Cpp2Rust: Automatic Translation of C++ to Safe Rust

LUCIAN POPESCU, FRANCISCO GOUVEIA, HENRIQUE PRETO, and JOÃO SILVEIRA, INESC-ID, Portugal and Instituto Superior Técnico, University of Lisbon, Portugal
DMYTRO HRYBENKO, Google, Germany
JOSÉ FRAGOSO SANTOS and NUNO P. LOPES, INESC-ID, Portugal and Instituto Superior Técnico, University of Lisbon, Portugal

About 70% of security vulnerabilities in widely deployed software originate from memory-safety bugs in languages such as C and C++. Despite decades of investment in mitigations, from static analysis and sanitizers to hardware isolation, attackers continue to exploit unsafe memory operations. A promising long-term solution is to migrate existing C++ codebases to memory-safe languages such as Rust, but doing so manually is prohibitively expensive and error-prone.

In this paper, we present Cpp2Rust, the first system capable of translating C++ programs into functionally equivalent and memory-safe Rust code automatically. By trading some performance for security, Cpp2Rust addresses the fundamental mismatch between C++'s unrestricted aliasing and Rust's ownership model by inserting runtime-enforced ownership and mutability checks, ensuring safety while preserving semantics. To mitigate the performance overhead of dynamic checks, we developed a suite of source-to-source optimizations for Rust code that eliminate redundant ownership operations and recover much of the lost performance.

We evaluate Cpp2Rust on two real-world C++ programs, totaling 13k lines of code: WOFF2, a font compression library, and Brunsli, a JPEG lossless compression library. Cpp2Rust achieves full memory safety with only a 2% performance penalty on WOFF2 compression, while being 6× slower on Brunsli due to heavy usage of pointer arithmetic. These results demonstrate that automated, semantics-preserving translation from C++ to safe Rust is practical for some safety-critical applications, offering a viable path toward eliminating memory-safety vulnerabilities in legacy systems.

CCS Concepts: • **Software and its engineering → Source code generation**; **Translator writing systems and compiler generators**; **Software evolution**; • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Automated Program Translation, Memory Safety Retrofitting, Safe Rust

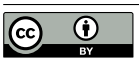
ACM Reference Format:
Lucian Popescu, Francisco Gouveia, Henrique Preto, João Silveira, Dmytro Hrybenko, José Fragoso Santos, and Nuno P. Lopes. 2026. Cpp2Rust: Automatic Translation of C++ to Safe Rust. *Proc. ACM Program. Lang.* 1, PLDI (June 2026), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Major software vendors, such as Google and Microsoft, have reported that about 70% of the security vulnerabilities in their products are related to memory-safety bugs [58, 81]. These vulnerabilities include both spatial and temporal memory-safety issues, such as out-of-bounds accesses, use-after-free errors, double frees, and data races.

Over the years, researchers and practitioners have developed a wide range of software- and hardware-based technologies to mitigate memory-safety attacks. These advances, from compiler

Authors' Contact Information: Lucian Popescu; Francisco Gouveia; Henrique Preto; João Silveira, INESC-ID, Portugal and Instituto Superior Técnico, University of Lisbon, Portugal; Dmytro Hrybenko, Google, Germany; José Fragoso Santos; Nuno P. Lopes, nuno.lopes@tecnico.ulisboa.pt, INESC-ID, Portugal and Instituto Superior Técnico, University of Lisbon, Portugal.



ACM 2475-1421/2026/6-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

instrumentation and hardened allocators [1, 2, 7, 14, 19, 20, 53, 63, 68, 76, 82] to pointer capabilities and trusted execution environments [13, 33, 64, 87, 88], have made exploitation significantly more difficult. Nevertheless, attackers continue to discover ways to bypass these defenses. Because security is often an all-or-nothing property, defending against individual classes of vulnerabilities remains an endless game of whack-a-mole.

Memory-safe languages eliminate most vulnerabilities related to memory safety by design. They achieve this by disallowing potentially unsafe operations at compile time through the type system, and at run time via compiler-generated instrumentation and through a runtime library (e.g., a garbage collector). However, due to the perceived performance costs of memory-safe languages, such as run-time checks, developers have traditionally implemented performance-critical components in non-memory-safe languages like C and C++. Unfortunately, these components are often also the most security-sensitive, leaving critical parts of the software ecosystem exposed to memory-safety vulnerabilities.

Rust, a relatively recent memory-safe language, stands out for its performance. Manual rewrites of legacy applications in Rust have shown that it can often match or even exceed the performance of the original application [3, 27, 49]. This, along with its memory safety, has sparked significant interest in the industry [15, 34, 48, 50, 69, 72, 84, 85]. Several organizations are already using Rust for writing performance- and security-sensitive software. For example, the Linux kernel now supports drivers written in Rust, marking a major step towards integrating the language in critical systems [11, 39, 52, 65].

Rewriting all software in memory-safe languages is neither quick nor practical. Large-scale rewrites often fail, which is why older languages like COBOL remain widely used [17, 41, 46, 79, 80, 90]. Even when feasible, manual rewrites risk introducing errors. Developers aiming to improve clarity may unintentionally alter functionality they do not fully understand, leading to bugs. Additionally, experts in the old language may not be proficient in the new one, and vice versa, further complicating the transition.

Automatically translating (or transpiling) applications to memory-safe languages is an emerging area of research aimed at simplifying the transition and avoiding the pitfalls of manual rewrites For example, C2Rust [40] is the leading tool for translating C code into Rust. However, C2Rust has a major drawback: it generates Rust code outside the language's safe subset, thereby inheriting most of the vulnerabilities present in the original program [89].

Translating code from a non-memory-safe language to safe Rust is generally undecidable due to the need to prove properties like ownership in the presence of aliasing. This explains why C2Rust generates unsafe Rust code. Recently, researchers have aimed to make the code produced by C2Rust "safer" by, e.g., replacing pointers (unsafe Rust) with references (safe Rust) [22, 23, 95]. However, as we have noted before, security is often an all-or-nothing property, meaning that leaving even small amounts of vulnerable code can still open the door for attackers to compromise the entire system.

In this paper, we take the opposite approach: we first generate safe Rust code, and then iteratively refine it to improve performance and readability. To sidestep the undecidability issue, we shift Rust's mutability and ownership checks to run time by using reference counting and dynamic mutability checks. This trades speed for security, a compromise many software vendors already make by enabling compiler-based hardening by default.

We also introduce a new source-to-source optimization tool for safe Rust programs that use dynamic mutability and ownership. The tool removes reference counters when it detects single ownership and converts dynamic mutability checks into static ones.

In summary, this paper makes the following contributions:

(1) Cpp2Rust, the first tool that can translate programs written in a subset of C++ into equivalent safe Rust programs automatically. Cpp2Rust supports functions, lambdas, enums, loops, pointer arithmetic, abstract classes, inheritance, virtual calls, and several STL types.
(2) A set of source-to-source optimizations for safe Rust code using dynamic ownership and mutability.
(3) Two case studies on translating two C++ applications *fully automatically*: WOFF2, a font compression library, and Brunsli, a lossless JPEG compressor.

2 Background On Rust

We begin by introducing three fundamental concepts of Rust—ownership, borrowing, and mutability tracking—that shape its unique balance of expressiveness, performance, and safety. These features, while powerful, present significant challenges when generating Rust code automatically.

2.1 Ownership and Borrowing

In Rust, each variable must have a single owner, and it must be known statically. This is a key feature that allows Rust to be memory safe without a run-time garbage collector. As a result, the compiler knows exactly when each object must be deallocated.

Ownership can be moved, but not shared [43, 44]. However, there is a mechanism to borrow an object temporarily by creating a non-owning reference. The following code snippet shows how ownership is tracked, moved, and borrowed:

```
fn moves(s: String) { ... }
fn borrows(s: &String) { ... }
fn f() {
  let x = String::from("foo");  // create a new string
  let y = x;                    // moves ownership; invalidates x
  let z = y.clone();            // makes a copy of y
  moves(y);                     // moves ownership; invalidates y
  borrows(&z);                  // temporary borrow; z is ok
  print!("{x}");                // compile-time error: x is invalid
  print!("{y}");                // compile-time error: y is invalid
  print!("{z}");                // ok
}
```

Function `moves` takes the ownership of the argument. Arguments are automatically deleted before the function returns. Hence, in function `f`, `y` is considered invalid after the call to `moves`.

2.2 Mutability

A well-typed Rust program is data-race free (with some caveats). This is ensured by restricting the program to have just immutable references to a variable, or a single mutable reference and no immutable references. The following example illustrates these constraints:

```
fn f() {
  let mut x = String::from("foo");  // create a new string
  let r: &String = &x;              // create 1st immutable reference
  let s: &String = &x;              // create 2nd immutable reference
  let t: &mut String = &mut x;      // error: there are live references
  print!("{x} {r} {s} {t}");        // all ok except t
}
```

In Rust, references are only live until the last use, unlike most languages where variables are live until the end of the scope. This allows several mutable references for a same variable to co-exist as long as the liveness of the variables are disjoint:

```
fn f() {
    let mut x = String::from("foo");  // create a new string
```

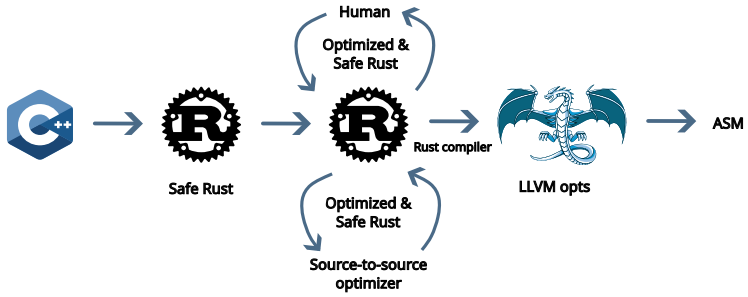


Fig. 1. Architecture of Cpp2Rust. A C++ program is first translated into memory-safe Rust. An automatic source-to-source optimizer then removes redundant ownership constructs and unnecessary boxing. A human may optionally edit the generated Rust code and rerun the optimizer to further improve performance. Finally, the resulting Rust code is compiled to assembly using the Rust compiler and its LLVM backend.

```
    let y: &mut String = &mut x;      // create a mutable reference to x
    y.push('a');                      // append a new character
                                      // last use of y; reference is destroyed

    // create another mutable reference to x. OK; there are no other live references
    let z: &mut String = &mut x;
    z.pop();
    let r: &String = &x;              // OK; there are no other live references
    print!("{r}");
}
```

We now illustrate why syntactic mutability tracking is challenging and requires programs to be written following a specific discipline. The following code does not compile, despite having no data races. Still, Rust forbids us from creating a second temporary mutable reference to pass to called functions.

```
// simple function; doesn't escape reference nor has concurrency
fn g(x: &mut String) {
    x.pop();
}
fn f() {
  let mut x = String::from("foo");  // create a new string
  let y: &mut String = &mut x;      // create 1st mutable reference
  g(&mut x);                        // error: can't create a 2nd mutable reference
  y.pop();                          // y reference live until here
  print!("{x}");
}
```

We hope this brief introduction to Rust's unique features—*syntactic* tracking of ownership and mutability—highlights the complexity involved in porting code from other languages to Rust. We illustrate these challenges further in the next section.

