Hayroll: A Modular Wrapper for Translating C Macros and Conditional Compilation to Rust

HAORAN PENG, University of Washington, USA
BARIS KASIKCI, University of Washington, USA
GILBERT LOUIS BERNSTEIN, University of Washington, USA
MICHAEL D. ERNST, University of Washington, USA

Previous C-to-Rust translators run the C preprocessor `cpp` on their input before translation. This discards configurability, and it loses programmer-defined abstractions expressed as C macros.

We present Hayroll, a modular wrapper that makes C-to-Rust translation preprocessor-aware without modifying the underlying translator. Hayroll consists of two cooperating layers. The conditional compilation translation layer uses symbolic execution to derive activation conditions for each line of the source code. It splits the program into a set of configuration-specific translation tasks that together cover every line of code. These tasks are then passed independently to the macro translation layer. The macro translation layer classifies macros, annotates macro-expanded nodes with tags, and sends the code through the underlying translator. It then reconstructs C macros as Rust functions or macros by retrieving these tags from the translator's output. Because the layers communicate only through task partitioning and source-code annotations, the underlying translator remains a black box; no invasive changes are required. Our implementation uses C2Rust, but the design is translator-agnostic.

We evaluated Hayroll on CRUST-Bench, LibmCS, and zlib. Hayroll successfully reconstructs most syntactical macros and avoids configuration explosion through symbolic execution. Hayroll's output passes the same tests as the underlying C-to-Rust transpiler. These results show that decoupled preprocessor analysis can restore configurability and abstraction to C-to-Rust translation in a practical and modular way.

CCS Concepts: • **Software and its engineering → Source code generation**; **Preprocessors**; **Automated static analysis**.

Additional Key Words and Phrases: C-to-Rust translation, transpiler, conditional compilation, macros, preprocessors, symbolic execution

ACM Reference Format:
Haoran Peng, Baris Kasikci, Gilbert Louis Bernstein, and Michael D. Ernst. 2026. Hayroll: A Modular Wrapper for Translating C Macros and Conditional Compilation to Rust. *Proc. ACM Program. Lang.* 10, PLDI, Article 198 (June 2026), 24 pages. https://doi.org/10.1145/3808276

1 Introduction

The C programming language underlies much of the world's software systems. The C programming language also underlies most of the world's software vulnerabilities, which primarily result from C's lack of memory safety. About 70% of critical vulnerabilities in Android and Chrome are due to memory unsafety [37]. Translating C programs to a memory-safe language, such as Rust, would

Authors' Contact Information: Haoran Peng, University of Washington, Seattle, USA, hrpeng@cs.washington.edu; Baris Kasikci, University of Washington, Seattle, USA, baris@cs.washington.edu; Gilbert Louis Bernstein, University of Washington, Seattle, USA, gilbo@cs.washington.edu; Michael D. Ernst, University of Washington, Seattle, USA, mernst@cs.washington.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART198
https://doi.org/10.1145/3808276

Original C Code

```c
#ifdef LIBMCS_FPU_DAZ
    #define __LIBMCS_FPU_DAZ
    static volatile float __volatile_onef = 1.0f;
#endif
#define GET_FLOAT_WORD(i,d)                      \
    do { ieee_float_shape_type gf_u;             \
         gf_u.value = (d); (i) = gf_u.word; \
    } while (0 == 1)
#define FLT_UWORD_IS_FINITE(x) ((x)<0x7f800000L)
float sinhf(float x) {
    #ifdef __LIBMCS_FPU_DAZ
        x *= __volatile_onef;
    #endif
    int32_t ix, jx;
    GET_FLOAT_WORD(jx, x);
    ix = jx & 0x7fffffff;
    if (!FLT_UWORD_IS_FINITE(ix)) {
        return x + x;
    } ...
```

C2Rust Output

```rust
pub unsafe extern "C" fn sinhf(mut x: libc::c_float) ->
libc::c_float {
    let mut ix: int32_t = 0;
    let mut jx: int32_t = 0;
    loop {
        let mut gf_u = ieee_float_shape_type { value: 0. };
        gf_u.value = x; jx = gf_u.word as int32_t;
        if !(0 as libc::c_int == 1 as libc::c_int) {
            break;
        }
    }
    ix = jx & 0x7fffffff as libc::c_int;
    if !((ix as libc::c_long) < 0x7f800000 as libc::c_long) {
        return x + x;
    } ...
```

Hayroll Output

```rust
#[cfg(feature = "LIBMCS_FPU_DAZ")]
static mut __volatile_onef: libc::c_float = 1.0f32;

unsafe fn GET_FLOAT_WORD(i: *mut int32_t, d: *mut libc::c_float) {
    loop {
        let mut gf_u = ieee_float_shape_type { value: 0. };
        gf_u.value = *d; *i = gf_u.word as int32_t;
        if !(0 as libc::c_int == 1 as libc::c_int) {
            break;
        }
    }
}

unsafe fn FLT_UWORD_IS_FINITE(x: *mut int32_t) -> libc::c_int {
    ((*(x) as libc::c_long) < 0x7f800000 as libc::c_long) as
libc::c_int
}

pub unsafe extern "C" fn sinhf(mut x: libc::c_float) ->
libc::c_float {
    #[cfg(feature = "LIBMCS_FPU_DAZ")]
    (x *= __volatile_onef);
    let mut ix: int32_t = 0;
    let mut jx: int32_t = 0;
    GET_FLOAT_WORD(&mut jx, &mut x);
    ix = jx & 0x7fffffff as libc::c_int;
    if FLT_UWORD_IS_FINITE(&mut ix as *mut int32_t) == 0 {
        return x + x;
    } ...
```

Fig. 1. C2Rust and Hayroll outputs on code from LibmCS [13]. Highlighting shows source correspondence. C2Rust removes the inactive `#ifdef` region and expands all macros, thus losing configurability and abstraction. Hayroll translates the `#ifdef` region to a `#[cfg]`-guarded region and reconstructs C macros as Rust functions, preserving configurability and abstraction.

eliminate the largest cause of defects and vulnerabilities. Such translation is widely recommended, including by government authorities [33].

Previous C-to-Rust translators [7, 11, 20, 26, 51] share a common weakness: they make little attempt to translate uses of the C preprocessor, even though it is an integral part of the C language. Its conditional compilation manages feature selection and platform-dependent behavior. Its macros define constants, generic functions, and other behavior [9]. Instead, previous C-to-Rust translators can only work on a pure C file. Their first step is to run the preprocessor `cpp` once.

Ignoring the C preprocessor leads to a translation that is incomplete and lacks programmer-defined abstractions. It is *incomplete* because only one variant is chosen for each conditional. The resulting Rust program cannot be compiled to have different behavior in different situations, as the C program could. It *lacks abstractions* that the programmer expressed as macros. Each macro use is replaced by its definition, which may be large and hard to understand. In fig. 1, the C2Rust [20] translation loses both the `#ifdef`-controlled choice and the macro abstraction, whereas Hayroll restores both.

To address the important problem of incomplete translations that destroy abstractions, we have created a system, Hayroll, that preserves most conditional compilation and macro abstractions through a C-to-Rust pipeline. Hayroll converts C `#if` directives into Rust `#[cfg]` attributes, and it converts C macros into Rust functions or macros. Hayroll stands for *HARVEST Annotator for Yielding Regions of Lexical Logic.*

Hayroll uses symbolic execution to determine under what conditions each macro definition may flow to a given use. It also determines the syntactic form of each macro definition: expression,

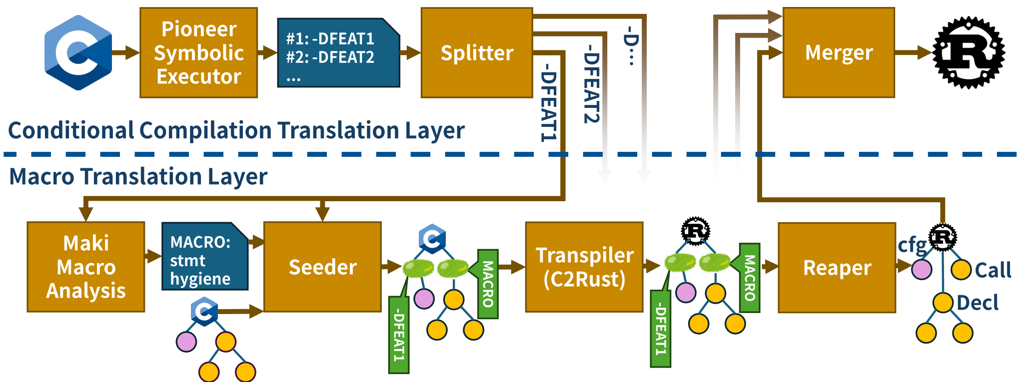

Fig. 2. Hayroll consists of two layers. The conditional compilation translation layer's Pioneer and Splitter symbolically interpret preprocessor directives and partition the program into configuration-specific translation tasks. The macro translation layer (Maki [36], Seeder, C2Rust [20], and Reaper) processes each task independently to translate macros under a fixed configuration. Finally, Merger combines all translated variants into a unified, configurable Rust program that reconstructs some C macros as Rust function or macros.

statement, declaration, etc. It cooperates with any macro-unaware C-to-Rust translator — with no need to modify the underlying translator — by tagging macro-relevant parts of the code, running the translator multiple times, and reconstructing abstractions in a single Rust codebase.

We evaluated Hayroll on 62,821 LoC. With respect to *correctness*, Hayroll's translation passes all the same tests as C2Rust, a state-of-the-art translator. With respect to *completeness*, Hayroll preserves 74% of syntactical macro invocations (53% of all macro invocations) into Rust code. With respect to *performance*, Hayroll takes roughly 0.7–2.9 seconds per kLoC, depending on the complexity of the codebase.

