereby eliminating unfair comparisons.

The running time is calculated between the entry and exit of the `main` function of the programs to exclude differences in the load and initialization time[9]. We report the average over 70 repeated trials, removing the first 10 rounds, which warm up hardware and OS caches, as well as the fastest and slowest 10 rounds of the remaining. The ratio of running time Rust compared to the corresponding C is given in Fig. 8. We have that for 5 out of the 8 C benchmarks, their most similar Rust translations run faster. 6 out of 8 have corresponding Rust translations well below 10% overheads, while the remaining 2 have about 10% and 40% overhead at `-O3` optimization level.

It is worth noting that our performance analysis is post-hoc. We did not ask users to measure or optimize for performance. We only specified correctness and safety as the objectives.

For Rust translations most similar to the original C code, the overhead is mostly within 20% and Rust is often faster.

C. Examples of Security-Enhancing Code Patterns

Certain code patterns in C are considered unsafe and are forbidden in safe Rust. However, there are often safe abstractions in Rust designed to achieve similar goals at low cost. We focus on two kinds of patterns present in our C programs, including mutable globals and unions, to analyze how users translate such code patterns into safe Rust.

Mutable Globals. Mutable global variables in C are prone to be corrupted and exploited in real-world attacks due to their extensive lifetime across many functions and relatively predictable address [35], [37]. Besides being a convenient target for attackers, mutable globals are also sources of various hard-to-find temporal memory errors, and are thus discouraged by various coding guidelines [1], [5], [7]. Safe Rust forbids plain mutable globals because they violate the AXM principle even in single-threaded programs. Otherwise, different call frames can obtain mutable references to globals with overlapping lifetimes, leading to memory errors and incorrect optimizations in Rust. To translate C code using globals into safe Rust, we see that users devised several strategies using safe Rust abstractions, which are listed below:

A) *From global to locals.* A performant strategy is to identify where the global is used first and last in the program. Then, one can replace the original global variable with a local (stack or heap) object spanning the lifetime of actual use. A reference can be passed into functions that need them. When multiple mutable globals are transformed this way, they can be grouped into a single `struct` object for better performance and to reduce code bloat. One reference to the grouped `struct` object with the original globals as fields at different offsets is sufficient. This optimizes for performance as it lowers the cost of parameter passing by reference for many moved globals with similar lifetimes of use in the original C code.

B) *Dynamic references.* The fallback strategy is to create dynamic references, declared either as thread locals or true globals in Rust. For thread-local variables, single-thread synchronization (e.g., `RefCell`) is still needed [11], but multi-thread synchronization (e.g., `Mutex`) is not necessary. Such references incur some runtime costs.

[7]The fix commit changed the ownership from borrowing (line 100 of the left side) to owning (line 188-192 of the right side) in `url.c`.

[8]We do not measure performance on unit tests for error handling.

[9]We also measure end-to-end performance. Details are in the Appendix C.

C) *Atomic integer types.* For mutable global variables that are integers, dynamic references are not needed if declaring them as `Atomic`. For `Atomic` types, special APIs can mutate their values without mutable references to them.

19 participants chose to move the globals to locally referenced objects. 11 of them also grouped multiple globals into an aggregate `struct` object. 3 users kept mutable globals as globals or thread-local variables through dynamic references. 4 used atomics for mutable globals. Code examples illustrating each strategy are shown in the Appendix in the extended version of this paper [39].

C Unions. Unions in C allow overlapping objects of incompatible types. However, programmers are responsible for ensuring that operations on unions are type safe. Otherwise, type safety violations can lead to both spatial and temporal memory errors [21]. In safe Rust, unions are not allowed, but various abstractions can be used to achieve similar functionality. 2 of the 8 C programs in our benchmarks use unions. Users translated C unions depending on different use cases:

A) *Zero-cost type casting (type punning [12]).* The string compression library `shoco` uses union to access integers as raw bytes. Its behavior depends on the endianness of the target architecture. Some users translate such code into safe Rust API calls (`to(from)_le(be)_bytes`) that convert between primitives and constant-size byte arrays. Note that the conversion is zero-cost since those APIs will be optimized away during compilation. They can also be used together with conditional compilation supported in safe Rust to match the endianness of the architecture. Examples are provided in Appendix D in the extended version [39] of this paper.
B) *Sum type (variant record).* Another C program `expr` uses unions to implement variant records (sum types). For such use cases, some users translate them into `enum` and access them using statically-checked `match` statements. Rust `enum` is memory efficient, and the memory occupied by a Rust `enum` depends on its largest discriminant, similar to unions in C. Please see Appendix D in the extended version [39] for examples.

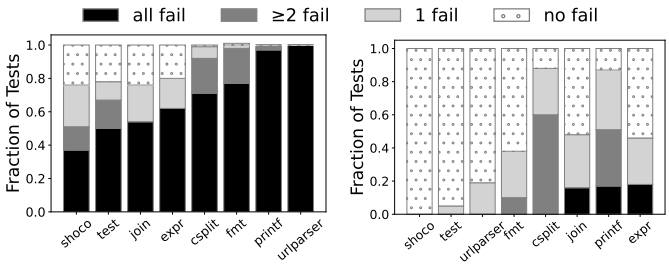
For relevant use cases of unions, 3 users used zero-cost type casting APIs in safe Rust, and 2 users used Rust enums. Another 3 users did not use those abstractions but emulated the C unions using structs and methods with higher overhead.

Users often translate known unsafe C code patterns to equivalent statically-checked low-cost safe Rust code.

VI. The Gap In Functional Correctness

As mentioned in Section III, each participant submitted their final translation and tested it with test cases they created. Most participants created tests that covered more than 85% of the original C program and reported that their final Rust translations passed their tests. We analyze how many behavioral differences are missed by tests self-created by participants.

We employ automated fuzz-testing to check for behavioral differences between Rust and the corresponding C source. We use `AFL++` [26] to generate tests for each of the 8 C programs for 1 hour per program. We then sample 300 distinct tests



(a) Greybox fuzzing tests (b) User-provided tests

Fig. 9: Rust translations of the same C program all fail on a large fraction of fuzz tests (Subfig (a)) and all pass on a large fraction of user-provided tests (Subfig(b)).

per program, except for programs with less than 300 tests available, in which case we took all tests[10]. On these tests, we compare each of the 31 Rust programs to its corresponding C program. We omit minor non-semantic discrepancies such as the error message format when computing the difference.

We find that *none of the 31 translations is fully equivalent* to the original C code. Fig. 9 shows that 37%-100% (68% on average) of the fuzz tests exhibit a discrepancy across different Rust translations of the same C program.

Many translations fail the same way. Given that all the translations have behavioral differences from the corresponding C code, we examine whether their failures are correlated. We count how many translations for the same C program fail on the *same* test case. Fig. 9 shows that nearly all Rust translations fail on about the same 38% tests on `shoco`, the same 50% on `test`, and so on. We investigated some of the tests that all translations failed, finding that they are often corner cases involving C library calls that are tedious to emulate in safe Rust, and hence missed by all translations. An illustrative example is provided in the Appendix B for interested readers.