3 Overview

In this section, we present by example our algorithm for translating a subset of C++ to safe Rust.

Fig. 1 shows the architecture of the proposed solution. First, we generate safe Rust code directly from a C++ AST in a mechanical and straightforward way. Then, we employ a source-to-source optimizer for safe Rust code that removes unneeded boxing and makes the code more idiomatic. Since this optimization task is undecidable in general, we envision having a loop with a human, where developers can make changes to the code and then re-run the source-to-source optimizer.

3.1 Variables

Let us start with a simple example of a function that adds two integers. On the left we have the input C++ code, and on the right the automatically generated code:

```cpp
int f() {
  int a = 2;
  int b;
  b = 3;
  return a + b;
}
```

```rust
pub fn f() -> i32 {
    let a: Value<i32> = Rc::new(RefCell::new(2));
    let b: Value<i32> = Rc::new(RefCell::new(0));
    *b.borrow_mut() = 3;
    return *a.borrow() + *b.borrow();
}
```

We translate every variable into the `Value<T>` type, which is an alias to `Rc<RefCell<T>>` and is defined in our runtime library. This is Rust's idiomatic way to defer ownership and mutability checks to run time. `Rc` is a reference counter, and thus enables shared ownership and creation of an arbitrary number of references. `RefCell` defers mutability checks to run time, allowing multiple mutable references with *syntactically* overlapping lifetimes to exist, as long as their lifetimes do not overlap at run time. Mutability checks are done by `borrow()` and `borrow_mut()`, which fail if the reference lifetime rules are violated.

The translated code is not very idiomatic, but it accurately reflects how our algorithm works. This ensures that any program can be translated. Afterwards, we run a source-to-source optimizer to eliminate unnecessary constructs, producing code that a Rust developer would naturally write:

```rust
pub fn f() -> i32 {
    let a: i32 = 2;
    let mut b: i32 = 0;
    b = 3;
    return a + b;
}
```

When calling a function, arguments passed by value in C++ are passed with their raw types in Rust (no `Value<T>` boxing). The same applies to the return value:

```cpp
int g(int a) {
  return a + 1;
}

int f() {
  return g(3);
}
```

```rust
pub fn g(a: i32) -> i32 {
    let a: Value<i32> = Rc::new(RefCell::new(a));
    return *a.borrow() + 1;
}
pub fn f() -> i32 {
    return g(3);
}
```

We add a preamble to function definitions that shadows the arguments with a boxed value, as all variables must be boxed for our translation algorithm to work. Boxing variables locally instead of passing boxed variables as arguments has the advantage that the unboxing optimization can work intra-procedurally. This design decision is justified by the fact that most variables can be unboxed since it is not common for variables to have their address taken.

3.2 Pointers and Arrays

Rust has both references and pointers. As we have seen, references carry a lot of syntactic restrictions in terms of tracking ownership and meeting the borrow rules. Translating arbitrary C++ code into Rust code using exclusively references is thus undecidable. Even when it is possible, it could require changing the structure of the program significantly. On the other hand, most operations involving pointers are not in the safe subset of Rust.

To workaround these issues, we introduce a `Ptr<T>` type in our runtime library, where `T` stands for the allocated type, or for the allocated element type in the case of arrays. The `Ptr<T>` type allows

pointer arithmetic, array-to-pointer decay, dangling references, etc, all in safe Rust. The definition of the type is as follows:

```
enum PtrKind<T> {
    #[default]
    Null,
    StackSingle(Weak<RefCell<T>>),
    StackArray(Weak<RefCell<Box<[T]>>>),
    HeapSingle(Weak<RefCell<T>>),
    HeapArray(Weak<RefCell<Box<[T]>>>),
    ...
}
pub struct Ptr<T> {
    offset: usize,
    kind: PtrKind<T>,
}
```

Essentially, a pointer is a pair containing an offset (for pointer arithmetic), and a *weak* reference to a `Value<T>`. Please ignore the distinction between heap and stack allocations for now.

A weak reference is obtained from an `Rc`. These references allow an `Rc` to be deleted when the strong count becomes zero. If a weak reference is accessed and the inner object was deleted, the program is aborted. Let us illustrate how pointers are translated with the following example:

```
void g(int *p) {
  *p = 4;
}

int f() {
  int a = 2;
  int b[3];
  int *p = &a;
  int *q = b + 1;
  g(q);
  // returns 6
  return *p + *q;
}
```

```
pub fn g(p: Ptr<i32>) {
    let p: Value<Ptr<i32>> = Rc::new(RefCell::new(p));
    *p.borrow().deref() = 4;
}
pub fn f() -> i32 {
    let a: Value<i32> = Rc::new(RefCell::new(2));
    let b: Value<Box<[i32]>> = Rc::new(RefCell::new(Box::new([0; 3])));
    let p: Value<Ptr<i32>> = Rc::new(RefCell::new(a.as_pointer()));
    let q: Value<Ptr<i32>> = Rc::new(RefCell::new(
        b.as_pointer().offset(1)));
    g(q.borrow().clone());
    return *p.borrow().deref() + *q.borrow().deref();
}
```

The example illustrates several key pointer-related features. We extend the `Value<T>` type so it offers an `as_pointer()` method, which essentially converts the `Rc` into a `Ptr` (a weak reference). The `Ptr<T>` type offers several methods, including `offset()`, which increments its `offset` field, and `deref()`, which returns a mutable reference to the underlying object (adjusted to the offset).

One fundamental feature shown in the example is that arrays have a single reference counter for the whole array (and not one per element). This reduces the overhead (memory and performance) of our translation. Hence, the `Ptr<T>` type holds a reference to the whole array plus an offset, indicating the exact element the pointer points to.

The reason for using weak rather than strong references for `Ptr<T>` is because we wanted to maintain precise execution of destructors. In C++, the common RAII pattern for resource acquisition/control requires executing the destructors in precise locations, otherwise the code may behave differently. For example, an object may release a mutex when the destructor is called. Hence, we could not have used strong references, as they could prolong the lifetime of some objects and execute the destructors later than expected. If a program retains a pointer to a destroyed object, the weak reference becomes dangling and will cause the program to abort if dereferenced. Therefore, this translation ensures memory safety, while reflecting the semantics of C++ accurately.

C++ references are non-null constant pointers, i.e., they can be used to change the data they point to, but the pointer itself cannot be changed. We leverage this restriction to skip emission of `Value<T>` boxing, and thus reduce the overhead:

```cpp
int f() {
  int a = 1;
  int &p = a;
  p = 2;
  return a; // 2
}
```

```rust
pub fn f() -> i32 {
    let a: Value<i32> = Rc::new(RefCell::new(1));
    let p: Ptr<i32> = a.as_pointer(); // no Value<T> boxing
    *p.deref() = 2;
    return *a.borrow();
}
```

3.3 Heap Memory Allocation

A first important note is that a Rust program that leaks memory is well typed. This simplifies the translation from C++. Calls to `malloc`/`new` are translated into `Ptr::alloc`, which is defined as follows:

```rust
impl<T> Ptr<T> {
    pub fn alloc(value: T) -> Self {
        let owner = Rc::new(RefCell::new(value));
        let weak = Rc::downgrade(&owner);
        let _ = Rc::into_raw(owner);
        Self {
            offset: 0,
            kind: PtrKind::HeapSingle(weak),
        }
    }
}
```

The trick here is that Rust allow us to leak a strong reference using the `Rc::into_raw()` function. Therefore, the `Rc` has a strong reference count of 1 when the function returns. The `Ptr::delete()` function can then recover that reference (using `Rc::from_raw()`) and delete it.

Allocation using `std::unique_ptr` is more straightforward, since deallocation happens automatically. The use of the `Option<T>` type is required because `std::unique_ptr` may hold a null pointer, while `Value<T>` must own an object.

```cpp
int f() {
  auto v = std::make_unique<int>(8);
  *v = 9;
  return *v;
}
```

```rust
pub fn f() -> i32 {
    let v: Value<Option<Value<i32>>> =
        Rc::new(RefCell::new(Some(Rc::new(RefCell::new(8)))));
    *v.borrow_mut().as_ref().unwrap().borrow_mut() = 9;
    return *v.borrow().as_ref().unwrap().borrow();
}
```

3.4 Inheritance and Virtual Dispatch

C++ and Rust offer different approaches to polymorphism. For example, Rust has traits, while C++ has abstract classes; the two are not directly comparable. Rust's traits only define an API, while C++ classes can also have common fields for derived classes. In the current version of Cpp2Rust, we support only the subset of C++ OOP idioms that can be mapped straightforwardly to Rust.

The following example has an abstract class (all methods are pure virtual), and a derived class that implements those methods. The call to function `getX` in function `f` does an implicit upcast to `Base*`. The call in `getX` is dispatched at run time since the compiler does not know statically to which class variable `x` refers to.

We implement a `PtrDyn<dyn T>` type in the runtime library to represent pointers to virtual classes. Since traits are not sized, Rust uses the `dyn` keyword to indicate that the type is not sized, and that it should use dynamic dispatch. The method `Ptr::to_strong()` upgrades the weak reference in the pointer to a strong reference, essentially transforming a `Ptr<T>` into a `Value<T>`. The method

`as_pointer_dyn()` transforms a `Value<dyn T>` into a `PtrDyn<dyn T>`. This gymnastics is required to workaround some of Rust's restrictions without imposing more overhead on the `Ptr<T>` type.

```
class Base {
public:
  virtual int get() = 0;
};
class Derived : public Base {
  int a;
public:
  int get() override {
    return a;
  }
};
int getX(Base *x) {
  return x->get();
}
int f() {
  Derived *p = new Derived;
  // implicit upcast to Base*
  return getX(p);
}
```

```
pub trait Base {
    fn get(&self) -> i32;
}
pub struct Derived {
    pub a: Value<i32>,
}
impl Base for Derived {
    fn get(&self) -> i32 { return *self.a.borrow(); }
}
pub fn getX(x: PtrDyn<dyn Base>) -> i32 {
    let x: Value<PtrDyn<dyn Base>> = Rc::new(RefCell::new(x));
    return x.borrow().deref().get();
}
pub fn f() -> i32 {
    let p: Value<Ptr<Derived>> = Rc::new(RefCell::new(
        Ptr::alloc(Derived::default())));
    return getX((p.borrow().to_strong() as Value<dyn Base>)
                    .as_pointer_dyn());
}
```

3.5 Optimizing Dynamic Ownership and Mutability Checks

Cpp2Rust generates code in a straightforward manner, without applying optimizations to the output. This design choice simplifies the implementation of Cpp2Rust, reducing its complexity and minimizing the risk of introducing optimization-related bugs. Instead of integrating optimizations directly into Cpp2Rust, we opted to develop a source-to-source optimizer for safe Rust programs. Although this optimizer is tailored for the code generated by Cpp2Rust, it can be used with any Rust program.

The main purpose of the `Value<T>` type is to facilitate the creation of multiple pointers to the same variable without violating Rust's mutability rules, which allow only one *syntactic* mutable reference at a time. However, if a variable never has its address taken, the `Value<T>` boxing is not needed. In such cases, the method `as_pointer()` is never invoked on the variable, and thus the optimizer removes the `Value<T>` boxing.

