Translation Tag Team: Formal Rules and LLMs Translate More Macros Together than Apart

Anonymous Author(s)

Abstract

Modern critical software infrastructure is largely written in C. Since C lacks memory safety, researchers are investigating automatic translation of C to safer languages like Rust. But real-world C software consists of more than just C code, often using named code fragments called macros which are not part of the C language proper. State-of-the-art techniques avoid translating macros by preprocessing C code first before translating it. But this approach produces translations that are dissimilar to the original C code, because preprocessing inlines all macro definitions. To preserve macro usage in translated code, we study the language features that macros and C share and distill them into the first formally-specified translator, MerC. To evaluate MerC, we introduce the first macro translation benchmark, MacroBench, with test cases based on macros randomly sampled from real-world C programs. We find that MerC supports 50% of MacroBench's macro test cases. We also use MacroBench to evaluate how effective large language models (LLMs) are at performing the previously-unstudied task of macro translation. LLMs translate 22% to 77% more of MacroBench than MerC, but with 8% and 28% of these translations being incorrect translations requiring additional validation by developers. In contrast, MerC only produces correct translations. Our key insight is that running MerC first then using LLMs on the remainder reaps greater benefits than using either technique alone. This *tag team* approach has an average failure rate 32% lower than that of LLMs, while also translating an average of 51% more test cases than MerC.

CCS Concepts

• **Software and its engineering → Preprocessors**.

Keywords

macros, C, program translation, program analysis

ACM Reference Format:

Anonymous Author(s). 2026. Translation Tag Team: Formal Rules and LLMs Translate More Macros Together than Apart. In . ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Modern critical software infrastructure is largely written in C. Since C lacks memory safety [5, 12, 69], researchers are investigating automatic translation of C to safer languages like Rust [4, 6, 14, 31, 33, 60]. However, real-world C code heavily uses [15, 53] *macros*[20], which are named code fragments and not part of the C language proper [26, 37, 50, 62]. Instead, the C preprocessor replaces macros in source code with their defined code fragments to produce pure C code. C tools have long-struggled with macros [26, 37, 50, 62], and translators are no exception [7–9, 32]. State-of-the-art C translators avoid macros by preprocessing first, then translating the resulting pure C code [7, 34, 61]. But this approach emits translations which inline all macros and lose their abstractions.

The problem is that by expanding macros first, translators end up translating a version of C code different from the original, and emit translated code that lacks the original code's macro abstractions. For example, the code below is from the Linux kernel source[1]:

```
gen6_pte_t pte = GEN6_GTT_ADDR_ENCODE(addr) | GEN6_PTE_VALID;
```

Syntactically, `GEN6_GTT_ADDR_ENCODE(addr)` resembles a C function call and `GEN6_PTE_VALID` a C variable, but both are actually macros. After preprocessing, the macros are inlined, producing the following C code:

```
gen6_pte_t pte = ((addr) | (((addr) » 28) & 0xff0)) |
    ((u32)(((((1UL))) « (0)) + ((int)(sizeof(struct {
      int : (-!!((sizeof(int) ==
        sizeof(*(8 ? ((void *)((long)(0) * 0l)) :
          (int *)8))) && ((0) < 0 || (0) > 31)));})))));
```

This barely-recognizable C code, which lacks the original programs's macros, is what translators ultimately end up translating to a new language, because they preprocess macros first. Macros are very common in real-world source code: Linux has over 50 thousand macro definitions and 500 thousand macro invocations, and C programs average one invocation every four lines of code [15].

There is little prior research on preprocessor macro translation. Mennie and Clarke describe a translation technique [44] for a restricted set of macros that act like C variables, omitting commonly-used function-like macros, but lack a rigorous evaluation or benchmark, and their translator's code is publicly unavailable. Visual Studio supports translating macros to C++ constant expression declarations [55], but lacking semantic analysis [53] gets translations wrong [57] and is designed for C++, not C. LLMs are potentially viable for macro translation [18, 28, 46, 51, 61] and some are specifically designed for code translation [27, 47]. But to the best of our knowledge, there is no prior study of their ability to translate macros. Moreover, there are no published macro translation benchmarks that would enable evaluating different translators.

We introduce the first formally-specified, semantics-aware macro translator. The key challenge is that while macro usage appears syntactically like C functions, their semantics can differ greatly [53]; translators cannot in general treat them like C functions and expect to produce equivalent code. The key insight to our approach is that by identifying those macro usages that behave like C functions, we can transform such macros to true C functions. The benefit is

[1] `linux-v6.2-rc2/drivers/gpu/drm/i915/gt/intel_gtt.h`

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

that existing C translators can then translate them like any other C function without special support for macros. We study macro and C language features and identify the specific conditions under which macros match the behavior of C functions, variables, or enums. We define five formal translation rules that capture these conditions and implement them in a new tool called MerC.

With no published benchmark designed for evaluating macro translation, we introduce a new benchmark suite called MacroBench sampled randomly from real-world C programs, independent from our formal translation rule specifications. MacroBench is a statistically significant (95% confidence level, 5% margin of error) random sample of 398 macro test cases stratified over 23 real-world C programs used in prior studies of C and macros [15, 26, 37, 53]. Each test case focuses on the translation of a single macro and contains its surrounding language context. MacroBench also contains more than one thousand real translations of its test cases produced during the evaluation in this paper. We semi-automatically checked each translation to record the translation's correctness, what language construct the tool attempts to translate each macro to, why each tool failed for incorrect translations, timings, and other metadata.

We then use MacroBench to evaluate how effective popular large language models (LLMs) [28, 46, 47] are at translating macros, and compare them to MerC. We specifically compare MerC to LLMs, and no other rule-based tools, because to the best of our knowledge MerC is the first publicly-available semantics-aware macro-to-C translator. Other rule-based tools either ignore macro semantics and emit incorrect translations [55], lack a publicly-available implementation [44], or do not translate macros to C [39, 43]. MerC translates all MacroBench's test cases in 37 seconds and produces no incorrect translations, but only emits translations for 50% of MacroBench's test cases. Meanwhile, LLMs emit translations for 22% to 77% more test cases than MerC, but spend 45 minutes to 2.5 hours on translation, and get 8% to 28% of translations wrong. The presence of incorrect translations means that developers validate all LLM translations to find correct ones, a substantial task for large codebases like the Linux kernel which contains thousands of macros. In contrast, MerC's semantics-aware translation rules only produce correct translations, obviating substantial validation.

We show that using MerC to translate macros first, and then "tagging in" LLMs to translate the remaining macros, reaps greater benefits than using either technique alone. We observe an 18% to 45% lower failure rate for this *tag team* approach than using LLMs alone and 30% to 80% more translations than using MerC alone. Counterintuitively, although LLMs have a much higher average failure rate on the remaining macros not supported by MerC (38% vs. 17%), it is less costly in both overall translation time and post-translation validation effort to reserve LLMs for these harder cases, rather than treating LLMs as all-purpose translators.

We make the following contributions:

- A comparison of macro and C semantics and a set of formal translation rules for common macro usage (Section 2).
- MerC, an implementation our translation rules (Section 3).
- MacroBench, the first benchmark specifically focused on preprocessor macro translation (Section 4).
- An evaluation and comparison of the macro translation capabilities of MerC and several popular LLMs (Section 5).

The anonymized, publicly-available artifact[2] includes MerC, MacroBench, experimental scripts, and data.

2 Translating Macros with MerC

We present formal translation rules specifying how and when to convert macro definitions into C definitions, and give examples of how MerC follows these rules to translate macros. We designed MerC's translation rules by studying the C preprocessor's ISO specifications [35, 36], its usage in large open source projects, and analyses of it in prior work [13, 15, 53].

2.1 Macro Definitions

Each macro definition has a C identifier as a name, and a possibly empty sequence of C tokens as a body. *Function-like* macro definitions also have a parenthesized, comma-separated list of C identifiers as arguments. *Object-like* macros do not. For example:

```
1 #define PI          3.14
2 #define ADD(a, b)   a + b
3 PI * (ADD(PI, 2))
```

Lines 1 and 2 define the object-like macro `PI` and the function-like macro `ADD()`. Line 3 preprocesses to `3.14 * (3.14 + 2)`.

