DOI:10.1145/3737696

Automatic C-to-Rust translation tools are helpful, but they produce unsafe and unidiomatic code. What can be done to address these issues?

BY JAEMIN HONG AND SUKYOUNG RYU

Automatically Translating C to Rust

IN THE SOFTWARE industry, legacy systems developed in older languages often evolve by being reimplemented in newer languages that offer modern language features. For example, Twitter migrated from Ruby to Scala to enhance performance and reliability;[29] Dropbox rewrote its Python backends in Go to leverage better concurrency support and faster execution;[18] and banking systems originally written in Cobol have evolved to Java or C# for easier maintenance and integration with modern infrastructures.[3]

One of the most critical language migrations needed today is the shift from C to newer languages, which can improve the reliability of important system programs. Despite its popularity in systems programming, C is infamous for its limited language-level safety mechanisms. C programs are prone to memory bugs, such as buffer overflow and use-after-free, which can lead to severe security vulnerabilities—exemplified by the Heartbleed bug[a] in OpenSSL. A large portion of vulnerabilities in legacy systems arise from memory bugs; for example, approximately 70% of the vulnerabilities in Microsoft's codebase are due to memory bugs.[28] Acknowledging these risks, even the White House recently recommended discouraging the use of C.[24]

The most promising migration target for C is Rust,[21] which provides strong safety guarantees while still allowing low-level memory control and high performance. Its safety guarantees are enabled by its ownership type system, which ensures the absence of memory bugs in programs that pass type checking.[15] Due to this advantage, Rust has been widely adopted in systems programming, as demonstrated by the development of clean-slate system programs such as garbage collectors,[20] Web browsers,[1] and operating systems.[2,16,19,22]

Recognizing Rust's potential, the industry has shown significant interest in migrating legacy systems from C to Rust. Such migration allows developers to detect previously unknown

a https://heartbleed.com/

» key insights

- **Migrating legacy systems developed in C to Rust is a promising way to enhance reliability, thanks to Rust's strong safety guarantees.**
- **Automatic translators can facilitate migration from C to Rust, but existing translators generate unsatisfactory code by relying on language features whose safety is not validated by the compiler and code patterns considered unidiomatic by Rust developers.**
- **Carefully designed static analyses and code transformations can improve automatic translation by replacing unsafe features and unidiomatic patterns with safe and idiomatic alternatives.**
- **Although LLMs are another promising approach to C-to-Rust translation, they often produce code with type errors or behavior different from the original code. Combining LLMs with static analyses is a potential future research direction.**

ILLUSTRATION BY VIKTOR KOEN

bugs through Rust's type checking. For instance, more than half of cURL's vulnerabilities could have been discovered if it had been rewritten in Rust.[12] Furthermore, after migration, the risk of introducing new bugs when adding functionalities is significantly reduced.

Several well-known system programs have already begun migrating from C to Rust. For example, a media decoder in VLC was ported to Rust,[4] and utility programs in GNU Coreutils are being rewritten in Rust.[13] Most notably, the Linux kernel has supported the use of Rust for kernel development since Linux 6.1,[27] with the first network driver in Rust introduced in Linux 6.8.[17]

To encourage the migration of real-world systems, the industry has been developing automatic C-to-Rust translators.[b,c,d] Because manual code translation is time-consuming and error-prone, these tools are essential. Among them, the most successful is C2Rust,[b] which is still actively developed and maintained. Software companies including Huawei,[32] and some open source projects[e,f] have used C2Rust to migrate their code.

As an automatic translator, C2Rust successfully handles syntactic discrepancies between the two languages. For example, variables are declared using the syntax `[type] [name];` in C but `let [name]: [type];` in Rust, and C2Rust rewrites variable declarations accordingly. Through such syntactic conversions, C2Rust produces syntactically valid Rust code, meeting the minimum expectation for a working translator.

That said, C2Rust's translation remains unsatisfactory for developers because it fails to generate code that fully leverages Rust's language features. The Rust code produced by C2Rust retains C's features, which Rust supports as a subset of its features. However, Rust also has its own features absent in C (hereafter referred to as *Rust features*). C2Rust does not incorporate Rust features into the translated code, negatively affecting migration quality in two aspects: safety and productivity.

Recognizing Rust's potential, the industry has shown significant interest in migrating legacy systems from C to Rust. Such migration allows developers to detect previously unknown bugs through Rust's type checking.

First, the safety of the translated code is not guaranteed because most C features are classified as *unsafe features*[g] in Rust, whose safety is not ensured by the Rust compiler. Safety can be guaranteed only by using Rust features classified as safe. This limitation is particularly significant, as the primary motivation for migrating to Rust is to enhance the reliability of the system.

Second, the productivity of the translated code is poor because it contains *unidiomatic code patterns* that do not adhere to Rust's programming idioms. Rust idioms rely heavily on Rust features to express high-level concepts. In contrast, C lacks equivalent features, leading C programmers to use peculiar code patterns that are considered unidiomatic in Rust. These patterns persist in the translated code, making it difficult for developers to comprehend and reducing overall productivity.