Replacing our smart pointer type (`Ptr<T>`) with Rust's native references is not trivial, particularly when pointers are passed as function arguments. To eliminate the overhead associated with smart pointers, all pointers related to a `Value<T>` must be removed to allow the variable to be unboxed. The all-or-nothing nature of this transformation makes the problem particularly challenging. Addressing this type of optimization is left for future work. We show an example below of the kind of code that is currently optimized:

```
fn f() -> i32 {
    let a: Value<i32> = Rc::new(RefCell::new(2));
    let b: Value<i32> = Rc::new(RefCell::new(0));
    // Cannot convert these two pointers to mutable
    // references because their liveness overlaps.
    let p: Ptr<i32> = b.as_pointer();
    let q: Ptr<i32> = b.as_pointer();
    *p.deref() = 3;
    *q.deref() = 4;
    return *a.borrow() + *b.borrow();
}
```

```
fn f() -> i32 {
    let a: i32 = 2;
    // The optimizer doesn't unbox this one
    let b: Value<i32> =
        Rc::new(RefCell::new(0));
    let p: Ptr<i32> = b.as_pointer();
    let q: Ptr<i32> = b.as_pointer();
    *p.deref() = 3;
    *q.deref() = 4;
    return a + *b.borrow();
}
```

4 Translating C++ to Safe Rust

In this section, we present the algorithm of Cpp2Rust to translate a subset of C++ to safe Rust. The algorithm supports the following C++11 constructs:

- Primitive types, enums, `const`/`constexpr`, `typedef`/`using`, namespaces, global and function-static variables, functions (overloading, default arguments, pass-by-value/reference).
- Pointers, function pointers, pointer arithmetic, arrays, `new`/`delete`, `std::unique_ptr`, `sizeof`.
- Classes/structs: fields, default constructors, copy/move constructors, destructors, static members, nested classes, default member initializers. Inheritance from pure-virtual base classes, virtual dispatch.
- Templates via full instantiation. Lambdas with capture-by-value and capture-by-reference.
- Casts: `static_cast`, C-style casts, `void*` round-trip casts.
- STL: `std::vector`, `std::map`, `std::string`, `std::deque`, `std::pair`, `std::initializer_list`, iterators, `std::move`, `memcpy`/`memset` (size must be a multiple of the pointee type).

Notable unsupported constructs include: `union`, `volatile`, `goto`, exceptions, bitfields, placement `new`, user-defined copy/move constructors, `dynamic_cast`, `const_cast`, base classes with fields or non-virtual methods, multiple inheritance, and multi-threaded code.

4.1 Translation of Types

We start by presenting the translation of C/C++ types to Rust types (Fig. 2). Basic types (`int`, `float`, etc) are translated one-to-one to their Rust counterparts (`i32`, `f32`, etc). C/C++ strings (`char` arrays and `std::string`) are converted into a vector of `u8` integers rather than using Rust's `String` type, which represents a well-formed UTF-8 string. C/C++ strings are low-level and support manipulation of individual bytes irrespective of the string encoding. For some projects, using Rust's `String` type would be more appropriate and more idiomatic. However, for projects like WOFF2 which use strings to hold binary data (fonts) that are not valid UTF-8 strings, we have to resort to byte vectors.

Composite types are boxed until the last type. For example, '`std::vector<int> v`' is translated to '`let v: Value<Vec<i32>>`'. This shares a single reference counter among all elements of the vector. Nested types, such as vectors of vectors '`std::vector<std::vector<int>> v`', are translated as '`let v: Value<Vec<Value<Vec<i32>>>>`', allowing the code to create a pointer to any element of the vector; we will expand on this later in this section.

Our runtime library offers a type called `AnyPtr` to represent `void*` pointers, which allows casts back to the original type and operations over `void*` pointers, such as `memcpy` and `memset`. Other pointers are represented using our smart-pointer type (`Ptr`), except for pointers to virtual classes. We translate such classes to Rust traits, which have no fields and thus are not sized. Rust requires the usage of the `dyn` keyword to hold pointers to traits, hence our runtime library exposes a dedicated type (`PtrDyn`) to avoid additional complexity and overhead in the generic `Ptr` type.

We formalize the translation of C/C++ types to Rust types as function $\mathcal{T}(T)$ that, given a C/C++ type as an argument, produces a Rust type. The translation rules are given in Fig. 2. We use T to range over C/C++ types and t to range over Rust types. The main translation function makes use of three auxiliary conversion functions: (1) $\mathsf{BoxIfComp}(t)$ that converts the type t to a `Value` type provided that it is composite (e.g., $\mathsf{BoxIfComp}$(`Vec<i32>`) = `Value<Vec<i32>>` and $\mathsf{BoxIfComp}$(`i32`) = `i32`); (2) $\mathsf{Pointee}(t)$ that extracts the pointee type of a boxed type (e.g., when applied to the type `[i32]`, this function returns `i32`); and (3) $\mathsf{IsPrimitive}(t)$ to check if t is a primitive type.

$$\textsc{Int}\quad \mathcal{T}(\text{int}) \triangleq \text{i32} \qquad \textsc{String}\quad \mathcal{T}(\text{std::string}) \triangleq \text{Vec<u8>}$$

$$\textsc{Unique Pointer}\quad \frac{\mathcal{T}(T) = t \qquad \text{Box}(t) = t'}{\mathcal{T}(\text{std::unique\_ptr<}T\text{>}) \triangleq \text{Option<}t'\text{>}}$$

$$\textsc{Const-Sized Array}\quad \frac{\mathcal{T}(T) = t \qquad \text{BoxIfComp}(t) = t'}{\mathcal{T}(T[n]) \triangleq [t'; n]} \qquad \textsc{Var-Sized Array}\quad \frac{\mathcal{T}(t) = T \qquad \text{BoxIfComp}(t) = t'}{\mathcal{T}(T[]) \triangleq \text{Box<}[t']\text{>}}$$

$$\textsc{Vector}\quad \frac{\mathcal{T}(T) = t \qquad \text{BoxIfComp}(t) = t'}{\mathcal{T}(\text{std::vector<}T\text{>}) \triangleq \text{Vec<}t'\text{>}} \qquad \textsc{Void*}\quad \mathcal{T}(\text{void}*) \triangleq \text{AnyPtr}$$

$$\textsc{Ptr - Non-Virtual Class}\quad \frac{\mathcal{T}(T) = t \qquad T \text{ is not a virtual class} \qquad \text{Pointee}(t) = t'}{\mathcal{T}(T*) \triangleq \text{Ptr<}t'\text{>}} \qquad \textsc{Ptr - Virtual Class}\quad \frac{\mathcal{T}(T) = t \qquad T \text{ is a virtual class} \qquad \text{Pointee}(t) = t'}{\mathcal{T}(T*) \triangleq \text{PtrDyn<dyn }t'\text{>}}$$

$$\textsc{Pointer Decay}\quad \frac{t = [t'; n] \textbf{ or } t = \text{Box<}[t']\text{>} \textbf{ or } t = \text{Vec<}t'\text{>} \textbf{ or } t = \text{Value<}t'\text{>}}{\text{Pointee}(t) \triangleq t'} \qquad \textsc{Pointer - Primitive}\quad \frac{\text{IsPrimitive}(t)}{\text{Pointee}(t) \triangleq t}$$

$$\textsc{Box - Static Array}\quad \frac{t = [t'; n]}{\text{Box}(t) \triangleq \text{Value<Box<}[t']\text{>>}} \qquad \textsc{Box - Default}\quad \frac{t \neq [t'; n]}{\text{Box}(t) \triangleq \text{Value<}t\text{>}} \qquad \textsc{BoxIfComp - Primitive}\quad \frac{\text{IsPrimitive}(t)}{\text{BoxIfComp}(t) \triangleq t}$$

$$\textsc{BoxIfComp - Composite}\quad \frac{\neg\text{IsPrimitive}(t) \qquad t' = \text{Box}(t)}{\text{BoxIfComp}(t) \triangleq t'}$$

Fig. 2. Definition of the $\mathcal{T}(T)$ function, which maps C/C++ types to Rust types (abbreviated).

4.2 Translation of Expressions

Fig. 3 presents the conversion of C++ expressions to Rust. The conversion functions $\mathcal{E}_\tau(e)$ return a pair containing a list of statements required to evaluate the expression, and a Rust expression denoting the value to of the original expression.

To simplify the presentation, the result of evaluating an expression is always stored in a temporary variable. However, our implementation optimizes the translation and avoids the creation of temporary variables whenever possible. For example, the assignment '`x = y + z`' is translated directly to '`*x.borrow_mut() = *y.borrow() + *z.borrow()`' as long as `y`, `z` are different than `x`. Rust's mutability and lifetime rules mean that the three borrows are evaluated in the same scope and thus the right-hand side cannot borrow the same variable to which it is assigned (e.g., '`*x.borrow_mut() = *x.borrow() + 1`' traps the program at run time).

The translation of expressions is separated in three cases: lvalues, rvalues, and addresses. Each case has its own associated translation function, respectively $\mathcal{E}_{\text{lval}}$, $\mathcal{E}_{\text{rval}}$, and $\mathcal{E}_{\text{addr}}$. The main difference between the translation of lvalues and rvalues is that for lvalues we need to use `borrow_mut()` to obtain a writable reference, whereas for rvalues using `borrow()` is sufficient.

RValue - Global

$$\frac{\mathsf{isGlobal}(v) \qquad \mathsf{fresh}(v') \qquad s' = \texttt{let } v' = *v\texttt{.with(Value::clone).borrow()}}{\mathcal{E}_{\mathsf{rval}}(v) \triangleq (s', v')}$$

RValue - Local

$$\frac{\neg\mathsf{isGlobal}(v) \qquad \mathsf{fresh}(v') \qquad s' = \texttt{let } v' = *v\texttt{.borrow()}}{\mathcal{E}_{\mathsf{rval}}(v) \triangleq (s', v')}$$

LValue - Local

$$\frac{\neg\mathsf{isGlobal}(v) \qquad \mathsf{fresh}(v') \qquad s' = \texttt{let } v' = *v\texttt{.borrow\_mut()}}{\mathcal{E}_{\mathsf{lval}}(v) \triangleq (s', v')}$$

AddrOf - Local

$$\frac{\neg\mathsf{isGlobal}(v) \qquad \mathsf{fresh}(v') \qquad s' = \texttt{let } v' = v\texttt{.as\_pointer()}}{\mathcal{E}_{\mathsf{addr}}(v) \triangleq (s', v')}$$

Binop

$$\frac{\mathcal{E}_{\mathsf{rval}}(e_i) = (s_i, v_i)\ |_{i=1}^{2} \qquad \mathsf{fresh}(v) \qquad s = s_1; s_2; \texttt{let } v = v_1 \oplus v_2}{\mathcal{E}_{\mathsf{rval}}(e_1 \oplus e_2) \triangleq (s, v)}$$

Call

$$\frac{\mathcal{E}_{\mathsf{rval}}(e_i) = (s_i, v_i)\ |_{i=1}^{n} \qquad \mathsf{fresh}(v) \qquad s = s_1; ...; s_n; \texttt{let } v = f(v_1, ..., v_n)}{\mathcal{E}_{\mathsf{rval}}(f(e_1, ..., e_n)) \triangleq (s, v)}$$

Deref

$$\frac{\mathcal{E}_{\mathsf{rval}}(e) = (s, v) \qquad \mathsf{fresh}(v') \qquad v' = *v\texttt{.deref()}}{\mathcal{E}_{\mathsf{lval/rval}}(*e) \triangleq (s, v')}$$

AddrOf-Deref

$$\frac{\mathcal{E}_{\mathsf{addr}}(e) = e'}{\mathcal{E}_{\mathsf{addr}}(*e) \triangleq e'}$$

AddrOf

$$\frac{\mathcal{E}_{\mathsf{addr}}(e) = e'}{\mathcal{E}_{\mathsf{rval}}(\&e) \triangleq e'}$$

Fig. 3. Selected cases from the definition of the functions $\mathcal{E}_{\mathsf{lval}}(e)$, $\mathcal{E}_{\mathsf{rval}}(e)$, and $\mathcal{E}_{\mathsf{addr}}(e)$, which map C++ expressions to Rust (respectively, for lvalues, rvalues, and addresses).