A macro invocation need not expand to a complete C statement:

```
1 #define MY_IF(X)    if (X)
2 MY_IF(x > 0) { ... }
```

Listing 1: `MY_IF()` macro definition and invocation.

Line 2 preprocesses to `if (x > 0)`, a fragment of a C if statement.

2.2 Determining Translatability

Name	Condition checked
DefinedGlobally(*name*)	*name* is defined outside a function.
CalledByName(ι)	ι uses call-by-name semantics.
CapturesEnv(ι)	ι accesses caller's environment.
IsConstExpr(ι)	ι is a constant C expression.
IsExpression(ι)	ι is a C expression.
IsICE(ι)	ι is a compile-time constant (ICE).
IsStatement(ι)	ι is a not an expression.
AddressRequired(ι)	ι must be an addressable expression.
SizeRequired(ι)	ι must be an expression with a size.
ConstExprRequired(ι)	ι must be a compile-time constant.
IsMeta(ι)	ι manipulates tokens.

Table 1: Informal function definitions for Figure 2. Each function accepts either a macro *name*, or an invocation or argument expansion ι, as input, and outputs a boolean.

Figure 1 presents a six-way Venn diagram comparing the C and C preprocessor semantic features MerC uses to decide which macro definitions are translatable to C variable, enum, and function definitions, and which macro arguments are translatable to C function arguments. A macro that only uses the semantic features of a C construct can be safely translated to a definition of that construct, without needing to change the macro's callsites. For instance, a

[2] https://doi.org/10.5281/zenodo.15021666

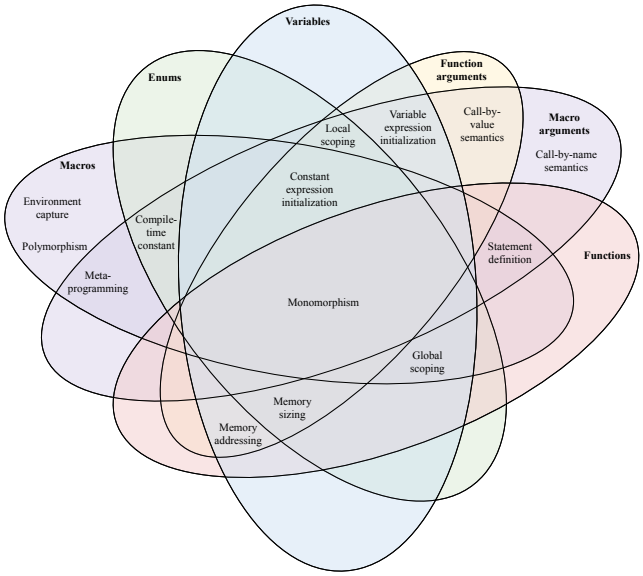


Figure 1: Semantics available to C and macro constructs.

macro that only uses features in Figure 1's overlapping regions between macros and enums would be safe to turn into a C enum by swapping its macro definition with an enum definition, without changing its callsites. This enables downstream C translators to translate C code while preserving macro abstractions, and helps remove needless macros from programs like the Linux kernel that have guidelines to prefer functions over macros when possible [64].

Figure 1 is formalized in Figure 2 as a set of inference rules for translating macros to C code. Each rule has the form:

$$\frac{p_0 \cdots p_n}{V_0 \cdots V_n \vdash A \rightsquigarrow B}$$

Where $p_0 \cdots p_n$ are premises that must all be true for the rule to apply, $V_0 \cdots V_n$ are context variables necessary for evaluating the rule's premises, and A and B are source and target syntax.

Rules may use any of the four context variables M, Γ, A, and Δ. M and Γ map each macro's name to its set of invocations and the types of the C expressions it expands to, respectively. A and Δ map each pair of macro name and argument name to the set of the argument's expansions and the types of the C expressions it expands to, respectively. A macro or macro argument that expands to a non-expression C statement maps to **void** in Γ or Δ, respectively.

Rule premises use helper functions typeset in SmallCaps font. Table 1 defines each function informally. MerC uses an existing macro analyzer [53] to compute these functions in practice. All premises and functions expect named macros expanding to whole C statements, and MerC does not translate macros like `MY_IF()` (Listing 1) that expand to C syntax fragments, or variadic macros.

2.3 Translating Object-like Macros

Rules I and II (Figure 2) translate to variables and enums, respectively, macros whose semantics match the rules' premises. For example, the object-like macro `QUIT`, which expands to the character `'q'`, satisfies Rule I's conditions for translation to a C variable:

```
1 #define QUIT 'q'
2 void check_quit(void) {
3     if (QUIT == getch()) exit(0);
4 }
```

Listing 2: `QUIT` macro definition and invocation.

Premise 1 checks that `QUIT` is defined outside of function boundaries, so that translating it to a C variable will not limit it to a specific scope. Premise 2 collects all `QUIT`'s invocations, in this case the one invocation on Line 3, into a set μ. Premises 3 and 4 check that `QUIT` does not capture identifiers from its caller's environment nor performs metaprogramming actions like token-pasting or stringification [21], as C variables lack these features. `QUIT` satisfies these premises, because it expands to the literal `'q'`. Premises 5 and 6 verify the macro is not used with the address-of (`&`) or `sizeof` operators, because differences between macro and C scoping rules mean that variables in macro expansions and function bodies have different memory addresses and sizes.

Premise 7 validates that `QUIT` is not used where a compile-time constant is required (e.g., in a case label), because C variables are not compile-time constants[3]. `QUIT` satisfies Premise 7 because it expands in an `if` condition (Line 3) where compile-time constants are not required. Premise 8 ensures that `QUIT`'s definition is a constant expression, since it will be translated to a global variable, per Premise 1, and C requires global initializers to be constant. `QUIT` satisfies Premise 8 because it expands to the character literal `'q'`.

Finally, MerC checks Premises 9-12 to ensure that the macro's type is not `void`—valid only for functions, not variables—and that the macro is *monomorphic*, i.e., all its expansions have the same type, since C variables are monomorphic. Expanding only to a character type (`char`), `QUIT` satisfies these premises. Since `QUIT` satisfies all premises, MerC can follow Rule I's conclusion to correctly rewrite the macro definition on Line 1 to a global C variable definition.

```
1 static const char QUIT = 'q';
2 void check_quit(void) {
3     if (QUIT == getch()) exit(0);
4 }
```

Note that *no other parts of the C source code change*. MerC's specification ensures that identical syntax can be used at macro invocation sites and the rewritten C variables, functions, or enums, e.g., Line 3's usage of `QUIT`.

Rule II identifies macro-to-`enum` translations. Unlike Rule I, it identifies invocations used where only compile-time constant expressions are permitted (Premise 19).

2.4 Translating Function-like Macros

Rules III and IV translate function-like macros to functions that do and do not return a value (`void` functions), respectively. Returning value affects function call site semantics. To demonstrate Rule III, consider the function-like macro `MUL()`, which expands to the expression `((n)) * ((m))`:

[3] C23, finalized Oct. 2024, permits compile-time constant variables with `constexpr` [36].