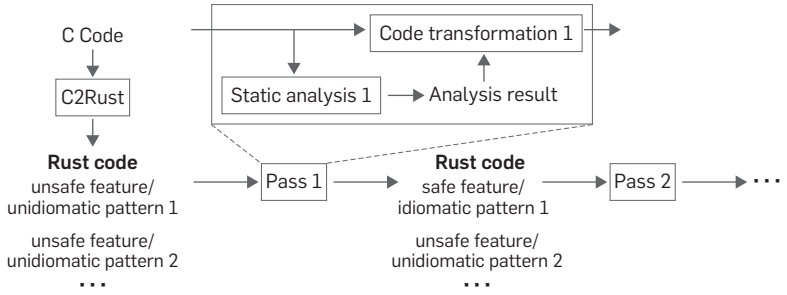
To maximize the benefits of migration, the translator must introduce Rust features into the code, replacing unsafe features with safe counterparts and unidiomatic patterns with idiomatic alternatives. This task is challenging due to the nature of Rust features, many of which require the source code to reveal information that is implicit in C. For example, Rust pointers indicate which ones have the right to deallocate their pointees, whereas C pointers do not. Consequently, introducing Rust features necessitates a precise understanding of the program's behavior.

The most feasible way to address the limitations of C2Rust is to gradually improve the translated code through multiple refinement passes, each aimed at replacing a specific feature or pattern, as illustrated in Figure 1. After C2Rust's initial translation, the Rust code contains various unsafe features and unidiomatic patterns. The code is then improved through each pass, specialized in introducing a Rust feature that can eliminate a specific unsafe feature or unidiomatic pattern. Each pass performs static analysis to gather information about the target feature or pattern and applies code transformations

b C2Rust: https://github.com/immunant/c2rust
c Citrus: https://gitlab.com/citrus-rs/citrus
d Corrode: https://github.com/jameysharp/corrode
e qcms: https://github.com/FirefoxGraphics/qcms/
f zwbra-rs: https://github.com/panstromek/zebra-rs

g The Rust Reference: Unsafety: https://doc.rust-lang.org/reference/unsafety.html

Figure 1. Improving C2Rust-translated code by replacing unsafe features and unidiomatic patterns.



to replace it with a better alternative based on the analysis results.

This approach of using multiple passes to transform code is analogous to the methodology employed by the usual program transformation tools. Compilers typically perform multiple optimization passes, each applying a specific optimization supported by static analysis. Decompilers, which convert machine code to high-level code like C, also use multiple passes, each recovering an abstraction found in source code—such as loops, functions, and types—by employing specialized static analysis. This strategy allows for a compositional approach to handling complex language features, rather than addressing them all at once.

In this article, we begin with code examples that illustrate the challenges in improving C2Rust-translated code. We then present the initial progress made by the research community, including our own, in developing refinement passes to remove unsafe features and unidiomatic patterns from C2Rust-translated code. So far, Emre et al.[5,6] and Zhang et al.[33] have worked on an unsafe feature, scalar pointers, while we have studied two unsafe features—locks[8] and unions with tags[10]—as well as one unidiomatic pattern, output parameters.[9] Next, we describe the remaining unsafe features and unidiomatic patterns that future research needs to address. Finally, we briefly discuss the potential of using large language models (LLMs) in C-to-Rust translation, highlighting recent results.[7,11,23,25,26,31]

Challenges in Translation

Here, we provide an overview of the challenges in C-to-Rust translation. We consider a toy C program and demonstrate how C2Rust translates it into Rust code that contains unsafe features and unidiomatic patterns. We then explain how the translated code can be improved by introducing appropriate Rust features, and why automating this improvement is challenging.

Listing 1 shows a C program that handles fractions. The `frac` struct represents fractions with two fields: `num` for the numerator and `den` for the denominator (line 1).

The function `frac _ to _ int` rounds down a fraction to an integer by dividing the numerator by the denominator (lines 2 to 4). It takes two parameters: `f`, a pointer to the `frac` to be converted, and `res`, a pointer to an integer where the result will be stored (line 2). This code pattern, where a parameter is used to store the output, is known as an *output parameter*. The return value of the function indicates whether the conversion was successful, rather than carrying the result itself. The function first checks whether the denominator is zero, which makes the conversion impossible, and if so, returns `-1` to indicate failure (line 3). Otherwise, it stores the result in `res` and returns `0` to indicate success (line 4).

The function `foo` constructs a fraction and converts it to an integer by calling `frac _ to _ int` (lines 5 to 8). It first allocates memory for a `frac` on the heap using `malloc` and initializes the numerator and denominator (line 6). Then, it defines the variable `res`, where the result of the conversion will be stored, calls `frac _ to _ int`, and prints the result only if the conversion succeeds (line 7). Finally, it modifies the denominator of the fraction and performs further computations (line 8).

Although this code works correctly, it is prone to memory bugs if the programmer makes a mistake. For instance, one might decide to add `free(f)` to line 4, just before `frac _ to _ int` returns. If `foo` then attempts to modify the denominator of the fraction after calling `frac _ to _ int`, it results in use-after-free by writing to memory that has already been freed.

