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

ANONYMOUS AUTHOR(S)

Existing C-to-Rust translators preprocess their input before translation, discarding macro structure and configurability. As a result, they are incomplete on configurable software and lose programmer-defined abstractions embedded in C macros.

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

Evaluated on CRUST-Bench, LibmCS, and zlib, InariRoll preserves correctness across configurations, successfully reconstructs most syntactical macros, and avoids configuration explosion through symbolic execution. These results show that decoupled preprocessor analysis can restore configurability and abstraction to C-to-Rust translation in a practical and modular way.

1 Introduction

The C programming language underlies many of the software systems in the world. 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 instantly eliminate the largest cause of defects and vulnerabilities. Such translation is widely recommended, including by government authorities [33].

Many C-to-Rust translators exist [7, 11, 20, 26, 51], but to the best of our knowledge, they all 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 first run the preprocessor (`cpp`) once, which replaces every preprocessor directive with a single definition, and then work on the pure C file.

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. For an example, see Figure 1.

To address the important problem of incomplete translations that destroy abstractions, we have created a system, InariRoll, that preserves most conditional compilation and macro abstractions through a C-to-Rust pipeline. InariRoll converts C `#if` directives into Rust `#[cfg]` attributes. It converts C macros into Rust functions or macros.

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

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;
    } ...
```

InariRoll 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: Example from LibmCS [13] comparing C2Rust and InariRoll outputs. Highlighting shows source correspondence. C2Rust removes the inactive `#ifdef` region and expands all macros, thus loses configurability and abstraction. InariRoll translates the `#ifdef` region to a `#[cfg]`-guarded region, and reconstructs C macros as Rust functions, preserving configurability and abstraction.

We evaluated InariRoll on 62,821 LoC. With respect to *correctness*, InariRoll's translation passes all the same tests as C2Rust [20], a state-of-the-art translator. With respect to *completeness*, InariRoll preserves 74% of syntactical macro invocations (53% of all macro invocations) into Rust code. With respect to *performance*, InariRoll takes roughly 1-5 seconds per kLoC, depending on the complexity of the codebase.

We summarize our contributions as follows:

- **C macro taxonomy for Rust translation.** We define a taxonomy that classifies C macros by their properties that matter for translation to Rust, and use it to distinguish macros that can be translated from those that must be expanded.
- **Conditional compilation resolution (Pioneer).** We formalize and build 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]`.
- **In-AST tagging for source correspondence (Seeder/Reaper).** We develop an in-AST tagging mechanism that designates AST nodes generated from macro expansions or guarded by conditional compilations, and embeds metadata inside the C source without auxiliary sidecar files. These tags preserve semantics and survive translation, allowing the Rust output to reconstruct its correspondence with macro expansions and conditional compilations.
- **Evaluation.** We implement the full system, InariRoll, as a wrapper over C2Rust and evaluate it on real-world codebases. The results show how the taxonomy, symbolic execution, and tagging components work together to preserve macro and configuration structure.

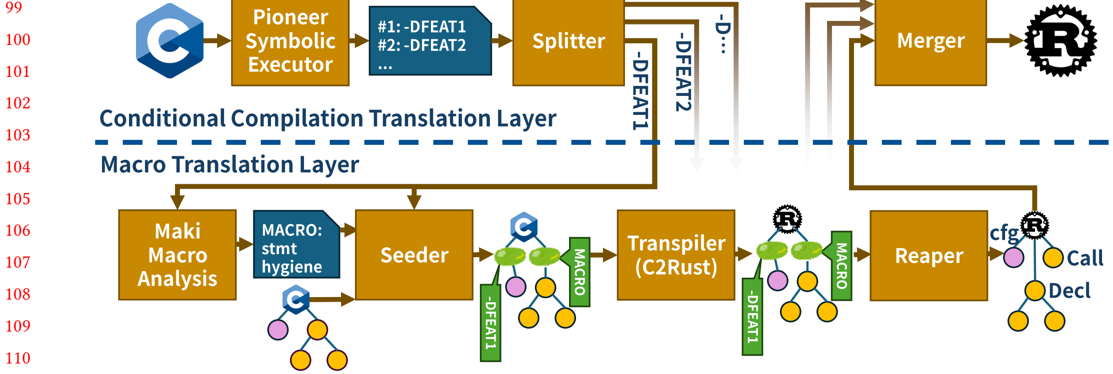

Fig. 2: InariRoll Pipeline. The pipeline consists of two layers. The conditional compilation translation layer (Pioneer and Splitter) symbolically interprets preprocessor directives and partitions the program into configuration-specific translation tasks. The macro translation layer (Maki, Seeder, C2Rust, and Reaper) processes each task independently to translate macros under a fixed configuration. Merger then combines all translated variants into a unified, configurable Rust program.

2 Design

2.1 Overview