Our contributions are:

- **Conditional compilation resolution (Pioneer).** We formalized and built a symbolic execution engine that computes, for each source line, a predicate over preprocessor definitions describing when the line is active. This enables translation from C conditional compilation to Rust `#[cfg]`.
- **Source-code tagging for source correspondence (Seeder/Reaper/Merger).** We developed a source-code tagging mechanism that identifies AST nodes generated from macro expansions or guarded by conditional compilation. It embeds metadata inside the C source without auxiliary sidecar files. These tags preserve semantics and survive translation, allowing the Rust output to reconstruct macro abstractions and conditional compilation while remaining decoupled from the underlying translator.
- **Evaluation.** We implemented the full system, Hayroll, as a wrapper around C2Rust and evaluated it on real-world codebases. The results show how symbolic execution and tag-based reconstruction work together to preserve configuration and macro structure.

2 Design

A C program is written in two different languages interleaved with each another. The C preprocessor `cpp` treats a file as a stream of tokens. The C programming language proper parses a pure (already preprocessed) C file into an AST. It is challenging to treat a C program holistically; most analyses ignore one of the two languages.

2.1 Overview

Design challenges. The C preprocessor introduces three fundamental challenges for a preprocessor-aware C-to-Rust translator. First, **conditional compilation** `#if`/`#ifdef` expressions may involve nested Boolean and arithmetic conditions, redefinitions, and dependent flags, so identifying the activation condition of each line requires symbolic analysis. Second, **C macro semantics** operate at the string and character level rather than at the AST level where existing AST-based translators work. A separate analysis must determine whether a macro is syntactical (it expands to a single AST node), what syntactic kind (if any) a macro expands to, and whether it uses features such as token pasting or stringification. Third, reconstructing macro structure in Rust requires knowing which output nodes correspond to which input expansions, yet current translators expose no **source correspondence** interface.

Design goals. Hayroll's goal is twofold. First, it *translates* a subset of macro expansions into Rust functions and macros and translates conditional compilation into Rust configuration (`#[cfg]`). The translatable subset depends on what C macro features are used (section 2.3.1). Second, it is *minimally invasive* to the macro-agnostic C-to-Rust transpiler: Hayroll requires no stable API or fixed implementation from the underlying tool.

The above challenges and goals motivate our modular wrapper design: instead of modifying the underlying translator, Hayroll resolves conditional compilation, analyzes macro semantics, tags the C program, and later recovers these tags from the Rust program, enabling macro-aware translation while treating the translator itself as a black box. Many C-to-Rust translators also operate as wrappers or post-processors of existing tools such as C2Rust to improve specific aspects of the Rust output [7, 26, 32, 51]. By adopting the same non-intrusive wrapping principle, our architecture naturally fits into this ecosystem. We recognize that other designs are possible. For example, our techniques could be implemented within a C-to-Rust translator. That would enable better source correspondence and could permit statement reordering, at the cost of per-translator implementation effort.

Assumptions about the underlying transpiler. Hayroll makes no assumptions about how the underlying C-to-Rust transpiler works. Its only assumption is about *seeded* C code (no-op tags that Hayroll inserts to mark code originating from conditional compilation ranges or from macro expansions). Each seeded subtree in the C AST must remain a subtree after translation. The underlying translator must preserve the node's subtree boundary: it does not arbitrarily reorder code across that boundary (e.g., hoisting code out of the subtree or sinking unrelated code into it), nor optimize away the seeds. This assumption is sufficient for Hayroll to derive source correspondence. It is always satisfied by C2Rust, which does line-by-line translation.

Design layers. Hayroll consists of two layers: (1) an outer layer for **conditional compilation translation**, and (2) an inner layer for **macro translation**. Wrapped inside these two layers is a preprocessor-agnostic C-to-Rust translator, such as C2Rust. Figure 2 shows the full pipeline.

2.2 Conditional Compilation Translation Layer (Pioneer, Splitter, Merger)

The conditional compilation layer consists of three components.

Pioneer performs symbolic execution over preprocessor conditionals (`#if`, `#ifdef`, etc.). It computes the enabling condition of each source line in terms of combinations of user-defined preprocessor flags. This analysis captures when each line of code is active, while avoiding the combinatorial explosion of brute-force exploration.

Splitter schedules multiple translation runs of the macro translation layer. Each run uses a configuration that is sufficient to cover a subset of lines. Together, these runs exercise every

Table 1. Macro attributes affecting translation to Rust. This is a subset of those defined in [36].

Attribute	Description
`syntactical`	Whether the expansion exactly aligns with a single AST node. Non-syntactical expansions cross AST boundaries or form AST fragments.
`astKind`	AST category of the expansion when syntactical.
`anyArgument-NotExprOrStmt`	Some argument is not an expression or a statement (e.g., a new identifier).
`hygienic`	The macro does not capture or reference local variables outside its lexical scope.
`isICE`	Indicates whether the expansion occurs in an integer-constant context (e.g., array size or enum value).
`usesPasting`	Uses token concatenation (`##`, e.g., `token##suffix`).
`usesStringification`	Uses stringification (`#x`).
`hasFunctionPointer`	The macro expands to a function name or a function pointer.
`argumentHas-SideEffect`	Some argument contains side effects, e.g., `i++`.
`hasControlFlowJumps`	Expansion contains control-flow jump statements: `return`, `break`, `continue`, or `goto`.
`anyArgumentNotExpr`	Some argument is not an expression (e.g., a statement block).

conditional region at least once. Each run outputs a Rust file that expresses part of the C program's full functionality.

Merger takes as input multiple Rust files, each of which is hard-coded to one configuration, that is, one set of user-supplied preprocessor flags. It combines them into a configurable Rust output.

2.3 Macro Translation Layer (Maki, Seeder, Reaper)

This inner layer of the wrapper performs macro-related analysis, instrumentation, and reconstruction. It consists of three components that operate in sequence: **Maki**, **Seeder**, and **Reaper**.

Maki [36] is a tool that analyzes macros at the string level. Maki classifies each macro *expansion* regarding its translatability: whether the macro is syntactical (expands into an AST subtree), its AST kind (expression, statement, declaration, or type), and whether it involves token pasting, stringification, or hygiene violations.

Seeder instruments the C source with lightweight seeds that record their macro or conditional compilation origins. Each seed is legal C source code whose run-time behavior is a no-op. This instrumented C code is passed to the underlying C-to-Rust translator.

After translation, **Reaper** reads seeds, which are now expressed in Rust syntax, from the Rust output and reconstructs macro definitions or conditional guards as Rust functions, macros, or `#[cfg]` attributes.

The seed mechanism provides a practical form of source correspondence without requiring changes to the translator, such as a new API that gives the source correspondence.

2.3.1 Translation into Rust Functions and Macros. Hayroll translates some C macros into Rust functions and some into Rust macros (`macro_rules!`). Tables 1 and 2 indicate the conditions for each. Other macros are unaffected by Hayroll: their definitions and uses appear unchanged in the program that Hayroll provides to the C-to-Rust translator.

Table 2. Which macro uses are translated by Hayroll.

Can translate to macro: `syntactical` ∧ `hygienic` ∧ (`astKind` ∈ {Expr, Stmt, Decl}) ∧ ¬`anyArgumentNotExprOrStmt` ∧ ¬`isICE` ∧ ¬`usesPasting` ∧ ¬`usesStringification` ∧ ¬`hasFunctionPointer`.
Can translate to function: Conditions for macros, plus: (`astKind` ∈ {Expr, Stmt}) ∧ ¬`argumentHasSideEffect` ∧ ¬`hasControlFlowJumps` ∧ ¬`anyArgumentNotExpr`.

Fig. 3. Definition chain example in LibmCS (`internal_config.h`).

```
#ifdef LIBMCS_FPU_DAZ
   #define __LIBMCS_FPU_DAZ
   static volatile double __volatile_one = 1.0;
   static volatile float __volatile_onef = 1.0f;
#endif
```

Fig. 4. Mutual exclusion in nested conditional directives in LibmCS (`internal_config.h`).

```
#ifdef LIBMCS_LONG_DOUBLE_IS_64BITS
   #define __LIBMCS_LONG_DOUBLE_IS_64BITS
   #ifdef LIBMCS_DOUBLE_IS_32BITS
      #error Cannot define both ...
   #endif
#endif
```

The requirement of being `syntactical` is inherent: when a macro expansion does not align with any C AST node, there is no Rust construct to which it can correspond.

By contrast, the lack of `Type`-kind support, integer-constant contexts, and constructs involving pasting, stringification, or function pointers stems from current limitations of our Seeder implementation, which cannot yet attach metadata to such AST nodes. These restrictions are therefore practical rather than fundamental, and could be addressed if future translation backends exposed finer-grained source-AST correspondences.

Similarly, handling unhygienic macros would require an explicit refactoring phase that reconstructs them into hygienic equivalents before translation.

3 System

3.1 Pioneer Symbolic Executor

3.1.1 Motivation. Conditional compilation in C introduces complicated logical structures. The motivation for building a symbolic executor to reason about it is threefold:

(1) Chained macro definitions and derived flags. Macros may define secondary symbols that the user should not configure directly. For example, in fig. 3, only the outer flag `LIBMCS_FPU_DAZ` should be touched by the user; defining the internal `__LIBMCS_FPU_DAZ` skips initialization code and yields inconsistent state. Such derived definitions cannot be captured through naive enumeration.

(2) Nested conditions and mutual exclusion. Nested directives encode logical constraints among flags. Figure 4 shows an exclusion relation between two configuration flags. The nested structure expresses the constraint that `LIBMCS_LONG_DOUBLE_IS_64BITS` and `LIBMCS_DOUBLE_IS_32BITS` shall never be defined at the same time. Naive enumeration would produce many configuration combinations that include these two flags, which will cause many unnecessary erroneous runs of the macro translation layer.