Breakdown of Behavioral Differences. The behavioral differences can be classified into three categories, including:

A) *I/O encoding/decoding errors.* Standard Rust APIs have different I/O formatting and encoding behavior from similar C APIs. For example, the input arguments and string APIs often assume UTF-8 encoding, which might abort the program if the string buffer is not valid UTF-8.
B) *Runtime safety aborts.* The Rust has runtime checks about spatial errors and null pointer dereferences that safely abort, while C may continue arbitrarily and crash.
C) *Logical differences.* Both C and Rust can finish the execution and normally exit, but the output is different.

Fig. 10 shows the distribution of category of differences for each of the 31 translations. Nearly all translations (27/31) behave differently on more than half of the newly created tests from fuzzing. Logical differences (category C) exist across all translations, highlighting the gap in functional equivalence. I/O encoding/decoding errors (category A) are frequent in general, while safety-violating inputs (category B) are more frequent in

[10]3 programs had fewer than 300 tests: `shoco` (200 tests), `csplit` (211 tests), and `urlparser` (22 tests)

Fig. 10: Behavioral differences across 31 translations with tests from greybox fuzzing. Legends from left to right mean "*equivalent behaviors*" (green), "*I/O enc. errors*" (gray), "*Runtime safety aborts*" (brown), and "*Logical differences*" (red).

translations of `shoco` and `urlparser`. Note that Category B cases do not lead to undefined behavior in Rust due to runtime safety checks, but exhibit behavioral differences to C. These differences are to be expected—Rust is eliminating potentially unsafe (undefined) behavior from untamed C code.

Differences in I/O. Although the gap between the translations and full functional equivalence is non-negligible and common, we find that some of the differences may not matter for many intended usage scenarios. One example is that the differences in I/O encoding/decoding behaviors are a major source of behavioral differences, as shown in the gray bars in Fig. 10. Such differences may not matter if the intended inputs to the programs are valid UTF-8 strings. The UTF-8 encoding is widely assumed in the Rust standard library on string-related APIs. Those APIs provide convenient ways to work with UTF-8 encoded strings compared with other encodings. If the intended use case requires processing other encodings, the same data types and APIs may no longer be useful. Rewriting of related API functions or external emulation would be necessary, leading to a more verbose translation.

Panic-free Translations. 7 out of 8 C benchmarks have at least one user-provided Rust translation that is panic-free[11]. The `urlparser` benchmark is an example where we have 2 translations that do not have any runtime safety aborts (brown bar in Fig. 10), while the remaining 3 translations have panics. On the contrary, the 2 panic-free translations avoid all use of `Option<T>::unwrap()`. These differences across Rust translations are only artifacts of how different users handle safety aborts, arising out of undefined behavior in C.

None of the Rust translations is fully equivalent to the corresponding C, but some differences are due to Rust programmers handling safety aborts differently. Several logical differences and incompatibilities in the external I/O remain.

VII. Efficacy of State-of-the-art Automatic Tools

We evaluate 4 state-of-the-art C to Rust translation tools on our C benchmarks to analyze whether their approach is effective and, when not, where the immediate gaps are.

A. Compiler-based Tools

Both Laertes and Crown are compiler-based systems that post-process the unsafe Rust code produced by the `c2rust` compiler [33]. Our results, shown in Table IV, confirm that while both tools can often produce Rust programs that compile, the generated code is still largely unsafe and far from safe Rust. We summarize their missing features next.

Fraction of Raw Pointers. The fraction of raw pointers that can be lifted by Laertes and Crown is limited. For Laertes, Most of the data references (95.3% on average) are still unsafe raw pointers[12]. For Crown, the majority of the references are also raw pointers, and safe Rust references account for less than 3.1% averaged over all the benchmarks.

Lifted Safe Rust Data Types. Both of the translation tools have limited support for higher-level data types. Both Laertes and Crown are limited to three types of Rust references that are closely related to C pointers. The three types include `Option<&T>` (similar to `const T*` in C), `Option<&mut T>` (similar to `T*`), and `Option<Box<T>>` (owning `T*`) where `T` is a C-compatible type. Most of the higher-level smart references that participants frequently use are out of the scopes of those tools, including `String`, `&str`, `Vec<T>`, `BufRead`, and so on, which are shown in Fig. 4.

Handling of C Library Calls. C library calls turn into FFI calls from Rust to C in the translation of both Laertes and Crown. C library functions generally do not have direct safe Rust mappings, and the translation is often not straightforward. Calling C functions through FFI is easy, but such handling of library calls permits unsafety. Such operations are only allowed in unsafe Rust. At the same time, C APIs often require raw pointers as parameters, which may also constrain related Rust code to operate on raw pointers. In contrast, users in our study addressed the API translation problem by utilizing safe Rust code in creative ways, as shown previously in Fig. 5.

Dealing with Aliasing and Lifetimes. Aliasing references in Rust can frequently violate borrowing rules and thus can be challenging to translate. Laertes and Crown aim to lift pointers to references while preserving the low-level control flow and data flow of the C program. The strategies highlighted in our work, for instance in Section IV-B, are not used by these tools. These prior tools do not aim to eliminate unsafe C code patterns such as mutable globals or unions that may be present in the `c2rust` compiler output used as their first stage.

Memory Safety. The end result of the code produced from prior compiler-based tools might be safer than C, but is not guaranteed to have full memory safety. Recall that there are 3 known vulnerabilities in our C benchmarks. The translation resulting from these tools eliminates 1 of these vulnerabilities, a read overflow of a global array, as shown in Fig. 11 in the Appendix A. The original global array declaration in

[11] Excluding the abort due to I/O encoding/decoding errors.
[12] Excluding data references in helper functions injected by Laertes.

TABLE IV: Comparison of Existing Tools on Our Benchmarks

Prog. \ Tools	Laertes				Crown				Flourine				Vert			
	Compile	Safe Code (%)		Passed Tests	Compile	Safe Code (%)		Passed Tests	Compile	Safe Code (%)		Passed Tests	Compile	Safe Code (%)		Passed Tests
		#Line	#Ref.			#Line	#Ref.			#Line	#Ref.			#Line	#Ref.	
`csplit`	✓	1.3%	3.2%	100%	✓	1.8%	3.2%	100%	✗	100%	100%	-	✗	97.2%	100%	-
`expr`	✓	1.3%	5.0%	95.3%	✗	1.8%	2.5%	-	✗	85.3%	100%	-	✗	98.8%	93.1%	-
`fmt`	✗	1.2%	3.0%	-	✓	1.8%	3.0%	100%	✗	73.1%	100%	-	✗	60.1%	100%	-
`join`	✓	0.9%	14.3%	100%	✗	1.4%	2.4%	-	✗	74.3%	100%	-	✗	94.3%	100%	-
`printf`	✓	1.4%	3.0%	100%	✓	2.1%	3.0%	100%	✗	77.1%	100%	-	✗	95.9%	100%	-
`test`	✓	1.0%	9.5%	100%	✓	1.6%	9.5%	100%	✗	81.8%	88.2%	-	✗	62.9%	84.6%	-
`shoco`	✓	1.2%	0%	100%	✓	0.3%	0%	100%	✗	96.3%	100%	-	✗	99.4%	100%	-
`urlparser`	✓	0%	0%	100%	✓	0.2%	1.3%	100%	✗	100%	100%	-	✗	84.1%	78.6%	-