Design challenges and motivation. C preprocessor features introduce three fundamental challenges for a preprocessor-aware C-to-Rust translator. First, **conditional compilation** is non-trivial to resolve: `#if`/`#ifdef` expressions may involve nested Boolean and arithmetic conditions, redefinitions, and mutually dependent flags, so identifying the activation condition of each line cannot be done by simple inspection. A detailed explanation with examples can be found at Section 3.1.1. Second, **C macro semantics** operate at the string and character level rather than at the AST level where existing AST-based translators work; determining whether a macro is syntactical, what syntactic kind it expands to, and whether it uses features such as token pasting or stringification requires a separate analysis. Third, reconstructing macro structure in Rust requires knowing which output nodes correspond to which input expansions, yet current translators (including C2Rust) expose no standard **source correspondence** interface. Together, these challenges motivate our modular wrapper design: instead of modifying the underlying translator, InariRoll resolves conditional compilation, analyzes macro semantics, tags the C AST, and later recovers these tags from the Rust AST, enabling macro-aware translation while treating the translator itself as a black box. Many existing efforts in C-to-Rust translation also operate as wrappers or post-processors of existing tools such as C2Rust to improve specific aspects [7, 26, 32, 51]. By adopting the same non-intrusive wrapping principle, our architecture naturally fits into this ecosystem.

Design goals. InariRoll's goal is twofold. First, it translates a *subset of macro invocations* into Rust constructs and translates conditional compilations into configuration (`#[cfg]`) in the output. Which exact subset is translatable depends on what C macro features are used, and that taxonomy is introduced in Section 2.4. Second, it achieves *minimum invasiveness* of the ongoing macro-agnostic transpiler: InariRoll requires no stable API or fixed internal structure from the downstream tool, only high-level concepts. This design keeps InariRoll compatible with a wider range of translators and resilient to their version changes.

Assumptions about the underlying transpiler. InariRoll makes no API-level assumptions. Its single structural assumption is about *seeded* AST nodes (lightweight tags that InariRoll inserts to mark code originating from macro expansions or from conditional compilation ranges). For

148 any seeded node, the downstream translator preserves the node's subtree boundary: it does not
149 arbitrarily reorder code across that boundary (e.g., hoisting code out of the subtree or sinking
150 unrelated code into it). In other words, each seeded subtree in the C AST remains a corresponding,
151 self-contained subtree after translation. This assumption is sufficient for InariRoll to derive source
152 correspondence and to rebuild conditional structure in Rust.

153 **Design layers.** InariRoll consists of two layers: (1) an outer layer for **conditional compilation**
154 **translation**, and (2) an inner layer for **macro translation**, including macro analysis, instrumenta-
155 tion, translation, and reconstruction. Wrapped inside these two layers is a preprocessor-agnostic
156 C-to-Rust translator, such as C2Rust. Together, these layers preserve the configurability and ab-
157 straction of C code without modifying the underlying translator. An overview of the full pipeline is
158 shown in Figure 2.
159

160 ## 2.2 Conditional Compilation Translation Layer (Pioneer, Splitter, Merger)

161 The outer layer of the wrapper handles conditional compilation in C. It performs reasoning over
162 preprocessor conditionals and issues multiple macro translation runs to cover all conditional
163 compilation ranges. This layer consists of three components: **Pioneer**, which analyzes conditions
164 symbolically; **Splitter**, which partitions translation tasks based on these conditions; and **Merger**,
165 which integrates the results back into a single configurable Rust output.

166 **Pioneer** performs symbolic execution over preprocessor conditionals (`#if`, `#ifdef`, etc.) to
167 compute the enabling condition of each source line in terms of combinations of user-defined
168 preprocessor flags. This analysis captures when each line of code is active, while avoiding the
169 combinatorial explosion of brute-force exploration. We formalize this symbolic execution algorithm
170 later in the paper.

171 Based on the predicates produced by Pioneer, the **Splitter** schedules multiple translation runs of
172 the macro translation layer. Each run uses a concrete assignment of preprocessor flags sufficient to
173 cover a subset of lines, ensuring that all conditional regions are exercised at least once. After all
174 runs complete, the **Merger** combines the results back into a single configurable Rust output that
175 reflects all possible branches.
176

177 ## 2.3 Macro Translation Layer (Maki, Seeder, Reaper)

178
179 The inner layer of the wrapper performs macro-related analysis, instrumentation, and reconstruc-
180 tion. It consists of three main components that operate in sequence: **Maki**, **Seeder**, and **Reaper**.