The need for a specific translation for addresses stems from the fact that the address-taking operation must be pushed to the innermost expressions to which it is applied, even when it is deeply nested inside a complex composite expression. For example, when translating the expression '`&(cond ? *a : *b)`', we cannot first evaluate `*a` and `*b` and then use `.as_pointer()` on top of the conditional expression. The expression has to be converted to:

'`if cond { a.as_pointer() } else { b.as_pointer() }`'

Finally, in Rust, non-basic types are not copyable automatically; the programmer needs to call `clone()` explicitly. For example, our `Ptr` type falls into that category. Hence, when evaluating expressions of non-basic types that are assigned to temporary local variables or passed as function arguments, we must clone them explicitly. As an optimization, Cpp2Rust tracks whether the result of an expression is fresh or not, to avoid adding unnecessary clones. For example, the result of calling `v.as_pointer()` is a fresh pointer, while evaluating a pointer variable (e.g., `*p.borrow()`) is not.

4.3 Translation of Statements

Fig. 4 shows the translation of C++ statements to Rust. Function declarations include a preamble that shadows the arguments with boxed counterparts, with the same name, but wrapped within a `Value<T>`. Arguments are passed unboxed and are boxed on function entry. This matches C++'s semantics of passing arguments by value, and allows our optimizer to remove most boxing with an intra-procedural analysis.

As an optimization, references are not boxed, since these are essentially immutable pointers. Taking the address of a reference amounts to copying its (pointer) value.

Global variables are mapped to thread-local storage because Rust does not support boxed `Value<T>` objects as true globals. The optimizer replaces such variables with constants when their address is never taken and their value is constant.

$$\textsc{Expression Statement}\quad \frac{\mathcal{E}_{\text{rval}}(e) = (s, -)}{\mathcal{S}(e) \triangleq s} \qquad \textsc{Assign}\quad \frac{\mathcal{E}_{\text{rval}}(e_r) = (s_1, x_1) \qquad \mathcal{E}_{\text{lval}}(e_l) = (s_2, x_2) \qquad s' = s_1; s_2; x_2 = x_1}{\mathcal{S}(e_l = e_r) \triangleq s'}$$

$$\textsc{Global Variable Declaration}\quad \frac{\text{isGlobal}(v) \qquad \mathcal{E}_{\text{rval}}(e) = (s, e') \qquad \mathcal{T}(T) = t \qquad \text{Box}(t) = t' \qquad s' = \texttt{thread\_local!(static } v : t' = \texttt{Rc::new(RefCell::new(\{ } s; e' \texttt{ \}))\ )}}{\mathcal{S}(T\ v = e) \triangleq s'}$$

$$\textsc{Local Variable Declaration}\quad \frac{\neg\text{isGlobal}(v) \qquad \mathcal{E}_{\text{rval}}(e) = (s, e') \qquad \mathcal{T}(T) = t \qquad \text{Box}(t) = t' \qquad s' = \texttt{let } v : t' = \texttt{Rc::new(RefCell::new(\{ } s; e' \texttt{ \}))}}{\mathcal{S}(T\ v = e) \triangleq s'}$$

$$\textsc{Function Declaration}\quad \frac{t = \mathcal{T}(T) \qquad t_i = \mathcal{T}(T_i)\ |_{i=1}^{n} \qquad s_i' = \mathcal{S}(s_i)\ |_{i=1}^{m} \qquad t_i' = \text{Box}(t_i)\ |_{i=1}^{n} \qquad s_i'' = \texttt{let } x_i : t_i' = \texttt{Rc::new(RefCell::new(}x_i\texttt{))}\ |_{i=1}^{n} \qquad s' = s_1''; \ldots; s_n''; s_1'; \ldots; s_m'}{\mathcal{S}(T\ f(T_1\ x_1, \ldots T_n\ x_n)\{s_1; \ldots; s_m\}) \triangleq \text{pub fn } f(x_1 : t_1', \ldots, x_n : t_n') \rightarrow t\ \{\ s'\ \}}$$

Fig. 4. Definition of the function $\mathcal{S}(s)$, which maps C++ statements to Rust (abbreviated).

Class declarations are mapped straightforwardly to Rust structs. Fields are boxed to allow the creation of pointers to each field individually. Because of the boxing, we also emit an implementation of the `Clone` trait for each struct. The default implementation increments the reference counter and creates a reference to the original field rather than copying the value. If all fields get unboxed, the custom `Clone` can also be removed.

We currently handle C++ templates by fully instantiating them, which leads to code duplication. It is possible to leverage Rust's generics to avoid some or all of these instantiations, but we leave this for future work.

4.4 Translation of C++ Standard Library Calls

Translating calls into the C++ standard library currently requires some manual effort. We designed a simple internal DSL to describe the mapping between C++ types and functions to their Rust equivalents. Some functions map one-to-one to Rust, while others require a sequence of statements. For example, since we map `std::string` to `Vec<u8>`, most method calls require some work to maintain the invariant that the last character is 0 (the string terminator). In Rust, blocks of statements are expressions, which makes it easier to map a single C++ expression to multiple Rust statements.

Below are a few examples showing how some C++ calls (left) are mapped to Rust (right). The Rust code uses % for placeholders: %o is the object itself, %p is a Ptr<T> derived from the object (e.g., decay a vector to a pointer), and %a* refers to all arguments. We also support a ? placeholder in the C++ code to match anything; in the case below it means the pattern applies to any kind of allocator.

```
{"std::make_unique(_Args &&...)", "Some(Rc::new(RefCell::new(%a*)))"},
{"std::unique_ptr<T, ?>::get", "%o.as_pointer()"},
{"std::vector::begin", "%p"},
{"std::basic_string::append(size_type, char)", R"({
    %o.pop(); // remove the \0 string terminator
    %o.extend(std::iter::repeat(%a1).take((%a0) as usize));
    %o.push(0); %p})"}, // re-add the \0 string terminator
```

4.5 Runtime Library

Programs generated by Cpp2Rust make use of a small runtime library that implements auxiliary types and functions to simplify the generated code. The library exports the following API:

- `Ptr<T>`, which holds a reference to the allocated type `T`, and thus no pointer casts are supported (except to `void*`). `Ptr<T>` is a pair of a weak reference and an offset, so we can share a single reference counter for a whole array/vector. `Ptr<T>` supports a rich API for heap memory (de)allocation, updating the offset, and dereferencing the object.
- `AnyPtr` is used to represent `void*` pointers. It supports casts back to the original `Ptr<T>` type, as well as C functions that operate over `void*` pointers, such as `memcpy` and `memset`.
- `PtrDyn<dyn T>`: similar to `Ptr<T>`, but used to hold a virtual class' pointer.
- Iterators: we use the `Ptr` type to represent iterators, and a dedicated `StringIterator` type for the translation of `std::string::iterator`. The difference of the string iterator is that it skips the last null character.
- Pre/post inc/dec functions. Rust does not have the equivalent of C's `++v` and `v++` operators: it has just a '`v += 1`' statement (not an expression!). Hence, we define functions that emulate C's behavior.

4.6 Correctness and Safety Guarantees of the Translated Code

Cpp2Rust guarantees two properties of the translated code: memory safety and functional equivalence with the original C++ program.

Memory safety is guaranteed by the Rust compiler, given that the generated code never contains the `unsafe` keyword. Rust's type system rules out spatial errors (out-of-bounds accesses), temporal errors (use-after-free, double-free), and data races. The runtime library has a single use of `unsafe`, in `Ptr::delete()`, which recovers the strong reference leaked by `Ptr::alloc()` via `Rc::from_raw()` and drops it. This is sound because `Ptr::delete()` only accepts pointers created by `Ptr::alloc()`. Because the pointer contains a weak reference to the underlying `Rc`, the leaked reference can be recovered when the strong reference count is 1 (otherwise it is 0—a double-free—which triggers a panic).

Functional equivalence is established by a refinement argument over program traces: for any execution of the C++ source, the translated Rust program must exhibit a behavior that refines it, meaning it may reduce nondeterminism but may not introduce new observable behaviors. We distinguish three cases:

(1) Memory-safe, UB-free C++ traces are translated in a mostly structural, one-to-one manner. A key invariant is that destructors execute at exactly the same program points in Rust as in C++. This is guaranteed because C++ pointers become weak references (`Ptr<T>`), which do not contribute to the strong reference count and therefore do not extend allocation lifetimes. Deallocation, and hence destructor invocation, occurs at the same point it would in C++.
(2) Traces executing memory-unsafe operations, such as dereferencing a freed pointer or an out-of-bounds access, constitute undefined behavior (UB) in C++ and are thus unconstrained: any Rust behavior, including a panic from `Ptr::deref()` or a failed `Weak::upgrade()`, is a valid refinement.
(3) Traces involving other UB are handled by providing well-defined behavior where practical (e.g., signed integer overflow uses wrapping semantics, and memory is zero-initialized), with panics in remaining cases such as division by zero.

A critical corollary is that the translated code cannot panic on a well-defined C++ input. In particular, `RefCell` borrow panics are structurally excluded. The translation ensures that every dereference borrows a `RefCell` only long enough to copy needed data into local temporaries, releasing the borrow before any subsequent operation. No borrow is held across a function call or control-flow

boundary. When two borrows on the same cell might otherwise overlap, including in left- and right-hand sides of assignments and function call arguments, the translation introduces additional temporary variables, ensuring at most one borrow is active on any cell at any program point.

Functional equivalence was validated empirically via unit tests and differential testing on WOFF2 and Brunsli.

5 Optimizing Referenced-Counted Rust Code

Due to the conservative nature of our reference-counted translation model, where every variable is pessimistically wrapped inside a `Rc<RefCell<T>>` type, checks that would usually execute at compile-time are shifted to run time, degrading performance. We developed a series of optimizations whose goal is to remove these run-time checks whenever possible, improving both the performance and readability of Cpp2Rust's output.

Our algorithm operates through an iterative approach, employing three auxiliary optimizations. Each optimization first identifies the variables of a specific type that can be safely simplified and then applies the corresponding simplification. These optimizations are applied in a loop until a fixed point is reached. More concretely, we introduce three optimization passes, each aiming to eliminate a specific type: `Rc<T>` (§5.1), `RefCell<T>` (§5.2), and `Option<T>` (§5.3).