(3) Efficiency and scalability. Symbolically capturing preprocessor control flow also addresses a severe efficiency problem. Naive enumeration of all possible macro combinations grows exponentially with the number of flags: for n Boolean flags, 2^n configurations exist, many of which are redundant or illegal. Pioneer avoids this explosion by computing logical premises once per code region instead of reprocessing every configuration.

Fig. 5. An example input for Pioneer.

```
#ifndef HEADER_GUARD
#define HEADER_GUARD
int f();
#endif
```

Fig. 6. The preprocessor-level AST created by Pioneer to represent the program of fig. 5.

```
preproc_ifndef "#ifndef HEADER_GUARD [then] #endif"
  then:
    preproc_define "#define HEADER_GUARD"
    c_tokens "int f();"
```

3.1.2 Input and Output. Pioneer's input is an unpreprocessed C compilation unit (file) that retains all preprocessor directives. It parses the source file into a *preprocessor-level* AST, rather than a C AST. Each directive such as `#if`, `#define`, or `#error` becomes a distinct AST node. All non-preprocessor tokens are represented as `c_tokens` nodes. As a result, Pioneer does not require the file to be syntactically valid as a full C AST before preprocessing; non-syntactical macro expansions remain opaque `c_tokens` rather than causing a parse failure.

For example, the program in fig. 5 is parsed into the preprocessor-level AST shown in fig. 6, where preprocessor directives and normal code tokens are explicitly separated.

Pioneer outputs a line-to-premise mapping `ProgramLine` ↦ `Premise` where each `Premise` is a Boolean-and-arithmetic expression over the definedness and value of preprocessor symbols. It represents the combination of preprocessor defines under which the corresponding source line is active, that is, when it will be preserved after preprocessing and seen by the downstream translator. It also outputs `forbidden` which is a formula of preprocessor defines that are explicitly disallowed by preprocessor error directives.

3.1.3 Representation of Execution States. Pioneer performs symbolic execution over the preprocessor AST. At the beginning of execution, each preprocessor define `-DNAME[=VAL]` is represented by two symbolic variables: one for its *definedness* and one for its *value*. The definedness variable is `true` if the macro is defined by the user on a given run and `false` otherwise; the value variable holds a symbolic integer value if the macro participates in numeric expressions. Consequently, a *symbolic expression* may combine both logical and arithmetic components. For example, conditions such as `#if defined(X) && (Y + 1 < 4)` are represented symbolically as formulas over these variables. Although later translation stages in Hayroll only accept Boolean expressions (because Rust's conditional compilation does not support arithmetic predicates), Pioneer's symbolic executor maintains full expressive power to reason about both definedness and macro values during analysis. In cases where a non-Boolean condition is encountered, later stages will emit a warning and concretize it, that is, translate a non-configurable version of the conditional by evaluating the expression with default values (undefined macros are treated as zero).

Pioneer's symbolic algorithm explores all execution paths of the preprocessor program under these symbolic assignments. Each node in this exploration corresponds to one possible logical state of the preprocessing process. Pioneer models every execution state as a triple of:

- `ProgramPoint`: the current position in the preprocessor AST.
- `SymbolTable`: the current macro definition environment, represented as a mapping from macro names to their definedness and value. Looking up a macro from a `SymbolTable` yields either a concrete value if it is defined, or a symbolic value denoted by the macro's name if not defined.
- `Premise`: the accumulated Boolean condition along the traversal path.

3.1.4 Algorithm. The symbolic executor performs a worklist search over all possible program states (fig. 7). Each state $S = (n, s, p)$. For each program point n, the transition function $F(n)$ generates successor states, see table 3.

Fig. 7. Pioneer symbolic executor algorithm. The transition function $F(n)$ is defined in table 3.

```
Input:
   astRoot : ProgramPoint

Algorithm:
   worklist : List<(ProgramPoint, SymbolTable, Premise)> := { (astRoot, emptySymbolTable, true) }
   lineToPremise : Map<ProgramPoint, Premise> := []
   forbidden : Premise := false
   while worklist is not empty do
      (n, s, p) := worklist.pop()
      if n is an "#error" directive then
         forbidden := forbidden || p
         continue
      if n is a c_tokens node then
         lineToPremise[n] := lineToPremise[n] || p
      newStates := F(n)(s, p)
      worklist = worklist + newStates

Output:
   (lineToPremise, forbidden)
```

Table 3. Transition rules $F(n)$ for preprocessor directives. Given cpp AST node n, $F(n)$ takes as input a symbol table s and a premise p. The output of $F(n)$ is a set of 0, 1, or 2 successor states. A state is a triple (`ProgramPoint`, `SymbolTable`, `Premise`).

cpp AST node	Transition rule
`#define m v`	$s' = s \cup \{m \mapsto v\}$; successor $(\text{next}(n), s', p)$.
`#undef m`	$s' = s \setminus \{m\}$; successor $(\text{next}(n), s', p)$.
`#if cond`	Parse *cond* under s into a symbolic expression. Successors: $(\text{then}(n), s, p \wedge cond)$, and $(\text{else}(n), s, p \wedge \neg cond)$ when an else branch exists or $(\text{next}(n), s, p \wedge \neg cond)$ when no else branch exists.
`#ifdef m`	Successors: $(\text{then}(n), s, p \wedge \text{def}(m))$, and $(\text{else}(n), s, p \wedge \neg\text{def}(m))$ when an else branch exists or $(\text{next}(n), s, p \wedge \neg\text{def}(m))$ when no else branch exists.
`#ifndef m`	Successors: $(\text{then}(n), s, p \wedge \neg\text{def}(m))$, and $(\text{else}(n), s, p \wedge \text{def}(m))$ when an else branch exists or $(\text{next}(n), s, p \wedge \text{def}(m))$ when no else branch exists.
`#elif cond`	Equivalent to a subsequent `#if cond` branch following the preceding `#if`.
`c_tokens`	Successor: $(\text{next}(n), s, p)$.
`#error`	No successor.
`EOF`	No successor.

3.1.5 Optimizations. Although symbolic execution of conditional directives could in principle lead to exponential branching, Pioneer incorporates several optimizations that make it practical for real-world C code.

Hybrid concrete execution and whitelist mode. Pioneer supports a hybrid execution model that combines symbolic and concrete evaluation of macros. In practice, a single compilation unit may include many header files from system libraries whose macros are unrelated to project-level configurability yet introduce large numbers of symbols and conditional branches. Naively symbolizing all of them would lead to severe state explosion and make execution infeasible. To address this, Pioneer allows users to specify a working directory. For source files outside that directory, their macros are treated as concrete values determined by the current build environment. This bounds the symbolic search space to project-relevant configurations. In addition, a *whitelist mode* allows the user to specify the exact set of macro names to be treated symbolically. All other macros remain concrete, enabling targeted exploration in large projects where full symbolic

Fig. 8. Splitter algorithm for generating configuration sets.

```
Input:
   lineToPremise : Map<ProgramPoint, Premise>
   forbidden: Premise
   nestingDepth : Map<ProgramPoint, Int>

Algorithm:
   tasklist : List<(ProgramLine, Premise, Int)> := sortDescendingBy(nestingDepth, lineToPremise)
   splits : List<Configuration> := {}
   for (l, p, d) in tasklist do # deepest line first
      config := solve(p && !forbidden) # the solver returns a satisfying model as configuration
      if config is UNSAT then
         continue # line l is unreachable
      splits.append(config)
      # remove lines already covered by this configuration
      for each (l2, p2, d2) in tasklist do
         if isTautology(evalWithModel(config, p2)) then
            tasklist.remove((l2, p2, d2))

Output:
   splits
```

analysis would be prohibitive. This is also helpful when the user only needs to preserve partial configurability.

State merging. When exploring a chain of `#if`/`#elif`/`#else` blocks, different branches sometimes leave the macro definition environment unchanged. Pioneer detects such cases and merges states that share the same symbol table by combining their premises with logical disjunction. This prevents the number of states from growing exponentially in common cases where conditions only test flags without setting any flags.

Symbol table reuse. Conceptually, each branching point could produce a full copy of the current symbol table, but in practice Pioneer avoids this. The symbol table is stored in a chained hash table that allows newly created states to share the unchanged portion of their parent environment. Only the modified part of the table is newly allocated, keeping both memory and time overhead sublinear in the number of branches.

3.2 Splitter

Purpose. Splitter partitions the configurable program translation into several configuration-specific macro translation tasks. Its goal is to cover all conditional regions in the source program at least once, so that each resulting configuration can be translated independently by downstream tools. It does not aim to produce as few splits as possible for a reason we will discuss later, but it still reduces the worst-case number of splits from exponential (by naive enumeration) to linear in the number of `#if` directives in the source code.

Input and output. Splitter takes as input the line-level premises and the set of globally forbidden flag combinations produced by Pioneer. It outputs several configurations, each representing a specific combination of `-D` flags.

Algorithm. As shown in fig. 8, the algorithm processes each line in order of decreasing nesting depth. It obtains a satisfying model for the line's premise using an external SAT/SMT solver and interprets that model as a concrete configuration. It then removes all other lines already satisfied by this configuration.

Discussion. A natural question is why Splitter does not attempt to cover as many lines as possible in a single configuration and produce as few splits as possible. In theory, one could greedily combine multiple premises that are jointly satisfiable, reducing the number of macro translation layer runs. In practice, however, such aggressive merging risks generating invalid configurations,

because mutually exclusive flags are common in C codebases. Some of these exclusions are explicitly guarded by `#error` directives as in the case of fig. 4, but many others are unguarded or merely documented in natural language. Those would not trigger preprocessor errors but would cause later translation errors. An example is two alternative function implementations with the same signature that appear in different `#if` blocks. By exploring one constrained premise at a time, Splitter reduces the possibility of encountering these hidden conflicts.

3.3 Maki Macro Semantics Analyzer

Maki [36] is a static analysis tool for C macros. Hayroll extends and integrates it.