181 **Maki** [36] is an existing tool. It analyzes macros at the string level. Maki extracts properties
182 of each macro expansion that affect its translatability: whether the macro is syntactical (aligned
183 with an AST subtree), its kind (expression, statement, declaration, or type), and whether it involves
184 token pasting, stringification, or hygiene violations. These descriptors are then used to guide
185 reconstruction in the final step. We will introduce in detail the taxonomy of C macros in section 2.4.

186 The **Seeder** instruments the C source with lightweight in-AST tags that record their macro
187 or conditional compilation origins. After translation, the **Reaper** reads these tags from the Rust
188 output and rebuilds macro definitions or conditional guards as Rust functions, macros, or `#[cfg]`
189 attributes. This mechanism provides a practical form of source correspondence without requiring
190 changes to the translator.
191

192 ## 2.4 C Macro Taxonomy

193 C macros exhibit a wide range of syntactic and semantic behaviors. Some expand to well-formed
194 AST units, while others span partial fragments or use preprocessor operators such as pasting or
195 stringification. These properties directly determine how a macro can be translated: syntactical and
196

197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245

Table 1: Macro attributes affecting translation to Rust.

Attribute	Description	Impact on Translation
`syntactical`	Whether the expansion exactly aligns with a single AST node. Non-syntactical expansions cross syntactic boundaries or form only partial fragments.	Must be true to translate.
`astKind` ∈ { Expr, Stmt, Decl, Type }	AST category of the expansion when syntactical.	Must be Expr / Stmt / Decl to translate due to Seeder's limitations.
`hygienic`	The macro does not capture or reference local variables outside its lexical scope.	Must be hygienic to translate.
`isICE`	Indicates whether the expansion occurs in an integer-constant context (e.g., array size or enum value).	Does not translate if true due to Seeder's limitations.
`usesPasting`	Uses token concatenation (##, e.g., `token##suffix`).	Does not translate if true due to Seeder's limitations.
`usesStringification`	Uses stringification (#x).	Does not translate if true due to Seeder's limitations.
`hasFunctionPointer`	The macro expands to a function name or a function pointer.	Does not translate if true due to Seeder's limitations.
`argumentHasSideEffect`	Any macro argument contains side effects such as `i++`, which may execute multiple times.	Does not block translation, but prevents function form.
`hasControlFlowJumps`	Expansion contains control-flow jump statements: `return`, `break`, `continue`, or `goto`.	Does not block translation, but prevents function form.
`anyArgumentNotExpr`	At least one argument is not an expression (e.g., a statement block).	Does not block translation, but prevents function form.

hygienic expression/statement macros may become Rust functions or macros; others such as non-syntactical expansions or those relying on pasting/stringification are left expanded. Because this forms a hierarchy of translatability, we classify macros by the attributes that influence translation outcomes. The attribute set is adapted from prior macro feature taxonomies [36], but we retain only the subset relevant to our system. See Table 1 for a full list of C macro attributes used by our system, and Table 2 for a full list of translatability requirements.

The untranslatable categories in Table 1 are due to different reasons. The requirement of being `syntactical` is inherent: when a macro expansion does not align with any AST node, there is no intuitive 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 Figure 3, only the outer flag `LIBMCS_FPU_DAZ`

Table 2: Translation levels and required conditions.

Translation Level	Required Conditions	Outcome
Translate as Expanded Form	Does not satisfy any of the following translation levels.	The macro is passed directly to the downstream C-to-Rust translator without being tagged by InariRoll.
Translate to Rust Macro	$\texttt{syntactical} \land \texttt{hygienic} \land (\texttt{astKind} \in \{\text{Expr}, \text{Stmt}, \text{Decl}\}) \land \neg\texttt{isICE} \land \neg\texttt{usesPasting} \land \neg\texttt{usesStringification} \land \neg\texttt{hasFunctionPointer}$. but not meeting the next-level conditions.	Reconstructed as a Rust `macro_rules!`.
Translate to Rust Function	Satisfies all conditions of the previous level and additionally: $(\texttt{astKind} \in \{\text{Expr}, \text{Stmt}\}) \land \neg\texttt{argumentHasSideEffect} \land \neg\texttt{hasControlFlowJumps} \land \neg\texttt{anyArgumentNotExpr}$.	Translated into a Rust function definition.

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
```