Running Example. To illustrate the inner workings of our optimization algorithm and associated analyses, we will use the following example. On the left is the original C++ code, and, on the right, the (non-optimized) code produced by Cpp2Rust.

```cpp
int f() {
  int a = 2;
  sum(a);
  a += 2;
  auto v = std::make_unique<int>(8);
  *v = 9;
  return a;
}
```

```rust
pub fn f() -> i32 {
    let a: Value<i32> = Rc::new(RefCell::new(2));
    sum(*a.borrow());
    *a.borrow_mut() += 2;
    let v: Value<Option<Value<i32>>> = Rc::new(
        RefCell::new(Some(Rc::new(RefCell::new(8)))));
    *v.borrow_mut().as_ref().unwrap().borrow_mut() = 9;
    return *a.borrow();
}
```

5.1 Rc Elimination

The reference counting smart pointer `Rc<T>` serves as a solution for scenarios where ownership needs to be shared. Unlike Rust's default single-owner model, `Rc<T>` implements shared ownership through a reference counting mechanism that tracks the number of references to a value, automatically deallocating memory when the last reference is dropped.

Given a Rust expression, our optimization first computes the set of `Rc` variables that do not have to be reference counted and then rewrites the expression to adjust their types. In a nutshell, a variable does not have to be of type `Rc<T>` if it holds the only reference to its corresponding value. Since precise static computation of variable aliasing is, in general, undecidable [22], we implement an over-approximating analysis that considers that a variable can be safely simplified if: (1) no method of the type `Rc` is called on the variable; (2) no new references are created to that variable; and (3) the variable is not passed as an argument to a function. Condition (3) is needed because the optimization is intra-procedural; thus, if a variable is passed as an argument to a function, we must pessimistically assume that it cannot be simplified.

In the running example, both variables `a` and `v` can be simplified since: no specific method of the `Rc<T>` type is applied on them, they are not cloned, and they are not used in function calls. Note that `borrow_mut` is a method of `RefCell`, not `Rc`, which means that condition (1) is satisfied.

$$\textsc{Non-RefCell Calls}\quad \frac{m \notin \{\texttt{RefCell::}\{\texttt{borrow, borrow\_mut}\}\}}{\vdash_{\texttt{RC}} \{\Omega\}\ x.m()\ \{\Omega \setminus \{x\}\}} \qquad \textsc{Borrow}\quad \vdash_{\texttt{RC}} \{\Omega\}\ \&x\ \{\Omega \setminus \{x\}\}$$

$$\textsc{Function Call}\quad \frac{\Omega_0 = \Omega \qquad \{\Omega_{i-1}\}\ E_i\ \{\Omega_i\}\ |_{i=1}^{n} \qquad \Omega' = \Omega_n}{\vdash_{\texttt{RC}} \{\Omega\}\ f(E_1, \ldots, E_n)\ \{\Omega'\}} \qquad \textsc{Assign Expr}\quad \frac{\vdash_{\texttt{RC}} \{\Omega\}\ E\ \{\Omega'\}}{\vdash_{\texttt{RC}} \{\Omega\}\ \texttt{let}\ x = E\ \{\Omega'\}}$$

Fig. 5. Rc-Elimination: $\vdash_{\texttt{RC}} \{\Omega\}\ E\ \{\Omega'\}$.

We formalize the analysis that identifies the variables to be simplified as a set of judgments of the form $\vdash_{\texttt{RC}} \{\Omega\}\ E\ \{\Omega'\}$, where Ω and Ω' denote the set of variables that can be *Rc-simplified* before and after analyzing expression E, respectively. The analysis processes one function at a time as follows: it places all function variables of `Rc` type in the initial set of variables to be simplified (Ω), and, as it analyzes the expressions in the function body, it removes the variables that do not meet conditions (1)-(3) from the current set, until only those that can be safely simplified are kept. Fig. 5 shows selected rules for Rc-Elimination. Condition (1) is enforced by the rule Non-RefCell Calls; condition (2) is enforced by the rule Borrow; and condition (3 is enforced by the rule Function Call. The rule Assign Expr illustrates a recursive application of the analysis.

Having computed the set of `Rc` variables that can be simplified, we remove the `Rc` wrapper type from those variables. Returning to the running example, as variables `a` and `v` can be simplified, their types are changed to `RefCell<i32>` and `RefCell<Option<Value<i32>>>`, respectively. We formalize this rewriting pass as a set of rules of the form $\Omega \vdash_{\texttt{RC}} E \rightsquigarrow E'$, meaning that the Rust expression E is rewritten to E' by simplifying the `Rc` variables in Ω.

$$\text{RC-Let}\quad \frac{x \in \Omega}{\Omega \vdash_{\texttt{RC}} \texttt{let}\ y = \texttt{Rc::new}(x) \rightsquigarrow \texttt{let}\ y = x}$$

5.2 `RefCell` Elimination

The `RefCell<T>` type enables interior mutability by shifting borrow checking to run time, allowing the program to have several syntactic references to a variable, as long as just one mutable reference is used at a time at run time. Like with the `Rc` type, using `RefCell` inccurs in run-time overhead.

In a nutshell, a variable does not have to be of type `RefCell<T>` if it is never subject to overlapping mutable borrows. As in the case of `Rc`, we implement an over-approximating analysis that considers that a variable can be safely optimized if: (1) no method is called on the variable except `borrow` and `borrow_mut`; and (2) the variable is not passed as argument in a function call. In general, we cannot simplify variables on which other `RefCell` methods are called since it is typically not possible to replicate their logic without `RefCell`'s internal state. In contrast, calls to `borrow` and `borrow_mut` can be removed because the code generated by Cpp2Rust never creates overlapping borrows.

As before, we formalize the identification of `RefCell` variables to be simplified as a set of judgments of the form $\vdash_{\texttt{RefCell}} \{\Theta\}\ E\ \{\Theta'\}$, where Θ and Θ' denote the set of variables that can be *RefCell-simplified* before and after analyzing expression E, respectively. The optimization processes one function at a time, setting the initial $\Theta = \{x^{immut} \mid x \in \text{vars}(fn)\}$ to be the set of all `RefCell` variables declared in the function annotated to be immutable, and progressively eliminating the variables from Θ that do not satisfy conditions (1) and (2). We show selected rules in Fig. 6. Variables in Θ are annotated with an (im)mutability modifier $\alpha ::= mut \mid immut$, which indicates whether or not its corresponding variable is immutable.

$$\textsc{Non-Borrow Calls}\quad \frac{m \notin \{\texttt{RefCell::}\{\texttt{borrow, borrow\_mut}\}\} \qquad x^{\alpha} \in \Theta}{\vdash_{\texttt{RefCell}} \{\Theta\}\ x.m()\ \{\Theta \setminus \{x^{\alpha}\}\}}$$

$$\textsc{Immut-Borrow Call}\quad \vdash_{\texttt{RefCell}} \{\Theta\}\ x.\texttt{borrow}()\ \{\Theta\}$$

$$\textsc{Mut-Borrow Call}\quad \vdash_{\texttt{RefCell}} \{\Theta\}\ x.\texttt{borrow\_mut}()\ \{\Theta \setminus \{x^{immut}\} \cup \{x^{mut}\}\}$$

Fig. 6. RefCell-Elimination: $\vdash_{\texttt{RefCell}} \{\Theta\}\ E\ \{\Theta'\}$.

Having computed the set of RefCell variables that can be simplified, we remove the RefCell wrapper type from those variables, along with the respective method calls. Returning to our example, as variables a and v can be simplified, their types are changed to i32 and Option<Value<i32>>, respectively. We formalize this rewriting pass as a set of rules of the form $\Theta \vdash_{\texttt{RefCell}} E \rightsquigarrow E'$, meaning that the Rust expression E is rewritten to E' by simplifying the RefCell variables in Θ. Below, we give an excerpt of our set of rewriting rules:

$$\textsc{Borrow}\quad \frac{m \in \{\texttt{borrow, borrow\_mut}\} \qquad x^{\alpha} \in \Theta}{\Theta \vdash_{\texttt{RefCell}} *x.m() \rightsquigarrow x}$$

$$\textsc{Refcell-Immut Call}\quad \frac{x^{immut} \in \Theta}{\Theta \vdash_{\texttt{RefCell}} \texttt{let}\ y = \texttt{Refcell::new}(x) \rightsquigarrow \texttt{let}\ y = x}$$

$$\textsc{Refcell-Init}\quad \frac{x^{mut} \in \Theta}{\Theta \vdash_{\texttt{RefCell}} \texttt{let}\ y = \texttt{RefCell::new}(x) \rightsquigarrow \texttt{let mut}\ y = x}$$

5.3 Option Elimination

The last optimization is used to remove instances of the Option type whenever a variable is guaranteed to be always different from None. As with the previous two cases, we model the variable identification process as a set of rules of the form $\{\Omega\}\ E\ \{\Omega'\}$, with Ω and Ω' containing the Option variables that can be simplified before and after the analysis of expression E, respectively. A variable can be simplified if: (1) it is never assigned to None, and (2) no Option method, except for unwrap, is called on it. We show below selected rules for Option elimination. Conditions (1) and (2) are enforced by the rules None-Assignment and Non-Unwrap Call, respectively.

$$\textsc{None-Assignment}\quad \frac{x \in \Omega}{\vdash_{\texttt{Option}} \{\Omega\}\ x = \texttt{None}\ \{\Omega \setminus \{x\}\}}$$

$$\textsc{Non-Unwrap Call}\quad \frac{x \in \Omega \qquad m \neq \texttt{Option::unwrap}}{\vdash_{\texttt{Option}} \{\Omega\}\ x.m()\ \{\Omega \setminus \{x\}\}}$$

After calculating the set of variables that can be safely simplified, we rewrite the body of each function to remove the Option type constructor where appropriate. In our example, the type of the variable v changes from Option<Value> to Value. We formalize this rewriting pass as a set of rules of the form $\Omega \vdash_{\texttt{Option}} E \rightsquigarrow E'$, meaning that the Rust expression E is rewritten to E' by simplifying the Option variables in Ω. Below, we give an excerpt of our set of rewriting rules:

$$\textsc{Option-Some}\quad \frac{x \in \Omega}{\Omega \vdash_{\texttt{Option}} \texttt{let}\ y = \texttt{Some}(x) \rightsquigarrow \texttt{let}\ y = x}$$

$$\textsc{Unwrap Call}\quad \frac{x \in \Omega}{\Omega \vdash_{\texttt{Option}} x.\texttt{unwrap}() \rightsquigarrow x}$$

6 Evaluation

We translated two C++ programs to evaluate Cpp2Rust with respect to run-time performance, memory usage, binary size, and generated code size (in lines of code). We also present a case study comparing Cpp2Rust with a manual translation of the WOFF2 font compression library.

Cpp2Rust is implemented as a clang-based tool, directly translating clang's AST into Rust code.[1] Cpp2Rust consists of 8.1k lines of C++, and a runtime library written in Rust (1.5k lines of code). The source-to-source optimizer is implemented in 3.6k lines of Rust, and uses the `syn` crate to parse Rust code into an AST, and the `prettyplease` crate to turn the optimized AST back to Rust code.