Rule I: Object-like macro to global variable

$$\frac{\begin{array}{lll}(1)\ \textsc{DefinedGlobally}(name) & (5)\ \forall \iota \in \mu : \neg\textsc{AddressRequired}(\iota) & (9)\ \gamma = \Gamma(name) \\ (2)\ \mu = M(name) & (6)\ \forall \iota \in \mu : \neg\textsc{SizeRequired}(\iota) & (10)\ |\gamma| = 1 \\ (3)\ \forall \iota \in \mu : \neg\textsc{CapturesEnvironment}(\iota) & (7)\ \forall \iota \in \mu : \neg\textsc{ConstExprRequired}(\iota) & (11)\ type \in \gamma \\ (4)\ \forall \iota \in \mu : \neg\textsc{IsMeta}(\iota) & (8)\ \forall \iota \in \mu : \textsc{IsConstExpr}(\iota) & (12)\ type \notin \{\textbf{void}\}\end{array}}{\text{M, } \Gamma \vdash \textbf{\#define } name\ body \rightsquigarrow \textbf{static const } type\ name = body\ \textbf{;}}$$

Rule II: Object-like macro to enum

$$\frac{\begin{array}{lll}(13)\ \textsc{DefinedGlobally}(name) & (17)\ \forall \iota \in \mu : \neg\textsc{AddressRequired}(\iota) & (21)\ |\gamma| = 1 \\ (14)\ \mu = M(name) & (18)\ \forall \iota \in \mu : \neg\textsc{SizeRequired}(\iota) & (22)\ type \in \gamma \\ (15)\ \forall \iota \in \mu : \neg\textsc{CapturesEnvironment}(\iota) & (19)\ \forall \iota \in \mu : \textsc{IsICE}(\iota) & (23)\ \textsc{SizeOf}(type) \leq \textsc{SizeOf}(\textbf{int}) \\ (16)\ \forall \iota \in \mu : \neg\textsc{IsMeta}(\iota) & (20)\ \gamma = \Gamma(name) & \end{array}}{\text{M, } \Gamma \vdash \textbf{\#define } name\ body \rightsquigarrow \textbf{enum \{ } name = body\ \textbf{\};}}$$

Rule III: Function-like macro to non-void function

$$\frac{\begin{array}{lll}(24)\ \textsc{DefinedGlobally}(name) & (29)\ \forall \iota \in \mu : \neg\textsc{SizeRequired}(\iota) & (34)\ type \in \gamma \\ (25)\ \mu = M(name) & (30)\ \forall \iota \in \mu : \neg\textsc{ConstExprRequired}(\iota) & (35)\ type \notin \{\textbf{void}\} \\ (26)\ \forall \iota \in \mu : \neg\textsc{CapturesEnvironment}(\iota) & (31)\ \forall \iota \in \mu : \textsc{IsExpression}(\iota) & (36)\ name, \text{A}, \Delta \vdash args \rightsquigarrow args' \\ (27)\ \forall \iota \in \mu : \neg\textsc{IsMeta}(\iota) & (32)\ \gamma = \Gamma(name) & \\ (28)\ \forall \iota \in \mu : \neg\textsc{AddressRequired}(\iota) & (33)\ |\gamma| = 1 & \end{array}}{\text{M, } \Gamma, \text{A, } \Delta \vdash \textbf{\#define } name\ \textbf{(}\ args\ \textbf{)}\ body \rightsquigarrow \textbf{static } type\ name\ \textbf{(}\ args'\ \textbf{)\ \{ return } body\ \textbf{; \}}}$$

Rule IV: Function-like macro to void function

$$\frac{\begin{array}{lll}(37)\ \textsc{DefinedGlobally}(name) & (42)\ \forall \iota \in \mu : \neg\textsc{SizeRequired}(\iota) & (47)\ type \in \gamma \\ (38)\ \mu = M(name) & (43)\ \forall \iota \in \mu : \neg\textsc{ConstExprRequired}(\iota) & (48)\ type \in \{\textbf{void}\} \\ (39)\ \forall \iota \in \mu : \neg\textsc{CapturesEnvironment}(\iota) & (44)\ \forall \iota \in \mu : \textsc{IsExpression}(\iota) \vee \textsc{IsStatement}(\iota) & (49)\ name, \text{A}, \Delta \vdash args \rightsquigarrow args' \\ (40)\ \forall \iota \in \mu : \neg\textsc{IsMeta}(\iota) & (45)\ \gamma = \Gamma(name) & \\ (41)\ \forall \iota \in \mu : \neg\textsc{AddressRequired}(\iota) & (46)\ |\gamma| = 1 & \end{array}}{\text{M, } \Gamma, \text{A, } \Delta \vdash \textbf{\#define } name\ \textbf{(}\ args\ \textbf{)}\ body \rightsquigarrow \textbf{static void } name\ \textbf{(}\ args'\ \textbf{)\ \{ }body\ \textbf{; \}}}$$

Rule V: Empty macro argument list

$$\frac{}{name, \text{A}, \Delta \vdash \quad \rightsquigarrow}$$

Rule VI: Non-empty macro argument list

$$\frac{(50)\ name, \text{A}, \Delta \vdash arg \rightsquigarrow arg' \quad (51)\ name, \text{A}, \Delta \vdash args \rightsquigarrow args'}{name, \text{A}, \Delta \vdash arg, args \rightsquigarrow arg', args'}$$

Rule VII: Non-empty macro argument

$$\frac{\begin{array}{lll}(52)\ \alpha = \text{A}(name, arg) & (56)\ \forall \iota \in \alpha : \neg\textsc{SizeRequired}(\iota) & (60)\ \delta = \Delta(name, arg) \\ (53)\ \forall \iota \in \alpha : \neg\textsc{CapturesEnvironment}(\iota) & (57)\ \forall \iota \in \alpha : \neg\textsc{ConstExprRequired}(\iota) & (61)\ |\delta| = 1 \\ (54)\ \forall \iota \in \alpha : \neg\textsc{IsMeta}(\iota) & (58)\ \forall \iota \in \alpha : \neg\textsc{CalledByName}(\iota) & (62)\ type \in \delta \\ (55)\ \forall \iota \in \alpha : \neg\textsc{AddressRequired}(\iota) & (59)\ \forall \iota \in \mu : \textsc{IsExpression}(\iota) & (63)\ type \notin \{\textbf{void}\} \end{array}}{name, \text{A}, \Delta \vdash arg \rightsquigarrow type\ arg}$$

Figure 2: Formal rules for MerC's translation of macros to C code.

```
1 #define MUL(X, Y) ((X)*(Y))
2 int *new_matrix(int n, int m) {
3     return calloc(MUL(n, m), sizeof(int));
4 }
```

Listing 3: `MUL()` macro definition and invocation.

Premises 24-30 are the same as those for object-like macros (Premises 1-7) explained above in Section 2.3.

Premise 31 verifies that `MUL()` is an expression, because C function calls and their return values must be expressions. `MUL()` satisfies Premise 31 because it expands to an arithmetic expression. Premises 32-35 check that `MUL()` has a monomorphic type, since C functions are monomorphic. `MUL()` satisfies premises 32-35 since its one invocation expands to a single integer type.

Premise 36 uses Rules VII-VI to translate untyped macro arguments to typed C function arguments. Rules V and VI translate lists of arguments, and Rule VII translates individual arguments. For instance, `MUL()`'s first argument, `X`. Rule VII first uses Premise 52 to collect `X`'s expansions. Premise 53 ensures that `X`, like C function arguments, does not capture identifiers from its caller's environment. `X` satisfies Premise 53 because it only refers to `n`, which is passed as `MUL()`'s `X` argument (Listing 3, Line 3). Like C variables, C function arguments lack preprocessor metaprogramming capabilities, and C function arguments behave differently from macro arguments

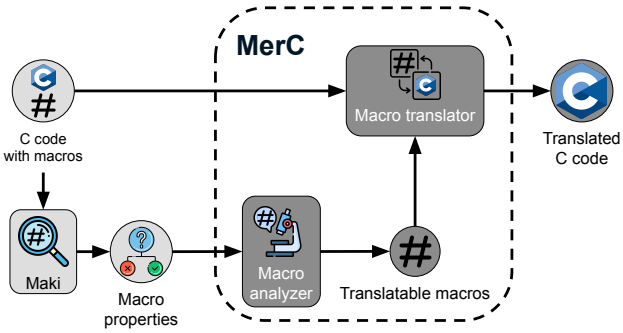


Figure 3: MerC architecture diagram. Circles are data. Dark gray rectangles are novel contributions. Maki is prior work.

when used with address-of (&) or size-of operators (Section 2.3). Premises 55-57 specify these conditions.

Premise 58 validates that the argument does not use call-by-name semantics, as function arguments are call-by-value. Call-by-name semantics evaluate an argument and its side-effects each time it is used in the macro's body, which may be many, or not at all if the macro appears in the unevaluated branch of a short-circuiting operation such as logical and (&&). In contrast, call-by-value semantics evaluate functions' arguments exactly once at their callsites. `X` satisfies Premise 58 because the first argument to the call to `MUL()` on Line 3 of Listing 3 does not have side-effects, and `MUL()`'s definition does not use `X` in a short-circuiting operation.

