



Lifting the Preprocessor with Oxidize: Structure-Preserving C-to-Rust Translation

Robbe De Greef
Vrije Universiteit Brussel
Brussels, Belgium
robbe.de.greef@vub.be

Théo Engels
Royal Military Academy
Brussels, Belgium
theo.engels@mil.be

Felix Van den Broucke
Vrije Universiteit Brussel
Brussels, Belgium
felix.van.den.broucke@vub.be

Ken Hasselmann
Royal Military Academy
Brussels, Belgium
ken.hasselmann@mil.be

Antonio Paolillo
Vrije Universiteit Brussel
Brussels, Belgium
antonio.paolillo@vub.be

Abstract

Migrating legacy C code to Rust is increasingly desirable, yet existing C-to-Rust translators preprocess macros, conditional compilation, and includes away before translation, producing flat and repetitive Rust output. Such output cannot serve as a *migration target*: a translated codebase that a development team can adopt as the new source of truth.

We present Oxidize, a structure-preserving C-to-Rust translator with macro-retention capabilities. Oxidize introduces *macro contraction* and *invoke-expand*, two new algorithms that lift textual preprocessor constructs into the translator's intermediate representation rather than expanding them away. Object- and function-like C macros are retained as Rust macros, with hygiene checks and per-call-site argument types handling Rust's scoping and typing.

Across nine real-world C codebases of varied size, Oxidize preserves substantially more macros than C2Rust and rust-bindgen, while remaining competitive on per-program execution-success rates across single-file translatability benchmarks.

CCS Concepts: • Software and its engineering → Incremental compilers; Preprocessors; Source code generation.

Keywords: Translation, Source-to-source, Transpilation, C, Rust, Preprocessor, Macros, Abstract Syntax Tree, Metaprogramming, Structure preservation

ACM Reference Format:

Robbe De Greef, Théo Engels, Felix Van den Broucke, Ken Hasselmann, and Antonio Paolillo. 2027. Lifting the Preprocessor with

Oxidize: Structure-Preserving C-to-Rust Translation. In *22nd European Conference on Computer Systems (EuroSys '27)*, April 19–23, 2027, Rabat, Morocco. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3842654.3848533>

1 Introduction

C remains the foundation of modern systems software, powering operating systems, device drivers, databases, and embedded platforms. Unfortunately, C's lack of memory safety is a dominant source of vulnerabilities: Microsoft and Google both report that about 70% of their severe security issues arise from memory-unsafe code [9, 18, 37]. The growing industry consensus—from recent White House statements [33] to the integration of Rust into the Linux kernel [21]—underscores the urgency of migrating away from C.

Rust is an attractive target language: it combines strong safety guarantees with high performance and a growing ecosystem. Rewriting millions of lines of legacy C by hand is infeasible, so automated C-to-Rust translation is appealing, and several translators (also called source-to-source compilers or transpilers) have emerged: e.g., C2Rust, CRUST, Corrode, and Citrus. The critical obstacle they all face is the C preprocessor. Macros, conditional compilation, and `#include` encode developer-defined abstractions, build configurations, and module boundaries—in short, the structure of the program as humans maintain it. Existing translators preprocess this structure away before translation, replacing macro invocations with their textual expansions and collapsing all configurations into a single build variant. The result is flat and repetitive Rust output that is difficult to read, debug, or evolve.

Migration targets need structure. We refer to the form of the translated code that a development team adopts as the new source of truth as a *migration target*. For Rust output to serve as a migration target, it must be (i) maintainable without referring back to the original C, (ii) readable enough to support direct code review and bug reports, and (iii) structured so that subsequent edits compose with existing patterns rather than fight them. Producing migration-grade



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

EuroSys '27, Rabat, Morocco

© 2027 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2971-3/2027/04

<https://doi.org/10.1145/3842654.3848533>

output therefore requires *structure preservation*: keeping the developer-defined abstractions visible in the translated code, instead of erasing them at the preprocessing step. This paper focuses on macros as one of these abstractions; conditional compilation and `#include` structure are addressed by related [24] and future work.

Listing 1 illustrates the point on a real example from the DOOM source code. The macro `logical_gamemission` encodes the relationship between the active build target and the displayed game; it is invoked at thirteen call sites across six source files. C2Rust, like other existing translators, inlines the macro at every call site, scattering the abstraction throughout the codebase; rust-bindgen does not emit a Rust binding because the macro body is not a plain constant expression. The result is thirteen identical ternaries to maintain, no longer linked to a single named definition. Preserving the macro—the focus of this paper—keeps the abstraction intact at a single, named site.

This paper. We present Oxidize, a structure-preserving C-to-Rust translator. At its core is an intermediate representation that retains preprocessor-level macro constructs throughout translation, together with two new algorithms—*macro contraction* and *invoke-expand*—that systematically translate object- and function-like C macros into Rust macros, even in complex codebases. We call this *lifting* the preprocessor: macro contraction and invoke-expand recover the textual expansions performed by the preprocessor as named, structured macro invocations in the IR, raising them from the token level to a level where they can be analyzed, typed, and translated. The resulting Rust code is semantically equivalent to the original C while remaining readable and maintainable. A prototype of Oxidize appeared as an earlier WIP paper [3]; this paper presents the complete macro-translation design and a full-scale evaluation.

We evaluate Oxidize along two axes: (1) *Macro preservation across real codebases*. We run Oxidize on the original build systems of nine real-world C codebases of varied size, including *sl*, *lolcat*, *nyancat*, *smaz*, *FIGlet*, *bzip2*, *zlib*, *libpng*, and *DOOM*; we compare per-file translation and macro retention against C2Rust and rust-bindgen: at equal file coverage with C2Rust (129 of 146 files), Oxidize retains 98% of macro definitions against 52%, and it is the only tool in this comparison that preserves function-like macros. (2) *Robustness across diverse program inputs*. On handwritten unit tests, LLM-generated programs, CodeNet samples, Csmith, and a Csmith extension we built (*Macsmith*) to emit macro-rich programs, Oxidize achieves end-to-end translation success competitive with the strongest rule-based baseline (96% against 99%).

Contributions. This paper provides two contributions:

- **Macro-preserving translation algorithms:** *macro contraction* at each invocation site, *invoke-expand* driven by a macro dependency graph for nested macro definitions,

```
// doomstat.h:69 (DOOM, github.com/ozkl/doomgeneric)
#define logical_gamemission
    (gamemission == pack_chex ? doom :
     gamemission == pack_hacx ? doom2 : gamemission)
// ... used at 13 call sites across 6 files,
// e.g. d_main.c:758:
if (logical_gamemission == doom) { /* ... */ }
```

- (a) Original C: the macro names a per-build-target abstraction used widely across the codebase.

```
// every call site becomes a fresh inlined ternary:
if (if gamemission == pack_chex { doom }
    else if gamemission == pack_hacx { doom2 }
    else { gamemission })
    == doom { /* ... */ }
```

- (b) C2Rust: the macro is inlined at every call site; the developer-named abstraction is gone, and the same logic is duplicated wherever the macro was used.

```
macro_rules! logical_gamemission { () => {
    (if gamemission == pack_chex { doom }
     else if gamemission == pack_hacx { doom2 }
     else { gamemission })
}; }
// ...
if logical_gamemission!() == doom { /* ... */ }
```

- (c) Oxidize: the macro definition becomes a Rust macro; each of the thirteen call sites preserves the original abstraction.

Listing 1. Why structure preservation matters for migration. The macro `logical_gamemission` from the DOOM source code encodes the relationship between the active build target and the displayed game; it is invoked at thirteen call sites across six source files. C2Rust translates the macro definition by inlining it at every call site; rust-bindgen drops the macro entirely from its output, because its body is not a pure constant. Oxidize preserves the macro as a Rust macro, so the abstraction remains a single editable definition.

and helper-trait *macro typing* that defers C’s implicit conversions to Rust’s trait resolver, hosted in a structure-preserving IR that retains developer macro abstractions through translation. Concurrent work by Peng et al. [24] achieves a related goal through a complementary architecture (a wrapper around an unmodified C-to-Rust translator); we discuss the differentiation in §2.5.

- **An extensive empirical evaluation:** nine real-world codebases evaluated on their original build systems, a per-class macro breakdown against rule-based translators including rust-bindgen, and an adversarial fuzzing study using Csmith and Macsmith.

2 Related Work

The idea of automating the migration of C codebases into Rust has gained traction in both academia and industry,

Table 1. Oxidize features compared with the related work. $\times$ = not supported, $+$ = planned, $\sim$ = not fully supported, $\checkmark$ = supported. *rust-bindgen* generates Rust FFI bindings rather than Rust translations of the C source; included here for the macro-handling comparison.

	Project	Features				
		C	C++	Macros		Comments
				Obj.	Fn.	
Rule-based	Corrode	$\checkmark$	$\times$	$\times$	$\times$	$\times$
	Citrus	$\sim$	$\times$	$\times$	$\times$	$\times$
	CRUST	$\sim$	$\sim$	$\times$	$\times$	$\checkmark$
	C2Rust	$\checkmark$	$\times$	$\checkmark$	$+$	$\times$
	Hayroll [24]	$\checkmark$	$\times$	$\checkmark$	$\sim$	$\times$
	Oxidize	$\checkmark$	$+$	$\checkmark$	$\checkmark$	$+$
	Cpp2Rust [25]	$\times$	$\checkmark$	$\times$	$\times$	$\times$
	bindgen [38]	$\checkmark$	$\sim$	$\sim$	$\times$	$\sim$
LLM	C2SaferRust	$\checkmark$	$\times$	$\times$	$\times$	$\times$
	Sactor	$\checkmark$	$\times$	$\times$	$\times$	$\times$

driven by the potential to eliminate memory-unsafe constructs while retaining performance. A recent CACM survey by Hong and Ryu [11] states the practical problem: existing C-to-Rust translators are useful, but their output is both *unsafe* (it relies on Rust features the compiler cannot verify) and *unidiomatic* (it does not exploit Rust’s high-level constructs). Two main approaches have emerged: rule-based translators, which rely on static analysis and syntactic rewriting, and LLM-based translators, which leverage large language models to generate idiomatic Rust. Both lose the preprocessor, for different reasons: rule-based tools translate the preprocessed source, so macros, conditional compilation, and source inclusion disappear before translation begins, while LLM-based tools do read the original source but inline or rewrite its macros, with no guarantee that an abstraction survives. Macro preservation, the focus of this paper, attacks the unidiomaticity gap at its root: expanding a macro turns one named abstraction into code repeated at every call site, which no Rust programmer would write, whereas Rust’s own macro system can host that abstraction directly. Earlier work on *macro processing* for source-to-source translation provides important antecedents, which we revisit in §2.4.