should be controlled by the user; defining the internal `__LIBMCS_FPU_DAZ` directly 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 issue many unnecessary error 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 must be analyzed, many of which are redundant or illegal. Pioneer avoids this explosion by computing logical premises once per code region instead of reprocessing every configuration.

3.1.2 Input and Output. Pioneer’s input is an unpreprocessed C compilation unit that retains all preprocessor directives. It parses the source file into a *preprocessor-directive-based* AST, rather than a C AST. Each directive such as `#if`, `#define`, or `#error` becomes a distinct, structured AST node, and any non-preprocessor tokens are represented as `c_tokens` nodes. This perspective allows Pioneer to symbolically reason about the control flow of conditional compilation itself.

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

Fig. 5: An example input for Pioneer.

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

Fig. 6: The preprocessor-level AST used by Pioneer.

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

Pioneer outputs a line-to-premise mapping $\texttt{ProgramLine} \mapsto \texttt{Premise}$ where each `Premise` is a Boolean-arithmetic combined expression over the definedness and value of preprocessor define 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 a `ForbiddenSet` which is a formula of preprocessor defines that are explicitly disallowed by preprocessor error directives.

3.1.3 Core Representation. Pioneer performs symbolic execution over the preprocessor AST to explore all possible behaviors under different user-provided macro definitions. 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 evaluates to `true` if the macro is defined by the user and to `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 InariRoll 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.

Pioneer's algorithm explores all execution paths of the preprocessor AST under these symbolic assignments. Each node in this exploration corresponds to one possible logical state of the preprocessing process. To represent these states systematically, Pioneer models every execution state as a triple $\texttt{State} = (\texttt{ProgramPoint}, \texttt{SymbolTable}, \texttt{Premise})$ where:

- `ProgramPoint`: the current position in the preprocessor AST.
- `SymbolTable`: the current macro definition environment, represented as a mapping from macro names to their definition values. 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 it is 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 (Figure 7). Each state $S = (n, s, p)$. For each program point n, the transition function $F(n)$ generates successor states. The transition function $F(n)$ is defined in Table 3.

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 selectively executes external macros concretely based on user-defined scope control. When a source file lies outside a specified working directory, its macros are treated as

Fig. 7: Pioneer symbolic executor algorithm.

```
Input:
   astRoot : ProgramPoint

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

Output:
   (lineToPremise, forbiddenSet)
```

Table 3: Transition rules for preprocessor directives.

Directive	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.
`#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.
`#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.
`#elif cond`	Equivalent to a subsequent `#if cond` branch following the preceding `#if`.
`c_tokens`	Successor $(\text{next}(n), s, p)$.
`EOF`	No successor.

concrete values determined by the current build environment. This bounds the symbolic search space to project-relevant configurations. In addition, a *whitelist mode* allows users 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 analysis would be prohibitive.

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 existing flags without redefining new ones.

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

Fig. 8: Splitter algorithm for generating configuration sets.

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

Algorithm:
   tasklist : List<(ProgramLine, Premise, Int)> := sortDescendingBy(nestingDepth, lineToPremise)
   splits : List<Configuration> := {}
   while tasklist is not empty do
      (l, p, d) := tasklist.pop_front() # deepest and earliest line
      model := solve(p && !forbiddenSet) # call SAT/SMT solver
      if model is UNSAT then
         continue # this line is unreachable
      splits.append(model)
      # remove all lines already covered by this model
      newTasklist : List<(ProgramLine, Premise, Int)> := []
      for each (l2, p2, d2) in tasklist do
         if not isTautology(evalWithModel(model, p2)) then
            newTasklist.append((l2, p2, d2))
      tasklist := newTasklist