C is syntactically translated into an equivalent global array declaration in Rust. Access to such an array (rather than via a raw pointer) is bounds-checked automatically in Rust. The remaining 2 vulnerabilities persist in the Rust translation produced by both these tools, one of which is a spatial violation and the other temporal. Raw C pointers (not integers) are involved in the unsafe code in the 2 cases, and Laertes and Crown are not yet able to lift them to safe Rust references.

State-of-the-art compiler-based C to Rust tools output translations that extensively use unsafe Rust while rigidly retaining many structural similarities to the original C code.

B. LLM-based Tools

There are 2 recent tools, Flourine and Vert, that are state-of-the-art for C to Rust translation utilizing large language models (LLMs). We report on our experience in running them on our 8 benchmark programs. Flourine explores 4 test-driven repair strategies in concert with LLMs. Vert uses a different approach. It creates two translations of the given C program, one is unsafe Rust decompiled from WASM, and the other is an LLM-generated safe Rust code. Then, it uses fuzzing and, if needed, model-checking techniques to find tests that exhibit differences and subsequently run an automatic repair. The work on Vert claims that if their tool terminates normally, the resulting translation is expected to be functionally equivalent to the C code. For evaluating Flourine, we choose their most stable configuration with GPT-4[13] as the LLM backend. For evaluating Vert, we use Claude[14], which is reported to be the best-performing LLM backend for their artifact.

We observe that code generated from these LLM-based tools is more idiomatic and uses safe Rust abstractions more often than compiler-based tools, and thus can serve the goal of useful translation aids for human developers on code snippets. However, neither tool generates a runnable translation for any of the C programs we consider, as shown in Table IV. This is because both tools have certain structural assumptions on the C code given to them as input. Specifically, both tools expect that the C program can be decomposed into components such that each can be *independently* translated and tested, since present LLMs work reasonably well on small code fragments.

The decomposition of C programs into such components and merging of their Rust translations are not automated by the tools. We encountered many difficulties when trying to emulate the decomposition strategies described in their respective works. These difficulties point to a broader technical challenge that may be of independent interest for further research.

Decomposition Failures. 6 out of 8 of our C benchmarks are real-world standalone programs, and the remaining 2 are libraries. Functions in standalone programs are connected by a call graph. We find it difficult to decompose such functions into independently testable modules using the strategies described in the works of Flourine or Vert. For example, if a function f calls g, then the component containing f is required to contain g since it is a dependency. This implies that on our standalone C program benchmarks, the function `main` depends on all other functions and the component containing it includes nearly the whole program, making it too large to obtain repairable translations from LLMs. The 2 library benchmarks are marginally better since there is no `main` function, but the challenge persists in part here as well.

We note that the decomposition strategy proposed in Flourine or Vert can produce multiple translations of the same function that are inconsistent and cannot be merged into one. Say we have two components, one containing function $\{f, g\}$ and another containing function $\{h, g\}$. The two components are translated independently and thus multiple Rust translations of the function g are obtained. We often find that these translations have conflicting type definitions and incompatible types that are difficult to merge into a single translation.

We are not aware of better ways to decompose our C programs in components small enough to feed to Flourine or Vert. We fed the whole program to these tools and the translated Rust code does not compile. We then experimented with several versions manually to best split each function in a separate component, while including only a minimal number of dependencies, such that the size of component is small enough to work with LLMs. We are able to produce Rust translations reported in Table IV with some compilable components, but unable to merge them back into a single compilable program.

State-of-the-art LLM-based C to Rust translators produce idiomatic safe Rust snippets, but not safe Rust programs that compile. Existing tools share a common challenge in devising workable decomposition strategies for long C code.

C. Do existing LLMs help in user-provided translation?

We revisit whether LLMs, taken standalone, are helpful to users who followed their own translation strategies. Recall that

[13]We use the model version gpt-4-0125.

[14]We use Claude 3.5 (claude-3-5-sonnet-20240620), which is the best available version of Claude at the time of our evaluation.

we placed no restrictions on the participants to employ external tools. We asked our participants to specify which tools they used and provide qualitative feedback on their experience. In their feedback, 31 of the 33 participants reported that they tried to use LLMs for assistance. 14 users mentioned that LLMs are helpful indirectly in the translation process, including tasks such as explaining the C code and suggesting Rust data types and APIs. However, most of the participants (20/31) reported that the code generated by LLMs is error-prone and hard to debug. 2 users reported that they abandoned LLMs for direct code translation and translated manually from scratch.

VIII. Extensibility of Findings to Real-world Code

To test whether our findings generalize beyond our chosen programs or our user group, we conduct a post-hoc analysis of a mature open-source Rust project `uutils/coreutils` [15] (`uutils`), which mirrors the GNU coreutils written in C. The `uutils` project aims for compatibility with GNU coreutils and passing the same GNU test suite. The `uutils` Rust repository has 17.6*k* Github stars and 5 of its most active contributors have Rust experience of more than 2 years at the time of this writing. We examine 6 `uutils` Rust programs that share names and functionality with the 6 `BSDCoreUtils` programs in our benchmarks. Similar to our user study, we compare those Rust programs to their C counterparts in GNU coreutils.

Program Dissimilarity. Recall that the Rust translations in our user study are dissimilar from the C source (Sec. IV-A). We find that a larger dissimilarity between the structure of `uutils` Rust programs and their corresponding GNU C programs compared with our user study is observed. We manually checked 142 and 165 C and Rust functions, respectively, across the 6 programs. We were only able to find 9 pairs of functions semantically similar, and they were all relatively small.[16]

Security-Enhancing Code Patterns. The code patterns involving mutable globals and unions in C are changed in Rust programs in our user study (Sec. V-C). It is the same for the `uutils` project. The original GNU C programs also have many global variables (on average 13.2 globals per program), and the corresponding Rust programs in `uutils` avoid global variables almost completely. These Rust programs group variables into `struct`-typed objects (strategy A for globals in Sec. V-C), more often than in our study. One program `expr`, which involves unions in C, is implemented with `Enum`s in safe Rust.