Purpose. Maki analyzes each individual macro invocation and reports a set of its properties. Hayroll keeps the subset that affects its translatability into Rust (table 1). In addition, our extended version of Maki also analyzes each conditional compilation region in the source code (that is, each range from `#if*` to the matching `#else` or `#endif`). This includes non-syntactical macro expansions: they are still detected and classified, but later left expanded rather than reconstructed.

Input and output. The input to Maki is an unpreprocessed C compilation unit and a specific configuration. For every macro expansion, Maki produces the attributes defined in section 2.3.1. For each conditional compilation region, only a simplified subset of attributes is recorded: whether the block is `syntactical` and, if so, what its `ASTKind` is. These attributes are later consumed by downstream components such as Seeder and Reaper to guide code region seeding and Rust function/macro reconstruction.

3.4 Seeder

3.4.1 Overview. Seeder runs after macro and conditional compilation analyses and immediately before the C-to-Rust translation. It instruments the original C source with tags, called *seeds*. The C-to-Rust translator translates the seeded C code into Rust, then Reaper and Merger read them to recover source correspondence without a sidecar metadata file. The inputs are one unpreprocessed C compilation unit, Maki's attributes for macro invocations and conditional compilation regions, and Pioneer's line-to-premise mapping. The output is an instrumented C source. The instrumentation does not add or remove cpp syntax (macro uses and # directives including macro definitions), but it does change the C source code.

Seeder inserts seeds for three kinds of entities: macro expansion sites, arguments of function-like macros, and conditional compilation ranges. For macros, each seed records the translation-relevant properties from section 2.3.1 together with source locations; for conditional ranges, it records the AST kind, guarded source range, and symbolic premise from Pioneer. All the seeding strategies are designed to preserve observable behavior, carry serialized metadata, and survive translation in recognizable shapes. Figure 9 shows a running example that contains a declaration macro, a statement macro, an expression macro, and a conditional region.

3.4.2 Seeding Strategies. Different categories of AST nodes require different seeding strategies. Seeder handles three classes of seeds: statement, expression, and declaration.

Statement seeding. For statements, Seeder uses a *sandwich* pattern: two no-op strings are inserted immediately before and after the original statement (fig. 10). The strings contain the seed metadata. In the running example, this strategy is used both for the conditional `#ifdef FEAT1` region and for the statement macro invocation `STMT_MACRO_INCR(a)` (fig. 9, line 11); the conditional region appears on lines 8–10. It does not alter control flow and produces a clear boundary that persists after translation. If the statement immediately follows an `if` or `else` (without braces), Seeder wraps the statement in braces.

Fig. 9. Input C program with macro decl/stmt/expr and a conditional region.

```
1  #define EXPR_MACRO_ADD(x, y) ((x) + (y))
2  #define STMT_MACRO_INCR(x) do { (x)++; }
       while (0)
3  #define DECL_MACRO_INT int a;
4
5  DECL_MACRO_INT
6
7  int main() {
8     #ifdef FEAT1
9     ++a;
10    #endif
11    STMT_MACRO_INCR(a);
12    return EXPR_MACRO_ADD(a, 5);
13 }
```

Fig. 10. Statement seeding.

```
1  *SEED_BEGIN;
2  ORIGINAL_STATEMENTS;
3  *SEED_END;
```

Fig. 11. Expression seeding for rvalues.

```
1  (
2     (*SEED) ?
3     (EXPR) :
4     (*(__typeof__(EXPR)*)(0))
5  )
```

Fig. 12. Expression seeding for lvalues.

```
1  (
2     *(
3        (*SEED) ?
4        (&(EXPR)) :
5        ((__typeof__(EXPR)*)(0))
6     )
7  )
```

Fig. 13. Declaration seeding.

```
1  const char * SEED_FOR_<MACRO_NAME> = SEED;
```

Expression seeding (rvalue). In C, every expression is either an lvalue or an rvalue. An lvalue denotes a storage location and can appear on the left-hand side of an assignment; an rvalue denotes a computed value and cannot be assigned to. This distinction is important for Seeder as the seeding transformation must preserve both value and assignability. Seeder therefore employs separate strategies for rvalues and lvalues.

Rvalue expressions are wrapped in a conditional expression whose guard is the seed string (fig. 11). On line 12 of fig. 9, this applies to the second argument `5` of `EXPR_MACRO_ADD(a, 5)`. Because the seed always evaluates to a nonzero value, the original expression is evaluated and returned unchanged. The alternate branch introduces a dummy expression of the same type via `__typeof__`, preserving type correctness. The generated Rust code exhibits a characteristic if-expression shape that Reaper can recognize.

An alternative seeding approach is to use C's comma operator. We could not use this because C2Rust splits comma-separated expressions into multiple statements, which violates Reaper's assumption that subtree shapes of seed nodes are preserved after translation.

Expression seeding (lvalue). Lvalues must remain assignable after seeding. Our approach leverages a simple lemma: every lvalue can be addressed, after which dereferencing the address yields an assignable object again. Seeder therefore reuses the rvalue seeding pattern, but wraps the original expression inside an address–dereference pair (fig. 12). In the running example, this applies to the argument `a` passed both to `STMT_MACRO_INCR` (fig. 9, line 11; fig. 14, line 16) and to `EXPR_MACRO_ADD` (fig. 9, line 12; fig. 14, line 27). The *then* branch takes the address of the original expression, and the *else* branch uses a null-typed pointer of the same type. Dereferencing the entire conditional restores the expression's lvalue semantics.

Declaration seeding. For declarations, the sandwich pattern cannot be used reliably because the underlying C-to-Rust translator reorders top-level declarations. In such cases, the relative position between a declaration and its surrounding seeds would be lost. To handle this, Seeder employs a fallback strategy, which it uses for the declaration macro `DECL_MACRO_INT` (fig. 9, line 5; fig. 14, line 5). For each declaration, C2Rust emits a companion declaration whose value is the seed string (fig. 13). C2Rust has an option to attach a `#[c2rust::src_loc = "line:col"]` attribute to every top-level item, allowing Reaper to locate the designated Rust declaration.

Fig. 14. Seeded C for fig. 9. Its behavior is identical to fig. 9.

```
#define EXPR_MACRO_ADD(x, y) ((x) + (y))
#define STMT_MACRO_INCR(x) do { (x)++; } while (0)
#define DECL_MACRO_INT int a;

DECL_MACRO_INT const char * SEED_FOR_DECL_MACRO_INT = "{ ... \"astKind\":\"Decl\", ... ,
    \"cuLnColBegin\":\"7:1\", ... }"; // Seed for DECL_MACRO_INT

int main()
{
  *"{ ... \"astKind\":\"Stmt\", \"begin\":true, ... }"; // Begin seed for conditional compilation
  #ifdef FEAT1
  ++a;
  #endif
  *"{ ... \"astKind\":\"Stmt\", \"begin\":false, ... }"; // End seed for conditional compilation

  *"{ ... \"astKind\":\"Stmt\", \"begin\":true, ... }"; // Begin seed for STMT_MACRO_INCR(a)
  STMT_MACRO_INCR(
    (*
      ((*"{ ... \"isArg\":true, \"astKind\":\"Expr\", \"isLvalue\":true, ... }"? // Seed for arg 1
      (&(a)) : ((__typeof__(a)*)(0)))
    )
  );
  *"{ ... \"astKind\":\"Stmt\", \"begin\":false, ... }"; // End seed for STMT_MACRO_INCR(a)

  return
  (
    *"{ ... \"astKind\":\"Expr\", \"begin\":true, ... }"? // Seed for EXPR_MACRO_ADD
    EXPR_MACRO_ADD(
      (*
        ((*"{ ... \"isArg\":true, \"astKind\":\"Expr\", \"isLvalue\":true, ... }"? // Seed for arg 1
        (&(a)) : ((__typeof__(a)*)(0))))
      ),
      (
        *"{ ... \"isArg\":true, \"astKind\":\"Expr\", \"isLvalue\":false, ... }"? // Seed for arg 2
        (5) : (*(__typeof__(5)*)(0))
      )
    ) : (*(__typeof__(EXPR_MACRO_ADD(a,5))*)(0))
  );
}
```

3.4.3 Limitations. Seeder currently has several practical limitations arising from its compiler-agnostic design. It does not implement annotating AST nodes whose kind is `Type`. For expressions, the current seeding mechanism cannot seed integer constant expressions (ICE) or function pointers. Cases involving token pasting or stringification are also difficult, since these operators change the syntactic category of tokens during expansion. These issues stem from the fact that Seeder operates without relying on any source-to-AST correspondence from the underlying compiler. If future translation infrastructures were to expose standardized node-level correspondence between pre- and post-translation ASTs, many of these limitations could be eliminated by bypassing Seeder's manual seeding mechanism altogether. Alternatively, Hayroll could be re-implemented within a C-to-Rust translator.