Output:
   splits
```

Purpose. The Splitter partitions the configurable program translation into several configuration-specific macro translation tasksİts 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 with regards to the number of `#if` directives in the source code.

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

Algorithm. As shown in Figure 8, the algorithm maintains a worklist of all unvisited lines. At each iteration it selects the deepest and most constrained line, obtains a model for its premise using an external SAT/SMT solver, and records the resulting assignment as a new configuration. It then removes all other lines already satisfied by this model, repeating the process until all satisfiable lines are covered.

Discussion. A natural question is why Splitter does not attempt to cover as many lines as possible into 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 large C codebases. Some of these exclusions are explicitly guarded by `#error` directives as in the case of Figure 4, but many others are unguarded or merely documented in natural language. Those would not trigger preprocessor errors but would cause later translation errors. For example, two alternative function implementations with the same signature put in different `#if` blocks. By exploring one constrained premise at a time, Splitter minimizes the possibility of encountering these hidden conflicts.

3.3 Maki Macro Semantics Analyzer

Maki [36] is an existing static analysis tool for C macros. InariRoll extends and integrates it.

Purpose. Maki analyzes each individual macro invocation and reports a set of its properties. InariRoll keeps the subset that affect its translatability into Rust, as introduced in Section 2.4. In

442 addition, our extended version of Maki also analyzes each conditional compilation region in the
443 source code (that is, each range from `#if` to the matching `#else` or `#endif`).
444 **Input and output.** The input to Maki is a single, unpreprocessed C compilation unit with a
445 specific configuration. For every macro expansion, Maki produces a descriptor containing the
446 attributes defined in Section 2.4. For each conditional compilation region, only a simplified subset
447 of attributes is recorded: whether the block is `syntactical` and, if so, what its `ASTKind` is. These
448 descriptors are later consumed by downstream components such as the Seeder and Reaper to guide
449 in-AST code region tagging and Rust function/macro reconstruction.
450

3.4 Seeder
451

452 *3.4.1 Role in the Pipeline.* Seeder runs after macro and conditional compilation analyses and
453 immediately before the C-to-Rust translation. It instruments the original C source so that the
454 tags survive translation and can later be consumed by Reaper/Merger. This solves the source
455 correspondence problem: even when a macro expands into a large Rust fragment, the resulting
456 Rust still carries enough structure to trace it back to the originating C site. The instrumented AST
457 nodes are called *seeds*.
458

459 *3.4.2 Design Goals.* **(1) In-AST tagging without behavior change.** Seeds are inserted into the C
460 source (not a sidecar file) and preserve observable behavior. **(2) Carry serialized semantic info.**
461 Each seed embeds a string that can encode the macro or conditional range's properties. **(3) Reaper-**
462 **and Merger-friendly shapes.** Seeds compile to characteristic code shapes in Rust, making them
463 easy to locate and align across splits.

464 *3.4.3 Input and Output.* The inputs are a single, unpreprocessed C compilation unit; Maki's
465 descriptors for macro invocations and for conditional compilation regions; and Pioneer's line-to-
466 premise mapping. The output is an instrumented C source that contains seeds.
467

468 *3.4.4 What to Seed.* **Macro expansion sites.** Every macro expansion that satisfies translatability
469 requirements is seeded. **Arguments of function-like macros.** Each actual argument node is
470 seeded so Reaper can later reconstruct macro arguments correctly. **Conditional-compilation**
471 **ranges.** For each region from `#if`/`#ifdef` to the matching `#else`/`#endif`, Seeder inserts seeds
472 that designate the region. Both active and inactive branches receive seeds, so Merger can align
473 corresponding regions across translation splits without performing unification.

474 *3.4.5 Information Embedded.* For **macros**: the properties relevant to translation from our C Macro
475 Taxonomy (Section 2.4), plus source locations (definition and expansion). For **conditional compi-**
476 **lation ranges**: AST kind (`Stmt`/`Expr`/`Decl`), source locations for the guarded range, and the premise
477 (symbolic condition) derived from Pioneer (which `-D` combinations make the region active).
478

479 *3.4.6 Seeding Strategies.* Different categories of AST nodes require different seeding strategies,
480 yet all of them are designed to satisfy the same three goals introduced earlier. Seeder handles three
481 classes of seeds: statement, expression, and declaration.
482 **Statement seeding.** For statements, Seeder uses a *sandwich* pattern: two no-op string tags
483 inserted immediately before and after the original statement (Figure 12). This form trivially satisfies
484 the three design goals — it does not alter control flow, allows embedding arbitrary metadata strings,
485 and produces a clear begin-end block that persists after translation. A minor corner case arises
486 when the statement immediately follows an `if` or `else` without braces; in such cases, Seeder wraps
487 the statement in braces to maintain proper scoping.
488 **Expression seeding (rvalue).** In C, every expression is either an lvalue or an rvalue. An lvalue
489 denotes a storage location and can appear on the left-hand side of an assignment; an rvalue denotes
490

Fig. 9: Declaration seeding.