Finally, Premise 59 ensures `X` is monomorphic, which `X` is because it expands to the integer expression `n`. Satisfying Rule VII's premises, both macro arguments `X` and `Y` can be translated following the rewrite rule to `int X` and `int Y`, respectively. Since `MUL()` and its arguments satisfy Rule III, the macro can be translated to:

```
1 static inline int MUL(int X, int Y) { return ((X)*(Y)); }
2 int *new_matrix(int n, int m) {
3     return calloc(MUL(n, m), sizeof(int));
4 }
```

As with object-like macros, this translation is a drop-in replacement for the macro and requires no changes to the macro's callsites. MerC also makes `MUL()`'s translation `inline` to inline its callsites at compile-time, preserving the original macro's performance.

3 MerC Implementation

Figure 3 shows the architecture of MerC, which inputs unpreprocessed C source code and emits unpreprocessed C source with supported macros translated to C constructs. MerC is implemented in 952 source lines of Python code. MerC implements the translation rule premises by taking fine-grained macro semantic properties from the Maki [53] tool's output and aggregating them to match the translation rule premises. When all a translation rule's premises apply, MerC rewrites the macro into the rule's target C construct. Because MerC's translation rules preserve the same syntax of macro callsites, MerC only ever needs to modify the definitions of supported macros to translate them to C constructs. Moreover, MerC retains macros' performance benefits by translating macros to `inline` functions. Downstream translators can then convert these inline functions to compile-time functions in the target language, such as Rust's own macros or `#[inline]` functions.

4 Macro Translation Benchmark

We introduce MacroBench, the first benchmark specifically focused on macro translation. MacroBench is designed to be realistic, varied, focused, and usable. We constructed MacroBench by randomly sampling macro usage from widely-varying, real-world C programs to ensure representativeness, independence from any particular translation techniques, and statistical significance.

4.1 Design Principles

We adhered to the following principles when creating MacroBench.

Realism. MacroBench's test cases should reflect real-world macro usage to best represent a translator's performance on C code. For realism we base MacroBench on the 26 real-world programs studied in the seminal work on macro analysis [15], removing 8 programs that use C++ or whose dependencies are no longer available[4]. We compensate for the removed programs by adding 5 modern ones: Linux, Lua, Python, SQLite, and FFMpeg, for a total of 23 programs, increasing the benchmark to 100,711 macro definitions[5].

We sample macros with stratified random sampling, where the strata are the source programs. We calculate the number of macros sampled from each program based on its number of definitions, so that all programs contribute at least one macro to MacroBench. Otherwise, small programs would likely contribute no macros, and large programs like the Linux kernel would dominate. To ensure representativeness, we choose a statistically-significant sample size of 398, for a 95% confidence level and 5% margin of error.

Macros in real-world code do not appear in isolation, but are surrounded by declarations, statements, and expressions. MacroBench preserves this context by including the declarations of any variables, functions, types, or other symbols the sampled macro references, along with expressions or statements invoking the sampled macro.

Variety. MacroBench's test cases should contain diverse macro usage to evaluate a translator's ability to handle the many kinds of macros real-world C uses. To achieve variety, we sample macros from a selection of C software spanning diverse software categories, including databases, programming languages, and window management. Furthermore, by stratifying macros by their source programs we ensure MacroBench includes macros from all programs, even small ones, since simple random sampling would likely exclude macros from small programs altogether. After sampling, we verified by hand that MacroBench contains myriad types of macros that have been previously documented by researchers [44], including named constants, function name aliases, generic functions, and metaprogramming macros. 22% are function-like and 78% are object-like, reflecting the distribution found by prior research [15]. Additionally, we check for diverse semantics [15, 39, 53], such as variable capture, token-pasting, control-flow modifications, and more. The complete listing of macros, originating programs, and additional metadata can be found in our accompanying artifact[2].

[4] gcc, gnuchess, gs, mosaic, plan, rasmol, workman, and zephyr.
[5] See the file `programs.txt` in our public artifact for a listing of each program's name, version, source lines of code, and number of invoked and sampled macro definitions.

Focus. MacroBench is designed to specifically measure macro translation capacity, without penalizing tools for lack of whole program translation or build system analysis. In particular, LLMs' limited context windows and cloud service input token limits preclude prompting them with entire programs, which may comprise millions of lines of code, only to evaluate macro translation capacity. Similarly, C programs use build automation to set compiler flags and use header files to share common macro definitions across C files. To evaluate macro translation with whole program translation would require a tool to handle these pragmatic issues in order to even observe macro usage. In contrast, MacroBench enables a baseline comparison of macro translation in isolation from whole program translation pragmatics. To accomplish this, we hand-slice the macro definition and invocations into a single C file per macro, preserving any local language usage that surrounds the macro and is used within the macro to retain its semantic context.

Usability. MacroBench is made to be readily accessible and usable for other researchers to evaluate macro translation to C or other languages. We organize MacroBench's test cases into directories based on the programs we sampled them from and include build automation that researchers and tool designers can use to integrate and automate their tool's translation of the benchmark. We make each test case a single file so that AI researchers may directly prompt models to perform translations without also providing header files, or whole programs as inputs. Finally, we include the evaluated tools' translations (Section 5) along with the semi-automatically validated results of the evaluation, including which translations are correct or incorrect, failure type, locations, descriptions, and other metadata, so that future researchers developing macro translations have a baseline for comparison. MacroBench is available publicly and is included in the accompanying artifact[2] for review.

4.2 Benchmark Creation Procedure

We used the following procedure to convert each sampled macro into a MacroBench test case. For each test case we created one C source file, and copied into it a single sampled macro's definition and invocation sites. Most macros were only invoked a handful of times, and for these test cases we included all invocations. But a small number (12%) of macros were invoked thousands of times across many C files, and to include them all would produce a file too large for input to LLMs. So for these macros, we used a random number generator to select a random sample of the macros invocations, and copied these into the test case file. Each such case is described in the macro benchmark listing in the accompanying artifact[2]. For macros invoked within C functions or other constructs, to maintain realistic semantics we copied the macro's surrounding language constructs, along with any types, variables, additional macros, and other symbols that the benchmark macro relies on. We included context recursively; if a macro depends on the declaration of some type A that itself depends on type B, we included declarations of both A and B in the benchmark. Similarly, since macros may be called within other macros, we preserve the calling context of nested benchmark macros by partially expanding all macro invocations until reaching the sampled benchmark macro, so that the benchmark macro's invocation is explicit.

5 Evaluation

We investigate these six research questions about MerC and LLMs.

RQ1 How many translations do tools attempt?
RQ2 How many translations are failures?
RQ3 How fast are translations?
RQ4 How fast and effective are MerC-LLM Tag Teams?
RQ5 What are LLMs' translations failures?
RQ6 How does prompting strategy affect LLM translations?

5.1 LLM Selection

We chose the following LLMs: GPT-4o [46] because it is popular and likely to be used by developers in practice; Claude 3.5 Sonnet [28] because it is designed for solving coding tasks [27]; and o1 Preview [47] for its focus on reasoning [48]. We use the GitHub Copilot Visual Studio Code extension [29] to translate macros with GPT-4o and Claude 3.5 Sonnet, because Copilot is a popular tool that reflects developer LLM usage. GPT-4o is its current default model. Copilot recently added support for Claude 3.5 Sonnet, a model that reportedly "excels at coding tasks" [27]. Since macro translation is a coding task, we also use Copilot with Claude 3.5 Sonnet. Finally, we include o1 Preview[6] to represent LLMs with reportedly superior reasoning [48]. Copilot currently imposes strict daily usage limits on this model [30], so we use OpenAI's own API [49] to evaluate it.

5.2 Experimental Setup and Methods

We performed the LLM translations using cloud services (Copilot and OpenAI's) and performed MerC translations on an off-the-shelf laptop with an 8-core Intel i7-11857G processor and 16GB of RAM.