Listing 2 shows the result of trans-

Listing 1.

```
1 struct frac { int num; int den; };
2 int frac_to_int(struct frac *f, int *res) {
3   if (f->den == 0) { return -1; }
4   *res = f->num / f->den; return 0; }
5 void foo() {
6   struct frac *f = malloc(sizeof(struct frac)); f->num = 42; f->den = 5;
7   int res = 0; if (frac_to_int(f, &res) == 0) { printf("%d\n", res); }
8   f->den = 7; ... }
```

Listing 2.

```
1 struct frac { num: i32, den: i32 }
2 unsafe fn frac_to_int(f: *mut frac, res: *mut i32) -> i32 {
3   if (*f).den == 0 { return -1; }
4   *res = (*f).num / (*f).den; return 0; }
5 unsafe fn foo() {
6   let f: *mut frac = malloc(size_of::<frac>()); (*f).num = 42; (*f).den = 5;
7   let res = 0; if frac_to_int(f, &mut res) == 0 { printf("%d\n", res); }
8   (*f).den = 7; ... }
```

Listing 3.

```
1 struct frac { num: i32, den: i32 }
2 fn frac_to_int(f: &frac) -> Option<i32> {
3   if (*f).den == 0 { return None; }
4   return Some((*f).num / (*f).den); }
5 fn foo() {
6   let f: Box<frac> = Box::new(frac { num: 42, den: 5 });
7   if let Some(res) = frac_to_int(&f) { println!("{}", res); }
8   (*f).den = 7; ... }
```

lating this C code to Rust using C2Rust. The generated code is syntactically valid Rust and preserves the behavior of the original C program. However, the translation is still unsatisfactory, for two main reasons: the use of *raw pointers*, which are an unsafe feature, and the use of *output parameters*, which are considered unidiomatic by Rust programmers.

First, using raw pointers prevents the compiler from guaranteeing the safety of the code, contradicting the main goal of translating to Rust. Rust's raw pointers are equivalent to C pointers. Since the compiler does not check the validity of raw pointers, memory bugs can still occur. In Listing 2, the types `*mut frac` and `*mut i32` indicate that `f` and `res` are raw pointers (lines 2 to 6). If the programmer mistakenly frees `f` before returning from `frac_to_int`, the compiler will not catch the error. Since both `frac_to_int` and `foo` use raw pointers, Rust requires them to be marked `unsafe` (lines 2 to 5), indicating that the programmer is responsible for ensuring the code's safety.

Second, output parameters are difficult to understand and error-prone. Since parameters are typically used for input, output parameters do not clearly convey that the function produces a result. Moreover, in this example, `frac_to_int` is a *partial function*—it may fail for some inputs. The function may not write a result to the output parameter, and `foo` must check the return value to determine whether to read `res`. If `foo` inadvertently reads `res` after the conversion has failed, it will print an arbitrary value (`0` in this case) that `res` held before the call. Despite these drawbacks, output parameters are a common pattern in C for implementing partial functions, as C lacks a language feature to express partial functions. However, as we will see shortly, Rust provides a better alternative, and the use of output parameters is considered unidiomatic in Rust.

Listing 3 shows an improved version of the translation. The improvement is mainly twofold: 1) the code uses `Box` and *references* instead of raw pointers, allowing the compiler to guarantee safety; and 2) it directly returns `Options` instead of using output parameters to implement partial functions, making the code more comprehensible and less error-prone.

First, by using `Box` and references, the absence of use-after-free is guaranteed at compile time. A `Box` is an *owning* pointer to a heap-allocated object. By owning the pointee, the pointer has the right to deallocate it. In contrast, a reference represents a *borrowing* pointer. It can access the pointee but does not have permission to deallocate it. While multiple references to the same object can coexist, only one `Box` can own the object at a time.

In Listing 3, `foo` has a `Box` to a `frac` (line 6), while `frac_to_int` takes a reference (line 2). This ensures that `frac_to_int` cannot deallocate the `frac` object, making it safe to use `f` after the call to `frac_to_int`. If one wants to deallocate `f` within `frac_to_int`, the function must take a `Box` instead of a reference. In that case, the compiler treats the ownership of `f` as transferred from `foo` to `frac_to_int` due to the function call. Since Rust's ownership type system forbids using a value after its ownership has been moved, using `f` in `foo` after the call will result in a type error.

Second, directly returning `Options` is the idiomatic way to express partial functions in Rust. `Option` is a type that represents the optional existence of a value. An `Option` value is either `None` or `Some(v)` for some value *v*, where `None` represents the absence of a value and `Some` represents the presence of a specific value. Partial functions can be expressed by returning `Options`, using `None` to indicate failure and `Some` to indicate success.

In Listing 3, `frac_to_int` returns an `Option<i32>`, which represents an optional integer (line 2). When the denominator is zero, the function returns `None` (line 3); otherwise, it returns `Some` containing the result (line 4). To handle the `Option` value returned by `frac_to_int`, `foo` uses `if let`, a syntactic construct in Rust designed for handling various types, including `Option` (line 7). This statement checks whether the right-hand side evaluates to `Some`, and if so, binds the inner value to `res` and executes the block to print the result. Since accessing the inner value of `Some` always requires explicit syntax such as `if let`, the programmer cannot mistakenly print an arbitrary value by neglecting to consider the failure case.