To estimate the performance upper bound of translated Rust code, Cpp2Rust includes a straightforward algorithm to emit *unsafe* Rust code (in addition to the safe translation algorithm described in Section 4). The generated code is similar to what C2Rust produces (i.e., uses raw pointers), but Cpp2Rust additionally supports C++ classes, virtual calls, and some C++ library classes.

The experiments were run on a server with an Intel Xeon E5-2630 v2 @ 2.60 GHz CPU with 64 GB of RAM, and Debian 12. We used `rustc` 1.89 to compile Rust files, and `clang` 20 to compile C++. Both compilers use the same backend (LLVM 20), which ensures they both have access to the same set of optimizations. All generated code was formatted using the `rustfmt` tool to ensure the number of lines of code reflects the usual Rust coding style. We used `brotli` 0.6.0 to run the C++ code, and the `brotli_sys` 0.3.2 crate for Rust code (which is a wrapper around the same `brotli` 0.6.0 library). This ensures fairness in the experiments.

To compute the running time of each benchmark, we run them 15 times. We then discarded the two extreme points, and computed the average.

6.1 Translating WOFF2 from C++ to Safe Rust

WOFF2 is a font compression library widely used to encode web fonts. Its decompression component is integrated into many browsers. We chose WOFF2 as the first case study for the following reasons:

- WOFF2 is security sensitive, since it processes arbitrary data from the internet, making it a good target for translation to a memory-safe language to mitigate security vulnerabilities.
- WOFF2 is small, but uses complex C++ features such as abstract classes, inheritance, and various containers of the C++ standard library.
- An engineer from Google had already translated WOFF2 to Rust by hand [16], giving us a baseline for comparison.

WOFF2 consists of a library that can be embedded in other projects (4.6k lines of C++), and three command-line programs: `woff2_info` (142 lines; reads a WOFF font and prints information about it), and `woff2_decompress` and `woff2_compress` (40 lines of code each; (de)compress fonts from/to the WOFF format).

Despite WOFF2 not being very large, it uses some challenging C++ features, which make it an interesting target for translation, namely:

- Some C++11 APIs: `std::vector`, `std::map`, `std::string`, `std::unique_ptr`, and iterators.
- Low-level features and optimizations, such as memcpy, and casts between pointer types to widen memory accesses.
- OOP: inheritance with virtual methods and uses non-default constructors.
- Reads and writes files.
- All memory allocations are done through `std::unique_ptr` (or implicitly through containers).
- Uses a 3rd-party library (Brotli, which we replaced with an existing crate). For compression, WOFF2 spends 81-98% of the time inside Brotli, and 3-6% when decompressing.

[1] Source code available at https://github.com/Cpp2Rust/cpp2rust.

Table 1. Summary of the implementations of WOFF2 in terms of number of lines of code (in thousands, kLoC), time in seconds to compress and decompress 128 TTF fonts, peak memory usage, and binary size. We show the results for the original C++ version, the safe and unsafe Rust code produced by Cpp2Rust, and the manual port to Rust. For the safe Rust code, we also show the results after applying our source-to-source optimizer and after using Clippy.

Version	kLoC	Compress (s)	Decompress (s)	Memory (MB)	Binary Size (MB)
C++	4.7	94.1	2.4	46.9	1.2
Cpp2Rust unsafe	5.4	88.4	0.6	46.9	1.2
Cpp2Rust safe	8.5	103.4	5.9	47.2	1.7
Cpp2Rust + Opts	8.0	98.5	2.9	46.0	1.5
Cpp2Rust + Opts + Clippy	7.7	95.8	3.2	46.9	1.5
Google's manual Rust port	6.2	~ C++	~ C++	n/a	n/a

We translated both the library and the three programs. Since Cpp2Rust translates whole programs into a single Rust file, we generate three Rust files, one for each program. The largest file (`woff2_info.rs`) has 8.5k lines of code (1.8× the size of the C++ code). After applying our source-to-source optimizer and Clippy, the number of lines reduces to 7.7k (1.6× the size of the C++ code). Table 1 summarizes the results.

Code Size. As expected, the unsafe version is the smallest since it more closely resembles the C++ code. In the safe version, our optimizer removes 642 uses of `Rc` and `RefCell` (71% of the total). The remaining uses cannot be removed because their address is taken, either explicitly in the original code or via references passed to other functions. Our optimizer is currently unable to remove uses of our `Ptr` type.

Clippy does several improvements to the generated code: removes duplicated or unneeded casts, removes unneeded `mut` keywords from variable declarations, removes unneeded parentheses, removes the `return` statement from the end of functions (in Rust, the last computed expression is returned if no `return` statement exists), removes unneeded calls to `into_iter()`, simplifies $0 + x$, simplifies an `else` branch containing an `if` statement into an `else if`, and replaces lambdas with a single function call into a reference to the function (e.g., replaces '`|| u8::default()`' into '`u8::default`'). These transformations shrink the programs by 270 lines (a 3.4% reduction).

The safe Rust code after all transformations still has 3k lines more than the original C++. There are several reasons for this:

- Rust's coding style occupies more space than Google's C++ style (used by WOFF2). For example, in Rust's coding style, function calls with many arguments are broken down to have a single argument per line, while Google's C++ style fills the whole line before overflowing to the next line.
- Function calls: because of the mutability and lifetime rules of Rust, we have to evaluate all arguments first and store the result in a `let` statement and only then do the call, resulting in one additional line per argument.
- We emit an implementation of the `Clone` trait for each class, even if the C++ implementation uses a default copy constructor. This is because we box fields, and thus the default `Clone` implementation increments the reference counter rather than doing a deep copy.

Performance. To compare the performance of the several Rust variants against C++, we measured the time to compress 128 common TTF fonts to the WOFF format, and then decompress them. We observe that the unsafe Rust variant is slightly faster than C++. The safe code produced by

Cpp2Rust is 10% slower at compression and 2.5× slower at decompression. These numbers improve to, respectively, 2% and 21% after using the source-to-source optimizer to remove most reference counters. We note that the optimizer provides a very significant speedup, over 2× for decompression. However, there is still room for further optimizations. For the manual port, Google reports only that their version achieves comparable performance to baseline C++.

Peak memory usage. There are no meaningful differences in terms of peak memory consumption. However, we do observe a slight reduction in memory consumption when using the source-to-source optimizer, which is expected since using `Rc` incurs in additional heap allocations.

Binary size. Once again, we observe that Rust binaries are larger than C++'s: even the unsafe version is 50% larger. Rust links libraries statically, which is good for performance, but increases the binary size. As expected, the safe Rust code results in a larger binary than the unsafe version (+42%) due to the additional safety checks and error handling. The source-to-source optimizer results in a smaller binary (-12%) since it removes a lot of the variable boxing.

Correctness. To guarantee that Cpp2Rust generated correct code, we compared the output of the C++ and Rust compression programs and ensured they matched byte-by-byte. We also checked that the decompressed WOFF file is equal to the original TTF font. This process actually uncovered a bug in Cpp2Rust's runtime library: our implementation of `memcpy` for types larger than 1 byte was copying the bytes with the wrong endianness.

Comparison with the manual port. The manual port required 12 days for a mechanical translation, and an extra 8 days to make the code more idiomatic. In contrast, Cpp2Rust automates the mechanical translation (takes only a few seconds), and the source-to-source optimizer does part of the rewrites to make the code more idiomatic.

The manual translation was done by first writing unsafe Rust code (using raw pointers) and then refactoring it to use (safe) slices. Cpp2Rust generates safe Rust code using its own `Ptr` type to manage pointers safely. We leave for future work improving the optimizer to replace uses of `Ptr` with more idiomatic constructs.

There were two significant refactorings done: using the `argh` crate to parse the command-line arguments (the C++ code parses them by hand), and using the `log` crate to translate calls to `fprintf(stderr, ...)`. Neither of these refactorings is done by Cpp2Rust.

The main advantage of Cpp2Rust is correctness: the code produced by Cpp2Rust is memory safe (no unsafe keyword emitted) and free of run-time errors for well defined C++ inputs. For example, Google's developers report that their manual port of WOFF2 to Rust introduced 20 bugs that were absent in the C++ version, including integer overflows, violation of mutability rules at run time, panics, and errors in the translation (mismatch in semantics). The violations of mutability rules at run time were considered the hardest to spot by looking at the source code. However, they can be found with a test suite with good coverage.

6.2 Translating Brunsli from C++ to Safe Rust

Brunsli is a lossless JPEG compression tool that uses several challenging C++ features beyond those used by WOFF2, namely:

- Address-taken global variables and static class variables, whose pointers are passed as function arguments.
- Lambdas and function pointers passed as arguments.
- Multi-dimensional C arrays whose addresses are taken.
- Extensive use of pointer arithmetic. Most functions receive pointers for input and output.

Table 2. Summary of the implementations of Brunsli in terms of number of lines of code (in thousands, kLoC), time in seconds to compress 100 images, peak memory usage, and binary size. We show the results for the original C++ version, the safe and unsafe Rust code produced by Cpp2Rust. For the safe Rust code, we also show the results after applying our source-to-source optimizer and after using Clippy.

Version	kLoC	Compress (s)	Memory (MB)	Binary Size (MB)
C++	8.3	5.3	18.4	1.0
Cpp2Rust unsafe	10.9	5.5	22.8	1.3
Cpp2Rust safe	15.2	272.4	360.9	2.8
Cpp2Rust + Opts	12.4	33.0	23.6	2.0
Cpp2Rust + Opts + Clippy	12.3	33.1	23.7	2.0

Table 2 summarizes the results. As with WOFF2, the performance of the unsafe Rust translation matches that of the original C++ code. However, the safe Rust translation is substantially slower. The slowdown stems primarily from Brunsli's extensive use of pointer arithmetic and references, which are translated to calls to `Ptr<T>::deref` in Rust. Even in the optimized version, 39% of the execution time is spent in `deref`.

The optimizer removes 87% of the `Value<T>` wrappers, failing to remove only 192 cases, which substantially reduces heap allocations. This results in a significant speedup of 8.2× compared to the unoptimized Rust code, which spends over a quarter of its time in heap allocation. We manually inspected the generated code and believe that a whole-program optimizer could eliminate most of the remaining uses of `Value<T>` and `Ptr<T>` by rewriting function arguments to use native Rust references.

6.3 Summary

These two case studies represent the largest applications automatically translated from C++ to safe Rust; no other tool has achieved full safety before. While our approach incurs a performance penalty, from as little as 2% to up to 6×, we believe this overhead is not fundamental and can be reduced through future work on two fronts. First, a global optimizer could transform `Ptr<T>` into native references. Second, improvements to the Rust compiler and LLVM backend could enable better optimization of `Rc` and `RefCell` containers, which are currently left unoptimized.

7 Related Work

Transpilers. C2Rust [40] is the most mature tool for translating C to Rust. It produces unsafe code using raw pointers. Corrode [73] aims to generate Rust code that is ABI compatible with the original, such that it can be used as a drop-in. Citrus [47] translates C syntax to Rust's without necessarily preserving the semantics and the generated code may not even compile. The goal is to produce similar code to the original one, which a developer can then fix by hand. CRAM [32] migrates C++ to idiomatic Rust semi-automatically via user-assisted refactoring. Scylla [28] compiles a restricted, alias-free subset of C to safe Rust with developer-guided restructuring for idiomatic output.