We performed LLM translations following these steps: (1) start a new chat with the model, (2) prompt the model to translate a test case, (3) record the translation time, and (4) save the translation in a new file. All LLMs were given the same zero-shot [63] prompt:

> translate this code to pure C, translating the macro *MACRO_NAME* to an enum, function, or variable without changing program semantics or callsites. You are NOT ALLOWED to change the callsites of the macro or create any new macros. Therefore, if it is not possible to translate the macro without preserving the callsites and semantics, do not perform any translation. Report a confidence level in percentage in the accuracy of the translation in a comment at the end within the code.
> *CODE*

`MACRO_NAME` and `CODE` are replaced with the name of the macro to translate and the code (wrapped in backticks) containing the macro from MacroBench, respectively. We developed the prompt iteratively, asking LLMs to translate Maki's [53] macro analysis unit tests and picking the one that worked best. We chose zero-shot prompting because it is a popular prompting strategy [59] that developers are likely to use in practice due to its ease-of-use and flexibility [42]. We determined the elapsed translation time for GPT-4o and Claude 3.5 Sonnet from Copilot's debugging output, subtracting the time the model received the request from the time it finished responding to it. For o1 Preview, we instrumented the request script to collect wall-clock time. To translate test cases with

[6]The full o1 release occurred after we started experiments.

MerC, we instrumented MacroBench's build system to run MerC on each source file and record the time taken on each test case.

A team of students trained by a co-author semi-automatically checked and labeled all 1,068 translations for correctness (RQ2) and categorized the failures (RQ5). All results were also validated by the same co-author, so that each translation was checked at least twice. All our analyses can be found in the accompanying artifact[2].

5.3 RQ1: How Many Translations do Tools Attempt?

The attempt rate is how many MacroBench test cases a tool attempted to translate, regardless of correctness. We consider a translation attempted when the tool translates the macro definition into a C declaration (variable, function, etc.). For instance, an LLM may refuse to translate or leave the input unchanged.

Figure 4a is the total number and percent of attempted translations for each tool. GPT-4o has the greatest attempt rate of 88%, while MerC has the lowest attempt rate, 50%. Claude 3.5 Sonnet and o1 Preview, designed for better coding and reasoning tasks [27, 48], have attempt rates between that of GPT-4o and MerC, attempting 61% and 70% of test cases, respectively. While all LLMs emit more translations than MerC, their translations are not always correct.

RQ1: All LLMs attempt more translations than MerC, but their correctness is not guaranteed.

5.4 RQ2: How Many Translations are Failures?

A failure is a translation that produces invalid or semantically different C code. The failure rate is the percentage of attempted translations that fail. A perfectly accurate translation would attempt all translations (100% attempt rate) without failures (0% failure rate).

Figure 4b shows failure rates for each tool on MacroBench. MerC attempts the fewest translations, 198, but emits no incorrect translations, for a failure rate of 0%. In contrast, all LLMs emit many wrong translations, between 8% and 28%. GPT-4o in particular has the highest failure rate of 28%, with 97 of its 350 translations being incorrect. The reasoning-style LLMs (Claude 3.5 Sonnet and o1 Preview) have lower failure rates than GPT-4o (13% and 8%, respectively), but also attempt fewer translations than GPT-4o as well (241 and 279 translations, respectively).

The downside of any non-zero failure rate, no matter how small, is that developers need to validate all LLM translations, *even the correct ones*, to ensure the correctness of the results, because LLMs do not distinguish correct from incorrect translations in their output. Although we prompted LLMs to report their confidence in the translation, they almost always reported a high confidence level, even for incorrect translations. Finally, 30 macros from MacroBench were never translated by any approach. These macros use features like token-pasting [19] that lack a direct C equivalent.

RQ2: MerC had no failures but translates only those macros for which it is specifically designed. In contrast, all LLMs made more translations but always produced some incorrect ones, forcing developers to manually verify all translations.

5.5 RQ3: How Fast Are Translations?

Figure 4c presents the total time each tool spent translating all test cases. Figure 4c shows that, despite running on a personal laptop, MerC is orders of magnitude faster than all LLMs, performing all translations in less than a minute. In contrast, GPT-4o takes 48 minutes to perform all translations, while the more accurate model, Claude 3.5 Sonnet, takes an hour and a half, and o1 Preview, the most accurate model, spends the most time at two and a half hours.

RQ3: The rule-based MerC is orders of magnitude faster than LLMs (seconds vs. hours).

5.6 RQ4: How Fast and Effective are MerC-LLM Tag Teams?

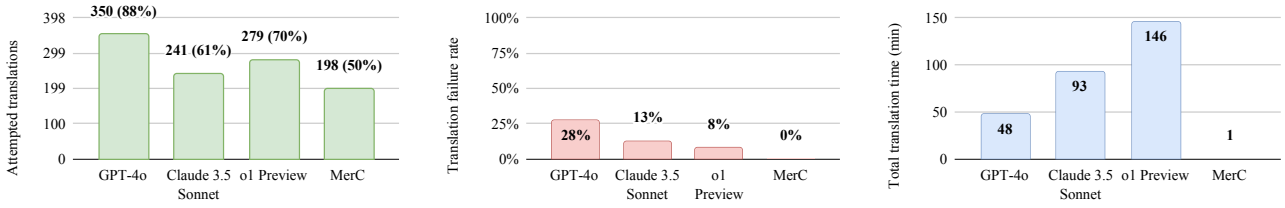
Neither MerC or LLMs are better in both number of translations and failure rate. But they have complementary strengths: MerC is fast and perfectly accurate while LLMs make more translations. We unite these complementary strengths into a *translation tag team* that first translates with MerC then "tags in" an LLM for the remaining macros MerC does not support. The tag team approach reaps greater benefits than either technique alone, decreasing translation time while increasing both the number and accuracy of translations.

The tag team works better than MerC or LLMs alone, because LLMs incorrectly translate some macros that fall within MerC's rules for common macro cases (there are 28 macros that MerC gets right that at least one LLM tool gets wrong). By avoiding LLM translation of macros that MerC already supports, developers need validate fewer translations for correctness. And by using LLMs to translate the remaining macros that MerC does not support, the tam team approach translates more macros overall.

Figure 5 compares MerC and the LLMs' attempts and failures to those of MerC-LLM tag teams. Because MerC only produces correct translations, the tag team approach only has an average 13% failure rate (7% to 22%), a 32% reduction in failures compared to using LLMs alone. Moreover, since the tag team approach supplements MerC's 198 translations with extra LLM translations, the total number of attempted translations is on average 51% higher (30% to 80%) than MerC's 198, with an average of 32% more correct translations to boot. Finally, by translating 198 macros correctly with MerC first before using LLMs to translate the remaining MacroBench macros, the number of LLM translations needing hand-validation drops by 66% on average for tag teams compared to using LLMs alone.

Similarly, MerC is orders of magnitude faster than LLMs, translating 50% of the benchmark in under a minute. Therefore, restricting LLM use to a subset of macros greatly reduces overall run-time. Figure 6 shows the total time each LLM spent translating the test cases not first translated by MerC. The MerC-LLM tag teams run in nearly half the time as the LLMs in isolation, with LLMs alone taking 48 minutes to 2.5 hours, and the MerC-LLM tag teams taking only 30 minutes to 1.3 hours.

RQ4: Using both MerC and LLMs averages 32% fewer failures than LLMs alone and 51% more translations than MerC alone.



(a) Attempted translations. (b) Translation failure rates. (c) Total translation times (minutes).

Figure 4: Attempt rates, failure rates, and translation times for LLMs and MerC when translating MacroBench's test cases.