Semantic Data Type Lifting. In our study, users lift raw pointers into various safe Rust data types, as shown previously in Fig. 4. We find that Rust data types used in `uutils` programs are similar to the types seen in this study. 83% of the data types in `uutils` programs also exist in translations by our users. The most frequent Rust types in `uutils` programs are string-related types such as `String` (18%), `&str` (14%), `OsString` (6%), and `OsStr` (4%). One difference is that `OsString` and `OsStr` types do not exist in translations by our users. These types bridge the gap between platform-native strings and Rust `String`s and are useful for `uutils/coreutils` as it aims to be cross-platform. Our user study only specified the requirements that the Rust code should work on Linux.

Library API Calls. Recall that in our user study, users translate many C library API calls using various strategies listed in Fig. 5. We also investigate how library API calls in GNU C coreutils are expressed in the `uutils` programs. Due to a large number of library API calls (561 in total) and large program dissimilarity, we focus on the top 10 frequently used APIs that covered 53% of all the API calls. We manually look into at least 5 call sites per API and check Rust code fragments that implement corresponding functionalities. Most API calls in C can be mapped to safe Rust API calls and expressions. Third-party crates are used for some APIs. One example is the `quote` C API that handles special characters. The Rust program implements the same functionality using `quote` API provided by the Rust crate `os_display`. We find no API calls handled using the API emulation strategy, however.

Dealing with Aliasing. Recall that users in our study use strategies including (A) *reference elision* and (B) *cloning* to deal with aliasing (Section IV-B). For the `uutils` programs, we find C code locations involving aliasing pointers and then check how a similar functionality is implemented in Rust. Due to large dissimilarities from the C source, we manually check around 20 places in C and find 4 with clear matching code fragments in Rust. Those code fragments are all examples of *reference elision*, and one worth mentioning is the mapping of a linked list in C to a `Vec` in Rust in the `csplit` program. Such type lifting eliminates code blocks with aliasing pointers that manipulate linked lists. We also find many instances of object cloning in Rust `uutils` to satisfy ownership constraints when calling functions. But none are specifically used for handling aliased references as was done in our user study.

Functional Equivalence. In Section VI, we reported that user translations are not fully equivalent to the C source and multiple differences are exposed by differential fuzzing. The phenomenon is observed in the `uutils` project, which currently passes around 80% of the GNU test suite. We find that for 5 of the 6 `uutils` programs we investigated, there are known correctness gaps with GNU Coreutils when using its test results, as reported in the `uutils` repository[17]. One program, `test`, seems to be close to functional equivalence since it passes the full GNU test suite. We further compare the Rust version of `test` to the GNU C version using differential fuzzing in a setup similar to Section VI. Among 300 fuzzer-generated tests sampled, we find 50.6% test cases exhibit different behaviors. With a deeper investigation, we find that at least 23.6% of the tests reveal non-trivial logical differences. More specifically, C and Rust programs have different results (i.e., return code) on 2.3% tests and semantically different error messages on 21.3% tests. We observe no I/O encoding errors or runtime safety aborts. We also find that the maintainers are aware of several semantic differences that need to be fixed according to comments in their additional test file.

Use of Unsafe Rust. Among the 6 Rust programs, only one program (`test`) has one line of unsafe Rust to call `libc::isatty`. This C library call checks if the open file descriptor is a terminal. It can perhaps be replaced with safe Rust APIs in third-party crates, such as `atty` or `termion`.

[15] We download a version on Sept. 26th, 2024 with commit hash a0d258d.

[16] Such large dissimilarity may be due to the fact that `uutils/coreutils` is re-implemented in Rust from scratch to avoid license issues.

[17] According to their `gnu-full-result.json` of commit a0d258d.

Most of the findings in our user study are also applicable to a Rust project mirroring GNU Coreutils written by experts.

IX. Discussion and Takeaways

Our analysis of user-provided translations highlights several points where automatic translation strategies deviate from those taken by human users. We reflect on why there is such a gap and summarize takeaways for automatic translation.

Separate Policy from Mechanism. Our observation is that there are many choices to be considered when translating a C program to Rust, i.e., there is no one-size-fits-all strategy to take. The resulting translations, while satisfying memory safety, can have varying levels of functional correctness, performance, and grace in handling runtime safety exceptions. The balance between them is a matter of *policy*. For example, the level of functional equivalence to achieve is one crucial policy decision. Full functional equivalence can require significant effort and may even go against the purpose of code migration [56]. To characterize what behaviors must be kept and what behaviors can change in the translation, we may need more thorough unit tests or other forms of specifications. Another example of an important policy decision is about the data types and APIs to use, which can lead to multiple ways of translating the same program with different trade-offs in performance, memory overhead, compatibility, and so on. In summary, users may want explicit control over policy decisions even when using automatic tools.

Improving LLM-based Search for Translations. Once policy decisions are clear, automatic mechanisms can enable search for translations. Our observation is that there are several immediate and open problems that, if addressed, can make C to Rust code automatic translator much more usable.

(1) Modeling of Data Types and APIs. So far, few of the previous work explicitly models data types and APIs in the Rust standard library, such as the ones summarized in Fig. 4. We believe it is essential to overcome the language differences between C and Rust and move away from unsafe code.
(2) Mergeable Decomposition. Finding the right way to decompose programs into smaller components is challenging but appears necessary for LLMs with limited context windows. At the same time, decomposition need not be one-shot. It is useful to explore if it is possible to incrementally transform identified components, such that a partial Rust translation can be used alongside the untranslated part of C.
(3) The Last Mile Problem. Some users reported in their (optional) qualitative feedback about their experience. 10 users reported that debugging was tedious in locating the root cause of certain semantic differences. 6 users mentioned that fixing some semantic differences involving library calls (e.g., `regex`) is not easy. 14 users mentioned that LLMs are ineffective in directly generating correct long code due to issues like type inconsistency across functions and multiple inter-related errors. 4 participants who used LLMs for assistance reported that they often needed to restart with clean context to get better output.

Promising Static or Dynamic Analysis. Based on our findings, we foresee 3 program analyses as immediately useful:

(1) Refactoring Global Variables Through Lifetime and Def-Use Analysis. Use of mutable globals differentiates C and safe Rust programs. As observed in our user study, many globals are accessed within a thread and can be moved to the heap or stack, possibly in grouped `struct`-typed objects for efficiency. Precise analysis of lifetime of globals is a promising next step.
(2) Lifting Data Types Semantically Before Lifting References. Data type analysis directly impacts how references are lifted to Rust. Code blocks and pointers in C can often be elided after type lifting, which is observed from both translations by our users and in our extensibility analysis on public code(e.g., Linked list pointer manipulations to `Vec` API calls). LLMs can be useful for pattern recognition tasks, according to qualitative feedback received from our users, and it would be promising to use them for type mapping suggestions of C objects to Rust.
(3) Lifting Unions. Automatic analysis to (a) tell apart different use cases of C unions and (b) associate tag values with valid fields for variant records can help convert unions to enums.