This example of improving C2Rust-translated code demonstrates why automating this process is challenging. Compared to C features, Rust features require more information to be explicitly expressed in the code. As a result, improving the code requires a deep understanding of the program's behavior. `Box` and references specify which pointer deallocates the object and which merely accesses it. Since raw pointers do not make this distinction, we must determine the purpose of each pointer to replace them with `Box` and references. Similarly, by returning `Options`, functions convey the intent of producing a result from a potentially failing computation. Since pointer-type parameters do not indicate whether they are used for input or output, or whether the output is partial, we must infer the purpose of each parameter to replace output parameters with `Options`.

Addressing this challenge motivates the use of static analysis, a technique for automatically discovering properties of a program from its code. It has been thoroughly studied in the programming languages literature, and a wide range of techniques have been developed to efficiently target different kinds of properties. Tools that process code, such as compilers, already make extensive use of static analysis to trans-

form code into a desired form while preserving its behavior. Therefore, we can leverage static analysis to extract the information required by Rust features and systematically replace unsafe features and unidiomatic patterns with safe and idiomatic alternatives.

Improving Translation Using Static Analysis

We now discuss the techniques developed by the research community to replace unsafe features and unidiomatic patterns in C2Rust-translated code with safe features and idiomatic patterns using static analysis. So far, three unsafe features, scalar pointers,[5,6,33] locks,[8] and unions with tags,[10] and one unidiomatic pattern, output parameters,[9] have been addressed.

For each feature or pattern, we briefly explain what it is, which Rust feature can replace it, and what information is required to enable that replacement. We also provide references to papers that propose static analyses for computing the necessary information, rather than discussing the analyses in detail. Interested readers can refer to those papers for more information.

Scalar pointers. Scalar pointers are raw pointers to non-array objects. As discussed earlier, they are considered an unsafe feature because they are prone to memory bugs.

Replacing scalar pointers with the safe features `Box` and references allows the compiler to guarantee the absence of memory bugs. `Box` and references distinguish between the owner and borrowers. As the owner is unique for each object, the owner can safely use the object without concern that other pointers might deallocate it. In addition, references often require *lifetime* annotations, which indicate how long they borrow objects. The compiler checks lifetimes to ensure borrowing ends before the owner deallocates the object.

Replacing raw pointers with `Box`/references requires two kinds of information. First, to decide which pointers should be converted to `Box` and which to references, we need to determine the owner of each object. Second, to assign appropriate lifetimes to references, we must compute the borrowing duration of each pointer. To this end, two static analyses have been proposed. Zhang et al.[33] proposed a technique to identify owners, and Emre et al.[5,6] developed a technique to determine lifetimes.

Compared to C features, Rust features require more information to be explicitly expressed in the code. As a result, improving the code requires a deep understanding of the program's behavior.

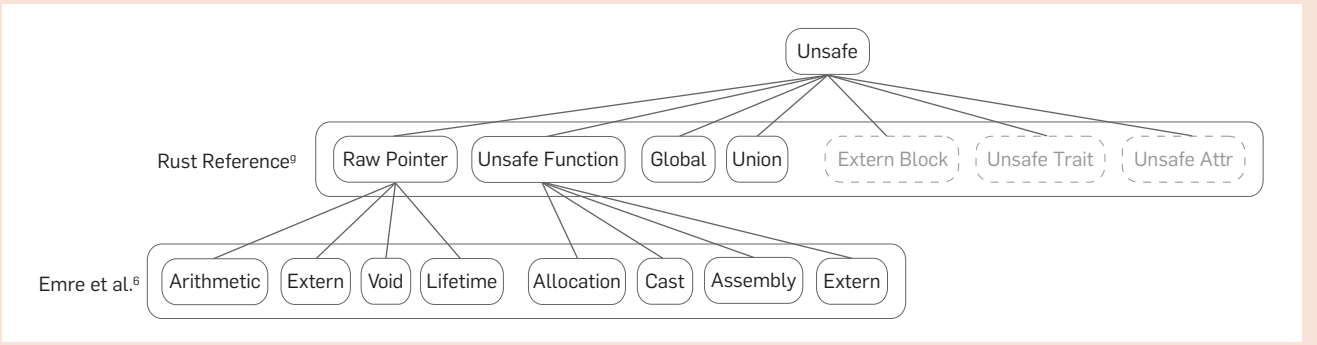
Locks. This feature involves the use of locks provided by the C standard library (libc) to protect data shared among multiple threads. In multithreaded programs, data races occur when multiple threads simultaneously read from and write to the same data. To prevent this, programmers define a lock for each shared data and make each thread acquire the lock before accessing the data and release it afterward. This allows exclusive access to the data because each lock can be held by at most one thread at a time.