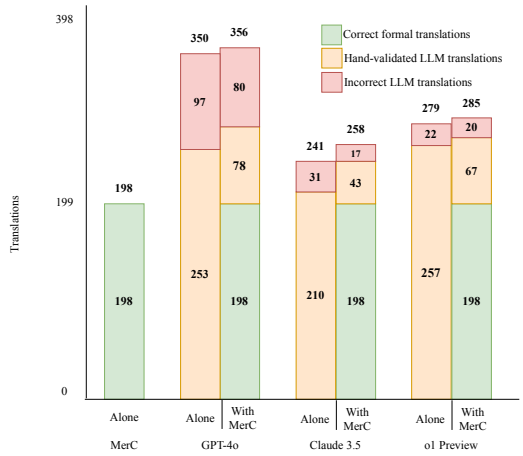


Figure 5: Types of translations performed by MerC, LLMs, and MerC-LLM tag teams.

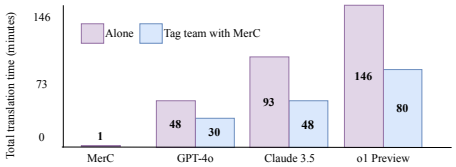


Figure 6: Total translation times for MerC, LLMs, and tag teams. Lower is better. MerC's 37 seconds rounds to 1 minute.

5.7 RQ5: What Are LLMs' Translation Failures?

We semi-automatically checked the correctness of each LLM and MerC translation along the following criteria, recording the first error found.

Compile-time behavior. Translated code must compile successfully without syntax or type errors. We automatically test for compilation errors with `gcc`, marking translations that fail to compile as compile-time failures. For example, consider o1 Preview's translation of the Emacs macro `FRAME_VISIBLE_P()`, which fails to compile because the translated version of `FRAME_VISIBLE_P()` references the struct `frame` before it is declared:

```
#define FRAME_VISIBLE_P(f)
    (f)->visible

struct frame {
    unsigned visible : 2;
};
```

Listing 4: Original `FRAME_VISIBLE_P()` macro.

```
static inline unsigned
FRAME_VISIBLE_P(struct frame *f){
    return f->visible;
}
struct frame {
    unsigned visible : 2;
};
```

Listing 5: Failed o1 Preview translation.

The translated code fails to compile because the function translation of `FRAME_VISIBLE_P()` references the struct type `frame`, which is not declared until later in the program. Macros may be defined before any types they reference because macros are dynamically scoped, but C functions, being statically scoped, must be defined after any types they reference.

Runtime behavior. All the translated macro's invocations must produce the same result as the original test case before translation. We compared each macro to its translated counterpart, and marked changes in macro behavior (e.g., changes to side-effects) as failures. For instance, GPT-4o's translation of the Emacs macro `Vafter_delete` changes the macro's runtime behavior by turning the macro into a completely new variable, whereas the original macro aliased an existing one (identifiers shortened for space):

```
#define Vafter_delete \
  globals.f_Vafter_delete
struct emacs_globals {
  Lisp_Object f_Vafter_delete;
};
extern struct emacs_globals
  globals;
```

Listing 6: Original `Vafter_delete` macro.

```
struct emacs_globals {
  Lisp_Object f_Vafter_delete;
};

extern struct emacs_globals
  globals;
Lisp_Object Vafter_delete;
```

Listing 7: Failed GPT-4o translation.

The translation fails to preserve the original program's behavior because the translation of `Vafter_delete` no longer aliases `globals.f_Vafter_delete`, but is rather an entirely new variable.

Non-functional behavior. The translated macro must preserve the program's memory layout and preprocessor semantics. For instance, translations that alter `#ifdef` behavior or mutate unions to structs fail to preserve non-functional program behavior, although these only represent a minority of failures (15% of all LLM failures). For example, Claude 3.5 Sonnet's translation of the FFmpeg macro `CONFIG_MOV_MUXER` converts a macro used in a preprocessor conditional into a C enum, changing the conditional's output:

```
#define CONFIG_MOV_MUXER 1
#if CONFIG_MOV_MUXER // true
const FFOutputFormat
ff_mov_muxer = {
    .p.name = "mov"
};
#endif
```

Listing 8: Original `CONFIG_MOV_MUXER` macro.

```
enum { CONFIG_MOV_MUXER = 1 };
#if CONFIG_MOV_MUXER // false
const FFOutputFormat
ff_mov_muxer = {
    .p.name = "mov"
};
#endif
```

Listing 9: Failed Claude translation.

The translated code alters the original program's non-functional behavior because it moves `CONFIG_MOV_MUXER` from the preprocessor namespace into the C namespace, altering the output of the program's `#if`. Since the original program defines `CONFIG_MOV_MUXER` to 1, the conditional will evaluate to true and declare the variable `ff_mov_muxer`. But because the translated code converts the macro to an enum, which is not a preprocessor symbol, the `#if` will evaluate to false, and `ff_mov_muxer` will never be declared.

	MerC	GPT-4o	Claude 3.5	o1 Preview
Compile Time	0	76	26	18
Runtime	0	4	2	2
Non-functional	0	17	3	2
Total	0	97	31	22

Table 2: Types of failed translations.

Table 2 presents the number of test cases that each tool translated incorrectly. MerC had no failures. For LLMs, the majority of translation failures were compile-time errors. Roughly half of all compile-time errors for GPT-4o and Claude 3.5 Sonnet were due to the use of non-constant values where constant values were required. This reflects a misunderstanding of C semantics and the common use of macros to define constants: macros get expanded before C compilation, so they can be used where C variables cannot.

For o1 Preview, however, the majority of compile-time errors (14 out of 18) were due to type errors, specifically undeclared references. On one hand, this indicates that o1 Preview is better at avoiding syntactic errors, but on the other it also suggests that the model misunderstands how the preprocessor interacts with C's typing rules. For example, Listing 5 presents an o1 Preview-translated macro argument which suffers a type error because it references a type declared later in the program. MerC avoids compile-time errors because its translation rules encode both C syntax and semantics.

RQ5: MerC correctly translates all attempted test cases, while LLMs' failed translations typically have compile-time errors.

5.8 RQ6: How Does Prompting Strategy Affect LLM Translations?

All LLM translations were conducted using a zero-shot prompt (Section 5.2). To evaluate whether a different prompting strategy would improve on failed LLM translations, we evaluated two additional popular prompting strategies, few-shot [3] and few-shot chain-of-thought (CoT) [68]. Few-shot prompts include example solutions of the task to solve [23], while few-shot CoT also adds explanations

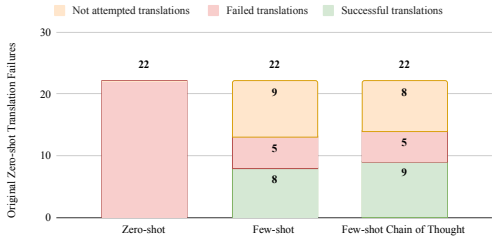


Figure 7: Comparison of the number of o1 Preview's failed translations under various prompting strategies.

of the reasoning steps followed in the examples [24]. We evaluate prompt engineering on o1 Preview specifically. Since it showed the best accuracy (RQ2) and has a reportedly high reasoning capacity [48], it stands to benefit the most from prompt engineering [38]. The few-shot and CoT prompt templates can be found in our accompanying artifact[2] in `macrobench/prompt_templates.txt`.

Figure 7 compares the number of o1 Preview translation failures with a zero-shot prompting strategy to the number of translation failures under few-shot and few-shot CoT prompting strategies. o1 Preview made 22 incorrect (out of 279 attempted) test case translations with the zero-shot prompt (Section 5.4). With few-shot, o1 Preview correctly translated eight of these cases, still incorrectly translated five of them, and did not attempt the remaining nine. Few-shot CoT was similar, only correctly translating one additional case that was not attempted with the few-shot prompt. While the prompting strategies helped reduce failed translations they also made o1 Preview more selective in avoiding translation altogether.

RQ6: Few-shot and few-shot CoT reduced failed translations but also resulted in fewer attempts on prior failed translations.

6 Threats to Validity

6.1 Internal threats

MerC translation scope. MerC only translates macros expanding to complete C constructs, and not those expanding to C syntax fragments. However, users can translate nested macro invocations by repeatedly running MerC on its own output to "peel back" layers of nested macro usage until no more translations occur.