2.1 Rule-Based Translators

Early tools such as Corrode [30], Citrus [16], and CRUST [31] demonstrated that syntactic C-to-Rust translation is feasible, but all remain incomplete and unmaintained. C2Rust [12] is the most mature and widely used; it produces largely unsafe Rust, with only partial support for object-like macros and none for function-like macros. As summarized in Table 1, none of these systems systematically preserve macros, making their output difficult to maintain.

2.2 LLM-Based Translators

Recent work has turned to LLM-assisted translation, including EvoC2Rust [39], GenC2Rust [40], C2RustTV [43], RustMap [2], C2SaferRust [19], and Sactor [44]. These systems aim to produce idiomatic or safe Rust by combining LLMs with static analysis, skeleton generation, or test suites. While promising, LLM-based translators are inherently non-deterministic and slower than rule-based tools. They incur a per-program API cost that rule-based pipelines do not, making them expensive by comparison. Their evaluations target different properties: compilation and test-pass rates at module granularity, idiomaticity, or less unsafe code than C2Rust emits. None reports macro preservation, the property §5.1 measures; our own §5.2 evaluation is single-file.

Industrial investment is following the same direction: Canonical and UKRI fund a multi-year University of Bristol project on a neurosymbolic C-to-Rust platform that combines LLM translation with fuzzing and formal-equivalence validation [29]; that group has studied LLM translation to Rust with differential fuzzing [5]. The project’s announcement casts translation that preserves C’s structure “too literally” as the obstacle to idiomatic output. “Structure”, however, names two different things. C’s low-level idioms should indeed give way to idiomatic Rust over time, and such refinement could operate on preserved structure (§6). Developer-defined abstractions, macros first among them, are the structure worth keeping: an abstraction expanded away is lost to every later pass, idiomatic or not (§1).

Our evaluation includes the two strongest models available at the time of writing, invoked through an agentic compile-and-fix loop (Claude Opus 4.7, OpenAI GPT-5.3 Codex), and the C2SaferRust refinement pipeline, to control the question “what does an LLM alone produce, with and without rule-based scaffolding?” Configuration is in §5.2.

2.3 Benchmarks and Datasets

Several recent efforts have proposed benchmarks to evaluate C-to-Rust translators. CRUST-Bench [14] contains 100 C repositories paired with manually written safe-Rust interfaces and test cases. C2Rust-Bench [32] distills 2,905 representative functions from a pool of 15,503. Large-scale datasets such as Project CodeNet [26] also provide diverse C programs for evaluation. We use Project CodeNet among our single-file datasets, and complement these benchmarks with generator-produced programs from Csmith and our macro-focused extension of it, which stress macro-heavy and corner-case constructs (§5.2). All of them are single-file by construction; our macro-preservation study instead runs on whole codebases with their original build systems (§5.1).

2.4 Macro Support in Source-to-Source Translation

The challenge of translating preprocessor macros while preserving program structure has been studied in earlier

source-to-source translation work. Most directly relevant is the *Rosetta Translator* of Andrews, Del Vigna, and Molloy [1], which translates pTAL—a proprietary block-structured language derived from Algol-60—to C++ while preserving macros, conditional compilation regions, and source inclusion. Rosetta introduces two notions central to its approach: a *virtual source* (the token stream the parser sees, after pre-processing) and a *fragment tree* (which records every macro expansion, parameter substitution, and source inclusion). The two trees are “joined at the tokens”: leaves of the syntax tree are also leaves of the fragment tree. Macro bodies are translated *at their use sites* rather than at their definition sites, since only at use can sufficient context be recovered for parsing and translation; the source generator then re-assembles the translated fragments into target text files.

Our *macro contraction* algorithm (§4.1) shares this insight—identify expansion sites in the post-preprocessing AST, then re-attach them to their definitions—but operates in a markedly harder setting. Rosetta’s source and target both use *unhygienic, text-based* preprocessor macros, so a translated macro body inherits the same identifier-capturing semantics as the original. Oxidize translates into Rust’s *hygienic macro_rules!* system, which forbids accidental capture, requires explicit type handling for type-generic patterns, and constrains where macro invocations may appear. The hygiene checks (§4), helper-trait macro typing, and cast macro are responses to that paradigm shift, and have no direct analogue in Rosetta.

Beyond translation, prior research has studied the C preprocessor through the lens of *analysis* and *refactoring*, e.g., by modeling preprocessor decision graphs, inferring safe refactorings, or parsing the unexpanded program [7, 8, 17, 20]. These approaches keep the preprocessor in place rather than translating its constructs into higher-level language features.

2.5 Concurrent Work: Hayroll

Concurrent with our work, Peng et al. [24] introduce *Hayroll*, a modular wrapper that adds macro and conditional-compilation awareness to an existing C-to-Rust translator (currently C2Rust) without modifying it. Hayroll has two layers: a symbolic executor (*Pioneer*) that derives per-line preprocessor activation conditions and translates `#if` and `#ifdef` directives into Rust `#[cfg]` attributes, and a string-level macro analyzer combined with a tag-based seed/harvest mechanism (*Seeder/Reaper*) that survives translation through the underlying tool, so that translated macro regions can be recovered as Rust `macro_rules!` or `fn` items. On three benchmarks (CRUST-Bench, LibmCS, zlib; 62.8K LoC), Hayroll reports 74% syntactical-macro retention, with a 20× translation runtime overhead over C2Rust alone [24, §5].

Hayroll’s contributions and ours are largely complementary, with two notable differences. *First*, Hayroll handles conditional compilation natively; Oxidize currently does not, though we discuss this as an extension path (§6). *Second*,

two of our techniques have no analogue in Hayroll: our *invoke-expand* algorithm (§4.2) classifies and resolves nested macro definitions, a case Hayroll explicitly does not handle [24, §6.1]; and our *macro typing* infrastructure (§4: helper traits, type parameters, and the cast macro) addresses C’s type-generic patterns (e.g., ADD macros that work on both integer and pointer operands), for which Hayroll emits one definition per type combination and Oxidize a single one.

2.6 Concurrent Work: Cpp2Rust

Also concurrent with our work, Popescu et al. [25] present *Cpp2Rust*, a translator from a subset of C++ to memory-safe Rust. The two systems address different source languages but share the strategic question of how to relate structure preservation to safety, and take opposite design points. Cpp2Rust parses C++ with libclang and emits Rust that uses a runtime library: every variable is wrapped in `Rc<RefCell<T>>`, and C++ pointers are translated to a smart pointer type that encodes offset arithmetic and aliasing through reference counting. Ownership and mutability checks that Rust would normally enforce at compile time are thereby deferred to runtime. An iterative source-to-source optimizer removes those wrappers where its intra-procedural analysis can prove they are unnecessary. The reported supported subset is sizeable but not comprehensive: full templates via instantiation, classes with default copy/move constructors, inheritance from pure-virtual bases, and a curated set of STL containers are handled, but exceptions, user-defined copy/move constructors, multi-threaded code, multiple inheritance, and several language idioms (unions, bitfields, `goto`, `volatile`, `dynamic_cast`) are excluded [25, §4]. On two real applications (WOFF2 and Brunsli; 13K LoC), the resulting safe Rust ranges from 2% slower than the original C++ to 6× slower, with a single unsafe block in the runtime library.

Cpp2Rust and Oxidize are complementary in two senses. First, their coverage differs: Cpp2Rust supports C++ features that Oxidize does not, while Oxidize preserves C macros, which Cpp2Rust, like other translators, expands away. Second, the two approaches embody opposite answers to the safety question: Cpp2Rust achieves safety up front, at the cost of runtime checks that its optimizer then removes where it can prove them unnecessary; Oxidize preserves the input’s structure so that later refinement passes can recover ownership, aliasing, and bounds information statically (§6).

3 Background

3.1 C vs. Rust in brief

C remains the lingua franca of systems software, but its permissive memory model—manual allocation, unchecked pointer arithmetic, implicit conversions, and pervasive undefined behavior—is a primary source of vulnerabilities and maintenance risk. In contrast, Rust enforces memory safety at compile time through *ownership*, *borrowing*, and *lifetimes*;

references are statically checked by the borrow checker, eliminating broad classes of memory errors while retaining performance comparable to C.

This semantic gap matters for automated migration. Syntactic translation is the easy part; reconciling language *semantics* is not. Commonly used C constructs such as implicit conversions, raw-pointer arithmetic, and unstructured control flow must be mapped to explicit and verifiably safe Rust constructs. Even seemingly benign differences—e.g., C’s integer promotions and “usual arithmetic conversions” that coerce operands toward a common type [13, §6.3], also documented in the GNU C reference manual [36]—can change program behavior if translated naively. Any practical C-to-Rust tool must therefore (1) preserve intent and structure for human maintainers, and (2) expose C’s implicit behaviors (e.g., promotions, macro expansions) as explicit constructs in the Rust output.

3.2 The C Preprocessor and Macros

A central obstacle to maintainable C-to-Rust translation is the C preprocessor. The C standard defines a multi-phase translation pipeline in which the first four phases are purely preprocessing: trigraph replacement, line splicing, tokenization (including comment removal), and directive handling (macro expansion, inclusion, and conditional compilation) [13, §5.1.1.2]. Critically, this process is *purely textual* and runs *before* C compilation, allowing macro invocations to appear anywhere in the source file without regard to C’s syntax.