Threats to Validity. Our user study is with a relatively small number of users and on relatively small programs. These present limits are an artifact of time constraints and the inherent difficulty of the task at hand for users who have limited prior experience with Rust. We point out that Rust has a relatively new developer community. A survey by JetBrains in 2023 involving over 26*k* developers worldwide reports the majority of Rust developers (56%) have less than 6 months of experience [3]. To partially assess which of our findings extend to code written by more experienced developers, we reported on our post-hoc analysis on more mature real-world Rust programs mirroring GNU C coreutils in Section VIII. Most of the key findings we highlight extend beyond our users.

X. Related Work

In practice, many motivations for code migration to new languages exist, such as legacy code modernization, compatibility improvement with newer platforms, N-versioning for fault tolerance, and so on. Memory safety is a unique driver for C to Rust translation. For example, Mozilla Firefox is actively migrating components written in C/C++ to Rust with memory safety as a key consideration [8]. Towards memory safety as a goal, a long line of memory defenses has been investigated. They achieve different trade-offs between performance, compatibility, and security.

Memory Safety in C/C++. SoftBound [44] and CETS [45] are compiler-based solutions that provide full memory safety in C. The performance overheads of enforcing full memory safety in software-based approaches are often reported to be higher than 50%. In recent years, specialized hardware features have emerged to accelerate both spatial [40], [48], [57], [60] and temporal checks [14], [60], but overheads below $10-20\%$ for full safety appear elusive. Runtime defenses for spatial safety can be acceptably low [30], but complete temporal safety still bears a bulk of the runtime overheads. Languages like Rust can help developers explicitly manage object lifetimes in a way that eliminates temporal memory management mistakes. In our user study, we see that users make heavy use of zero-cost abstractions for temporal safety. Owing to the costs of full memory safety, partial safety defenses have been investigated and have a rich history [52]. Prominent among these are CFI [54], ASLR [53], stack canaries [20], guard pages [17], and DEP [55] which have found wide deployments. These defenses have good performance characteristics but do

not rule out all memory safety errors. For example, exploits that bypass these deployed defenses without violating control-flow properties [6], [31] are known [13].

Another approach is based on proactive discovery of bugs and subsequent automatic repair at the source C code level. Greybox fuzzing [26], [29], symbolic execution [16], [49], and their combination [43] for finding security vulnerabilities is an active area of research. Automatic localization of buggy code [32], [50] and generation of suggestions for fixes are being actively explored [28], [51]. This find-and-fix approach offers a continual process to improve software quality and reduce patching effort once flaws are discovered.

Memory Safe Dialects for C. Writing code that is free of memory safety bugs is a desirable goal. Several works focus on finding language subsets or extensions of C/C++ that are easier to statically analyze and dynamically retrofit safety checks than untamed C. CCured explored spatial safety via "fat" pointers and relies on garbage collection for temporal memory safety [47]. Xu et. al achieved temporal memory safety using a global capability store instead of a garbage collector [58]. Cyclone has similar spatial safety mechanisms as CCured but uses memory regions for temporal safety [34]. Flow-sensitive type qualifier analysis can leverage user-provided type annotations for static analysis [27]. More recently, work on the Checked C language [22] aims to enable mixed legacy C pointers with safe ones for incremental migration, by allowing parts of the code to be type-annotated and proven safe. However, it does not provide full memory safety. Efforts to dispatch more spatial safety statically at compile-time are underway, which can further reduce costs of spatial safety [19], [38], [41]. There are also efforts to introduce temporal safety into Checked C with runtime overheads of about 30% or more [62]. Overall, while designing safe C dialects continues to be a promising endeavor, low-overhead designs that eliminate all memory safety bugs have yet to be found.

Translating C to Rust. Safe Rust abstractions force developers to move away from raw C pointers. The abstractions offered by safe Rust—lifetime [9], ownership [18], and the AXM principle [9]—force a significant departure from untamed C or C dialects in how programmers write code. In the previous sections, we have compared various previous work [24], [25], [59], [61], and scalable solution is still an open challenge.

XI. Conclusion

We have presented the lessons learned from a user study on how users can translate C programs to safe Rust, with good performance and security gains. Our analysis reveals that they share a high-level approach and common specialized strategies to overcome the challenges that encumber automatic tools. Zero-cost abstractions are ubiquitously used, significantly reducing the costs of runtime temporal safety checks, which highlights why Rust offers a promising road ahead.

Acknowledgment

We thank all the participants in this user study. We also thank the anonymous reviewers and our shepherd for giving us valuable feedback on an earlier draft of this paper. This research is supported in part by the research funds of the Crystal Centre at the National University of Singapore and the Cisco University Research Program Fund, a corporate advised fund of Silicon Valley Community Foundation.

References

[1] "C++ core guidelines." [Online]. Available: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#i2-avoid-non-const-global-variables

[2] "A command-line benchmarking tool." [Online]. Available: https://github.com/sharkdp/hyperfine

[3] "Developer echosystem 2023: Rust." [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2023/rust/

[4] "Dynamically sized types." [Online]. Available: https://doc.rust-lang.org/reference/dynamically-sized-types.html

[5] "Google c++ style guide: Static and global variables." [Online]. Available: https://google.github.io/styleguide/cppguide.html#Static_and_Global_Variables

[6] "The heartblead bug (cve-2014-0160)." [Online]. Available: https://heartbleed.com/

[7] "Linux kernel coding style." [Online]. Available: https://www.kernel.org/doc/html/next/process/coding-style.html

[8] "Oxidation." [Online]. Available: https://wiki.mozilla.org/Oxidation

[9] "References and borrowing - the rust programming language." [Online]. Available: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

[10] "Thread local storage (rust)." [Online]. Available: https://doc.rust-lang.org/src/std/thread/local.rs.html#37

[11] "Thread local storage (rust standard library)." [Online]. Available: https://doc.rust-lang.org/src/std/thread/local.rs.html

[12] "Type punning." [Online]. Available: https://en.wikipedia.org/wiki/Type_punning

[13] ""the web/local" boundary is fuzzy: A security study of chrome's process-based sandboxing," *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:7573477

[14] "Arm MTE architecture: Enhancing memory safety," 8 2019. [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety

[15] "Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code," Jan. 2024, arXiv:2308.03109 [cs]. [Online]. Available: http://arxiv.org/abs/2308.03109

[16] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Symposium on Operating Systems Design and Implementation*, 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:2520229

[17] T. cker Chiueh and F.-H. Hsu, "Rad: a compile-time solution to buffer overflow attacks," *Proceedings 21st International Conference on Distributed Computing Systems*, pp. 409–417, 2001. [Online]. Available: https://api.semanticscholar.org/CorpusID:32026510

[18] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad, "Ownership types: A survey," in *Aliasing in Object-Oriented Programming*, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:15940253

[19] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, "Dependent types for low-level programming," in *Proceedings of the 16th European Symposium on Programming*, ser. ESOP'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 520–535.