LLM selection. Section 5's results on how well LLMs translate macros should represent real-world developer LLM usage. To this end, we evaluate GPT-4o because it is popular, Claude 3.5 Sonnet because it is designed for coding tasks [27], and o1 Preview [48] for its focus on reasoning. To further reflect developer LLM usage, we evaluate GPT-4o and Claude 3.5 Sonnet with the GitHub Copilot Visual Studio Code Extension [29]. Although Copilot also supports o1 Preview, it currently imposes strict daily usage limits on the model [30], so we use OpenAI's own API [49] to evaluate it[6].

LLM prompting. To ensure Section 5 evaluates LLMs with a quality prompt, we designed our prompt iteratively by asking LLMs to translate small random samples of macros from Maki's [53] macro analysis test cases until all LLMs performed the task as instructed.

LLM performance results. To ensure Section 5.5 accurately portrays LLM performance on macro translation, we ran the LLMs using cloud services, since real developers are likely to lack dedicated AI machines, and instead use LLMs directly from their providers.

Independence of the benchmark. To prevent biasing MacroBench towards MerC, we designed the benchmark independently from MerC's translation rules, with randomly selected test cases to avoid even inadvertently specializing MerC to MacroBench. MacroBench was created after MerC was developed, and by a separate author.

Inter-rater agreement. We did not use a formal inter-rater agreement metric when identifying and classifying failed translations in Section 5. To address this threat, a student and a co-author checked each translation, and then met to resolve discrepancies and arrive at a final mutually agreed label for each translation. Afterwards, the same co-author verified that all translation labels were consistent.

6.2 External threats

Benchmark representativeness. To ensure Section 5's findings about macro translation generalize to real developer scenarios, we draw MacroBench's test cases from the 23 programs described in Section 4, which vary in size, complexity, and application domains.

Translating whole programs. MacroBench represents real-world macro translation tasks, but factors out practical challenges of whole program translation so as to enable evaluating LLMs, whose limited input windows do not support large C programs as translation inputs. In contrast, although MerC is designed to translate single compilation units and not whole programs, it can be applied to whole programs by translating each of their compilation units individually. We use this strategy to explore how well MerC translates macros appearing organically in the 23 real-world C programs from which MacroBench was drawn. Translations were run in parallel on a server with 2x AMD EPYC 7742 64-Core and 512GB RAM. MerC took 9.5 days to analyze all programs, most of which was spent on the Linux kernel, which required 8.5 days. The median analysis time was about 2 minutes per program. After analysis, MerC took 2 minutes to convert all 8,211 macros, with a median translation time of 1 second. We check correctness by compiling the translated programs and running their built-in test suites. All 23 successfully compile and pass tests after MerC translation. Out of 100,711 total invocations, MerC finds and translates 8,211 of them (8%). This percentage is low partly because some macro definitions are defined in header files shared across compilation units, and so can't be translated without additional specifications for the semantics of whole program macro usage. Another reason is that MerC only translates macros that are not called within other macros; this limitation can be overcome by repeatedly using MerC to translate programs until no more translations occur. This approach would always terminate, as macros have bounded recursion [22].

7 Related Work

Translating C preprocessor macros. Mennie and Clarke described the first automated macro translation tool in 2004 [44]. Their tool is publicly unavailable, and only translates constant object-like macros into C variables and enums, whereas MerC also translates function-like macros into C functions. Macroscope [43] translates C preprocessor usage to the ASTEC language, but fails to help translate macro-laden C code to safer languages since ASTEC is itself a preprocessing language. Kumar et al. discussed "rejuvenating" C++ programs by translating C preprocessor macros to C++ [39]. Unlike MerC, their translation is not fully automatic, translates to C++ instead of C, and is sometimes incorrect because it ignores macro invocation semantics. c2rust supports translating some C preprocessor macros to Rust [10], but this feature has correctness issues [11, 58]. Visual Studio can refactor certain macro definitions into constant expression declarations [55], but this feature has bugs [57]. Macroni [52, 65] translates C preprocessor macros to LLVM's MLIR [40], but not to C.

Translating C preprocessor conditionals. Tools exist for translating C preprocessor conditionals [25, 41, 54, 56, 66], but they do not translate macros, and therefore suffer the same issues as other C translators. Such tools expand macros in C code before translating it, preventing downstream C translators from preserving macros in their translations. MerC could be used to translate macros first before translating preprocessor conditionals to enable macro-aware variability analysis and verification.

Analyzing C preprocessor usage. Badros and Notkin presented the first preprocessor-aware C source code analyzer, Pcp^3 [2], in 2000. Ernst et al. used PCp^3 to conduct the first empirical study of C preprocessor usage [15]. Although they found 75% of macros to be constants or expressions, this provides an incomplete picture of macro translatability since their analysis only considered macro definitions, and ignored invocation semantics. Dietrich presented CppSig for inferring macros' function signatures from their invocations [13], and inferred return types and argument types for over 50% of Linux kernel macro definitions. Pappas and Gazzillo constructed a framework of 26 Boolean properties for measuring macro portability, and built on CppSig to implement it as a tool called Maki [53]. MerC in turn uses Maki's properties as input for deciding if and how to translate macros to C language constructs.

Translating code with LLMs. There exist many code-generating LLMs [1, 16, 17, 28, 45, 47, 67, 70], with some models designed specifically for translation [71]. Pan et al. compared the effectiveness of seven models for translating code in seven datasets spanning four programming languages [51], and found GPT-4o to emit the greatest percentage of correct translations. Section 5.4 shows that the newer Claude 3.5 Sonnet and o1 Preview models boast even lower translation failure rates than GPT-4o, demonstrating how these more recent models improve on prior ones.

8 Data-Availability Statement

We make MerC, MacroBench, and our experimental scripts and results publicly available in our anonymized artifact[2]. The artifact contains all macros used to create MacroBench, their originating programs, how many of their invocations are included in MacroBench, and additional metadata. This metadata includes each tool's translation speed, whether it translated each macro correctly, reasons why failed macro translations were unsuccessful, and what C constructs each tool attempted to translate each macro to. The artifact also provides our zero-shot, few-shot and chain-of-thought prompt templates in `macrobench/prompt_templates.txt`.

References

[1] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars! arXiv:2301.03988 [cs.SE] https://arxiv.org/abs/2301.03988

[2] Greg Badros and David Notkin. 2000. A Framework for Preprocessor-Aware C Source Code Analyses. *Software Prac. Experience* 30 (July 2000). doi:10.1002/(SICI)1097-024X(20000710)30:8<907::AID-SPE324>3.3.CO;2-9

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[4] Elliot Chance. 2021. c2go. https://github.com/elliotchance/c2go.

[5] CISA. 2023. The Case for Memory Safe Roadmaps. https://www.cisa.gov/sites/default/files/2023-12/The-Case-for-Memory-Safe-Roadmaps-508c.pdf.

[6] Correct Computation Inc. 2021. 3C. https://github.com/correctcomputation/checkedc-clang/.

[7] Correct Computation Inc. 2021. 3C. https://github.com/correctcomputation/checkedc-clang/issues/400.

[8] Correct Computation Inc. 2021. 3C. https://github.com/correctcomputation/checkedc-clang/issues/40.

[9] Correct Computation Inc. 2021. 3C. https://github.com/correctcomputation/checkedc-clang/issues/439.

[10] Stephen Crane. 2020. Document and (potentially) stabilize –translate-const-macros. https://github.com/immunant/c2rust/issues/304.

[11] Stephen Crane. 2023. long double #defines incompatible with –translate-const-macros. https://github.com/immunant/c2rust/issues/829.

[12] DARPA. 2024. Translating All C TO Rust (TRACTOR). https://sam.gov/opp/1e45d648886b4e9ca91890285af77eb7/view.

[13] Christian Dietrich. 2021. *CppSig: Extracting Type Information for C-Preprocessor Macro Expansions.* Association for Computing Machinery, New York, NY, USA, 62–68. https://doi.org/10.1145/3477113.3487268

[14] 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

[15] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng.* 28, 12 (Dec. 2002), 1146–1170. doi:10.1109/TSE.2002.1158288

[16] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[17] Raymond Li et al. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL] https://arxiv.org/abs/2305.06161

[18] Dwayne Forde. 2024. Working with AI (Part 2): Code Conversion. https://blog.withmantle.com/code-conversion-using-ai/

[19] Free Software Foundation, Inc. 2025. Concatenation (The C Preprocessor). https://gcc.gnu.org/onlinedocs/cpp/Concatenation.html.