Preprocessor directives. The main #-directives [13, §6.10] are:

Macro definitions. Object-like macros bind an identifier to a replacement token sequence; function-like macros do the same but may contain macro arguments that are expanded in the body [13, §6.10.3]. Both expand textually [34].

Conditional compilation. `#if/#elif/#else` evaluate constant expressions, including with the defined operator (and shorthands `#ifdef/#ifndef`) to test macro presence [13, §6.10.1]. Using macros in the condition of these constructs can further enhance their expressiveness.

Inclusion. `#include` copies header contents into the current translation unit and is widely used for modularization.

Undefinition. `#undef` removes a binding; later redefinitions are legal and affect later expansions [13, §6.10.3.4].

Expansion semantics. The preprocessor performs a left-to-right scan over preprocessing tokens, expanding identifiers bound as macros; the currently expanding macro is *disabled* to prevent infinite recursion, then re-enabled after its body is fully processed [13, §6.10.3.4]. Arguments to function-like macros are parsed as balanced token sequences (tracking nested parentheses) and substituted into the replacement list. Because expansion is textual and context-free w.r.t. C’s AST, macros can be invoked in positions that would be ill-formed

in C prior to expansion (e.g., substituting punctuators or partial expressions). This enables powerful metaprogramming but discards structural information a translator might rely on.

Hygiene. Because the C preprocessor expansion is purely textual, it does not preserve lexical scoping. Macros may introduce identifiers that shadow or capture user-defined names, leading to subtle bugs. Rust macros are *hygienic* by construction, preventing such capture; this semantic gap must be handled during translation.

Why this breaks translators. Most C-to-Rust translators invoke the preprocessor first and feed only the *expanded* C program to their front end. This simplifies parsing but discards critical structure: *constants* become inlined literals, obscuring intent; *generic patterns* encoded as function-like macros turn into ad hoc code clones; *configuration logic* (`#if` trees) collapses into a single build variant, hindering portability; *name- and scope-related tricks* (stringification `#`, token-pasting `##`) lose their provenance, complicating faithful translation. The result is “flat” Rust output that is hard to read, reason about, or evolve: precisely what migration projects cannot afford.

3.3 Rust Macros

Rust also supports macros, but they are *structured*, *hygienic*, and integrated into the language. Declarative `macro_rules!` macros match over parsed token trees, expand within the AST after parsing, and are validated by fragment specifiers (e.g., `$e:expr`) [28]. Hygiene prevents accidental capture and ensures identifiers refer to their intended bindings. Procedural macros are written as ordinary Rust functions inside dedicated `proc-macro` crates: they receive a stream of input tokens and emit a stream of output tokens, letting arbitrary Rust code run at compile time for transformations beyond what `macro_rules!` pattern matching can express (e.g., custom `#[derive]` implementations). Unlike the preprocessor, Rust macro invocations are allowed only in specific syntactic positions; expansion is staged (a worklist queues nested invocations), and there is no disabled-macro context akin to the preprocessor because recursion is controlled by expansion depth.

3.4 Implications for C-to-Rust Translation

Two implications follow:

1. Macro preservation is essential for maintainability. For Rust output to serve as a *migration target* (cf. §1)—a translated codebase that a development team adopts as the new source of truth—it must satisfy three properties: (i) maintainability without referring back to the original C; (ii) readability sufficient to support code review and bug reports; and (iii) structural compatibility with subsequent edits, so that future patches compose with existing patterns rather than fight them. A translator that preprocesses macros away erases design intent and configuration structure, producing

flat output that fails (i)–(iii). A macro-aware pipeline that *retains* macros at the Rust level (where possible) keeps those abstractions available in the output: a necessary condition for maintainability, readability, and structural compatibility (see §6).

2. Implicit C semantics must be made explicit. Pre-processor choices, integer promotions, and arithmetic conversions [13, §6.3] should become explicit Rust constructs (types, casts, or cfg-gated items) to avoid semantic drift.

Why the preprocessor is hard to “refactor away”. Historically, the C preprocessor was introduced after C’s initial design, modeled on assembler macro facilities and implemented as a separate tool [27], which explains its loose integration and text-level operation.

Prior research has mostly targeted *analysis* and *refactoring* with the preprocessor in place rather than translating its constructs into higher-level language features (see §2.4 for details). As a result, existing C-to-Rust tools typically side-step macro preservation entirely—the core gap this paper addresses.

3.5 What Oxidize Translates: Macro Classes

The C preprocessor’s textual operation means a macro body may contain arbitrary token sequences—not necessarily valid C expressions or statements—and a macro invocation may appear in any source context. To enable translation rather than blind expansion, Oxidize commits to a precise scope.

Accepted: expression-like macros. Oxidize targets *expression-like macros*: macros whose body, after resolution of any nested macros (§4.2), parses as a valid C expression. This subset spans both object-like macros (e.g., constants and conditional ternaries such as `logical_gamemission` from Listing 1) and function-like macros (e.g., `MAX(a, b)` or `ADD(x, y)`). A macro invocation is translated only when its invocation site also occurs in a valid C expression context; the macro-contraction algorithm of §4.1 identifies the qualifying invocation sites.

Conservatively expanded: everything else. Macros falling outside this subset are *conservatively expanded*: their text is inlined exactly as the C preprocessor would, just as in existing translators. Specifically, Oxidize expands rather than translates macros that are (i) empty, (ii) statement-like, (iii) unparseable as a single expression after `invoke-expand`, (iv) only invoked in non-expression contexts (e.g., as a partial syntactic fragment such as Listing 3), (v) failing hygiene checks (§4), (vi) type-defining (`#define TYPE int`), or (vii) using stringification (`#`) or token-pasting (`##`). This guarantees that Oxidize *never fails because of an unsupported macro*: the worst case for an unsupported macro is the same flat output prior tools produce.

Empirical distribution. Expression-like macros dominate in practice. Two representative codebases illustrate the trend: FIGlet (68 of 86 macros, 79%) and DOOM (889 of 999, 89%) classify as expression-like under our libclang-based analyzer;

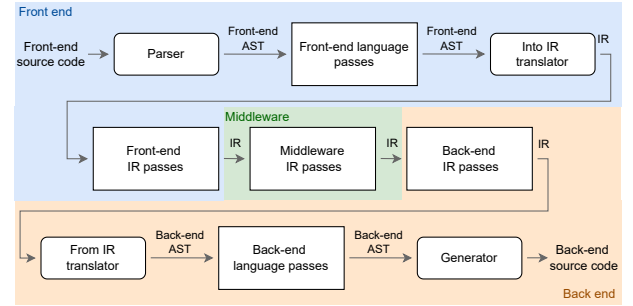


Figure 1. Architecture of Oxidize. Unlike typical translators that map C directly to Rust, Oxidize adopts a compiler-style three-phase design. The **front end** (blue) parses C into an annotated IR, the **middleware** (green) applies macro-aware passes and future safety analyses, and the **back end** (orange) generates readable Rust code.

the remainder are empty, unparseable, or statement-like. Our broader evaluation across nine real-world codebases (§5.1) confirms this pattern and breaks the accepted subset down further—constant-valued vs. non-constant object macros, and function-like macros by typing pattern—to expose where Oxidize provides value over `rust-bindgen`, which translates only the constant-valued slice.

Extension paths. The macro-contraction approach extends naturally to two classes that are presently conservatively expanded. *Type macros* can be supported by applying contraction to types in addition to expressions. *Statement macros* in the common `do { ... } while(0)` idiom can be supported by recognizing the wrapper and translating the body as a Rust block expression. We discuss both in §6.

4 Design of Oxidize & Macro Translation

Architecture in brief. Oxidize is a three-phase translator (front, middle, back end; Figure 1) operating over an explicit, AST-like intermediate representation (IR) that mirrors a Rust AST but carries C-specific nodes (raw pointer arithmetic, unstructured control flow, preprocessor traces) and macro-related annotations. The front end parses C with `libclang`, then runs the four incremental passes presented below (contraction, `invoke-expand`, hygiene, typing) to lift macro structure into the IR. Further front-end passes restructure unstructured control flow (`goto`, `fallthrough` `switch`) with a `Stackifier`-style algorithm [15, 42], rewriting irreducible loops into a dispatch loop; translate variadic function definitions to Rust’s C-variadic functions; and handle function pointers, unions, and anonymous records. Bitfields remain unsupported (§6). The middle end is currently empty; the back end serialises the IR to Rust via the `Ruast` library. The architectural choice driving this paper is the explicit IR: it splits macro recognition and translation into small, separable passes rather than a monolithic source-to-source rewrite.

<pre>#include <ctype.h> int foo() { return isspace('a'); }</pre> (a) Original	<pre>// [...] the expansion of <ctype.h> int foo() { return ((*__ctype_b_loc())[__int]((__int)('a')) & (unsigned short int) _ISspace); }</pre> (b) Preprocessed
--	--

Listing 2. C program testing if the character 'a' is a space.

```
1 #define MINUS_ONE -1
2 int foo() {
3   // expands to 'return 1 -1;':
4   return 1 MINUS_ONE;
5 }
```

Listing 3. Syntactically correct C program using an expression-like macro in a non-expression context.

Implementation details (pass scheduling, IR node taxonomy, back-end emission) appear in our technical report [4].

C programs rely heavily on the preprocessor: an empirical study of 1.4 MLOC across 26 packages found that about 25% of all lines expand a macro [6]. Macros fill long-standing gaps in the language. Object-like macros serve as compile-time constants without creating linker symbols, avoiding errors in header files where `const` is unsafe. Function-like macros provide type-generic code in the absence of templates; since C11, they are also used with `_Generic` to emulate overloading.

Most translators sidestep this complexity by invoking the preprocessor upfront [12, README.md]. While this simplifies parsing, it destroys macro structure. Listing 2 illustrates the effect: a call to the standard `isspace` macro expands into low-level code that is correct but nearly unreadable. Such output undermines the goal of producing maintainable Rust. `isspace` is used here for illustration; Oxidize’s scope is to preserve developer-defined macros and their use sites in the migration target. Replacing C standard-library macros and functions with their idiomatic Rust counterparts (e.g., `isspace` → `char::is_ascii_whitespace`) is a separate, complementary goal we revisit in §6.