3.5 Reaper

Goal. Reaper performs the inverse process of Seeder. It reads the Rust files produced by the C-to-Rust translator and reconstructs macro and conditional compilation from the seeds left by Seeder. For macros, Reaper creates Rust macro or function definitions; for conditional compilation, it attaches `#[cfg]` attributes to the corresponding code blocks. Reaper is run once on each split of the program, so conditional regions contain at most one active branch. Reaper does not merge

Fig. 15. Resulting Rust after C2Rust translates fig. 14. Characteristic shapes are preserved for Reaper/Merger.

```
#[c2rust::src_loc = "7:1"]
pub static mut a: libc::c_int = 0;
#[c2rust::src_loc = "7:29"]
pub static mut SEED_FOR_DECL_MACRO_INT: *const libc::c_char =
    b"{ ... \"astKind\":\"Decl\", ... , \"cuLnColBegin\":\"7:1\", ... }\0" as *const u8
        as *const libc::c_char; // Seed for DECL_MACRO_INT

#[c2rust::src_loc = "9:1"]
unsafe fn main() -> libc::c_int {
    *(b"{ ... \"astKind\":\"Stmt\", \"begin\":true, ... }\0" as *const u8 as *const libc::c_char);
           // Begin seed for conditional compilation
    a += 1;
    *(b"{ ... \"astKind\":\"Stmt\", \"begin\":false, ... }\0" as *const u8 as *const libc::c_char);
           // End seed for conditional compilation

    *(b"{ ... \"astKind\":\"Stmt\", \"begin\":true, ... }\0" as *const u8 as *const libc::c_char);
           // Begin seed for STMT_MACRO_INCR
    *if *(b"{ ... \"isArg\":true, \"astKind\":\"Expr\", \"isLvalue\":true, ... }\0" as *const u8 as
           *const libc::c_char) as libc::c_int != 0 // Seed for arg 1
    { &mut a } else { 0 as *mut libc::c_int } += 1;
    *(b"{ ... \"astKind\":\"Stmt\", \"begin\":false, ... }\0" as *const u8 as *const libc::c_char);
           // End seed for STMT_MACRO_INCR

    return if *(b"{ ... \"astKind\":\"Expr\", \"begin\":true, ... }\0" as *const u8 as *const
           libc::c_char) as libc::c_int != 0 // Seed for EXPR_MACRO_ADD
    {
        *(if *(b"{ ... \"isArg\":true, \"astKind\":\"Expr\", \"isLvalue\":true, ... }\0" as *const
               u8 as *const libc::c_char) as libc::c_int != 0 // Seed for arg 1
        { &mut a } else { 0 as *mut libc::c_int })
        +
        (if *(b"{ ... \"isArg\":true, \"astKind\":\"Expr\", \"isLvalue\":false, ... }\0" as *const
               u8 as *const libc::c_char) as libc::c_int != 0 // Seed for arg 2
        { 5 as libc::c_int } else { *(0 as *mut libc::c_int) })
    } else { *(0 as *mut libc::c_int) };
}
```

alternative branches; conditional-compilation seeds remain in place for Merger to combine multiple splits later.

Input and output. The input is the Rust output from the underlying C-to-Rust translator. Reaper requires no sidecar metadata file: all reconstruction information is encoded within the translated source. Reaper's output is a Rust file where (1) some C macros are reconstructed into either Rust macros or Rust functions, and (2) conditional code regions are annotated with `#[cfg]` guards.

Algorithm. Reaper scans the translated AST, identifies seeds, and reconstructs the seeded regions into higher-level constructs. The regions are extracted into Rust function or macro definition bodies, and the seeds are replaced with calls to these definitions. The reconstructed example in fig. 16 shows the result for the running example.

Macro definition consistency. A key challenge arises in ensuring that multiple invocations of the same C macro definition can be reconstructed into a single Rust definition. In principle, one C macro definition may correspond to several expansion sites. However, after translation, the resulting Rust constructs are not always structurally identical, breaking the one-to-many correspondence between a definition and its invocations.

We observed two primary causes of this inconsistency:

(1) Type divergence. C macros often rely on implicit type promotion or ad-hoc polymorphism. Two invocations expand to syntactically identical C code, but after translation, Rust's stricter typing rules can produce extra explicit type conversions. For example, if `EXPR_MACRO_ADD` is invoked once with `int` arguments and once with `long long` arguments, C's integer promotion rules would allow both cases, but the translated Rust code contains an explicit cast in only one version. In such cases,

Fig. 16. Reaper output after reconstruction of fig. 15.

```
use ::libc;
macro_rules! DECL_MACRO_INT {
   () => {
      #[no_mangle]
      pub static mut a: libc::c_int = 0;
   }
}
DECL_MACRO_INT!();

unsafe fn EXPR_MACRO_ADD(x: *mut libc::c_int, y: libc::c_int) -> libc::c_int {
   *(x) + (y)
}

unsafe fn STMT_MACRO_INCR(x: *mut libc::c_int) {
   *x += 1;
}

unsafe fn main() -> libc::c_int {
   // Reaper leaves seeds for conditional compilation in place. Merger will handle them later.
   *(b"{ ... \"astKind\":\"Stmt\", \"begin\":true, ... }\0"
      as *const u8 as *const libc::c_char); // Begin seed
   #[cfg(feature = "FEAT1")]
   (a += 1);
   *(b"{ ... \"astKind\":\"Stmt\", \"begin\":false, ... }\0"
      as *const u8 as *const libc::c_char); // End seed

   STMT_MACRO_INCR(&mut a);
   return EXPR_MACRO_ADD(&mut a, 5 as libc::c_int);
}
```

Reaper separates these invocations and reconstructs multiple Rust definitions, one for each distinct parameter type combination.

(2) Lvalue/rvalue divergence. Even when argument types match, the lvalue/rvalue status of macro arguments can still differ. In the running example, the macro `EXPR_MACRO_ADD` is invoked with the first argument `a` (an lvalue) and the second argument `5` (an rvalue). The reconstructed Rust function therefore treats the first parameter as a mutable pointer (`*mut T`) and the second as an immutable value. When multiple invocations exist, Reaper analyzes each of the argument uses: if any invocation passes an rvalue for a parameter, that parameter is reconstructed as an rvalue in Rust; only when all invocations use lvalues does Reaper infer a mutable pointer type. This rule maximizes the macro body's accessibility to a modifiable argument, unless the user has written an invocation that passes an rvalue to the argument, which itself acts as the proof that the macro body does not need that argument to be modifiable.

Expression-level conditional compilation. Rust's built-in conditional attribute `#[cfg]` can appear on declarations or statements, but not before arbitrary expressions. The built-in macro `cfg!()` however evaluates to a Boolean at runtime and can safely appear inside expressions. When the C `#if` guards an expression, Reaper reconstructs the guard using the `cfg!()` macro instead of `#[cfg]`. See fig. 17 for an example. Here, the guard is embedded within the expression's conditional structure. Although `cfg!()` is technically evaluated at runtime, it can still possibly be optimized away during compile time.

3.6 Merger

Goal. Merger is the final stage of the Hayroll pipeline. It receives multiple individually translated Rust outputs produced under different configuration combinations by the macro translation layer, and it combines them into a single configurable Rust program.

Fig. 17. Expression-level conditional compilation reconstructed using `cfg!()`.

```
a = if cfg!(feature = "FEAT1") {
        1 as libc::c_int
    } else {
        *(0 as *mut libc::c_int)
    };
```

Input and output. Each merge operation takes two Rust outputs and produces one unified result. At the pipeline level, this process repeats iteratively: each new translation is merged with the accumulated result until all splits have been combined into a single output. The final output is one Rust file that reintroduces configuration choices using `#[cfg]` attributes or `cfg!()` macros.

Algorithm. The merging process operates over three categories of program elements:

(1) Top-level declarations. For global items such as functions or static variables, the two inputs are treated as complementary: new declarations from either side are added to the result, and duplicates are retained from either one of the versions. Any identically named top-level items are assumed to have identical content.

(2) Statement-level conditional regions. Conditional blocks are matched by their remaining seeds in the post-Reaper program. For each distinct `#[cfg]` feature condition, Merger includes one copy; placeholders corresponding to inactive branches are replaced by the matching active blocks from other splits.

(3) Expression-level conditionals. For expressions reconstructed using the `cfg!()` macro (see fig. 17), Merger simply extends the conditional chain: if one branch contains an additional configuration, it is appended as a new `else if` clause in the unified output.

Because the seeds include source location, Merger does not need to infer correspondences or perform AST unification. It directly matches conditional ranges in the two Reaper outputs according to their original source location indicated by the seed info.

Our running example does not contain multiple configurations, so we do not show a separate complete merged output. In this case, the merged result is simply fig. 16 with lines 19–21 and 24–25 removed. If the original C program had included an `#else` branch for `FEAT1`, Merger would insert that branch immediately after line 23.

3.7 Correctness

Hayroll's correctness relies on contracts between components. At a high level, both layers follow the same pattern: an analyzer extracts information, a decomposition step partitions the problem according to that information, and a later reconstruction step reassembles the translated results.

The conditional compilation translation layer. Pioneer computes activation premises for conditional compilation regions. Given these premises, Splitter emits a set of concrete configurations that jointly cover every region that Pioneer marked reachable. Assuming the macro translation layer correctly translates each emitted task, Merger can then reconstruct one Rust program that covers the original configurable behaviors.

The macro translation layer. Maki reports the relevant properties of each macro expansion. Seeder then inserts seeds according to this information. Assuming the underlying translator preserves the semantics of the seeded program and respects the seeded-subtree assumption from section 2, Reaper can recover the seeded regions one by one and reconstruct the corresponding Rust macros or functions.

These contracts also clarify what happens when some information is unavailable or unusable. If Pioneer misses a conditional compilation region, that region will be translated with one default configuration, reducing configurability but not blocking translation of the surrounding program.

Likewise, if a macro expansion is non-syntactical or otherwise unsuitable for reconstruction, Hayroll leaves it expanded instead of reconstructing it as a Rust abstraction. In both cases, the result is a conservative loss of recovered structure rather than a failure of the remaining pipeline: other regions can still be translated and the final Rust output remains compilable.

4 Implementation

Hayroll is implemented as a mixed-language toolchain. The C-side components (Pioneer, Splitter, Seeder, and extended Maki) are written in C++. The Rust-side components (Reaper and Merger) are written in Rust. Pioneer and Splitter rely on `tree-sitter` (v0.25.10) [44] and a modified `tree-sitter-c` [45] grammar (`tree-sitter-c_preproc`) and use the Z3 SAT/SMT solver (v4.13.4) [38] for symbolic reasoning and model enumeration. Maki is written as a Clang/LLVM (v17.0.6) analysis pass. The underlying transpiler is C2Rust (0.20.0) [20]. Reaper and Merger use the rust-analyzer (2024-12-16) [23] library for parsing and AST transformation. Hayroll is open-sourced at https://github.com/UW-HARVEST/Hayroll.