[20] Free Software Foundation, Inc. 2025. Macros (The C Preprocessor). https://gcc.gnu.org/onlinedocs/cpp/Macros.html.

[21] Free Software Foundation, Inc. 2025. Stringizing (The C Preprocessor). https://gcc.gnu.org/onlinedocs/cpp/Stringizing.html.

[22] Free Software Foundation, Inc. 2025. Stringizing (The C Preprocessor). https://gcc.gnu.org/onlinedocs/cpp/Self-Referential-Macros.html.

[23] Vrunda Gadesha. 2024. What is few shot prompting? https://www.ibm.com/think/topics/few-shot-prompting.

[24] Vrunda Gadesha and Eda Kavlakoglu. 2024. What is chain of thoughts (CoT)? https://www.ibm.com/think/topics/chain-of-thoughts.

[25] Florian Garbe. 2017. *Performance Measurement of C Software Product Lines.* Master's thesis. University of Passau.

[26] 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

[27] GitHub. 2024. Bringing developer choice to Copilot with Anthropic's Claude 3.5 Sonnet, Google's Gemini 1.5 Pro, and OpenAI's o1-preview. https://github.blog/news-insights/product-news/bringing-developer-choice-to-copilot/#anthropics-claude-3-5-sonnet.

[28] GitHub. 2024. Claude 3.5 Sonnet is now available in public preview. https://github.blog/changelog/2024-11-01-claude-3-5-sonnet-is-now-available-to-all-copilot-users-in-public-preview/.

[29] GitHub. 2025. GitHub Copilot. https://marketplace.visualstudio.com/items?itemName=GitHub.copilot.

[30] GitHub. 2025. Prototyping with AI models. https://docs.github.com/en/github-models/prototyping-with-ai-models#rate-limits.

[31] 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

[32] Immunant. 2018. c2rust. https://github.com/immunant/c2rust/issues/16.

[33] Immunant inc. 2023. c2rust. https://github.com/immunant/c2rust.

[34] Immunant inc. 2023. c2rust. https://github.com/immunant/c2rust/tree/3e0183e4d56f9d4d003f1ea2e9a814648debf307.

[35] ISO/IEC JTC 1. 2024. *Information technology – Programming languages – C.* Standard 9899:1990. International Organization for Standardization.

[36] JTC1/SC22/WG14. 2024. *Information technology – Programming languages – C.* Standard 9899:2024. International Organization for Standardization.

[37] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 805–824. doi:10.1145/2048066.2048128

[38] KratosLab. 2025. Prompt Engineering for OpenAI's O1 and O3-mini Reasoning Models. https://kratoslab.com/prompt-engineering-for-openais-o1-and-o3-mini-reasoning-models/.

[39] Aditya Kumar, Andrew Sutton, and Bjarne Stroustrup. 2012. Rejuvenating C++ programs through demacrofication. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. Institute of Electrical and Electronics Engineers, 445 Hoes Ln, Piscataway, NJ 08854, United States, 98–107. doi:10.1109/ICSM.2012.6405259

[40] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 445 Hoes Ln, Piscataway, NJ 08854, United States, 2–14. doi:10.1109/CGO51591.2021.9370308

[41] Alex Lazar and Jean Melo. 2017. C Reconfigurator. https://github.com/itu-square/c-reconfigurator.

[42] Yinheng Li. 2023. A Practical Survey on Zero-shot Prompt Design for In-context Learning. In *Proceedings of the Conference Recent Advances in Natural Language Processing - Large Language Models for Natural Language Processings (RANLP)*. INCOMA Ltd., Shoumen, BULGARIA, Shoumen, Bulgaria, 641–647. doi:10.26615/978-954-452-092-2_069

[43] Bill McCloskey and Eric Brewer. 2005. ASTEC: A New Approach to Refactoring C. *SIGSOFT Softw. Eng. Notes* 30, 5 (Sept. 2005), 21–30. doi:10.1145/1095430.1081712

[44] Christopher A. Mennie and Charles L. A. Clarke. 2004. Giving meaning to macros. *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* 12 (2004), 79–85. https://api.semanticscholar.org/CorpusID:26483462

[45] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474 [cs.LG] https://arxiv.org/abs/2203.13474

[46] OpenAI. 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/.

[47] OpenAI. 2024. Introducing OpenAI o1-preview. https://openai.com/index/introducing-openai-o1-preview/.

[48] OpenAI. 2024. Learning to reason with LLMs. https://openai.com/index/learning-to-reason-with-llms/.

[49] OpenAI. 2024. Overview - OpenAI API. https://platform.openai.com/docs/overview.

[50] Yoann Padioleau. 2009. Parsing C/C++ Code without Pre-Processing. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009* (York, UK) *(CC '09)*. Springer-Verlag, Berlin, Heidelberg, 109–125. doi:10.1007/978-3-642-00722-4_9

[51] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. 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 82, 13 pages. doi:10.1145/3597503.3639226

[52] Brent Pappas. 2023. Holy Macroni! A recipe for progressive language enhancement.

[53] Brent Pappas and Paul Gazzillo. 2024. Semantic Analysis of Macro Usage for Portability. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, 1601 Broadway, 10th Floor New York, NY 10019-7434, 229–240. doi:10.1145/3597503.3623323

[54] Zachary Patterson, Zenong Zhang, Brent Pappas, Shiyi Wei, and Paul Gazzillo. 2022. SugarC: scalable desugaring of real-world preprocessor usage into pure C. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 12 pages. doi:10.1145/3510003.3512763

[55] Augustin Popa. 2018. Convert Macros to Constexpr. https://devblogs.microsoft.com/cppblog/convert-macros-to-constexpr/.

[56] H. Post and C. Sinz. 2008. Configuration Lifting: Verification meets Software Configuration. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, USA, 347–350. doi:10.1109/ASE.2008.45

[57] Sanjeev Reddy. 2018. Auto refactor of a macro followed by a comment to a constexpr put the semicolon after the comment. https://developercommunity.visualstudio.com/t/auto-refactor-of-a-macro-followed-by-a-comment-to/354205.

[58] Remmirad. 2023. Problematic macro translation with –translate-const-macros. https://github.com/immunant/c2rust/issues/803.

[59] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. 2025. The Prompt Report: A Systematic Survey of Prompt Engineering Techniques. arXiv:2406.06608 [cs.CL] https://arxiv.org/abs/2406.06608

[60] Jamey Sharp. 2017. Corrode. https://github.com/jameysharp/corrode/tree/84c6d7b5dedb32093ada9420ebd9f020828af6c5.

[61] 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

[62] Diomidis Spinellis. 2003. Global Analysis and Transformations in Preprocessed Languages. *IEEE Trans. Softw. Eng.* 29, 11 (Nov. 2003), 1019–1030. doi:10.1109/TSE.2003.1245303

[63] Vrunda Gadesha Meredith Syed. 2025. What is zero-shot prompting? https://www.ibm.com/think/topics/zero-shot-prompting.

[64] The Linux Kernel. 2024. Linux kernel coding style. https://github.com/torvalds/linux/blob/master/Documentation/process/coding-style.rst#12-macros-enums-and-rtl. Section 12, Macros, Enums and RTL.

[65] Trail of Bits. 2023. Macroni. https://github.com/trailofbits/macroni.

[66] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming* 85, 1, Part 2 (2016), 125–145. doi:10.1016/j.jlamp.2015.06.007 Formal Methods for Software Product Line Engineering.

[67] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. doi:10.18653/v1/2021.emnlp-main.685

[68] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 57 Morehouse Ln, Red Hook, NY 12571, United States, 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

[69] White House Office of the National Cyber Director. 2024. Back to the Building Blocks: A Path Toward Secure and Measurable Software. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.

[70] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/3520312.3534862

[71] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 5673–5684. doi:10.1145/3580305.3599790