Unfortunately, libc's locks are considered an unsafe feature in Rust because they fail to prevent data races when programmers make mistakes. They may access shared data without holding the appropriate lock, either by acquiring the wrong lock or by not realizing that the lock is not held.

Replacing libc locks with the Rust standard library's locks—considered a safe feature in Rust—addresses this issue. In Rust, a lock and the data it protects are combined into a single value. Since both the lock and the data are accessed via the same variable, programmers cannot mistakenly acquire the wrong lock when accessing shared data. In addition, Rust introduces the notion of a *guard*, which serves as explicit evidence that a lock is held. A guard is essentially a special kind of pointer to the lock-protected data, and dereferencing the guard is the only way to access the data. A guard is created when a lock is acquired, and when the guard is deallocated, its destructor automatically releases the lock. As a result, protected data can be accessed only while the lock is held. If a programmer attempts to access the data using a guard that has already been deallocated, the compiler raises a type error because the guard's ownership has been lost due to deallocation.

To replace libc locks with Rust locks, two kinds of information are needed. First, to merge each lock with the protected data into a single value, we must identify which data is protected by which lock. Second, to introduce guard values at the correct locations, we need to determine which locks are held at each program point. We pro-

Figure 2. Classification of unsafe features in C2Rust-translated code.



posed a static analysis to collect this information.[8]

Unions with tags. Unions are a compound data type consisting of multiple fields that share the same memory storage. Since all fields in a union share the same memory location, writing to one field overwrites the value previously stored in any other field. Thus, if a field other than the last-written one is read, the stored value is *reinterpreted* as the type of the read field.[14] Such reinterpretation can lead to bugs such as invalid memory access, so unions are considered an unsafe feature in Rust.

To mitigate this problem, programmers often accompany unions with *tags*—integer values that indicate the last-written field. They update the tag when writing to a union field and check it before reading to determine which field to access. However, programmers may incorrectly update the tag when writing to a union field or read the wrong field while neglecting the tag value.

This problem can be resolved by replacing unions and tags with *tagged unions*, a safe feature in Rust. A tagged union is defined by enumerating possible tags and specifying the type associated with each tag. This makes the relationship between tags and the types of stored values explicit. As a result, the compiler can ensure that the tag is correctly set when a tagged union value is constructed, and that the value is interpreted as the correct type according to its tag when used.

Replacing unions accompanied by tags with tagged unions requires two kinds of information. First, to merge a union and its tag into a single tagged union, we must identify where the tag for each union is stored. Second, to enumerate the tags and their corresponding types in the tagged union definition, we need to determine which tag values correspond to which union fields. We proposed a static analysis to collect this information.[10]

Output parameters. Output parameters are a code pattern whereby parameters are used to produce results. They are useful for implementing partial functions, as discussed earlier, and functions that produce multiple values. A function can produce multiple values by returning one value and writing the others to output parameters. However, output parameters are considered unidiomatic in Rust because they hinder code comprehension by failing to clearly convey their role as outputs.

Rust programmers prefer to return values directly, rather than relying on output parameters, to improve code readability. To implement the aforementioned kinds of functions using direct returns, Rust programmers employ `Options` and tuples. Using an `Option` as a return type indicates the possibility of failure, while using a tuple reveals that the function produces multiple values.

Replacing output parameters with direct returns requires two kinds of information. First, to decide whether a parameter should be replaced with a direct return, we must determine whether it is intended as an output. Second, to decide whether to use an `Option` or a tuple, we must determine whether each output parameter is partial. To collect this information, we proposed a static analysis tailored for this purpose.[9]

Remaining Features and Patterns

We now turn our attention to the remaining unsafe features and unidiomatic patterns to be addressed by future research. Our discussion mainly focuses on unsafe features, as Rust defines a clear set of unsafe features, making it possible to analyze which ones have been addressed and which remain to be tackled. In contrast, such analysis is difficult for unidiomatic patterns because programming idioms are inherently subjective. For this reason, we just briefly introduce a few unidiomatic patterns at the end of this section.

We begin by classifying unsafe features, as shown in Figure 2. According to the Rust Reference, there are seven kinds of unsafe features in Rust,[g] listed in the second row of the figure: dereferencing raw pointers, calling unsafe functions, accessing mutable global

Table. List of unsafe features in C2Rust-translated code.

Feature	Relevant Categories	Addressed
Scalar pointer	Lifetime pointer, allocation function	✓
Array pointer	Arithmetic pointer, allocation function, extern function	×
Lock	Extern pointer, extern function, global	✓
File	Extern pointer, extern function	×
Subprocess	Extern function	×
Union with tag	Union	✓
Void pointer argument	Void pointer, cast	×

variables, reading union fields, declaring extern blocks, implementing unsafe traits, and using unsafe attributes.

Of these, we exclude three categories—extern blocks, unsafe traits, and unsafe attributes—from our consideration. First, extern blocks are used only to declare the signatures of extern functions, while extern functions are already included in the unsafe functions category, making it redundant. Furthermore, unsafe traits do not appear in C2Rust-translated Rust code. Finally, unsafe attributes only affect linking by controlling function names emitted to the binary, rather than directly impacting program semantics. Tools such as ResolveImports[6] can eliminate the need for unsafe attributes by modifying the code to use built-in import mechanisms instead of relying on linking.