```c
const char * TAG_FOR_<MACRO_NAME> = TAG;
```

Fig. 10: Expression seeding for rvalues.

```c
(
   (*TAG) ?
   (EXPR) :
   (*(__typeof__(EXPR)*)(0))
)
```

Fig. 11: Expression seeding for lvalues.

```c
(
   *(
      (*TAG) ?
      (&(EXPR)) :
      ((__typeof__(EXPR)*)(0))
   )
)
```

Fig. 12: Statement seeding.

```c
*TAG_BEGIN;
ORIGINAL_STATEMENTS;
*TAG_END;
```

Fig. 13: Input C program with macro decl/stmt/-expr and a conditional region.

```c
#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

int main() {
   #ifdef FEAT1
   ++a;
   #endif
   STMT_MACRO_INCR(a);
   return EXPR_MACRO_ADD(a, 5);
}
```

a computed value and cannot be assigned to. This distinction is important for Seeder as the seeding transformation must preserve both value and assignability. We therefore employ separate strategies for rvalues and lvalues.

Rvalue expressions are wrapped in a conditional expression whose guard is the tag string (Figure 10). Because the tag 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. This design meets all three goals: semantics are preserved, the tag string can carry arbitrary information, and the generated Rust code exhibits a Rust if-expression shape that Reaper can easily recognize.

Expression seeding (lvalue). Lvalues must remain assignable after seeding. Our approach leverages a simple lemma: every lvalue can be addressed, and dereferencing its address again yields an assignable object. Seeder therefore reuses the rvalue seeding pattern, but wraps the original expression inside an address–dereference pair (Figure 11). 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 (C2Rust) reorders top-level declarations during translation. In such cases, the relative position between a declaration and its surrounding tags would be lost. To handle this, Seeder employs a fallback strategy: for each declaration, it emits a companion declaration whose value is the tag string. C2Rust (when turning on this option) attaches a `#[c2rust::src_loc = "line:col"]` attribute to every top-level item, allowing the Reaper to locate the designated Rust declaration.

3.4.7 Example Walk-through. Figure 13 shows a C program that contains macro-based declarations, statements, expressions, and a conditional compilation region. After Seeder inserts tags as described above, the instrumented C source in Figure 14 preserves identical semantics while embedding all macro and conditional boundaries as serialized strings. Finally, the resulting Rust output in Figure 15 demonstrates that these seeds survive the translation, providing the structural anchors used later by Reaper and Merger to reconstruct the original preprocessor-level structures.

Fig. 14: Seeded C after applying Seeder.

```
#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 * TAG_FOR_DECL_MACRO_INT = "{ ... \"astKind\":\"Decl\", ... ,
     \"cuLnColBegin\":\"7:1\", ... }"; // Tag for DECL_MACRO_INT

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

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

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

3.4.8 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 tagging mechanism cannot tag 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 entirely 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 tagging mechanism altogether.

3.5 Reaper

Goal. Reaper performs the inverse process of Seeder. It reads the Rust files produced by the translator and reconstructs macro and conditional compilation from the in-AST tags left by Seeder. For macros, Reaper groups tagged regions into macro or function definitions; for conditional compilation, it attaches `#[cfg]` attributes to the corresponding code blocks. At this stage, each Reaper instance operates on a single split of the program, so conditional regions contain at most one active branch. Reaper does not yet merge alternative branches; the tags remain in place for the Merger to combine multiple splits later.