5 Evaluation

5.1 Methodology

We evaluated the correctness, effectiveness, and performance of Hayroll.

To evaluate **correctness**, we ran tests. We ran each translated Rust program against the C tests. This is possible because C2Rust (and thus Hayroll) outputs Rust code that is ABI-compatible with the original C program. For object programs that are configurable via conditional compilation, we ran the tests once under each preprocessor define combination that Splitter emits. All object programs pass all tests except for a special case discussed in section 5.3.

To evaluate **effectiveness**, we counted the number of macro invocations successfully translated for each macro category, and what Rust code structures (Rust function or Rust macro) they are translated into. A translation is considered successful when Hayroll does not leave it expanded as-is. We also report why Hayroll leaves some macro invocations expanded by C2Rust.

To evaluate **performance**, we report the per-thread time spent on each component of Hayroll for translating the subject programs, normalized to milliseconds per 1000 LoC. The LoC is counted independently for each compilation unit, that is, the C source file and all its included headers.

5.2 Subject Programs

We evaluated Hayroll on three C codebases: CRUST-Bench, LibmCS, and zlib. Together they contain 62,821 lines of C code (non-blank, non-comment, including header files, but excluding system header files).

CRUST-Bench [22] consists of 100 C programs, each paired with tests and a manually written Rust counterpart. Because CRUST-Bench's Rust side was designed for LLM-based translators rather than transpilers, our evaluation uses only its C source and test portions. Before translation, we excluded programs on which the baseline C2Rust failed, that is, programs that failed to compile in C, failed during C2Rust translation, or the resulting Rust code failed to compile or pass tests. After filtering, 33 programs remain, covering 149 C files and 81 C header files with 19,633 lines of C code and 3,113 lines of header code. CRUST-Bench contains no conditional compilation, so it mainly serves to evaluate macro reconstruction and end-to-end correctness.

LibmCS (v1.2.0) [13] is an implementation of the C mathematical library for critical systems. Our evaluation exercises all of its preprocessor flags except those related to complex number support, which C2Rust does not support. The four preprocessor defines explored are: `LIBMCS_FPU_DAZ`, `LIBMCS_DOUBLE_IS_32BITS`, `LIBMCS_LONG_DOUBLE_IS_64BITS`, and

Table 4. Macro translation outcomes for CRUST-Bench by syntactic category.

Outcome	All	Syntactical	Expr.	Stmt.	Decl.	Type	Non-syn.
Total macros	1407	1143	417	605	32	89	264
Successfully translated	699 (50%)	699 (61%)	351 (84%)	348 (58%)	0 (0%)	0 (0%)	0 (0%)
→ Rust function	427 (61%)	427 (61%)	326 (93%)	101 (29%)	0 (0%)	0 (0%)	0 (0%)
→ Rust macro	272 (39%)	272 (39%)	25 (7%)	247 (71%)	0 (0%)	0 (0%)	0 (0%)
Left expanded	708 (50%)	444 (39%)	66 (16%)	257 (42%)	32 (100%)	89 (6%)	264 (100%)

Table 5. Macro translation outcomes for LibmCS by syntactic category.

Outcome	All	Syntactical	Expr.	Stmt.	Decl.	Type	Non-syn.
Total macros	597	590	217	373	0	0	7
Successfully translated	574 (96%)	574 (97%)	201 (93%)	373 (100%)	0 (0%)	0 (0%)	0 (0%)
→ Rust function	322 (56%)	322 (56%)	144 (72%)	178 (48%)	0 (0%)	0 (0%)	0 (0%)
→ Rust macro	252 (44%)	252 (44%)	57 (28%)	195 (52%)	0 (0%)	0 (0%)	0 (0%)
Left expanded	23 (4%)	16 (3%)	16 (7%)	0 (0%)	0 (0%)	0 (0%)	7 (100%)

Table 6. Macro translation outcomes for zlib by syntactic category.

Outcome	All	Syntactical	Expr.	Stmt.	Decl.	Type	Non-syn.
Total macros	1930	1090	820	250	0	20	840
Successfully translated	820 (42%)	820 (75%)	696 (85%)	124 (50%)	0 (0%)	0 (0%)	0 (0%)
→ Rust function	762 (93%)	762 (93%)	679 (98%)	83 (67%)	0 (0%)	0 (0%)	0 (0%)
→ Rust macro	58 (7%)	58 (7%)	17 (2%)	41 (33%)	0 (0%)	0 (0%)	0 (0%)
Left expanded	1110 (58%)	270 (25%)	124 (15%)	126 (50%)	0 (0%)	20 (1%)	840 (100%)

`LIBMCS_LONG_IS_32BITS`. LibmCS itself does not include test cases, so we adapted tests from OpenLibm [34]. The same filtering criteria as in CRUST-Bench are applied: we retain only tests that pass both the native C build and the C2Rust baseline build. LibmCS includes 190 C files (9,748 lines of code) and 18 C header files (1,373 lines of code). This benchmark evaluates Hayroll 's ability to reconstruct macros and handle conditional compilation logic.

Zlib (v1.3.1) [28] is a widely used data compression library with a large number of optional build configurations controlled by preprocessor flags. The size of all available flags would overwhelm Pioneer, so we enabled Pioneer whitelist mode, restricting symbolic analysis to five configuration flags: `HAVE_HIDDEN`, `NO_STRERROR`, `NO_snprintf`, `NO_vsnprintf`, and `ZLIB_CONST`. Besides size concerns, other flags were excluded also because they require non-Boolean macro values (e.g., string or integer assignments), are platform-specific, enable conditionally defined macros, which are beyond the designed capabilities of Hayroll and C2Rust. Zlib contains 41 C files with 17,252 lines of code and 27 C header files with 11,702 lines of code. It includes its own test suite, which we used in experiments. This benchmark demonstrates Hayroll 's ability to reconstruct macros and Pioneer's selective symbolic execution capability under challenging realistic software sizes.

5.3 Results

Correctness. All translated programs produced by Hayroll pass their corresponding test suites under every preprocessor-define combination emitted by Splitter with one exception. The only exception occurred in LibmCS when the `LIBMCS_DOUBLE_IS_32BITS` flag was enabled: under this configuration, several mathematical functions failed precision checks in the adapted OpenLibm tests. This deviation is expected, because the tests assume 64-bit double precision, while this flag

Table 7. Macro translation failure reasons. Percentages are relative to all kept-as-is macros in each benchmark.

Failing reason	CRUST-Bench	LibmCS	zlib
Non-syntactical	264 (37%)	7 (30%)	840 (76%)
Argument non-syntactical	260 (37%)	2 (9%)	120 (11%)
Uses stringification	140 (20%)	0 (0%)	0 (0%)
Unhygienic	131 (19%)	16 (70%)	164 (15%)
Requires integral constant expression	127 (18%)	0 (0%)	104 (9%)
Unsupported AST kind	89 (13%)	0 (0%)	20 (2%)
Uses token pasting	20 (3%)	0 (0%)	0 (0%)
Argument function pointer	13 (2%)	0 (0%)	0 (0%)

Table 8. Pipeline performance per component (ms/kLoC, share of total runtime).

Component	CRUST-Bench	LibmCS	zlib
Pioneer (symbolic eval)	32 (5%)	807 (28%)	351 (17%)
Splitter	0 (0%)	45 (2%)	1 (0%)
Maki (C macro analysis)	535 (75%)	1409 (49%)	1425 (67%)
Seeder	3 (0%)	28 (1%)	10 (0%)
C2Rust	36 (5%)	172 (6%)	88 (4%)
Reaper	54 (8%)	165 (6%)	126 (6%)
Merger	51 (7%)	239 (8%)	118 (6%)
Aggregate	**711 (100%)**	**2865 (100%)**	**2119 (100%)**

explicitly reduces precision to 32 bits. Therefore, the observed failure reflects the intended semantics of the configuration and further validates the correctness of our translation.

Effectiveness. Tables 4–6 summarize macro translation outcomes across syntactic categories. Overall, Hayroll achieves a high success rate on syntactical macros. Across all three benchmarks, syntactical macros account for 2,823 out of 3,934 total macros (72%), and among them, 2,093 (74%) were successfully translated. Non-syntactical macros are left expanded as expected. Among the benchmarks, LibmCS shows the highest success rate (97% of syntactical macros translated), reflecting its disciplined macro usage. By contrast, zlib contains a large fraction of non-syntactical macros, which lowers its overall translation ratio despite comparable results on syntactical macros.

Table 7 further classifies the reasons for expanded-as-is macros according to our macro taxonomy. Note that one such case may involve multiple causes and that the percentages do not add to 100%. The dominant causes are non-syntactical expansions and unhygienic macros. Other common reasons include the use of integer constant expressions and unsupported AST kinds (`Type`). `Type` and `Decl` are not common in our test programs. Hayroll does not handle `Type` macros. All `Decl` macros appear in CRUST-Bench, all of which involve function pointers and thus were expanded as-is. These findings confirm that translation quality strongly depends on disciplined macro usage.

Performance. Table 8 reports the per-component runtime. Across all benchmarks, Maki macro analysis dominates the runtime, accounting for 49–75% of total execution time. The overall overhead compared to a plain C2Rust translation is roughly 20×, but this is a one-time translation cost.

The cost distribution also reflects each benchmark's configurability. In CRUST-Bench, which contains no conditional compilation, Pioneer's symbolic execution takes only 5% of total runtime. In contrast, symbolic evaluation becomes more significant for LibmCS (28%) and zlib (17%).

In configurable codebases, Pioneer can be instructed to adapt its exploration strategy to the scale of the project. For smaller systems such as LibmCS, which defines 4 configuration flags, a naive enumeration would cause a combinatorial explosion of 2^4 variants. Pioneer's symbolic execution automatically detects mutually exclusive and independent relationships among these flags, reducing the space to only 4 splits. For large and heavily configurable projects such as zlib, however, full symbolic exploration remains infeasible due to the number and complexity of available flags. In such cases, we employ whitelist-based and hybrid symbolic-concrete execution modes to restrict exploration to a user-specified subset of configuration options.