Additionally, as shown in the third row, Emre et al.[6] further classify certain unsafe features. They categorize raw pointers into arithmetic pointers, extern pointers, void pointers, and lifetime pointers. Arithmetic pointers are those used in pointer arithmetic, extern pointers are those passed to or returned from extern functions, and void pointers are cast from or to `void *`. Lifetime pointers include all raw pointers not in the other categories. They also categorize unsafe functions into allocation functions, casts, assembly, and extern functions. Allocation functions include `malloc` and `free`, casts refer to unsafe casting through `std::mem::transmute`, and assembly refers to the use of inline assembly code. Extern functions cover all functions not classified elsewhere, mainly those in libc. By definition, their classification is exhaustive.

While this classification helps us understand which unsafe features exist, each category is not well-suited to be handled by a single static analysis. This is because some categories are still too broad, and some are deeply interconnected with others. For example, scalar pointers encompass both lifetime pointers and allocation functions used for non-array objects.

For this reason, we provide a list of unsafe features, each of which can be addressed by a single static analysis. This is shown in the table, which indicates which categories of unsafe features are relevant to each feature and whether the feature has already been addressed. This list is intended to complement the earlier classification. Researchers can refer to this list to identify targets for future work and then consult the classification to assess how much of the overall set of unsafe features would be covered. Note that the list is not exhaustive and can be extended by investigating the remaining areas in the classification. We now briefly explain each unaddressed feature in the list:

We can leverage static analysis to extract the information required by Rust features and systematically replace unsafe features and unidiomatic patterns with safe and idiomatic alternatives.

- *Array pointers.* This feature involves arithmetic pointers, allocation functions, and extern functions used to handle arrays. Arithmetic pointers can be replaced by `Vec` and `String`, which are owning pointers to arrays, or by slices, which are borrowing pointers to arrays. Allocation functions can be replaced by functions such as `Vec::new` and `String::new`. Extern functions like `memcpy` can be replaced by Rust standard library functions such as `copy_from_slice`.
- *Files.* This feature involves extern pointers and extern functions used for interacting with the file system. Extern pointers include pointers to streams of type `FILE`, and extern functions include stream-related operations such as `fopen` and `fclose`. These can be replaced by types provided in the Rust standard library, such as `File` for open files and the `Read`/`Write` traits for readable/writable streams, along with their methods.
- *Subprocesses.* This feature involves extern functions for creating, waiting on, and communicating with subprocesses, such as `fork`, `exec`, `wait`, and `pipe`. These can be replaced by the `Command` type in the Rust standard library and its methods.
- *Void pointer arguments.* This feature involves the use of void pointers as function arguments and cast from or to void pointers. These can be replaced with generic functions in Rust.

We now introduce a few unidiomatic patterns. This list can be extended by future research:

- *Index-based loop.* This pattern involves iterating over arrays using indices, which is considered unidiomatic in Rust because it can lead to out-of-bounds access. Index-based loops can be replaced by iterator-based

loops, where the iterator automatically terminates the iteration when the end of the array is reached.

▸ *Manual resource cleanup.* This pattern involves explicitly invoking a resource cleanup routine when an object is deallocated. It is considered unidiomatic in Rust because developers may forget to invoke the cleanup routine, especially in functions with early returns. Manual resource cleanup can be replaced by automatic cleanup using destructors, which are invoked automatically when an object is deallocated.

Using LLMs for Translation

While thus far in this article we have focused on using static analysis to improve translation, leveraging LLMs is emerging as another promising approach for C-to-Rust translation. Recent advances in machine learning have enabled the development of powerful LLMs, which are capable of performing code-related tasks, including code completion, summarization, bug detection, and automated repair. Applying LLMs to code translation is a natural application of their capabilities. In this section, we present the results of recent studies on LLM-based C-to-Rust translation[7,11,23,25,26,31] and discuss possible research directions in this area, including the use of static analysis and LLMs in a complementary manner.

Existing studies follow similar workflows for translation, summarized as *split-translate-check-fix*. First, the input C program splits into smaller pieces, usually top-level items such as functions or type definitions. This is necessary because entire programs are often too large to translate at once, particularly due to the context-length limits of LLMs. Second, the LLM translates each piece to Rust. In this step, the code is often augmented with additional information, such as declarations of used variables and functions. Third, the correctness of the translation is checked. Type checking ensures the translated code is free of type errors; if it passes, testing or model checking verifies behavioral equivalence between the original and translated code. Finally, if the translation is incorrect, the LLM fixes the code, and this process repeats until the translation is correct. One small exception to this workflow is the work by Nitin et al.,[23] where the LLM takes C2Rust-translated code as input and attempts to improve it, rather than translating from C directly.

While LLMs can generate Rust code using safe features and idiomatic patterns, they often produce incorrect translations. For this reason, it is crucial to verify the correctness of the translation and fix it if necessary.