Oxidize instead translates macros directly into Rust constructs. By retaining their intent rather than their expansion, Oxidize produces code that remains both semantically faithful and comprehensible.

As established in §3.5, Oxidize accepts *expression-like macros*: macros whose body parses as a valid C expression after invoke-expand resolution and whose invocation sites occur in expression contexts. All other macros are conservatively expanded. The remainder of this section presents the algorithms that recognize and translate this subset.

4.1 Macro-Contraction Algorithm

Restricting to expression-like macros addresses many cases, but not all. Two challenges remain: (1) expression-like macros may still be invoked in *non-expression* contexts,

producing syntactically invalid Rust; and (2) invoking the preprocessor discards macro structure.

Listing 3 illustrates the first issue. The macro `MINUS_ONE` expands to `-1`, a valid C expression on its own. But the invocation `return 1 MINUS_ONE;` places the macro immediately after a literal: the C preprocessor tolerates this, yet the resulting token stream is not a valid expression context before expansion. Classification must therefore be *per invocation site*, not *per definition*: the `MINUS_ONE` definition is accepted (body parses as an expression) and Oxidize emits a Rust `macro_rules!` for it; the line-4 invocation is nonetheless conservatively expanded because its post-preprocessing source range fuses with the adjacent literal into a single `1 -1` subtraction node in the C AST, with no expression-context node covering the expansion. Two invocations of the same definition can thus receive different verdicts, driven by surrounding AST shape. To address these challenges we introduce the *macro-contraction algorithm*. The key idea is to preprocess the file as usual, but to contract back expression-like macros into explicit invocations in the AST. This enables us to (i) reuse mature preprocessor implementations while (ii) preserving only macro invocations in valid expression contexts.

The algorithm proceeds as follows:

- Preprocess the file with a macro-aware frontend (e.g., libclang), recording each macro definition and every expansion site with its start/end offsets. All candidate macros are placed in a *valid macro list*.
- Parse each macro body as a C expression. If parsing fails, remove it from the valid list.
- Parse the preprocessed file into a C AST.
- Traverse the AST. For each expression node, check whether its source range matches a recorded expansion. If so, and if the corresponding macro is in the valid list, replace the expansion with an explicit macro invocation. Otherwise, leave the expansion as-is.

This process contracts away spurious expansions while retaining macros that are both syntactically valid and semantically recoverable, laying the groundwork for further passes such as dependency analysis (§4.2).

4.2 Invoke-Expand Algorithm

While the macro-contraction algorithm retains correct macro invocations, it can misclassify certain macros as non-expression-like because their bodies contain other macros that have not yet been expanded. Listing 4 illustrates this case: the macro `FOO` is not a valid expression unless the nested macro `PLUS` is expanded first.

To resolve such cases, we introduce the *invoke-expand algorithm*. The key idea is to reason about nested macro usage through a *Macro Dependency Graph* (MDG). The MDG is a directed acyclic graph $G = (V, E)$ where each node $v \in V$ corresponds to a macro definition, and each edge $(u, v) \in E$ indicates that the body of u contains an invocation of v .

```

1 #define THREE 3
2 #define PLUS +
3 #define BAZ THREE
4 #define FOO(a) a PLUS BAZ
5 int main() {
6     return FOO(2);
7 }

```

Listing 4. Example C program with nested macros. `FOO` depends on `PLUS` and `BAZ`; at first sight `FOO` is not expression-like, but after expanding `PLUS` it becomes one.

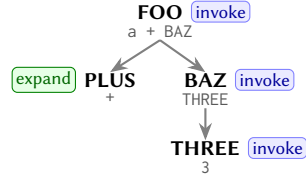


Figure 2. Macro Dependency Graph (MDG) for Listing 4. The invoke-expand algorithm classifies `PLUS` as *expand* and `FOO`, `BAZ`, `THREE` as *invoke*, yielding a valid expression-like macro.

Cyclic dependencies are not permitted in the MDG. If recursive macro expansion is detected, the corresponding macro invocation is conservatively classified as *expand*. This restriction is necessary because the recursion semantics of C preprocessor macros do not directly correspond to Rust macro expansion semantics.

The algorithm processes each macro definition as follows.

1. Graph construction. The source file is scanned and every macro definition is added as a node. If the body of a macro *M* references another macro *N*, we add an edge (*M*, *N*) to the MDG. A fresh MDG is built for every C translation unit.

2. Depth-first traversal. Every macro definition serves in turn as a starting point, since the MDG generally has several roots; results are memoised, so each macro is visited once. The traversal is depth-first, and each macro *M* is classified in post-order, after all macros it references. We attempt to parse its body as a C expression. If successful, *M* is tagged as *invoke* (to be preserved as a macro invocation). If parsing fails, *M* is tagged as *expand* (to be replaced by its fully expanded body).

3. Inductive handling. Because *M* is classified only after its dependencies, the tags are assigned in a topological order of the MDG, and the classification of *M* builds inductively on the decisions made for its dependencies.

The result is that only the minimal set of macros needed to yield valid expression-like macros are expanded. In the example of Listing 4, `PLUS` is classified as *expand*, while `FOO` becomes *invoke*, yielding a valid expression at its call site: Oxidize emits `macro_rules! FOO { ($a:expr) => { $a + BAZ!() }; }` (simplified; the emitted definition also carries the type parameters of §4.4). Figure 2 shows the corresponding MDG, with nodes annotated by their final classification.

4.3 Macro Hygiene

Because macro invocations are purely text-based copy-and-paste operations, they lack any notion of scoping or hygiene. This makes them susceptible to shadowing and other hygiene

```

1 #define STRCPY(x,y) strcpy((x),(y))
2
3 int main() {
4     int (*strcpy)(char*, char*) = otherfunc;
5     char a[] = "Hello";
6     char b[] = "abc";
7     STRCPY(a, b); // Expands to strcpy((a), (b));
8     // No safe Rust equivalent.
9     // Would resolve to local variable, not the libc strcpy.
10 }

```

Listing 5. Example of an unhygienic C macro that cannot be safely translated to Rust because of identifier shadowing.

issues. Rust macros, in contrast, are hygienic by design, so this semantic gap must be addressed during translation.

Listing 5 shows a concrete case. Here, the macro `STRCPY` expands to a call to `strcpy`, but within `main` the identifier `strcpy` has been shadowed by a local function pointer. The expansion therefore resolves to the *local* identifier rather than the standard library function, silently changing the semantics of the code. Such captures occur in practice: Linux v2.6.32 defined `VMALLOC_END` on ia64 as an alias for the global `vmalloc_end`, and `mm/vmalloc.c` later declared a local `const unsigned long vmalloc_end` initialised from `VMALLOC_END`, so after expansion the variable was initialised from itself; the kernel’s fix was to drop the macro [10].

To ensure correctness, Oxidize enforces a hygiene discipline on macros: all identifiers used in a macro body must either be macro parameters or globally defined symbols that are not shadowed at invocation sites. If this condition is violated, the invocation is replaced by its preprocessor expansion, kept from the front end, and a *translation warning* is issued. This prevents unsafe shadowing from leaking into the Rust code.

While such macros are legal in C, they are generally considered a *code smell* since their semantics depend on fragile interactions with the surrounding scope. By flagging and expanding them, Oxidize preserves semantics while signaling potential maintainability issues to the developer. Expansion at this stage remains possible despite having already passed the contraction phase, because it is performed directly on the parsed AST: the macro invocation is simply replaced by the macro body’s AST.

4.4 Macro Typing

Macros lack any information on the types of their arguments, since they are merely token sequences without any associated type data. This limitation is not unique to C—all syntactic macro systems (including Rust’s) expand purely on tokens. In C, however, the effect is magnified by the language’s implicit conversions: integer promotions, usual arithmetic conversions, and pointer arithmetic all change semantics depending on operand types. In regular code, a translator can analyze types and insert explicit casts to preserve behavior. Inside macros, this context is missing, so naive translation can yield compilation errors or diverging semantics.

<pre>#define ASN(x,y) \ x = y void foo() { int x; char y = 12; ASN(x, y); }</pre> (a) C	<pre>macro_rules! ASN { (\$x:expr,\$y:expr)=> { \$x = \$y } } fn foo() { let mut x: i32; let mut y: i8 = 12; ASN!(x, y); }</pre> (b) Naive Rust (broken)	<pre>macro_rules! ASN { (\$x:expr,\$y:expr, \$ty:ty)=> { \$x = \$y as \$ty } } fn foo() { let mut x: i32; let mut y: i8 = 12; ASN!(x, y, i32); }</pre> (c) Corrected Rust
---	---	---

Listing 6. Assignment macro: C source (a), naive Rust translation that breaks because the implicit char-to-i32 coercion is not preserved (b), and the corrected Rust that adds an explicit as cast through a typing parameter (c).

```
1 #define ADD(x, y) x + y
2 void foo() {
3     int x = ADD(1, 2);
4     int* y = ADD(&x, 1);
5 }
```

Listing 7. Example of a C program using an addition operator in a macro resulting in both an addition and a pointer arithmetic operation being performed in said macro.

To bridge this gap, Oxidize employs three complementary strategies: *macro type parameters*, *helper traits*, and a generic *cast macro*. Together, they ensure that expression-like macros remain both correct and idiomatic when translated into Rust. **Macro Type Parameters.** Some macros hide implicit casts. For example, the ASN macro in Listing 6 performs an assignment where C would insert an implicit conversion from char to int. In Rust, the naive translation fails because no implicit cast is available. The key observation is that while we cannot infer the exact type in advance, we can detect that a cast *must* occur. Oxidize therefore extends the macro with an explicit type parameter, inferred at each call site, as shown in Listing 6c. This preserves correctness without losing readability.