589
590 **Fig. 15: Resulting Rust after translation (characteristic shapes 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 TAG_FOR_DECL_MACRO_INT: *const libc::c_char =
    b"{ ... \"astKind\":\"Decl\", ... , \"cuLnColBegin\":\"7:1\", ... }\0" as *const u8
        as *const libc::c_char; // Tag 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 tag for conditional compilation
    a += 1;
    *(b"{ ... \"astKind\":\"Stmt\", \"begin\":false, ... }\0" as *const u8 as *const libc::c_char); //
         End tag for conditional compilation

    *(b"{ ... \"astKind\":\"Stmt\", \"begin\":true, ... }\0" as *const u8 as *const libc::c_char); //
         Begin tag 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 // Tag for argument 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 tag 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 // Tag 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 // Tag for argument 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 // Tag for argument 2
        { 5 as libc::c_int } else { *(0 as *mut libc::c_int) })
    } else { *(0 as *mut libc::c_int) };
}
```

614
615
616
617 **Input and output.** The input is the Rust output from the underlying C-to-Rust translator. Reaper
618 requires no sidecar metadata file: all reconstruction information is encoded within the translated
619 source. The output is a Rust file where (1) C macros are reconstructed into either Rust macros or
620 Rust functions, and (2) conditional code regions are annotated with `#[cfg]` guards.
621 **Algorithm.** Most of Reaper's logic is straightforward: it scans the translated AST, identifies
622 seeds, and rolls the seeded regions back into higher-level constructs. The reconstructed example in
623 Figure 16 shows the result for the same input used by Seeder.
624 **Macro definition consistency.** A key challenge arises in ensuring that multiple invocations of
625 the same C macro definition can be reconstructed into a single Rust definition. In principle, one C
626 macro definition may correspond to several expansion sites. However, after translation, the resulting
627 Rust constructs are not always structurally identical, breaking the one-to-many correspondence
628 between definition and invocations.
629 We observed two primary causes of this inconsistency:
630 *(1) Type divergence.* C macros often rely on implicit type promotion or ad-hoc polymorphism.
631 Two invocations expand to syntactically identical C code, but after translation, Rust's stricter typing
632 rules can produce extra explicit type conversions. For example, if `EXPR_MACRO_ADD` is invoked once
633 with `int` arguments and once with `long long` arguments, C's integer promotion rules would allow
634 both cases, but the translated Rust code will be attached an extra explicit type casting on only one
635 of the version. In such cases, Reaper separates these invocations and reconstructs multiple Rust
636 definitions, one for each distinct parameter type combination.
637

638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659

Fig. 16: Reaper output for the running example after reconstruction.

```
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 {
   // Tags for conditional compilation are kept for Merger to handle
   *(b"{ ... \"astKind\":\"Stmt\", \"begin\":true, ... }\0"
      as *const u8 as *const libc::c_char); // Begin tag
   #[cfg(feature = "FEAT1")]
   (a += 1);
   *(b"{ ... \"astKind\":\"Stmt\", \"begin\":false, ... }\0"
      as *const u8 as *const libc::c_char); // End tag