In this workflow, the main challenges arise from the checking and fixing steps. While LLMs can generate Rust code using safe features and idiomatic patterns, they often produce incorrect translations. For this reason, it is crucial to verify the correctness of the translation and fix it if necessary. Note that these challenges are completely different from those in static-analysis-based approaches, where C2Rust typically generates correct but unsafe and unidiomatic code, and the challenges lie in designing appropriate static analyses to improve the translation. We now turn to a discussion of how existing studies have approached checking and fixing.

In the checking step, the main challenges lie in verifying behavioral equivalence, as type checking can be easily performed using the compiler. To check equivalence, some approaches compare entire programs. Nitin et al.[23] used end-to-end test cases from the original program to test the translated code, while Eniser et al.[7] employed a fuzzer to compare the I/O behavior of the original and translated programs. Other approaches focus on comparing the behavior of individual functions. Shetty et al.[25] generated unit tests using LLMs. They collect arguments and corresponding return values for each C function, translate these values to Rust using the LLM, execute the Rust function with the translated arguments, and compare its output with the translated return value. In contrast, Yang et al.[31] used property-based testing (PBT) and model checking. They compile C functions to WebAssembly and lift the compiled code to Rust. Using this lifted code as an oracle for correctness, they apply PBT and model checking to validate LLM-translated code. While verifying individual functions is a promising direction, future research is needed to handle complex types, not just simple ones like integers, arrays, and their pointers.

For the fixing step, existing techniques provide the LLM with information about what went wrong, such as type error messages or test results. Un-

fortunately, this is often insufficient, failing to resolve all issues. For example, our work[11] translated GNU packages with fewer than 10k lines of code (LOC) using GPT-4o mini, observing that 44% of functions were uncompilable. Shiraishi and Shinagawa[26] translated programs with fewer than 5k LOC using Claude 3.5 Sonnet, resulting in 27% of top-level items being uncompilable. Yang et al.[31] translated competitive programming solutions using Claude 2; out of 520 functions, only 339 passed type checking, 209 passed PBT, 193 passed bounded model checking, and just 15 passed full model checking. These results highlight the need for more effective techniques to fix LLM-translated code.

One promising direction is to augment the code with more helpful information during the *translation* step, enabling the LLM to generate code that more closely resembles the correct version, rather than relying on difficult fixes afterward. An example of this approach is the work by Shetty et al.,[25] which performs dynamic analysis to collect information about pointers and provides it to the LLM. They successfully translated Zopfli, consisting of more than 3k LOC, into Rust code that compiles and passes all unit tests, demonstrating the effectiveness of this approach.

This suggests exploring the combination of static analysis and LLMs in C-to-Rust translation. While static analyses have already been developed to improve C2Rust-translated code, applying the results through handwritten rules often produces verbose code. Instead, we can provide the analysis results to LLMs to guide their translation and generate more developer-friendly code.

Conclusion and Outlook

Migrating from C to Rust is a promising way to enhance the reliability of legacy system programs. Automatic translators are key to this migration, as they can significantly reduce human effort. Unfortunately, existing translators like C2Rust fall short of producing safe and idiomatic Rust code. In this article, we presented the initial progress made by the research community in improving translation through static analysis.

However, much work remains to be done in this area, which is gaining increased interest from researchers. Notably, DARPA (Defense Advanced Research Projects Agency) has announced a research program focusing on automatic C-to-Rust translation.[30] The most important future work involves addressing various unsafe features and unidiomatic patterns that have not yet been covered. Combining LLMs with static analysis is also a promising research direction.

Acknowledgments

We would like to thank all members of the KAIST Programming Language Research Group (PLRG) for their collaboration. This research was supported by National Research Foundation of Korea (NRF) (2022R1A2C200366011 and 2021R1A5A1021944), Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (2022-0-00460, 2023-2020-0-01819, and 2024-00337703), and Samsung Electronics Co., Ltd (G01210570).