There are several tools that work on the output of C2Rust to remove unsafe annotations. CRustS [54] uses pattern matching to replace unsafe constructs with safe equivalents. Crown [95] and Emre et al. [22, 23] infer ownership and lifetimes to convert raw pointers into (safe) references. These efforts aim to improve the safety of translated unsafe Rust code, while Cpp2Rust takes the opposite approach, starting with unoptimized but safe Rust code, and iteratively optimizing it.

Another area of research aims at making the code more idiomatic, such as replacing tagged unions with Rust enums [37] and eliminating output parameters via algebraic data types [36]. Rustlancet [92] repairs ownership violations, and Concrat [35] translates C lock APIs for concurrency.

SBD [51] offers a hardening solution to isolate unsafe Rust code from the safe parts of a program.

Large Language Model (LLM)-based translation. VERT [91] combines LLMs with rule-based methods to generate idiomatic, formally verified Rust code via a repair-and-build loop. Hong et al. [38] use LLMs for migrating C types into Rust-native types. RustAssistant [18], originally designed for fixing compile errors, was extended by Eniser et al. [24] into a full C-to-Rust translator using iterative repair and validation. PR^2 [30] replaces raw pointers and calls to `malloc` with references and boxes, respectively. LAC2R [77] uses Monte Carlo tree search paired with an equivalence checker to find refinements. SafeTrans [26] uses an LLM to fix compilation errors.

LLMs have also been applied to translate C into safer dialects to improve memory safety: MSA [59] rewrites C to CheckedC [21]. C2SaferRust [62] uses an LLM to convert unsafe constructs into safe equivalents in code generated by C2Rust. Syzygy [74] integrates dynamic analysis with LLMs to ensure safe, equivalent Rust output. Shiraishi et al. [75] enhance translation via context-aware code segmentation which structures the input code more effectively. Subsequent work focused on improving accuracy and speed over traditional rule-based techniques [5, 10, 55, 66, 67, 86, 94].

Hardening of C/C++ code. Language dialects like CheckedC [21], Cyclone [42], and CCured [61], enforce partial memory safety at the language level. 3C [56] semi-automatically translates C to CheckedC using whole-program static analysis to infer bounds. Compiler-based instrumentation offers automated safety at the expense of run-time performance [12, 25, 31, 70, 96]. MetaSafe adds protection to Rust's smart pointers metadata to prevent tampering from unsafe code [45].

Unsafe-to-safe code translation. Mossienko et al. [60] propose a COBOL-to-Java translator that sacrifices functional equivalence for maintainability. Other approaches [29, 78] focus on full equivalence while adding an extra step to increase the readability of the generated code. Similar approaches exist for Fortran code to make it more maintainable and performant [9, 71] or to overcome its lack of type safety [83]. Automatic translation of enterprise applications written in C++ to Java has also been proposed [4, 6, 8, 57]. Translating programs into languages with run-time garbage collection is generally easier than into Rust, due to its more restrictive type system.

Emscripten [93] translates C/C++ applications into JavaScript. However, this translation is more akin to compilation, as the generated JavaScript code is not meant to be readable.

8 Conclusion

We presented Cpp2Rust, the first tool capable of automatically translating a subset of C++ into *safe* Rust. To further improve performance and make the generated code more idiomatic, we also introduced a source-to-source optimizer for safe Rust code.

We evaluated Cpp2Rust by translating two real-world C++ applications, totaling 13k lines of code. Our results show that, for some applications, the code produced and optimized by Cpp2Rust matches the performance of the original C++ code, while providing Rust's memory-safety guarantees.

Acknowledgments

This work was supported in part by national funds through Fundação para a Ciência e a Tecnologia (FCT), under projects UID/50021/2025 (DOI: https://doi.org/10.54499/UID/50021/2025) and UID/PRR/50021/2025 (DOI: https://doi.org/10.54499/UID/PRR/50021/2025), an unrestricted gift from Google, and a fellowship from "la Caixa" Foundation (ID 100010434, code DFI25-00467F).

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 4 (Nov. 2009). doi:10.1145/1609956.1609960

[2] Junho Ahn, Jaehyeon Lee, Kanghyuk Lee, Wooseok Gwak, Minseong Hwang, and Youngjin Kwon. 2024. BUDAlloc: defeating use-after-free bugs by decoupling virtual address management from kernel. In *USENIX Security*. https://www.usenix.org/system/files/usenixsecurity24-ahn.pdf

[3] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the Servo Web Browser Engine Using Rust. In *ICSE*. doi:10.1145/2889160.2889229

[4] Nadera S. Beevi, M. Reghu, D. Chitraprasad, and S. S. Vinodchandra. 2014. MetaJC++: A flexible and automatic program transformation technique using meta framework. *Central European Journal of Engineering* 4, 3 (Sept. 2014), 316–325. doi:10.2478/s13531-013-0161-2

[5] Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. 2024. Verified Code Transpilation with LLMs. In *NeurIPS*. https://proceedings.neurips.cc/paper_files/paper/2024/file/48bb60a0c0aebb4142bf314bd1a5c6a0-Paper-Conference.pdf

[6] Preeti Bhatt, Harmunish Taneja, and Kavita Taneja. 2020. SSCCJ: System for Source to Source Conversion from C++ to Java for Efficient Computing in IoT Era. In *ICIIL*. doi:10.1007/978-981-15-3020-3_35

[7] Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurélien Francillon, and Apostolis Zarras. 2018. Smashing the Stack Protector for Fun and Profit. In *IFIP SEC*. doi:10.1007/978-3-319-99828-2_21

[8] Frank Buddrus and Jörg Schödel. 1998. Cappuccino — A C++ to Java translator. In *SAC*. doi:10.1145/330560.331015

[9] Mateusz Bysiek, Aleksandr Drozd, and Satoshi Matsuoka. 2016. Migrating Legacy Fortran to Python While Retaining Fortran-Level Performance through Transpilation and Type Hints. In *PyHPC*. doi:10.1109/PyHPC.2016.006

[10] Xuemeng Cai, Jiakun Liu, Xiping Huang, Yijun Yu, Haitao Wu, Chunmiao Li, Bo Wang, Imam Nur Bani Yusuf, and Lingxiao Jiang. 2025. RustMap: Towards Project-Scale C-to-Rust Migration via Program Analysis and LLM. arXiv:2503.17741

[11] Shao-Fu Chen and Yu-Sung Wu. 2022. Linux Kernel Module Development with Rust. In *DSC*. doi:10.1109/DSC54232.2022.9888822

[12] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. 2022. ViK: practical mitigation of temporal memory safety violations through object ID inspection. In *ASPLOS*. doi:10.1145/3503222.3507780

[13] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2017. Secure Processors Part I: Background, Taxonomy for Secure Enclaves and Intel SGX Architecture. *Foundations and Trends in Electronic Design Automation* 11, 1–2 (2017). doi:10.1561/1000000051

[14] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. 2003. PointguardTM: protecting pointers from buffer overflow vulnerabilities. In *USENIX Security*. https://www.usenix.org/legacy/events/sec03/tech/full_papers/cowan/cowan.pdf

[15] Mohan Cui, Shuran Sun, Hui Xu, and Yangfan Zhou. 2024. Is unsafe an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming. In *ICSE*. doi:10.1145/3597503.3639136

[16] Felipe de Albuquerque Mello Pereira. 2022. Lessons learned from porting the WOFF2 library from C++ to Rust. https://www.youtube.com/watch?v=kcMAiTg5j1w

[17] Alessandro De Marco, Valentin Iancu, and Ira Asinofsky. 2018. COBOL to Java and Newspapers Still Get Delivered. In *ICSME*. doi:10.1109/ICSME.2018.00055

[18] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, Rishi Poddar, and Aseem Rastogi. 2025. RustAssistant: Using LLMs to Fix Compilation Errors in Rust Code. In *ICSE*. doi:10.1109/ICSE55347.2025.00022

[19] Dinakar Dhurjati and Vikram Adve. 2006. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE*. doi:10.1145/1134285.1134309

[20] Gregory J. Duck and Roland H. C. Yap. 2016. Heap bounds protection with low fat pointers. In *CC*. doi:10.1145/2892208.2892212

[21] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *SecDev*. doi:10.1109/SecDev.2018.00015

[22] Mehmet Emre, Peter Boyland, Aesha Parekh, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2023. Aliasing Limits on Translating C to Safe Rust. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 94 (April 2023). doi:10.1145/3586046

[23] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating C to safer Rust. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 121 (Oct. 2021). doi:10.1145/3485498

[24] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards Translating Real-World Code with LLMs: A Study of Translating to Rust. arXiv:2405.11514

[25] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. 2021. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *USENIX Security*. https://www.usenix.org/system/files/sec21-farkhani.pdf

[26] Muhammad Farrukh, Smeet Shah, Baris Coskun, and Michalis Polychronakis. 2025. SafeTrans: LLM-assisted Transpilation from C to Rust. arXiv:2505.10708

[27] Tommaso Fontana, Sebastiano Vigna, and Stefano Zacchiroli. 2024. WebGraph: The Next Generation (Is in Rust). In *WWW*. doi:10.1145/3589335.3651581

[28] Aymeric Fromherz and Jonathan Protzenko. 2026. Scylla: Translating an Applicative Subset of C to Safe Rust. *Proc. ACM Program. Lang.* OOPSLA (Oct. 2026).

[29] Shubham Gandhi, Manasi Patwardhan, Jyotsana Khatri, Lovekesh Vig, and Raveendra Kumar Medicherla. 2024. Translation of Low-Resource COBOL to Logically Correct and Readable Java leveraging High-Resource Java Refinement. In *LLM4Code*. doi:10.1145/3643795.3648388

[30] Yifei Gao, Chengpeng Wang, Pengxiang Huang, Xuwei Liu, Mingwei Zheng, and Xiangyu Zhang. 2025. PR^2: Peephole Raw Pointer Rewriting with LLMs for Translating C to Safer Rust. arXiv:2505.04852

[31] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2022. DangZero: Efficient Use-After-Free Detection via Direct Page Table Access. In *CCS*. doi:10.1145/3548606.3560625

[32] GrammaTech. 2023. CRAM: C++ to Rust Assisted Migration. https://www.grammatech.com/publication/cram-c-to-rust-assisted-migration/

[33] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. 2023. The Arm Morello Evaluation Platform—Validating CHERI-Based Security in a High-Performance System. *IEEE Micro* 43, 3 (2023), 50–57. doi:10.1109/MM.2023.3264676

[34] Sandra Höltervennhoff, Philip Klostermeyer, Noah Wöhler, Yasemin Acar, and Sascha Fahl. 2023. "I wouldn't want my unsafe code to run my pacemaker": An Interview Study on the Use, Comprehension, and Perceived Risks of Unsafe Rust. In *USENIX Security*. https://www.usenix.org/system/files/usenixsecurity23-holtervennhoff.pdf

[35] Jaemin Hong and Sukyoung Ryu. 2023. Concrat: An Automatic C-to-Rust Lock API Translator for Concurrent Programs. In *ICSE*. doi:10.1109/ICSE48619.2023.00069