Helper Traits. Other macros overload operators in ways that depend on operand types. Consider the ADD macro in Listing 7, used both for integer addition and for pointer arithmetic. Oxidize pre-generates a trait module once per translation: for 16 of C’s binary operators, one generic trait, such as `HelperAdd<T>` (Listing 8), where `T` is the right-hand operand type, `Self` the left-hand one, and the associated `Output` the result type. Each pair of primitive operand types gets an impl applying C’s usual arithmetic conversions [13, §6.3.1.8]; combinations C forbids (floating-point operands of bitwise operators) are pruned. Pointer arithmetic is handled by a similar mechanism in a separate pass. At a call site `ADD(x, y)` Oxidize emits `x.helper_add(&y)` (Listing 9), and Rust’s trait resolver selects the impl matching the operand types, so Oxidize never has to infer them itself.

Cast Macro. Finally, some C casts cannot be mapped directly to Rust’s `as` operator (notably arrays and function pointers,

```
1 trait HelperAdd<T> {
2     type Output;
3     unsafe fn helper_add(&self, rhs: &T) -> Self::Output;
4 }
5 impl HelperAdd::<i32> for i32 {
6     type Output = i32;
7     unsafe fn helper_add(&self, rhs: &i32) -> Self::Output {
8         *self + *rhs
9     }
10 }
11 impl<T> HelperAdd::<i32> for *mut T {
12     type Output = *mut T;
13     unsafe fn helper_add(&self, rhs: &i32) -> Self::Output {
14         unsafe { self.offset(*rhs as isize) }
15     }
16 }
17 unsafe fn foo() {
18     let mut x: i32 = 1.helper_add(&2);
19     let y: *mut i32 = (&raw mut x).helper_add(&1);
20 }
```

Listing 8. Rust helper trait `HelperAdd` emulates C’s overloaded `+` semantics (as used in Listing 7). The same syntactic construct (`helper_add`) performs either integer addition or pointer arithmetic, depending on operand types.

```
1 trait HelperAdd<T> // ...
2 macro_rules! ADD {
3     ($x: expr, $y: expr) => {
4         $x.helper_add(&$y)
5     }
6 }
7 unsafe fn foo() {
8     let mut x: i32 = ADD!(1, 2);
9     let y: *mut i32 = ADD!(&raw mut x, 1);
10 }
```

Listing 9. Translated ADD macro from Listing 7 in Rust. The macro expands to calls to `helper_add`, preserving C’s type-dependent behavior within a single syntactic construct.

```
1 macro_rules! helper_cast {
2     (regular, $lhs: expr, $rhs: ty) => {
3         $lhs as $rhs
4     };
5     (array, $lhs: expr, $rhs: ty) => {
6         $lhs.as_ptr() as $rhs
7     };
8 }
9 // at each invocation the translator fills in the kind:
10 helper_cast!(regular, x, i64)
11 helper_cast!(array, buf, *const u8)
```

Listing 10. Implementation of a (here simplified) generic Rust casting macro `helper_cast`, which abstracts different categories of casts. The `regular` case performs primitive casts directly, while the `array` case lowers an array to a pointer using `as_ptr()` before casting. Additional variants can be added to support other C-to-Rust cast semantics; the last two lines show invocations as the translator emits them.

which require a sequence such as `as_ptr()` + slice coercion + pointer cast). Oxidize provides a generic `helper_cast` macro (Listing 10) that dispatches on an explicit “kind” argument the translator fills in at each invocation.

These three strategies cover the main classes of typing issues in macro translation. Type parameters make implicit

casts explicit, helper traits capture operator overloading idioms, and the cast macro handles C’s corner-case conversions. Combined with macro contraction, invoke-expand, and hygiene, they complete the macro-preservation pipeline in Oxidize, allowing even complex expression-like macros from real-world C codebases to be rendered as correct Rust.

In practice, the four passes above run sequentially per C translation unit: contraction, invoke-expand, hygiene checks, then typing. The result is Rust code faithful to the semantics of the original C while exposing implicit behaviours in a maintainable form.

5 Evaluation

Our evaluation answers two questions. **RQ1:** *Can Oxidize preserve macros across real-world C codebases better than existing translators?* **RQ2:** *How robust is Oxidize compared to prior work across diverse datasets of C programs?* §5.1 addresses RQ1 on macro-heavy real-world applications; §5.2 addresses RQ2 across six single-file datasets and closes with a performance-overhead note.

5.1 Macro Preservation (RQ1)

We evaluate Oxidize’s ability to preserve macros across real-world, macro-heavy codebases. The evaluation answers two complementary questions per codebase: *coverage* (how much of the source each tool turns into compilable Rust) and *retention* (of the macros that fall within scope, how many each tool preserves as explicit Rust constructs). The two questions admit different denominators and therefore separate figures. **Selected programs.** We evaluate nine codebases selected by the following criteria: open-source, pure C, a self-contained build system, and an end-to-end smoke validation for the resulting binaries (in-tree tests where the codebase ships them; a manual run otherwise: for doomgeneric, launching and briefly playing the game). The nine (146 source files, 141 kLoC, see Figure 3 for per-codebase sizes) span interactive utilities (sl, lolcat, nyancat), text processing (FIGlet), compression (zlib, bzip2, smaz), system libraries (libpng), and a full game engine (doomgeneric).

Methodology. For each codebase we run a build interception pass that records every translation unit (TU)’s compilation flags, then invoke each translator on every TU and compile its output with rustc, so that only translations rustc accepts are counted. Macro definitions and call sites are enumerated by walking each TU with libclang into a per-codebase *macro census*. Each translator’s Rust output is parsed with tree-sitter to identify surviving definitions (macro_rules! or pub const items) and surviving invocations.

We compare Oxidize against C2Rust (default and experimental¹) and rust-bindgen. The two C2Rust variants appear alongside Oxidize as the columns in Figure 3; rust-bindgen

¹Both --translate-const-macros and --translate-fn-macros set to experimental.

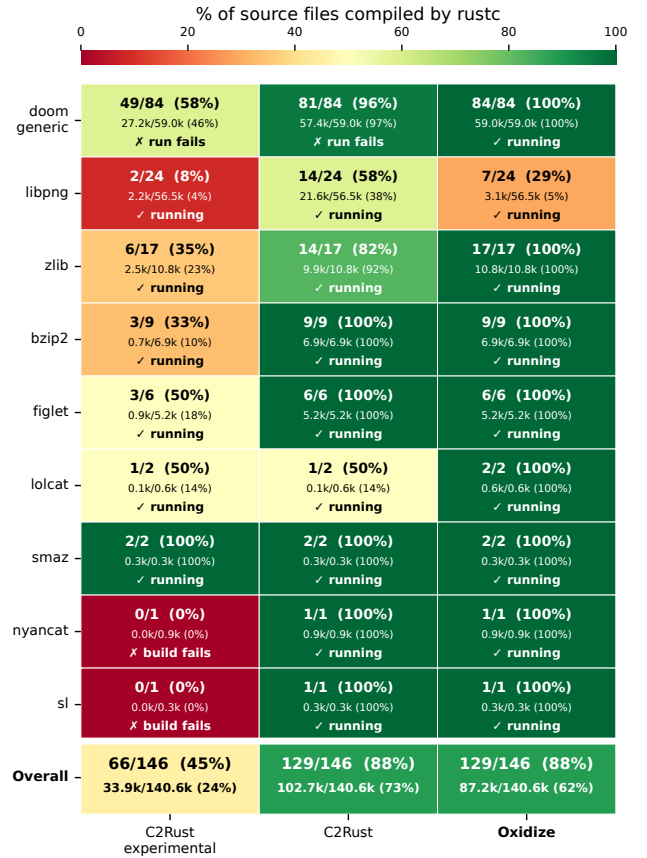


Figure 3. Per-codebase coverage of files accepted by rustc after each tool’s translation. Cells show the number of source files (top, in bold) and source LoC (bottom), both in absolute counts and as a percentage of the codebase total, plus the manual smoke outcome of the hybrid binary (third line).

emits FFI bindings rather than function bodies and therefore joins the comparison only in the retention figure (Figure 4). Hayroll [24] appears in neither figure; we discuss it separately below. LLM-backed tools are not part of RQ1: per-macro retention is ill-defined for output that inlines or rewrites macros freely, and agentic translation of whole codebases remains costly: porting Bun’s 535k lines of Zig to Rust took up to 64 parallel Claude agents 11 days and about \$165,000 in API usage [35], whereas our entire single-file LLM evaluation cost about \$550 of usage credit (§5.2).

Coverage. Figure 3 reports, per codebase, the fraction of source files whose translated Rust is accepted by rustc. Oxidize and C2Rust each translate **129 of 146 source files (88%)** pooled across the nine codebases; C2Rust experimental reaches 66/146 (45%). Oxidize reaches 100% file coverage on eight of the nine codebases (sl, nyancat, lolcat, smaz, FIGlet, zlib, bzip2, and doomgeneric); on doomgeneric alone this represents 84/84 files and 59 kLoC of game-engine source. Its single lower-coverage outlier is libpng at 7/24

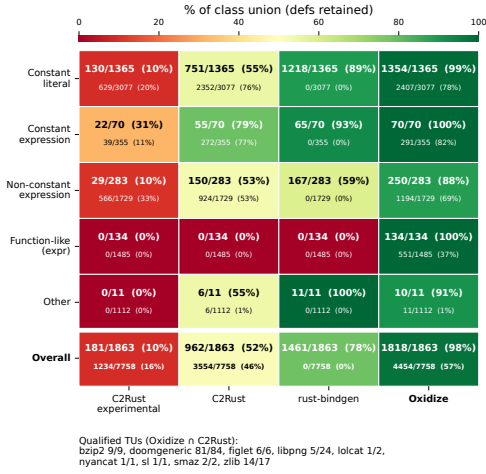


Figure 4. Macro retention on the per-TU intersection where both Oxidize and C2Rust pass rustc. Cells: distinct definitions retained (top, bold) and surviving call-sites (bottom).