[20] C. Cowan, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security Symposium*, 1998. [Online]. Available: https://api.semanticscholar.org/CorpusID:2358856

[21] G. J. Duck and R. H. C. Yap, "Effectivesan: type and memory error detection using dynamically typed c/c++," *SIGPLAN Not.*, vol. 53, no. 4, p. 181–195, Jun. 2018. [Online]. Available: https://doi.org/10.1145/3296979.3192388

[22] A. S. Elliott, A. Ruef, M. W. Hicks, and D. Tarditi, "Checked c: Making c safe by extension," *2018 IEEE Cybersecurity Development (SecDev)*, pp. 53–60, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:53413266

[23] M. Emre, P. Boyland, A. Parekh, R. Schroeder, K. Dewey, and B. Hardekopf, "Aliasing limits on translating c to safe rust," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 551–579, 2023.

[24] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating c to safer rust," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–29, 2021.

[25] H. F. Eniser, H. Zhang, C. David, M. Wang, B. Paulsen, J. Dodds, and D. Kroening, "Towards translating real-world code with llms: A study of translating to rust," *arXiv preprint arXiv:2405.11514*, 2024.

[26] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, ser. WOOT'20. USA: USENIX Association, 2020.

[27] J. S. Foster, T. Terauchi, and A. Aiken, "Flow-sensitive type qualifiers," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002, pp. 1–12.

[28] X. Gao, Y. Noller, and A. Roychoudhury, "Program repair," 2022. [Online]. Available: https://arxiv.org/abs/2211.12787

[29] Google, "Oss-fuzz vulnerabilities github repository," accessed: July, 2024. [Online]. Available: https://github.com/google/oss-fuzz-vulns

[30] F. Gorter, T. Kroes, H. Bos, and C. Giuffrida, "Sticky tags: Efficient and deterministic spatial memory error mitigation using persistent memory tags," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 217–217.

[31] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[32] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 539–554, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:106401501

[33] Immunant, "c2rust: Migrate C code to Rust," https://github.com/immunant/c2rust, accessed: July 4, 2023.

[34] T. Jim, G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *USENIX Annual Technical Conference, General Track*, 2002. [Online]. Available: https://api.semanticscholar.org/CorpusID:5958340

[35] B. Johannesmeyer, A. Slowinska, H. Bos, and C. Giuffrida, "Practical {Data-Only} attack generation," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1401–1418.

[36] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the rust programming language," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–34, 2017.

[37] G. Li, H. Zhang, J. Zhou, W. Shen, Y. Sui, and Z. Qian, "A hybrid alias analysis and its application to global variable protection in the linux kernel," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4211–4228.

[38] L. Li, Y. Liu, D. Postol, L. Lampropoulos, D. Van Horn, and M. Hicks, "A formal model of checked c," in *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*, 2022, pp. 49–63.

[39] R. Li, B. Wang, T. Li, P. Saxena, and A. Kundu, "Translating c to rust: Lessons from a user study," *arXiv preprint arXiv:2411.14174*, 2024.

[40] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "Pac it up: Towards pointer integrity using arm pointer authentication," *ArXiv*, vol. abs/1811.09189, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:53721743

[41] A. Machiry, J. Kastner, M. McCutchen, A. Eline, K. Headley, and M. W. Hicks, "C to checked c by 3c," *Proceedings of the ACM on Programming Languages*, vol. 6, pp. 1 – 29, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247748774

[42] D. Magdaleno, "Bsd coreutils is a port of many utilities from bsd to linux and macos," accessed: April, 2024. [Online]. Available: https://github.com/DiegoMagdaleno/BSDCoreUtils

[43] R. Majumdar and K. Sen, "Hybrid concolic testing," *29th International Conference on Software Engineering (ICSE'07)*, pp. 416–426, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:6760091

[44] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Softbound: highly compatible and complete spatial memory safety for c," in *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:248719

[45] ——, "Cets: compiler enforced temporal safety for c," in *International Symposium on Mathematical Morphology and Its Application to Signal and Image Processing*, 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:914358

[46] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[47] ——, "Ccured: type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, pp. 477–526, 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:8303920

[48] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel {MPK})," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 241–254.

[49] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!" in *USENIX Security Symposium*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:221178890

[50] S. Shen, A. Kolluri, Z. Dong, P. Saxena, and A. Roychoudhury, "Localizing vulnerabilities statistically from one exploit," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 537–549. [Online]. Available: https://doi.org/10.1145/3433210.3437528

[51] Y. SONG, X. GAO, W. LI, W.-N. CHIN, and A. ROYCHOUDHURY, "Provenfix: Temporal property guided program repair," 2024.

[52] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[53] T. P. Team, "Address space layout randomization." [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[54] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm," in *USENIX Security Symposium*, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:6034518

[55] A. van de Ven and I. Molnar, "Exec shield," 2004.

[56] C. Verhoef and A. Terekhov, "The realities of language conversions," *IEEE Software*, vol. 17, no. 6, pp. 111–124, Dec. 2000. [Online]. Available: http://ieeexplore.ieee.org/document/895180/

[57] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," *2015 IEEE Symposium on Security and Privacy*, pp. 20–37, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:7346980

[58] W. Xu, D. C. DuVarney, and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of c programs," in *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, 2004, pp. 117–126.

[59] A. Z. H. Yang, Y. Takashima, B. Paulsen, J. Dodds, and D. Kroening, "VERT: Verified Equivalent Rust Transpilation with Large Language Models as Few-Shot Learners," May 2024. [Online]. Available: http://arxiv.org/abs/2404.18852

[60] J. Yu, C. Watt, A. Badole, T. E. Carlson, and P. Saxena, "Capstone: A capability-based foundation for trustless secure memory access (extended version)," *ArXiv*, vol. abs/2302.13863, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257220126

[61] H. Zhang, C. David, Y. Yu, and M. Wang, "Ownership guided c to rust translation," *arXiv preprint arXiv:2303.10515*, 2023.

[62] J. Zhou, J. Criswell, and M. W. Hicks, "Fat pointers for temporal memory safety of c," *Proceedings of the ACM on Programming Languages*, vol. 7, pp. 316 – 347, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:251903483

APPENDIX

A. Case Studies with Memory Safety Vulnerabilities

Case Studies. We took a closer look at the translations of `shoco` and `urlparser`, which are the two C programs with spatial and temporal memory errors. We investigate if the memory errors are eliminated or detected at runtime in the Rust translation.

Case Study 1: `shoco` library. The C string data compression library `shoco` has one spatial memory vulnerability (CVE-2017-11367)[18]. The global array `packs` will be overread when the byte to be decompressed is malformed. In this case, the return value by the header decoding function will be larger than the length of `packs` array. When the return value is used to index `packs`, the global buffer overflow happens, as highlighted in line 5 in Fig 11 (a).