[36] Jaemin Hong and Sukyoung Ryu. 2024. Don't Write, but Return: Replacing Output Parameters with Algebraic Data Types in C-to-Rust Translation. *Proc. ACM Program. Lang.* 8, PLDI, Article 176 (June 2024). doi:10.1145/3656406

[37] Jaemin Hong and Sukyoung Ryu. 2024. To Tag, or Not to Tag: Translating C's Unions to Rust's Tagged Unions. In *ASE*. doi:10.1145/3691620.3694985

[38] Jaemin Hong and Sukyoung Ryu. 2024. Type-migrating C-to-Rust translation using a large language model. *Empirical Software Engineering* 30, 3 (2024). doi:10.1007/s10664-024-10573-2

[39] Jaemin Hong, Sunghwan Shim, Sanguk Park, Tae Woo Kim, Jungwoo Kim, Junsoo Lee, Sukyoung Ryu, and Jeehoon Kang. 2024. Taming shared mutable states of operating systems in Rust. *Sci. Comput. Program.* (2024). doi:10.1016/j.scico.2024.103152

[40] Galois Inc and Immunant Inc. 2024. C2Rust: Migrate C code to Rust. https://github.com/immunant/c2rust

[41] Anna Irrera. 2017. Banks scramble to fix old systems as IT "cowboys" ride into sunset. https://www.reuters.com/article/us-usa-banks-cobol/banks-scramble-to-fix-old-systems-as-it-cowboys-ride-into-sunset-idUSKBN17C0D8/

[42] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX ATC*. https://www.usenix.org/legacy/publications/library/proceedings/usenix02/full_papers/jim/jim.pdf

[43] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (Dec. 2019). doi:10.1145/3371109

[44] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017). doi:10.1145/3158154

[45] Martin Kayondo, Inyoung Bang, Yeongjun Kwak, Hyungon Moon, and Yunheung Paek. 2024. METASAFE: compiling for protecting smart pointer metadata to ensure safe rust integrity. In *USENIX Security*. https://www.usenix.org/system/files/usenixsecurity24-kayondo.pdf

[46] Ravi Khadka, Prajan Shrestha, Bart Klein, Amir Saeidi, Jurriaan Hage, Slinger Jansen, Edwin van Dis, and Magiel Bruntink. 2015. Does Software Modernization Deliver What It Aimed for? A Post Modernization Analysis of Five Software Modernization Case Studies. In *ICSME*. doi:10.1109/ICSM.2015.7332499

[47] Kornel. 2017. Citrus: C to Rust syntax converter. https://lib.rs/crates/citrus

[48] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is theft: experiences building an embedded OS in rust. In *PLOS*. doi:10.1145/2818302.2818306

[49] Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. 2024. An empirical study of rust-for-Linux: the success, dissatisfaction, and compromise. In *USENIX ATC*. https://www.usenix.org/system/files/atc24-li-hongyu.pdf

[50] Ruishi Li, Bo Wang, Tianyu Li, Prateek Saxena, and Ashish Kundu. 2024. Translating C To Rust: Lessons from a User Study. arXiv:2411.14174

[51] Shaowen Li and Hiroyuiki Sato. 2025. SBD: Securing safe rust automatically from unsafe rust. *Sci. Comput. Program.* 243, C (April 2025). doi:10.1016/j.scico.2025.103281

[52] Zhaofeng Li, Vikram Narayanan, Xiangdong Chen, Jerry Zhang, and Anton Burtsev. 2024. Rust for Linux: Understanding the Security Impact of Rust in the Linux Kernel. In *ACSAC*. doi:10.1109/ACSAC63791.2024.00054

[53] Hans Liljestrand, Zaheer Gauhar, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. 2019. Protecting the stack with PACed canaries. In *SysTEX*. doi:10.1145/3342559.3365336

[54] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, and Ahmed E. Hassan. 2022. In Rust we trust: a transpiler from unsafe C to safer Rust. In *ICSE*. doi:10.1145/3510454.3528640

[55] Feng Luo, Kexing Ji, Cuiyun Gao, Shuzheng Gao, Jia Feng, Kui Liu, Xin Xia, and Michael R. Lyu. 2025. Integrating Rules and Semantics for LLM-Based C-to-Rust Translation. arXiv:2508.06926

[56] Aravind Machiry, John Kastner, Matt McCutchen, Aaron Eline, Kyle Headley, and Michael Hicks. 2022. C to checked C by 3c. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 78 (April 2022). doi:10.1145/3527322

[57] Scott Malabarba, Premkumar Devanbu, and Aaron Stearns. 1999. MoHCA-Java: a tool for C++ to Java conversion support. In *ICSE*. doi:10.1145/302405.302918

[58] Matt Miller. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. In *BlueHat*. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

[59] Nausheen Mohammed, Akash Lal, Aseem Rastogi, Rahul Sharma, and Subhajit Roy. 2025. LLM Assistance for Memory Safety. In *ICSE*. doi:10.1109/ICSE55347.2025.00023

[60] Maxim Mossienko. 2003. Automated COBOL to Java recycling. In *CSMR*. doi:10.1109/CSMR.2003.1192409

[61] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526. doi:10.1145/1065887.1065892

[62] Vikram Nitin, Rahul Krishna, Luiz Lemos do Valle, and Baishakhi Ray. 2025. C2SaferRust: Transforming C Projects into Safer Rust with NeuroSymbolic Techniques. arXiv:2501.14257

[63] Gene Novark and Emery D. Berger. 2010. DieHarder: securing the heap. In *CCS*. doi:10.1145/1866307.1866371

[64] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. In *SIGMETRICS*. doi:10.1145/3219617.3219662

[65] Shane Panter and Nasir Eisty. 2024. Rusty Linux: Advances in Rust for Linux Kernel Development. In *ESEM*. doi:10.1145/3674805.3690756

[66] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *NIPS*. https://proceedings.neurips.cc/paper/2020/file/ed23fbf18c2cd35f8c7f8de44f85c08d-Paper.pdf

[67] Baptiste Roziere, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging Automated Unit Tests for Unsupervised Code Translation. arXiv:2110.06773

[68] Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *NDSS*. https://www.ndss-symposium.org/wp-content/uploads/2017/09/A-Practical-Dynamic-Buffer-Overflow-Detector-Olatunji-Ruwase.pdf

[69] Leon Schuermann, Arun Thomas, and Amit Levy. 2023. Encapsulated Functions: Fortifying Rust's FFI in Embedded Systems. In *KISV*. doi:10.1145/3625275.3625397

[70] Kostya Serebryany, Chris Kennelly, Mitch Phillips, Matt Denton, Marco Elver, Alexander Potapenko, Matt Morehouse, Vlad Tsyrklevich, Christian Holler, Julian Lettner, David Kilzer, and Lander Brandt. 2024. GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production. In *ICSE-SEIP*. doi:10.1145/3639477.3640328

[71] Keith Seymour and Jack Dongarra. 2001. Automatic translation of Fortran to JVM bytecode. In *JGI*. doi:10.1145/376656.376833

[72] Ayushi Sharma, Shashank Sharma, Sai Ritvik Tanksalkar, Santiago Torres-Arias, and Aravind Machiry. 2024. Rust for Embedded Systems: Current State and Open Problems. In *CCS*. doi:10.1145/3658644.3690275

[73] Jamey Sharp. 2016. Corrode: A C to Rust translator. https://github.com/jameysharp/corrode

[74] Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A. Seshia, and Koushik Sen. 2024. Syzygy: Dual Code-Test C to (safe) Rust Translation using LLMs and Dynamic Analysis. arXiv:2412.14234

[75] Momoko Shiraishi and Takahiro Shinagawa. 2024. Context-aware Code Segmentation for C-to-Rust Translation using Large Language Models. arXiv:2409.10506

[76] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. FreeGuard: A Faster Secure Heap Allocator. In *CCS*. doi:10.1145/3133956.3133957

[77] HoHyun Sim, Hyeonjoong Cho, Yeonghyeon Go, Zhoulai Fu, Ali Shokri, and Binoy Ravindran. 2025. Large Language Model-Powered Agent for C to Rust Code Translation. arXiv:2505.15858

[78] Harry M. Sneed. 2010. Migrating from COBOL to Java. In *ICSM*. doi:10.1109/ICSM.2010.5609583

[79] Harry M. Sneed and Katalin Erdoes. 2013. Migrating AS400-COBOL to Java: A Report from the Field. In *CSMR*. doi:10.1109/CSMR.2013.32
[80] Toshio Suganuma, Toshiaki Yasue, Tamiya Onodera, and Toshio Nakatani. 2008. Performance pitfalls in large-scale java applications translated from COBOL. In *OOPSLA*. doi:10.1145/1449814.1449822
[81] Adrian Taylor, Andrew Whalley, Dana Jansens, and Nasko Oskov. 2021. An update on Memory Safety in Chrome. https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html
[82] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security*. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-tice.pdf
[83] Wim Vanderbauwhede. 2022. Making legacy Fortran code type safe through automated program transformation. *Journal of Supercomputing* 78 (Feb. 2022), 2988–3028. doi:10.1007/s11227-021-03839-9
[84] Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. 2020. RusTEE: Developing Memory-Safe ARM TrustZone Applications. In *ACSAC*. doi:10.1145/3427228.3427262
[85] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards Memory Safe Enclave Programming with Rust-SGX. In *CCS*. doi:10.1145/3319535.3354241
[86] Jun Wang, Chenghao Su, Yijie Ou, Yanhui Li, Jialiang Tan, Lin Chen, and Yuming Zhou. 2025. Translating to a Low-resource Language with Compiler Feedback: A Case Study on Cangjie. *IEEE Transactions on Software Engineering* (2025), 1–23. doi:10.1109/TSE.2025.3594908
[87] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *S&P*. doi:10.1109/SP.2015.9
[88] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: revisiting RISC in an age of risk. In *ISCA*. doi:10.1145/2678373.2665740
[89] Si Wu, Huyao Yang, and Baojian Hua. 2025. Security Risks of Transpiling C Programs to Rust. In *TrustCom*.
[90] Aiko Yamashita. 2017. Modernization from a Maintenance Process Perspective: Challenges and Lessons Learned. *J. Adv. Inf. Technol.* 8, 2 (May 2017). doi:10.12720/jait.8.2.107-113
[91] Aidan Z. H. Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. 2024. VERT: Verified Equivalent Rust Transpilation with Large Language Models as Few-Shot Learners. arXiv:2404.18852
[92] Wenzhang Yang, Linhai Song, and Yinxing Xue. 2024. Rust-lancet: Automated Ownership-Rule-Violation Fixing with Behavior Preservation. In *ICSE*. doi:10.1145/3597503.3639103
[93] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *OOPSLA*. doi:10.1145/2048147.2048224
[94] Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. 2025. Scalable, Validated Code Translation of Entire Projects using Large Language Models. *Proc. ACM Program. Lang.* 9, PLDI, Article 212 (June 2025). doi:10.1145/3729315
[95] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. 2023. Ownership Guided C to Rust Translation. In *CAV*. doi:10.1007/978-3-031-37709-9_22
[96] Jie Zhou, John Criswell, and Michael Hicks. 2023. Fat Pointers for Temporal Memory Safety of C. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 86 (April 2023). doi:10.1145/3586038