(29%), where C2Rust reaches 14/24. Pooled by LoC, C2Rust leads with 73% against Oxidize’s 62%: Oxidize’s remaining failures concentrate on large files (libpng’s 17 failing files average 3.1 kLoC each). Beyond per-file acceptance, each cell’s third line reports a manual end-to-end smoke outcome. For every (codebase, tool) pair we link a *hybrid binary*: translated TUs as Rust objects, untranslated TUs as their original C objects. The tag records whether that binary builds, links, and passes its smoke validation (running), or where it first fails (build / link / run). Oxidize yields a running binary on all nine codebases, C2Rust on eight, and C2Rust experimental on six. The exception is telling: C2Rust translates 81 of doomgeneric’s 84 files and the binary links, but it panics at start-up on an out-of-bounds index in translated code. Compiling in isolation does not imply a runnable binary.

Macro population. The nine codebases collectively define **2862 macro definitions across 14 419 call sites**, including 277 function-like-expression and 95 function-like-statement definitions; four of the nine declare more than 25 function-like macros each (libpng alone declares 190). The full per-codebase, per-class breakdown is in our technical report [4].

Macro retention. Figure 4 reports per-class retention restricted to the per-TU intersection where both Oxidize and C2Rust pass rustc’s compilation (120 files across the nine codebases, covering 1863 distinct macros in the per-class union). The intersection is the only file set on which a per-tool retention comparison is dimensionally fair: tools are scored on the same TUs, so a difference in coverage cannot inflate or depress retention.

The main result is the *function-like-expression* row: Oxidize retains all **134 definitions** and **551 of the class’s 1485 call sites (37%)**; every other tool retains **0** on both axes. This

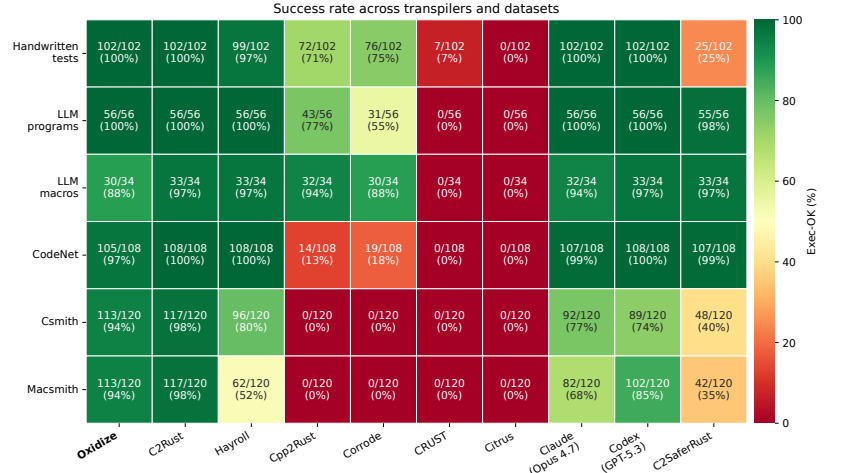


Figure 5. Exec-OK rates: rows are datasets, columns are translation tools (Oxidize and C2Rust first, then the other rule-based tools, then the LLM-backed ones). Cells show *successful runs / total runs (percentage)*.

is the class that motivates the contribution: function-like macros are a routine maintainability construct and no other tool in the figure preserves them at any rate. The two axes use different denominators. Definitions are scored against a per-class *union*, the macros that at least one tool retains, so a macro no translator can keep penalises none of them; call sites are scored against every call site of the class in scope. Two consequences follow. Statement-like macros have no row at all: none of the 67 invoked in the qualified TUs is retained by any tool, Oxidize included, since they fall outside the expression-like subset of §3.5. And within the classes shown, 1 constant-literal, 31 non-constant-expression, 38 function-like-expression and 185 other macros are invoked but retained by nobody: they sit outside the definition denominators, but their call sites count against every tool.

On the object-like classes, the split shows how close rust-bindgen, which can only emit constants, comes to Oxidize. For *constant-literal* macros (1365 in the union), Oxidize retains 1354 (99%) and rust-bindgen 1218 (89%); C2Rust trails at 751 (55%) and C2Rust experimental at 130 (10%). *Constant-expression*—object-like macros whose body is a non-literal constant such as (1 << SHIFT)—saturates at 70/70 for Oxidize versus 65 (93%) for bindgen and 55 (79%) for C2Rust. *Non-constant-expression*—object-like macros whose body names an identifier, a syntactic class that also catches aliases of other constants (cf. Listing 1)—is where the constant-folding tools fall behind: out of 283, Oxidize retains 250 (88%), bindgen 167 (59%), and C2Rust 150 (53%). Oxidize therefore beats bindgen on every constant class while remaining the only tool in Figure 4 that preserves function-like macros.

C2Rust’s lower retention on constant macros is partly an artifact of its translation strategy rather than its inability to

translate such macros. C2Rust only translates macros that are referenced in the code as it requires the usage of the macro to determine the type of the generated constant. As a result, C2Rust does not provide a configuration flag to force the translation of unused macros and scores significantly lower than Oxidize and rust-bindgen.

Pooled across all classes, Oxidize retains 1818/1863 (98%) of the union’s macro definitions, against rust-bindgen’s 1461/1863 (78%), C2Rust’s 962/1863 (52%), and C2Rust experimental’s 181/1863 (10%).

C2Rust experimental: invocations without definitions. The 10% pooled retention for C2Rust experimental is the visible end of a structural failure: the experimental function-macro mode emits macro *invocations* (`NAME! ( . . . )`) at every call site but never emits the corresponding `macro_rules! definitions`, so the output cannot compile. Figure 3 shows the same result on the coverage axis: 66/146 files pass `rustc` under C2Rust experimental against 129/146 for default C2Rust. **Bindgen as a bindings generator.** We include rust-bindgen in the retention comparison as the baseline, but with the caveat that bindgen emits FFI bindings (extern declarations and `pub const` items) rather than implementation code. Bindgen never emits function bodies; for that reason it does not appear in Figure 3. On Figure 4, bindgen trails Oxidize on every constant row and reaches 0% on every function-like row, by design rather than by implementation gap.

Hayroll on whole-codebase inputs. We also ran Hayroll [24] (§2.5) on all nine codebases, with a 15-minute timeout per file. It translates 88 of the 146 files, and 72 of them compile (49%, against 129 (88%) for Oxidize). On `doomgeneric`, 56 of its 84 files compile, fewer than C2Rust alone (81), whose output it refines, and the binary faults at start-up, as C2Rust’s does. On `zlib` and `libpng` it completes 2 of 41 files, and 34 hit the timeout; Hayroll’s authors report this limit and evaluate `zlib` with exploration restricted to five configuration flags [24, §5, §6.1]. Translating the nine codebases took Hayroll at least 11 hours, 9.5 of them in the 38 files cut off at the timeout, against one hour for Oxidize and about a minute for C2Rust. Our retention metric, which counts macro definitions and invocations in the output, finds none in Hayroll’s: Hayroll reconstructs macros as Rust functions specialised per argument type or as `macro_rules!` [24, §2.3.1, §3.5], and none survives as a macro on our codebases. Measured this way, Hayroll trails Oxidize on coverage, retention and translation time. Since Hayroll does not complete 38 files within the timeout and preserves no macro that our metric can count, we report its results here rather than as a column of Figures 3 and 4.

Conclusion. Across the nine codebases, Oxidize and C2Rust translate the same share of files (129/146), and on the 120 files where both succeed Oxidize retains 98% of macro definitions against 52% for C2Rust and 78% for rust-bindgen. It exceeds rust-bindgen on every constant macro class and is the only tool in Figure 4 with non-zero retention on function-like

macros (all 134 function-like-expression definitions and 37% of the class’s call sites). These results answer RQ1: at equal coverage, Oxidize preserves the macro structure central to a maintainable Rust port at substantially higher rates than existing rule-based tools.

5.2 Translatability across Datasets (RQ2)

Macro preservation (RQ1) is central to maintainability, but a translator is only useful if it produces correct, executable code. We therefore assess *translatability*: the ability to translate single-file C programs into Rust programs that compile under `rustc` and reproduce the original program’s output and exit code, measured across diverse datasets.

We evaluate Oxidize alongside nine other translators, grouped left-to-right in Figure 5: the main rule-based baseline C2Rust [12]; the concurrent academic translators Hayroll [24] and Cpp2Rust [25]; the legacy rule-based translators Corrode [30], CRUST [31], and Citrus [16]; and LLM-backed tools (Claude Opus 4.7, Codex GPT-5.3, both invoked through an agentic compile-and-fix loop, and C2SaferRust [19]).

Datasets. We evaluate on six single-file datasets. *Handwritten unit tests* (102 programs) cover core C constructs (structs, enums, pointers, control flow). *LLM-generated programs* (56) add algorithmic tasks. *LLM-generated macros* (34) stress advanced macros (X-macros, variadics). *CodeNet* (108) samples real submissions from the IBM CodeNet collection [26] as a real-world workload. *Csmith* (120) and *Macsmith* (120) are random-program generators tuned to stress different translator surfaces: upstream Csmith [41] (with `goto`, without bitfields) probes control-flow translation; Macsmith extends Csmith with object- and function-like macro emission to probe macro handling on adversarially generated inputs.

Evaluation protocol. For each program we measure three outcomes: (1) *Translate-OK*: the tool emits Rust code; (2) *Compile-OK*: the emitted Rust code compiles under `rustc`; and (3) *Exec-OK*: the resulting binary’s standard-output and exit code match the C reference. Our main metric is *Exec-OK*: a successful *Exec-OK* requires the prior two to succeed. All experiments run on identical hardware, in per-tool Docker environments composed with `pythiner` [22], with fixed flags (`gcc -O3` for the C reference, `cargo build --release` for Rust), and no manual edits are applied to translator outputs.