This spatial memory issue is detected at runtime in all Rust translations from our users. Since array `packs` is read-only, most translations lift it to a constant global array with type `[Pack; 3]`. Rust's spatial bounds checking on arrays ensures any out-of-bound array access will not happen. The Rust compiler inserts runtime checks before the index access `PACKS[mark]`. This is because it cannot statically infer the validity of this access since the value of `mark` is determined by the input. Fig 11 (b) shows one translation, where the binary will panic if the buffer overflow is about to happen.

It is worth noting that existing `c2rust`-based tools (including LAERTES and CROWN) are also able to eliminate this vulnerability. They perform the same syntactic transformation in their first stage to convert the global array declaration in C to an equivalent declaration in Rust. Such array declarations are bounds-checked on access. Since their translation of this global array and the access are similar, we show one example of the translated code snippet by their tools relevant to the vulnerability in Fig. 11 (c).

Case Study 2: `urlparser` library. The `urlparser` C library (commit `a65623ad`) has multiple memory-related vulnerabilities, including a spatial memory vulnerability that causes information leaks and a temporal memory vulnerability that can potentially cause programs using this library to crash. Both vulnerabilities have been fixed in their latest version.

Temporal memory errors. This library aliases the input URL string pointer and stores it as a field in its custom structure when parsing this string. In addition to the parsing API, it offers APIs that read the parsing result, such as the `inspect` API. This is problematic because this library does not own the memory referred to by this aliased pointer. Use-after-free can happen if a program using this library calls the parse API providing a string that does not live long enough before calling other APIs, such as `inspect`. Fig. 12 (a) shows the related source code and a small example program that triggers use-after-free. This issue has been fixed in commit `752635e`.

This temporal issue is eliminated in all Rust translations because of the temporal safety statically enforced by borrowing rules and the `String` type. Any code pattern that can result in double-free or use-after-free cannot be compiled. To satisfy borrow rules, user translations copy the input string so that the struct representing the parsing result can hold an owning reference to the string, as shown in Fig. 12 (b). This copy is automatically deallocated when the lifetime of the parsing result (i.e., `UrlData`) ends.

[18]https://nvd.nist.gov/vuln/detail/CVE-2017-11367

(a) The source code related to a spatial memory issue

```
1  static const Pack packs[PACK_COUNT] = { ... };
2
3  size_t shoco_decompress(...) {
4      while (in < in_end) {
5          if (mark < 0) { ... }
6          else {
7              if (o + packs[mark]...) // Global buffer overflow!
8          }
9      }
10 }
```

(b) Safe Rust translation of the above C code by our user

```
1  const PACKS: [Pack; PACK_COUNT] = [ ... ];
2
3  fn shoco_decompress(...) {
4      while in_index < in_end {
5          if mark < 0 { ... }
6          else {
7              if o_index + PACKS[mark as usize]... // BoF caught
       by runtime check
8          }
9      }
10 }
```

(c) Rust translation by compiler-based tools (LAERTES/CROWN)

```
1  static mut packs: [Pack; 3] = ...;
2
3  pub unsafe extern "C" fn shoco_decompress(...) {
4     ...
5     while in_0 < in_end {
6        mark = ...;
7        if ... {}
8        else {
9              if o.offset(packs[mark as usize].bytes_unpacked as
       size) > out_end ... // BoF caught by runtime check
10       }
11    }
12 }
```

Fig. 11: Case Study 1: The spatial memory issue in the `shoco` library is detected in all Rust translations.

Spatial memory errors. This library can overread the input string and print extra memory value if the input URL is malformed. The root reason is that it directly increments the URL pointer to skip the scheme separator `://` without checking if the separator exists in the URL. When it does not exist, the pointer is incremented to be outside of the input string buffer. This bug is caught at runtime by all Rust translations. This pointer is lifted to `&str` or `&String` types, which enforce spatial bounds checking on access. Several other heap buffer overflow issues exist as well, all due to saving strings to allocated memory regions with insufficient space, i.e., having an allocated size smaller than the string length. Such spatial issues do not exist in safe Rust translations by users due to automated memory management.

B. Examples of Logical Translation Errors by Users

Here we describe several logical translation errors we observed in translations of `fmt` program, along with a possible fix for each.

1) Logical Errors that Require More Thorough Testing

(a) Code example related to the temporal memory issue

```c
// urlparser.c
url_data_t *url_parse (char *url) {
  url_data_t *data = malloc(sizeof(url_data_t));
  data->href = url; // Store the pointer
}
void url_data_inspect (url_data_t *data) {
  printf("#url␣=>\n");
  printf("␣␣␣␣.href:␣\"%s\"\n", data->href); // use the pointer
}

// poc.c
int main() {
    // our_url points to a URL string stored on the heap
    url_data_t *parsed = url_parse(our_url);
    assert(parsed);
    free(our_url);
    url_data_inspect(parsed); // Use-after-free here!
}
```

(b) One corresponding Rust translation of the above C code

```rust
fn url_parse(url: &str) -> Option<UrlData> {
    // data is a default UrlData instance
    data.href = Some(url.to_string());
    // href has Option<String> type
    // ...
}
fn url_data_inspect(data: &UrlData) {
    println!("#url␣=>");
    println!("␣␣␣␣.href:␣{:?}", data.href); // Owned String. No
      temporal issue.
}
```

Fig. 12: Case Study 2: The temporal memory issue in the `urlparser` library is eliminated in all Rust translations.

The full program of the translation (Version A) we have seen before (Fig. 3), although passing all tests with 85% code coverage, has at least two more semantic discrepancies from the source program. Surprisingly, the bugs are in code lines covered by passing tests. We have two failing tests demonstrating each of the translation bugs in Fig. 13. One bug is revealed when there are multi-byte characters rather than just ASCII characters in the input. Another bug is revealed when there are odd numbers rather than even numbers of padding spaces. The unit tests written by the user miss those cases.

The first discrepancy is due to calling the wrong API when computing the display width of the characters. The translation used `char::len_utf8` which computed the size in bytes of a Unicode character instead of display width. For the frequent ASCII characters, the size in bytes happens to be 1, which coincides with their display size. However, the failing input "`z\u00df\u6c34\U0001d10b`" is an example where the size in bytes is different from the display width. It is not revealed by the passing unit tests which only include ASCII characters. The fix is to use the `c.width` method, as shown in Fig. 13.

The second discrepancy is due to an off-by-one error in the computation of padding spaces, making the Rust translation correct only when the number of spaces is even. The Rust program uses a slightly different computation to split the padding spaces into leading and trailing ones, compared to the C program. While the C program behaves like a round-up division, the Rust program takes the floor division. The passing test does not reveal this difference. The correct fix is shown in Fig. 13.