References

1. Anderson, B. et al. Engineering the Servo web browser engine ussing Rust. In *Proceedings of the 38th Intern. Conf. on Software Engineering Companion*. ACM (2016), 81–89.
2. Boos, K. et al. Theseus: An experiment in operating system structure and state management. In *Proceedings of the 14th USENIX Conf. on Operating Systems Design and Implementation*. USENIX (2020).
3. Carey, S. How companies are moving on from Cobol. Infoworld (May 10, 2021); https://www.infoworld.com/article/2265785/howcompanies-are-moving-on-from-cobol.html.
4. Crane, S. Porting C to Rust for a fast and safe AV1 media decoder. Prossimo Blog (Sep. 9 2024); https://www.memorysafety.org/blog/porting-c-to-rust-for-av1/
5. Emre, M. et al. Aliasing limits on translating C to safe Rust. *Proc. ACM Program. Lang. 7*, OOPSLA1 (Apr. 2023).
6. Emre, M. et al. Translating C to safer Rust. *Proc. ACM Program. Lang. 5*, OOPSLA (Oct. 2021).
7. Eniser, H.F. et al. Towards translating real-world code with LLMs: A study of translating to Rust. *arXiv:2405.11514 [cs.SE]* (2024); https://arxiv.org/abs/2405.11514
8. Hong, J. and Ryu, S. Concrat: An automatic C-to-Rust lock API translator for concurrent programs. In *Proceedings of the 45th Intern. Conf. on Software Engineering*. IEEE Press (2023), 716–728.
9. Hong, J. and Ryu, S. Don't write, but return: Replacing output parameters with algebraic data types in C-to-Rust translation. *Proc. ACM Program. Lang. 8*, PLDI (Jun. 2024).
10. Hong, J. and Ryu, S. To tag, or not to tag: Translating C's unions to Rust's tagged unions. In *Proceedings of the 39th IEEE/ACM Intern. Conf. on Automated Software Engineering*. ACM (2024), 40–52.
11. Hong, J. and Ryu, S. Type-migrating C-to-Rust translation using a large language model. *Empirical Software Engineering 30*, 1 (Oct. 2024).
12. Hutt, T. Would Rust secure cURL? (Jan. 16, 2021); https://blog.timhutt.co.uk/curl-vulnerabilities-rust/
13. Isaiah, A. Rewriting the GNU Coreutils in Rust. LWN.net (Jun. 8, 2021); https://lwn.net/Articles/857599
14. ISO. ISO/IEC 9899:1999 Programming languages – C (1999).
15. Jung, R. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang. 2*, POPL (Dec. 2017).
16. Lankes, S., Breitbart, J., and Pickartz, S. Exploring Rust for unikernel development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*. ACM (2019), 8–15.
17. Larabel, M. The first Rust-written network PHY driver set to land in Linux 6.8. Phoronix (Dec. 17, 2023); https://www.phoronix.com/news/Linux-6.8-Rust-PHY-Driver
18. Lee, P. Open sourcing our Go libraries. Dropbox.Tech (Jul. 1, 2014); https://dropbox.tech/infrastructure/open-sourcing-our-go-libraries
19. Levy, A. et al. Multiprogramming a 64KB computer safely and efficiently. In *Proceedings of the 26th Symp. on Operating Systems Principles*. ACM (2017), 234–251.
20. Lin, Y. et al. Rust as a language for high performance GC implementation. In *Proceedings of the 2016 ACM SIGPLAN Intern. Symp. on Memory Management*. ACM (2016), 89–98.
21. Matsakis, N.D. and Klock, F.S. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conf. on High Integrity Language Technology*. ACM (2014), 103–104.
22. Narayanan, V. et al. RedLeaf: Isolation and communication in a safe operating system. In *Proceedings of the 14th USENIX Conf. on Operating Systems Design and Implementation*. USENIX (2020).
23. Nitin, V. et al. C2SaferRust: Transforming C Projects into safer Rust with neurosymbolic techniques. *arXiv:2501.14257 [cs.SE]* (2025); https://arxiv.org/abs/2501.14257
24. Office of the National Cyber Director. Back to the Building Blocks: A Path Toward Secure and Measurable Software (2024); https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf
25. Shetty, M. et al. Syzygy: Dual code-test C to (safe) Rust translation using LLMs and dynamic analysis. *arXiv:2412.14234 [cs.SE]* (2024); https://arxiv.org/abs/2412.14234
26. Shiraishi, M. and Shinagawa, T. Context-aware code segmentation for C-to-Rust translation using large language models. *arXiv:2409.10506 [cs.SE]* (2024); https://arxiv.org/abs/2409.10506
27. De Simone, S. Linux 6.1 officially adds support for Rust in the kernel. InfoQ (Dec. 20, 2022); https://www.infoq.com/news/2022/12/linux-6-1-rust
28. Thomas, G. A proactive approach to more secure code. MSRC Blog (Jul. 16, 2019); https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/
29. Venners, B. Twitter on Scala: A conversation with Steve Jenson, Alex Payne, and Robey Pointer. Artima (Apr. 3, 2009); https://www.artima.com/articles/twitter-on-scala
30. Wallach, D. TRACTOR: Translating all C to Rust. DARPA (2024); https://www.darpa.mil/program/translating-all-c-to-rust
31. Yang, A.Z.H. et al. VERT: Verified equivalent Rust transpilation with large language models as few-shot learners. *arXiv:2404.18852 [cs.PL]* (2024); https://arxiv.org/abs/2404.18852
32. Yu, Y., d'Antras, A., and Bui, N.D.Q. Our Rust mission at Huawei (2021); https://trusted-programming.github.io/2021/02/07/our-rust-mission-at-huawei.html
33. Zhang, H. et al. Ownership guided C to Rust translation. In *Computer Aided Verification*. C. Enea and A. Lal (Eds.). Springer, Cham (2023), 459–482.

Jaemin Hong (jaemin.hong@kaist.ac.kr) is a research associate at KAIST, Daejeon, South Korea.

Sukyoung Ryu is a professor at KAIST, Daejeon, South Korea.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2025 Copyright held by the owner/author(s).

Watch the authors discuss this work in the exclusive *Communications* video. https://cacm.acm.org/videos/translating-c-to-rust