Overall, the data demonstrate that Hayroll maintains practical translation performance.

6 Limitations and Discussions

6.1 Limitations

First, **Seeder cannot annotate certain AST node kinds**, including type nodes, integer constant expressions, token pasting, stringification, and macros that expand to function pointers (see section 3.4.3 for details). These could possibly be mitigated by having an underlying transpiler that provides a source correspondence interface. Second, **nested macro definitions are not reconstructed** because Maki cannot reliably analyze the semantics of inner macro calls. Third, **macros whose definitions vary across `#ifdef` branches are not handled**. This could be addressed by providing our macro translation layer with extra information from the conditional compilation translation layer, so that each macro invocation is aware of the premise for its definition. Fourth, **Pioneer does not scale well enough to fully explore the configuration space of larger codebases like zlib**. Pioneer is successful in exploring selected configurations of zlib with its whitelist mode, but experiences state explosion when trying to explore all its configurations.

6.2 Discussion of Non-syntactical Macros

Although non-syntactical macros are not reconstructed as Rust abstractions, they do not disrupt Hayroll's correctness. Pioneer parses only preprocessor directives and leaves ordinary code as opaque `c_tokens`, so non-syntactical macros do not cause parse failure. Maki detects all macro expansions, including non-syntactical ones, because it is implemented as a Clang plugin that hooks into every macro expansion event. Seeder does not annotate non-syntactical macros; they are left expanded for the downstream translator. Because there are no seeds for them, they are not reconstructed as Rust abstractions, but their translated forms are still functionally correct.

In our benchmarks, the vast majority of non-syntactical macros did not interact with conditional compilation. We observed one notable interacting pattern: C++ compatibility wrappers that conditionally surround an entire file with `extern "C"` fragments. Since our target is standard C-to-Rust translation, these wrappers were conservatively ignored and did not affect correctness in our benchmarks.

6.3 Threats to Validity

Although Hayroll is designed to be independent of any particular C-to-Rust translator, the current prototype is implemented specifically with C2Rust and we have not plugged other translators into Hayroll. Other translators, especially LLM-based or aggressively optimizing ones, may not preserve the syntactic properties that Hayroll depends upon, in which case the seeds may fail to align with the translated Rust code. A source correspondence interface provided by the underlying translator would free Hayroll from the dependency on Seeder and solve this threat.

7 Related Work

7.1 Translating C to Rust

One of the earliest and most relied-on transpilers from C to Rust is Immunant's C2Rust tool [20]. Over the last 1–2 years, DARPA's TRACTOR program [5] has supercharged the already active research area of C-to-Rust translation. Prior to the explosion of interest in LLMs, most of this work built on C2Rust following more traditional transpiler approaches to language translation. However, recent years have seen a large amount of work seeking to exploit LLMs for more idiomatic and safe Rust translations at the expense of correctness guarantees.

Earlier C-to-Rust translation work [7, 26] built directly on Immunant's C2Rust tool, attempting to post-process the output to become safer. Almost all subsequent approaches we are aware of that transpile without use of an LLM have similarly been implemented as a C2Rust post-processor. Naturally much of this work has focused on analyzing references/pointers to try to make them safer [6, 51], through alias analysis, inference of lifetimes, ownership, etc. Hong and Ryu in particular have scoped out a variety of specific post-processing sub-problems, including lock discipline [14], argument promotion into output parameters [15], translation of C unions into Rust enums [16], and translation of I/O API usage [18]. Wu and Demsky [47] address the problem of translating C pointers with ambiguous type into parametrically polymorphic Rust code. Notably, they modify the C2Rust compiler directly rather than implementing their method as a post-process. Fromherz and Protzenko [11] are a rare exception to the above trend, instead focusing on a subset of C and aiming to provide higher formal guarantees of correctness than C2Rust. Regardless, Hayroll can be applied in conjunction with almost any of the above approaches because it wraps the underlying transpiler. This could be done by wrapping the innermost translation or the innermost translation composed with various post-processor passes.

While some LLM-based translation work has also chosen to post-process C2Rust output [12, 32] to eliminate uses of unsafe Rust features, the majority of LLM-based translation directly accepts the C source code as input. Different systems have been assembled from different combinations of a common set of strategies and considerations: decomposition, validation loops, and provision of analysis information. In order to handle large amounts of code, many systems identify boundaries, such as function signatures or types on which to decompose [2, 17, 41, 54] the translation problem. These interfaces, or code skeletons are first translated with or without LLMs. Because LLM-based translations have no correctness guarantees, validation and verification [49, 53] are much more serious concerns. Fuzzing combined with differential testing between an oracle and translation [8] is one popular approach. Translation of existing C tests, or LLM generation of tests [24] are additional popular approaches. Bai and Palit [1] are especially notable for using symbolic execution rather than fuzzing as a basis for differential testing. Validation failures are frequently used in iterative feedback loops [10, 42] to improve translation quality. While most validation work uses non-LLM validation methods, some work has explored using LLMs to generate validation code as well [19] thereby eliminating all reliable ground truth from the translation process. Finally, various static and dynamic analyses can be performed prior to translation and then supplied as extra information to aid translation or validation steps [27, 35, 40, 48, 50]. Some work also focuses on prompt engineering strategies [52], or separately asking an LLM to summarize a program prior to asking for a translation of that program [29].

Prior work does not address the challenges of macro translation. However, we will briefly mention two papers currently under review. EvoC2Rust [46] is a notable exception among LLM translation work for considering function-like macros and include directives among the kinds of code structures singled out for translation. Their work does not address conditional compilation, nor the full variety

of macro uses considered in Hayroll. In a "work in progress" report, DeGreef et al. [4] discuss the handling of Expr macros, addressing a subset of the considerations handled in Hayroll.

7.2 The C Preprocessor

Ernst et al. [9] conducted the first empirical study of C preprocessor use. Along with Maki, it forms the basis for our taxonomy of C macro attributes affecting C-to-Rust translation. Liebig et al. [25] and Medeiros et al. [31] conducted later studies, with Medeiros giving specific consideration to conditional compilation as it relates to build system configurations.

Our semantic analysis of preprocessor usage is conducted using Maki [36]. "Un-preprocessing" [3] is an approach to adding C preprocessor support to C program modification tools, such as refactoring tools. Their system architecture resembles the Hayroll architecture, but only works on C-to-C program transformations, not inter-language translations. Kästner et al. [21] describe a symbolic execution engine for the preprocessor that is very similar to our Pioneer subsystem, but makes a few different decisions about how to implement their symbolic execution.

The problem of covering different conditional compilations has most frequently been studied from the perspective of testing systems across multiple configurations. Rothberg et al. 2016 [39] (and many other authors) seek to test the Linux kernel under different configurations. While attempting to test different configurations of a piece of software, their concern is coverage of dynamic behavior, rather than static source for the purpose of translation. Medeiros et al. [30] survey and compares 10 different sampling algorithms for variant testing. These algorithms express different coverage criteria for configuration variations, as expressed using conditional compilation. Tartler et al. [43] use the *statement coverage* heuristic, which resembles the Pioneer symbolic execution approach.

8 Conclusion

We introduced Hayroll, a modular wrapper that restores macro structure and conditional compilation to C-to-Rust translation without modifying the underlying translator. By combining symbolic execution of preprocessor directives with a seed-based reconstruction pipeline, Hayroll preserves configuration behavior and recovers macro abstractions that existing tools discard. Our evaluation on CRUST-Bench, LibmCS, and zlib shows that this approach is correct across configurations, translates most syntactical macros, and scales to real codebases through symbolic reasoning and hybrid execution. While limited by current tagging and compiler-agnostic design, Hayroll demonstrates that decoupled preprocessor analysis can make C-to-Rust translation both macro-aware and practical.

9 Data Availability Statement

We open-source Hayroll at https://github.com/UW-HARVEST/Hayroll. An archive version is available at https://doi.org/10.5281/zenodo.19067228.

10 Acknowledgments

We thank Wuihee Yap, Mark Roberts, and Megan Frisella for helping with the evaluation. We also thank Khyber Sen for testing Hayroll and providing feedback and fixes.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590132.

References

[1] Yubo Bai and Tapti Palit. 2025. RustAssure: Differential Symbolic Testing for LLM-Transpiled C-to-Rust Code. arXiv:2510.07604 [cs.SE] https://arxiv.org/abs/2510.07604

[2] 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. In *Engineering of Complex Computer Systems*, Yuan Zhou, Sin G. Teo, Xiaofei Xie, Zuohua Ding, and Yang Liu (Eds.). Springer Nature Switzerland, Cham, 283–302.

[3] Yufeng Cheng, Meng Wang, Yingfei Xiong, Zhengkai Wu, Yiming Wu, and Lu Zhang. 2017. Un-preprocessing: Extended CPP that works with your tools. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware* (Shanghai, China) *(Internetware '17)*. Association for Computing Machinery, New York, NY, USA, Article 3, 10 pages. doi:10.1145/3131704.3131715

[4] Robbe De Greef, Attilio Discepoli, Esteban Aguililla Klein, Théo Engels, Ken Hasselmann, and Antonio Paolillo. 2025. Towards Macro-Aware C-to-Rust Transpilation (WIP). In *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Seoul, Republic of Korea) *(LCTES '25)*. Association for Computing Machinery, New York, NY, USA, 57–61. doi:10.1145/3735452.3735535

[5] Defense Advanced Research Projects Agency (DARPA). 2024. *Translating All C TO Rust (TRACTOR)*. Technical Report. U.S. Department of Defense. https://sam.gov/opp/1e45d648886b4e9ca91890285af77eb7/view Broad Agency Announcement, BAA HR001124S0035.