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

660 661 662 663 664 665 666

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)
   };
```

667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686

(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 when 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 compile time 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 Figure 17 for an example. Here, the guard is embedded within the expression's conditional structure. Though by definition being runtime-evaluated, `cfg!()` can still possibly be optimized away during compile time.

3.6 Merger

Goal. Merger forms the final stage of the InariRoll pipeline. It receives the individually translated Rust outputs produced under different configuration combinations by the inner macro translation layer and combines them into a single configurable Rust program.

Input and output. Each merge operation takes two translated 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 is straightforward. It 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 existing version. 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 Figure 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.

4 Implementation

InariRoll is implemented as a mixed-language toolchain. The C-side components (Pioneer, Splitter, Seeder, and the 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`) to parse preprocessor ASTs 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] The Reaper and Merger use the rust-analyzer (2024-12-16) [23] library for parsing and AST transformation. We plan to open-source the entire toolchain.

5 Evaluation

5.1 Methodology

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

To evaluate **correctness**, we perform testing. We ran each translated Rust program against the C tests. This is possible because C2Rust (and thus InariRoll) outputs Rust code that is ABI-compatible with the original C program. For object programs that are configurable via conditional compilation, we run the tests once under each set of preprocessor define combination that Splitter emits. All object programs pass all tests except for a special case to be 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 InariRoll does not leave it expanded

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

Outcome	All	Syntactical	Expr.	Stmt.	Decl.	Type	Non-syn.
Total macros	1407	1142	416	605	32	89	265
Successfully translated	697 (50%)	697 (61%)	351 (84%)	346 (57%)	0 (0%)	0 (0%)	0 (0%)
→ Rust function	425 (61%)	425 (61%)	326 (93%)	99 (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	710 (50%)	445 (39%)	65 (16%)	259 (43%)	32 (100%)	89 (6%)	265 (100%)

as-is. We also report the distributions of reasons why InariRoll leaves some macro invocations expanded as-is according to our macro taxonomy.

To evaluate **performance**, we report the per-thread time spent on each component of the InariRoll pipeline 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 evaluate InariRoll 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).

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 outcome 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 `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. The library includes 190 C files (9,748 lines of code) and 18 C header files (1,373 lines of code). This benchmark evaluates InariRoll '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 InariRoll 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 InariRoll 's ability to reconstruct macros and Pioneer's selective symbolic execution capability under challenging realistic software sizes.

785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833

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

Outcome	All	Syntactical	Expr.	Stmt.	Decl.	Type	Non-syn.
Total macros	611	604	223	381	0	0	7
Successfully translated	584 (96%)	584 (97%)	203 (91%)	381 (100%)	0 (0%)	0 (0%)	0 (0%)
→ Rust function	323 (55%)	323 (55%)	141 (69%)	182 (48%)	0 (0%)	0 (0%)	0 (0%)
→ Rust macro	261 (45%)	261 (45%)	62 (31%)	199 (52%)	0 (0%)	0 (0%)	0 (0%)
Left expanded	27 (4%)	20 (3%)	20 (9%)	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%)

Table 7: Comparison of macro translation failing reasons across benchmarks. Percentages are relative to all kept-as-is macros in each benchmark.

Failing reason	CRUST-Bench	LibmCS	zlib
Non-syntactical	265 (37%)	7 (26%)	840 (76%)
Argument non-syntactical	262 (37%)	5 (19%)	120 (11%)
Uses stringification	140 (20%)	0 (0%)	0 (0%)
Unhygienic	131 (18%)	17 (63%)	164 (15%)
Requires integral constant expression	126 (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)	57 (5%)	1199 (28%)	451 (20%)
Splitter	95 (9%)	411 (10%)	91 (4%)
Maki (C macro analysis)	653 (62%)	1591 (37%)	1371 (62%)
Seeder	4 (0%)	36 (1%)	8 (0%)
C2Rust	59 (6%)	272 (6%)	79 (4%)
Reaper	94 (9%)	299 (7%)	113 (5%)
Merger	86 (8%)	455 (11%)	110 (5%)
Aggregate	**1048 (100%)**	**4264 (100%)**	**2223 (100%)**

5.3 Results

Correctness. All translated programs produced by InariRoll pass their corresponding test suites under every preprocessor-define combination emitted by Splitter except for 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 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. Table 4, Table 5 and Table 6 summarize macro translation outcomes across syntactic categories. Overall, InariRoll achieves a high success rate on syntactical macros. Across all three benchmarks, syntactical macros account for 2,836 out of 3,948 total macros (72%), and among them, 2,101 (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 purely 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 percentage numbers 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. InariRoll 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, the Maki macro analysis dominates the runtime, accounting for 37–62% of total execution time. The overall overhead compared to a plain C2Rust translation is roughly 18×, but this cost is a one-time translation effort.

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 (20%).

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 InariRoll maintains practical translation performance while scaling to real-world configurable software.

6 Limitations and Threats to Validity

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 3.4.8 for detail). 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 handled 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 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.

Threats to validity. Although InariRoll is designed to be independent of any particular C-to-Rust translator, the current prototype is implemented specifically with C2Rust. These may have made assumptions about stable AST structures and preserved expression and statement boundaries through translation. Other translators, especially LLM-based or aggressively optimizing ones, may not preserve these properties, in which case the Seeder's tags may fail to align with the translated Rust code. A source correspondence interface provided by the underlying translator would free InariRoll from the dependency on the 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, InariRoll 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 concurrently seeking publication and addressed to the topic. 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 InariRoll. In a "work in progress" report, DeGreef et al. [4] discuss the handling of Expr macros, addressing a subset of the considerations handled in InariRoll.

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 InariRoll 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 in criterion.

8 Conclusion

We introduced InariRoll, 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 macro taxonomy and tag-based reconstruction pipeline, InariRoll 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, InariRoll demonstrates that decoupled preprocessor analysis can make C-to-Rust translation both macro-aware and practical.

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

981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029

[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.

1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078

[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