LLM tool configuration. The LLM-backed tools run under a configuration fixed by our evaluation pipeline: a prompt template (identical for both agentic tools, see our technical report [4]) providing the C source, the reference test cases and the required output path, with the source’s directory linked into the working directory so that headers resolve; pinned model identifiers (Claude Opus 4.7, Codex GPT-5.3); and each tool’s own agentic compile-and-fix loop, at default settings except Codex’s reasoning effort, set to *high*. The 10-minute per-file cap applied to every tool bounds each loop; there is no iteration limit. The prompt templates, per-tool settings, and the pipeline are part of our artifact.

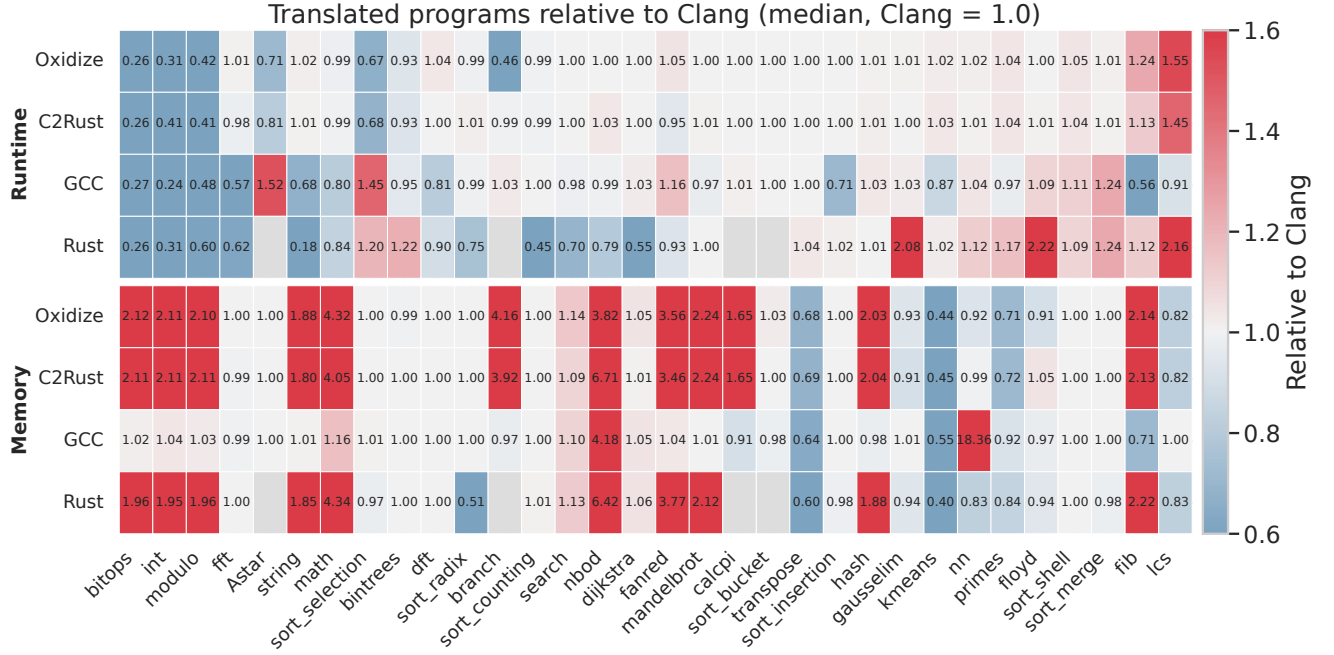


Figure 6. Runtime (top) and memory (bottom) overhead of the translated programs relative to Clang (baseline = 1.0): median over 10 runs per benchmark (columns, ordered by runtime overhead) and configuration (rows); grey cells have no valid measurement. Oxidize closely tracks Clang in runtime, with most benchmarks within 5–10% and no systematic slowdown compared to C2Rust. In memory, Oxidize follows C2Rust; the benchmarks that use more memory than C (e.g., math, mandelbrot, nbod) do so in every Rust configuration, idiomatic Rust included: a language/runtime effect rather than a translation artifact.

Results. Figure 5 reports per-(dataset, tool) Exec-OK rates. Aggregated across the 540 single-file programs in the six datasets, the two main rule-based tools are closely matched: C2Rust passes **533/540 (99%)** and Oxidize passes **519/540 (96%)**. Both reach 100% on Handwritten and the LLM programs, and stay above 90% on the harder datasets (Csmith and Macsmith). Considered alongside Figure 4, this is the central RQ2 finding: Oxidize preserves the macro structure of the input at substantially higher rates than the other rule-based tools, while staying within three points of C2Rust’s Exec-OK. Hayroll reaches 99/102 on Handwritten, 56/56 and 33/34 on the LLM programs and 108/108 on CodeNet (454/540 pooled), but only 62/120 on the macro-dense Macsmith programs: the Rust it generates for reconstructed macros does not always compile (e.g., `#define M_8 (void*)0` becomes a function with no return type). The LLM-backed tools tell a different story. *Claude* and *Codex* achieve perfect or near-perfect Exec-OK on Handwritten, the LLM programs, and CodeNet. On the random-program datasets they degrade more visibly: 92/120 (77%) and 89/120 (74%) on Csmith, 82/120 (68%) and 102/120 (85%) on Macsmith. *C2SaferRust* is near-perfect on the LLM programs and CodeNet but reaches only 25/102 on Handwritten and about 40% on the random sets. *CRUST* and *Citrus* are at or near zero everywhere; *Corrode* holds up on

the small sets but drops to 19/108 on CodeNet and to zero on the random ones. All three are included for completeness.

Cost and operational considerations. Beyond per-program success rates, three operational properties separate LLM-backed translators (Claude, Codex, C2SaferRust) from rule-based ones. *Wall-clock per file:* across the 540 single-file programs, the mainline rule-based tools have median translation times under two seconds (C2Rust 0.37 s, Oxidize 1.45 s) and p_{95} below 30 s. The LLM-backed tools sit one to two orders of magnitude higher: median 18.9 s for Claude, 47.4 s for Codex, with p_{95} in the few-minute range and occasional timeouts at the ten-minute per-file cap. *Cost:* the LLM evaluation for this paper incurred approximately \$550 net of VAT in third-party inference API charges, with per-file costs ranging from a few cents to several dollars depending on file size and the agentic loop’s iteration count. *Code privacy:* LLM-backed translation routes the C source through a third-party inference API; for codebases under non-disclosure agreements or export controls, this rules out the deployment shape unless an on-premise model is substituted. None of these properties affect the syntactic translation quality reported in Figure 5, but together they constrain the operational viability of LLM-backed translation in practice; the rule-based tools (including Oxidize) face none of them.

Performance overhead. A separate concern is whether Oxidize’s output runs as fast as the original C: a translator that produces correct but slow Rust would not be a viable migration path. We compared the runtime and memory of Oxidize’s output against the original C (compiled with `gcc -O3`) and against hand-written Rust on a standard microbenchmark suite, using `benchkit` [23]. Figure 6 reports the per-benchmark runtime and memory overhead. In runtime, Oxidize’s output tracks Clang within 5–10% on most workloads, as C2Rust does; in memory it matches C2Rust almost entirely, and the benchmarks on which it uses more memory than C are those on which every measured Rust configuration does. The full methodology is in our technical report [4].

Conclusion. Oxidize achieves Exec-OK rates competitive with C2Rust across the six single-file datasets (96% versus 99% pooled), while preserving function-like macros, which C2Rust does not (§5.1). The LLM-backed tools are competitive on raw Exec-OK on small and real-world datasets but degrade on adversarial inputs and carry operational constraints (wall-clock, cost, code privacy) that rule-based tools do not. RQ2 is thereby answered: Oxidize translates correctly across diverse single-file workloads without sacrificing the maintainability properties RQ1 measures.

6 Limitations

C language features. Some C language features remain unsupported, most notably bitfields. While Rust does not support bitfields directly, various crates exist that provide support through procedural macros. Supporting bitfields using such helper crates remains future work in Oxidize.

Unsupported macro classes. As detailed in §3.5, Oxidize accepts the expression-like macro subset and conservatively expands the rest, so translation never fails on macros. The two most prominent unsupported classes are *type macros* (`#define TYPE int`) and *statement macros*. Both are amenable to the same macro-contraction technique: type macros require running contraction over types in addition to expressions, and the prevalent `do { ... } while(0)` idiom can be recognized and translated as a Rust block expression. Neither is implemented in the current prototype; we expect both to be straightforward extensions. Statement macros that *declare local variables in the enclosing scope* (rather than within a `do { ... } while(0)` wrapper) need one more step: Rust’s macro hygiene hides local bindings a macro body introduces, so each declared name must become an extra identifier parameter, as Oxidize already does for types (§4).

Oxidize also does not currently support the C preprocessor’s stringification (`#`) and token-pasting (`##`) operators, whose semantics do not map cleanly onto Rust’s macro system. In addition, conditional-compilation constructs are resolved during preprocessing rather than translated into Rust `#[cfg(...)]` attributes. Preserving configuration structure across translation remains future work.

Maintainability evidence. We measure macro preservation, a necessary condition for maintainable output: an abstraction expanded away can no longer be maintained as one. We do not measure maintainability itself; a developer study of review and post-translation edit effort, and of readability relative to human-written Rust, remains future work.

Code quality. Although Oxidize improves readability compared to prior translators, its output still contains unsafe blocks and non-idiomatic constructs. Hong and Ryu argue for a chain of *refinement passes* over translator output, each replacing a specific unsafe or unidiomatic pattern with a Rust-native equivalent [11]; Oxidize’s structure-preserving IR is designed to host such passes natively rather than as post-processing over expanded code. The `isspace` example of Listing 2 illustrates the kind of low-level expansion that would benefit from a future *standard-library-mapping* pass that recognizes idiomatic C constructs and substitutes their Rust counterparts (e.g., `char::is_whitespace`).

7 Conclusion

We presented Oxidize, the first complex-macro-preserving C-to-Rust translator [3]. Unlike prior tools, Oxidize contracts, types, and preserves macros with their original intent, enabling semantic equivalence and developer-readable code. Our evaluation shows that this approach allows Oxidize to translate substantially more macros than competing translators and achieves robust translatability across datasets.