[6] 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), 29 pages. doi:10.1145/3586046

[7] 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), 29 pages. doi:10.1145/3485498

[8] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2025. Towards Translating Real-World Code with LLMs: A Study of Translating to Rust. arXiv:2405.11514 [cs.SE] https://arxiv.org/abs/2405.11514

[9] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An empirical analysis of C preprocessor use. *IEEE TSE* 28, 12 (Dec. 2002), 1146–1170.

[10] Muhammad Farrukh, Smeet Shah, Baris Coskun, and Michalis Polychronakis. 2025. SafeTrans: LLM-assisted Transpilation from C to Rust. arXiv:2505.10708 [cs.CR] https://arxiv.org/abs/2505.10708

[11] Aymeric Fromherz and Jonathan Protzenko. 2024. Compiling C to Safe Rust, Formalized. arXiv:2412.15042 [cs.PL] https://arxiv.org/abs/2412.15042

[12] Yifei Gao, Chengpeng Wang, Pengxiang Huang, Xuwei Liu, Mingwei Zheng, and Xiangyu Zhang. 2025. PR2: Peephole Raw Pointer Rewriting with LLMs for Translating C to Safer Rust. arXiv:2505.04852 [cs.SE] https://arxiv.org/abs/2505.04852

[13] GTD GmbH. 2025. LibmCS. https://gitlab.com/gtd-gmbh/libmcs.

[14] Jaemin Hong and Sukyoung Ryu. 2023. Concrat: An Automatic C-to-Rust Lock API Translator for Concurrent Programs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 716–728. doi:10.1109/ICSE48619.2023.00069

[15] 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), 25 pages. doi:10.1145/3656406

[16] Jaemin Hong and Sukyoung Ryu. 2024. To Tag, or Not to Tag: Translating C's Unions to Rust's Tagged Unions. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 40–52. doi:10.1145/3691620.3694985

[17] Jaemin Hong and Sukyoung Ryu. 2024. Type-migrating C-to-Rust translation using a large language model. *Empirical Softw. Engg.* 30, 1 (Oct. 2024), 38 pages. doi:10.1007/s10664-024-10573-2

[18] Jaemin Hong and Sukyoung Ryu. 2025. Forcrat: Automatic I/O API Translation from C to Rust via Origin and Capability Analysis. arXiv:2506.01427 [cs.SE] https://arxiv.org/abs/2506.01427

[19] Ali Reza Ibrahimzada, Brandon Paulsen, Reyhaneh Jabbarvand, Joey Dodds, and Daniel Kroening. 2025. MatchFixAgent: Language-Agnostic Autonomous Repository-Level Code Translation Validation and Repair. arXiv:2509.16187 [cs.SE] https://arxiv.org/abs/2509.16187

[20] Immunant, Inc. 2018. C2Rust: Automatic Translation from C to Rust. https://github.com/immunant/c2rust. Accessed: 2025-10-28. Original release 2018-04-20..

[21] Christian Kästner, Paolo G. Giarrusso, and Klaus Ostermann. 2011. Partial preprocessing C code for variability analysis. In *Proceedings of the 5th International Workshop on Variability Modeling of Software-Intensive Systems* (Namur, Belgium) *(VaMoS '11)*. Association for Computing Machinery, New York, NY, USA, 127–136. doi:10.1145/1944892.1944908

[22] Anirudh Khatry, Robert Zhang, Jia Pan, Ziteng Wang, Qiaochu Chen, Greg Durrett, and Isil Dillig. 2025. CRUST-Bench: A Comprehensive Benchmark for C-to-safe-Rust Transpilation. arXiv:2504.15254 [cs.SE] https://arxiv.org/abs/2504.15254

[23] The Rust Programming Language. 2025. rust-analyzer. https://github.com/rust-lang/rust-analyzer.

[24] Tianyu Li, Ruishi Li, Bo Wang, Brandon Paulsen, Umang Mathur, and Prateek Saxena. 2025. Adversarial Agent Collaboration for C to Rust Translation. arXiv:2510.03879 [cs.SE] https://arxiv.org/abs/2510.03879

[25] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development* (Porto de Galinhas, Brazil) *(AOSD '11)*. Association for Computing Machinery, New York, NY, USA, 191–202. doi:10.1145/1960275.1960299

[26] 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 *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 354–355. doi:10.1145/3510454.3528640

[27] Yuchen Liu, Junhao Hu, Yingdi Shan, Ge Li, Yanzhen Zou, Yihong Dong, and Tao Xie. 2025. LLMigrate: Transforming "Lazy" Large Language Models into Efficient Source Code Migrators. arXiv:2503.23791 [cs.PL] https://arxiv.org/abs/2503.23791

[28] Jean loup Gailly and Mark Adler. 2025. zlib. https://www.zlib.net/.

[29] 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 [cs.SE] https://arxiv.org/abs/2508.06926

[30] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 643–654. doi:10.1145/2884781.2884793

[31] Flávio Medeiros, Iran Rodrigues, Márcio Ribeiro, Leopoldo Teixeira, and Rohit Gheyi. 2015. An empirical study on configuration-related issues: investigating undeclared and unused identifiers. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Pittsburgh, PA, USA) *(GPCE 2015)*. Association for Computing Machinery, New York, NY, USA, 35–44. doi:10.1145/2814204.2814206

[32] 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 [cs.SE] https://arxiv.org/abs/2501.14257

[33] Office of the National Cyber Director (ONCD). 2024. *Back to the Building Blocks: A Path Toward Secure and Measurable Software*. Technical Report. The White House, Washington, DC. https://bidenwhitehouse.archives.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/ Press Release: Future Software Should Be Memory Safe.

[34] The JuliaMath Organization. 2025. OpenLibm. https://github.com/JuliaMath/openlibm.

[35] Guangsheng Ou, Mingwei Liu, Yuxuan Chen, Xueying Du, Shengbo Wang, Zekai Zhang, Xin Peng, and Zibin Zheng. 2025. Enhancing LLM-based Code Translation in Repository Context via Triple Knowledge-Augmented. arXiv:2503.18305 [cs.SE] https://arxiv.org/abs/2503.18305

[36] Brent Pappas and Paul Gazzillo. 2024. Semantic Analysis of Macro Usage for Portability. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 21, 12 pages. doi:10.1145/3597503.3623323

[37] Alex Rebert and Christoph Kern. 2024. Secure by design: Google's perspective on memory safety. *Technical report, Google Security Engineering* (2024).

[38] Microsoft Research. 2025. Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[39] Valentin Rothberg, Christian Dietrich, Andreas Ziegler, and Daniel Lohmann. 2016. Towards scalable configuration testing in variable software. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Amsterdam, Netherlands) *(GPCE 2016)*. Association for Computing Machinery, New York, NY, USA, 156–167. doi:10.1145/2993236.2993252

[40] 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 [cs.SE] https://arxiv.org/abs/2412.14234

[41] Momoko Shiraishi and Takahiro Shinagawa. 2024. Context-aware Code Segmentation for C-to-Rust Translation using Large Language Models. arXiv:2409.10506 [cs.SE] https://arxiv.org/abs/2409.10506

[42] 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 [cs.PL] https://arxiv.org/abs/2505.15858

[43] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static analysis of variability in system software: the 90,000 #ifdefs issue. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 421–432.

[44] The tree-sitter Project. 2025. tree-sitter. https://github.com/tree-sitter/tree-sitter.

[45] The tree-sitter Project. 2025. tree-sitter-c. https://github.com/tree-sitter/tree-sitter-c.

[46] Chaofan Wang, Tingrui Yu, Chen Xie, Jie Wang, Dong Chen, Wenrui Zhang, Yuling Shi, Xiaodong Gu, and Beijun Shen. 2025. EvoC2Rust: A Skeleton-guided Framework for Project-Level C-to-Rust Translation. arXiv:2508.04295 [cs.SE] https://arxiv.org/abs/2508.04295

[47] Xiafa Wu and Brian Demsky. 2025. GenC2Rust: Towards Generating Generic Rust Code from C. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 90–102. doi:10.1109/ICSE55347.2025.00127

[48] Qingxiao Xu and Jeff Huang. 2025. Optimizing Type Migration for LLM-Based C-to-Rust Translation: A Data Flow Graph Approach. In *Proceedings of the 14th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (Seoul, Republic of Korea) *(SOAP '25)*. Association for Computing Machinery, New York, NY, USA, 8–14. doi:10.1145/3735544.3735582

[49] 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 [cs.PL] https://arxiv.org/abs/2404.18852

[50] Zhiqiang Yuan, Wenjun Mao, Zhuo Chen, Xiyue Shang, Chong Wang, Yiling Lou, and Xin Peng. 2025. Project-Level C-to-Rust Translation via Synergistic Integration of Knowledge Graphs and Large Language Models. arXiv:2510.10956 [cs.SE] https://arxiv.org/abs/2510.10956

[51] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. 2023. Ownership Guided C to Rust Translation. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 459–482.

[52] Ruxin Zhang, Shanxin Zhang, and Linbo Xie. 2025. A systematic exploration of C-to-rust code translation based on large language models: prompt strategies and automated repair. *Automated Software Engineering* 33, 1 (2025), 21. doi:10.1007/s10515-025-00570-0

[53] Han Zhou, Yu Luo, Mengtao Zhang, and Dianxiang Xu. 2025. C2RustTV: An LLM-based Framework for C to Rust Translation and Validation. In *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*. 1254–1259. doi:10.1109/COMPSAC65507.2025.00158

[54] Tianyang Zhou, Haowen Lin, Somesh Jha, Mihai Christodorescu, Kirill Levchenko, and Varun Chandrasekaran. 2025. LLM-Driven Multi-step Translation from C to Rust using Static Analysis. arXiv:2503.12511 [cs.SE] https://arxiv.org/abs/2503.12511

Received 2025-11-14; accepted 2026-04-03