2) Semantic Differences that are Not Easy to Fix

Passing Test (Translation A)

```
Executed command: ./fmt -c -w 10
Input: "Center"
C Output: "  Center" // 2 leading padding spaces
Rust Output: "  Center" // same as C
```

Failing Test 1

```
Executed command: ./fmt -c -w 10
Input: "z\u00df\u6c34\U0001d10b"
C Output and Rust Output have different display
    width
```

Failing Test 2

```
Executed command: ./fmt -c -w 10
Input: "Center*"
C Output: "  Center*" // 2 leading padding spaces
Rust Output: " Center*" // off-by-one error!
```

The bugfix (Translation A)

```
...
+ use unicode_width::UnicodeWidthChar;
if bytes_read == 0 { break; }
let len: usize = p1.trim().chars().map(
    |c| if c == '\t' { ' ' } else { c }
- ).map(char::len_utf8).sum();
+ ).map(|c| c.width().unwrap_or(1)).sum();
let padding =
- (config.goal_length - len) / 2;
+ (config.goal_length - len + 1) / 2;
...
```

Fig. 13: The Translation Version A wrongly computes the display width. Part (a): 1 passing and 2 failing tests when aligning the input centrally in a 10-byte line (executed command: `./fmt -c -w 10`). Part (b): Bugfix for two semantic discrepancies.

There is another difficult-to-fix semantic discrepancy that exists in all user translations for the `fmt` program. They all omit the functionality of replacing invalid Unicode characters with `?`. With a deeper investigation, we find that this is not easy to fix due to various differences in data types and APIs across the two languages.

Fig. 14 highlighted the branch of the C code to handle this functionality, which was previously skipped in Fig. 2 for simplicity. All users assigned to this program omit this functionality in their translation.

With closer examination, it turns out that this behavior is not easily implementable when using the `String` type to represent `p1`. In Rust, characters stored in a `String` data type are limited to Unicode characters rather than arbitrary raw bytes. When using the Rust API `read_line` to read a `String` from the stream, it returns either a `String` instance or a `Err` result if there are invalid characters. When returning an `Err`, there is no API to tell where the invalid characters are. With a more careful search, we find an API `from_utf8_lossy` on the `String` type in Rust that is closer to what we want. This API can create a `String` from a buffer with each point of decoding error marked with `?`. However, we soon realize that this API also does not bridge the semantic gap. The C code not only outputs `?` at the position of invalid characters, but also outputs the same number of `?` as the number of invalid bytes. The

```c
static void
center_stream(FILE *stream, const char *name)
{
    char *p1, *p2;
    wchar_t wc;
    size_t len; /* Display width of the line. */
    int w1;     /* Display width of one character. */
    int w2;     /* Length in bytes of one character. */

    while ((p1 = get_line(stream)) != NULL) {
        len = 0;
          for (p2 = p1; *p2 != '\0'; p2 += w2) {
              // ... omitted

              if ((w2 = mbtowc(&wc,p2,MB_CUR_MAX))==-1) {
                  (void)mbtowc(NULL,NULL,MB_CUR_MAX);
                  *p2 = '?';
                  w2 = 1;
                  w1 = 1;
              } else if ((w1 = wcwidth(wc)) == -1)
                  w1 = 1;

            // ... omitted
        }
        // ... print whitespace padding
        puts(p1);
    }
    // ...
}
```

Fig. 14: A highlighted block of code of the C example program (`fmt`) that deals with invalid characters (previously omitted for simplicity).

```rust
use unicode_width::UnicodeWidthChar;
fn center_stream<R: BufRead>(mut stream: R, _name: &str, config:
      &Config) {
    let mut p1: Vec<u8> = vec![];
    while let Ok(bytes) = stream.read_until(b'\n', &mut p1) {
        if bytes == 0 { break; }

        let mut res = String::new();
        let mut idx = 0;
        loop {
            let remain: &[u8] = &p1[idx..];
            match remain.utf8_chunks().next() {
                Some(s) => {
                    let valid = s.valid();
                    if valid.is_empty() {
                        res.push_str("?");
                        idx += 1;
                    } else {
                        res.push_str(valid);
                        idx += valid.len();
                    }
                },
                None => {
                    break;
                }
            }
        }
        let len = ...;
        // ... remaining lines omitted
    }
}
```

Fig. 15: The fixed Translation A where the data type of `p1` is refined from `String` to `Vec<u8>`. The code to parse the Unicode character is updated accordingly.

`from_utf8_lossy` API is not behaving the same way as the corresponding C code—it replaces each chunk of invalid bytes as one `?`. For example, for the invalid byte sequence `"\x7a\xc3\x9f\xe6\xb0\xc3\xf0\x9d\x84\x8b"`, the Rust String API `from_utf8_lossy` outputs a string with two `"?"` while the C code outputs three `"?"`. In summary, we cannot find APIs on `String` to preserve the same semantics.

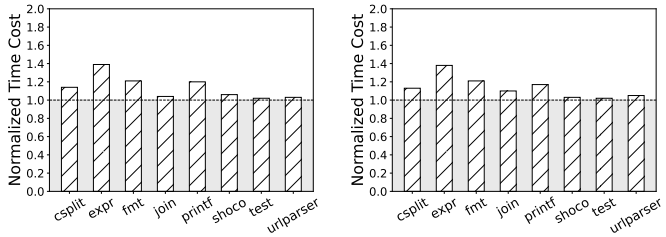
As a result, we can consider refining the choice of Rust data types for `p1`. The type of `p1` should carry sufficient information and support APIs to identify (1) the locations of invalid characters, and (2) the number of invalid bytes at each location.

Many Rust data types can represent a string but not all of them can meet our requirements. One possible choice is `std::ffi::OsString`, which is a platform-native string type that may hold invalid UTF-8 characters. `Vec<u8>` is another choice that can store invalid Unicode characters and supports per-byte control. Here we use `Vec<u8>` as an example.

Now the question is about how to use `Vec<u8>` to achieve the intended program behavior, i.e., to parse Unicode characters and replace invalid ones with the correct number of question marks. One option is to use `std::str::Utf8Chunks` structure from the Rust standard library. This structure allows iteratively check `u8` bytes referred by an immutable slice and convert them to Unicode characters if valid. Fig. 15 shows the fixed version of Translation A with the correct Unicode character parsing logic when using `Vec<u8>` type for `p1`. Because of this new choice of type, the original code to read lines and parse characters needs to be updated accordingly. We update multiple lines in the method calls on `stream` to read the input and iteratively save the line-separated input into a `Vec<u8>`. The eventually fixed translation (shown in Fig. 15) can pass the tests (behave the same as the C program) on the aforementioned invalid byte sequence.

C. End-to-End Performance Results of User Translations

Besides the performance results explained in Sec. V-B, we also measure the end-to-end performance on tests using hyperfine [2], which considers the load/start-up time of programs, in addition to the execution of the main functionality. The average overhead is around 13% across the eight Rust translations. Details of the performance measure for each program are shown in Fig. 16. This measure might not reflect the performance we care about since initialization costs are usually amortized for long-running programs and libraries.



(a) Compiled with `-O2` (b) Compiled with `-O3`

Fig. 16: End-to-end running time measured by hyperfine [2] of the most similar Rust translation compared to the original C at different optimization levels. The time for the baseline C (dashed line) is normalized to 1.