Future work. We plan to broaden the range of target applications to include drivers, databases, and other system-level frameworks; to extend the macro-contraction technique to type macros and statement macros; to expand both *performance benchmarking* and *translatability testing* so that future evaluations can be standardized across multiple languages and workloads; and to explore LLM-assisted authoring of individual refinement passes over the IR, where each generated pass is small and checkable, as an alternative to LLM translation of whole programs.

Outlook. By preserving macros, Oxidize shows structure-preserving translation to be practical on macro-heavy codebases, a necessary condition for maintainable migration. Our longer-term vision is to evolve Oxidize into a general framework for structure-preserving migration across programming languages—one that preserves the semantics of legacy code while keeping it open to ongoing maintenance.

Acknowledgments

We thank Attilio Discepoli and Aaron Bogaert for their contributions to early drafts of this work, the anonymous reviewers for their constructive feedback, and the EuroSys’27 artifact evaluators for their time and diligence. This work is supported by the Belgian Royal Higher Institute for Defence (RHID) under the Defence-related Research Action (DEFRA), contract no. 24DEFRA001 (FORCES project).

References

- [1] Kristy Andrews, Paul Del Vigna, and Mark Molloy. 1996. Macro and File Structure Preservation in Source-to-Source Translation. *Software: Practice and Experience* 26, 3 (March 1996), 281–292. doi:10.1002/(SICI)1097-024X(199603)26:3<281::AID-SPE14>3.0.CO;2-0
- [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. doi:10.48550/arXiv.2503.17741 arXiv:2503.17741 [cs].
- [3] 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
- [4] Robbe De Greef, Théo Engels, Felix Van den Broucke, Ken Hasselmann, and Antonio Paolillo. 2026. Lifting the Preprocessor with Oxidize: Structure-Preserving C-to-Rust Translation (Technical Report). arXiv:2609.30062 [cs.PL]
- [5] 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. doi:10.48550/arXiv.2405.11514 arXiv:2405.11514 [cs].
- [6] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering* 28, 12 (Dec. 2002), 1146–1170. doi:10.1109/TSE.2002.1158288
- [7] Alejandra Garrido and Ralph Johnson. 2013. Embracing the C preprocessor during refactoring. *Journal of Software: Evolution and Process* 25, 12 (2013), 1285–1304. doi:10.1002/smr.1603
- [8] Paul Gazzillo and Robert Grimm. 2012. SuperC: parsing all of C by taming the preprocessor. *SIGPLAN Not.* 47, 6 (June 2012), 323–334. doi:10.1145/2345156.2254103
- [9] Google Security Blog. 2021. An update on Memory Safety in Chrome. <https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html>. Accessed: 2025-09-02.
- [10] Tejun Heo. 2009. ia64: don't alias VMALLOC_END to vmalloc_end. <https://git.kernel.org/linus/126b3fcdcd350cad9700908d0ad845084e26a31>. Linux kernel commit 126b3fcdcd3, merged in v2.6.33.
- [11] Jaemin Hong and Sukyoung Ryu. 2025. Automatically Translating C to Rust. *Commun. ACM* 68, 11 (Oct. 2025), 58–65. doi:10.1145/3737696
- [12] Immunant. 2025. immunant/c2rust. <https://github.com/immunant/c2rust>
- [13] ISO. 1999. *Programming Languages — C*. International Standard ISO/IEC 9899:1999. ISO/IEC. <https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf> Also known as C99.
- [14] Anirudh Khatri, 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. Second Conference on Language Modeling (COLM). arXiv:2504.15254 doi:10.48550/arXiv.2504.15254
- [15] LeaningTech. 2024. Solving the Structured Control Flow Problem Once and for All. Medium blogpost. <https://medium.com/leaningtech/solving-the-structured-control-flow-problem-once-and-for-all-5123117b1ee2>.
- [16] Kornel Lesiński. 2024. Citrus / Citrus · GitLab. <https://gitlab.com/citrus/citrus>.
- [17] Christopher A. Mennie and Charles L. A. Clarke. 2004. Giving Meaning to Macros. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC)*. IEEE, Bari, Italy, 79–88. doi:10.1109/WPC.2004.1311050
- [18] Microsoft Security Response Center. 2019. We need a safer systems programming language. <https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/>. Accessed: 2025-09-02.
- [19] 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>
- [20] Jeffrey L. Overbey, Farnaz Behrang, and Munawar Hafiz. 2014. A foundation for refactoring C with macros. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) (FSE 2014). Association for Computing Machinery, New York, NY, USA, 75–85. doi:10.1145/2635868.2635908
- [21] Shane Panter and Nasir Eisty. 2024. Rusty Linux: Advances in Rust for Linux Kernel Development. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, Barcelona Spain, 496–502. doi:10.1145/3674805.3690756
- [22] Antonio Paolillo. 2026. pythainer: composable and reusable Docker builders and runners for reproducible research. *Journal of Open Source Software* 11, 117 (2026), 9059. doi:10.21105/joss.09059
- [23] Antonio Paolillo, Mats Van Molle, and Ken Hasselmann. 2026. benchkit: A Declarative Framework for Composable Performance Evaluation of System Software. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE '26)*. Association for Computing Machinery, Florence, Italy, 170–183. doi:10.1145/3777884.3796997
- [24] 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), 25 pages. doi:10.1145/3808276
- [25] Lucian Popescu, Francisco Gouveia, Henrique Preto, João Silveira, Dmytro Hrybenko, José Fragoso Santos, and Nuno P. Lopes. 2026. Cpp2Rust: Automatic Translation of C++ to Safe Rust. *Proc. ACM Program. Lang.* 10, PLDI, Article 188 (June 2026), 25 pages. doi:10.1145/3808266
- [26] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. arXiv:2105.12655 [cs.SE] <https://arxiv.org/abs/2105.12655>
- [27] Dennis M. Ritchie. 1993. The Development of the C Language. *ACM SIGPLAN Notices* 28, 3 (March 1993), 201–208. doi:10.1145/155360.155580
- [28] Rust Compiler Team. 2025. Rust Compiler Development Guide: Macro Expansion. <https://rustc-dev-guide.rust-lang.org/macro-expansion.html>. Online documentation; accessed: 2025-08-21.
- [29] Jon Seager. 2026. Investing in automated C-to-Rust translation. <https://discourse.ubuntu.com/t/investing-in-automated-c-to-rust-translation/86369> Ubuntu Community Hub; announcement of the Canonical / UKRI / University of Bristol project. [Online; accessed 2026-09-17].
- [30] Jamey Sharp. 2017. Jameysharp/Corrode. <https://github.com/jameysharp/corrode>
- [31] Nishanth Shetty, Nikhil Saldanha, and M. N. Thippeswamy. 2019. CRUST: A C/C++ to Rust Transpiler Using a “Nano-parser Methodology” to Avoid C/C++ Safety Issues in Legacy Code. In *Emerging Research in Computing, Information, Communication and Applications*, N. R. Shetty, L. M. Patnaik, H. C. Nagaraj, Prasad Naik Ham-savath, and N. Nalini (Eds.). Springer Singapore, Singapore, 241–250. doi:10.1007/978-981-13-5953-8_21

- [32] Melih Sirlanci, Carter Yagemann, and Zhiqiang Lin. 2025. C2RUST-BENCH: A Minimized, Representative Dataset for C-to-Rust Transpilation Evaluation. doi:10.48550/arXiv.2504.15144 arXiv:2504.15144 [cs].
- [33] Stack Overflow. 2024. In Rust We Trust: White House Office Urges Memory Safety. Online. <https://stackoverflow.blog/2024/12/30/in-rust-we-trust-white-house-office-urges-memory-safety/> Accessed: 2025-05-12.
- [34] Richard M. Stallman and Zachary Weinberg. 2025. *The C Preprocessor*. Free Software Foundation. <https://gcc.gnu.org/onlinedocs/cpp.pdf> Accessed July 2025.
- [35] Jarred Sumner. 2026. Rewriting Bun in Rust. <https://bun.com/blog/bun-in-rust> Bun Blog. [Online; accessed 2026-09-18].
- [36] GNU C Development Team. 2022. GNU C Language Introduction and Reference Manual — Common Type. https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Common-Type.html. Accessed: 2025-08-21.
- [37] The Chromium Project. 2025. Memory Safety. <https://www.chromium.org/Home/chromium-security/memory-safety/>. Accessed: 2025-09-02.
- [38] The Rust-lang Project. 2025. rust-bindgen: Automatically generates Rust FFI bindings to C (and some C++) libraries. <https://github.com/rust-lang/rust-bindgen>. Version 0.72.1, accessed 2026-04-25.
- [39] Chaofan Wang, Tingrui Yu, Jie Wang, Dong Chen, Wenrui Zhang, Yuling Shi, Xiaodong Gu, and Beijun Shen. 2026. EVOC2RUST: A Skeleton-guided Framework for Project-Level C-to-Rust Translation. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, Rio de Janeiro, Brazil, 94–105. arXiv:2508.04295 doi:10.1145/3786583.3786856
- [40] 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)*. IEEE Computer Society, Los Alamitos, CA, USA, 90–102. doi:10.1109/ICSE55347.2025.00127
- [41] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2009. Csmith, a random generator of C programs. <https://github.com/csmith-project/csmith> [Online; accessed 2025-09-08].
- [42] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (Portland, Oregon, USA) (OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 301–312. doi:10.1145/2048147.2048224
- [43] 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)*. IEEE, Toronto, ON, Canada, 1254–1259. doi:10.1109/COMPSAC65507.2025.00158 ISSN: 2836-3795.
- [44] Tianyang Zhou, Haowen Lin, Somesh Jha, Mihai Christodorescu, Kirill Levchenko, and Varun Chandrasekaran. 2025. SACTOR: LLM-Driven Correct and Idiomatic C to Rust Translation with Static Analysis and FFI-Based Verification. arXiv:2503.12511 [cs.SE] <https://arxiv.org/abs/2503.12511>