Semantic Analysis of Macro Usage for Portability

Brent Pappas
pappasbrent@knights.ucf.edu
University of Central Florida
Orlando, United States

Paul Gazzillo
paul.gazzillo@ucf.edu
University of Central Florida
Orlando, United States

ABSTRACT

C is an unsafe language. Researchers have been developing tools to port C to safer languages such as Rust, Checked C, or Go. Existing tools, however, resort to preprocessing the source file first, then porting the resulting code, leaving barely recognizable code that loses macro abstractions. To preserve macro usage, porting tools need analyses that understand macro behavior to port to equivalent constructs. But macro semantics differ from typical functions, precluding simple syntactic transformations to port them. We introduce the first comprehensive framework for analyzing the portability of macro usage. We decompose macro behavior into 26 fine-grained properties and implement a program analysis tool, called Maki, that identifies them in real-world code with 94% accuracy. We apply Maki to 21 programs containing a total of 86,199 macro definitions. We found that real-world macros are much more portable than previously known. More than a third (37%) are easy-to-port, and Maki provides hints for porting more complicated macros. We find, on average, 2x more easy-to-port macros and up to 7x more in the best case compared to prior work. Guided by Maki's output, we found and hand-ported macros in three real-world programs. We submitted patches to Linux maintainers that transform eleven macros, nine of which have been accepted.

CCS CONCEPTS

• **Software and its engineering** → *Semantics*; **Preprocessors**; **Automated static analysis**.

KEYWORDS

macros, C, program analysis

ACM Reference Format:
Brent Pappas and Paul Gazzillo. 2024. Semantic Analysis of Macro Usage for Portability. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3623323

1 INTRODUCTION

C is an unsafe language with millions of lines of critical software infrastructure implemented in it. Researchers have been developing (semi-)automated tools to port C to safer language, such as c2rust [25], 3c [6], and c2go [4]. Transforming the original, unpreprocessed code is hard, because C programs are written in two languages, C itself and the C preprocessor language, which is used extensively in real-world C programs [12]. Existing porting tools instead resort to first preprocessing the source file, then porting the preprocessed code, due to long-standing obstacles caused by preprocessor usage [7–9, 24].

The problem with preprocessing first is that the source code is barely recognizable after preprocessing. The following is an example of function macro usage from the lua [31] source:

```
1 #define ALPHABIT        0
2 #define MASK(B)         (1<<(B))
3 #define testprop(c,p) (luai_ctype_[(c)+1]&(p))
4 #define lislalpha(c) testprop(c,MASK(ALPHABIT))
5 if(lislalpha(ls->current))
```

While the highlighted call to `lislalpha` on line 5 looks syntactically like a function call, it is a macro invocation, which *expands* the macro, i.e., performs text substitution of the macro's definition on line 4 while also substituting its parameters into the macro body. After preprocessing, macro definitions are gone and the macro call is reduced to a lengthy series of arithmetic operations on magic constant values:

```
if((luai_ctype_[(ls->current)+1] & ((1 « (0)))))
```

This preprocessed source is what tools end up having to port, losing the macro function abstractions from the original source.

To preserve macro usage mixed with C, without having to preprocess it away, porting tools need to be able to understand macro usage before porting to equivalent constructs in the new language. For instance, some function-like macros, such as `MASK` on line 2 above, behave like C functions once the code is preprocessed and compiled. Indeed, prominent coding standards even recommend using a C function instead of a macro when it already behaves like a C function [5, 49]. Such macro usage has a straightforward transformation: create a function of the same name, infer the type of the arguments(s) [11], and put the macro body in a return statement. For instance, `MASK` would become the following, where the highlighted code is the exact contents of the original macro's body:

```
int MASK(int B) { return (1«(B)); }
```

But function-like macros, not part of the C language proper, do not always behave like C functions. Porting tools cannot, in general, apply the simple syntactic substitution used for `MASK` without first identifying whether it is correct to do so. Macro semantics differs from C [45] in calling convention and scoping rules; macros are call-by-name and have dynamic scoping, whereas C functions are call-by-value and have static scoping. For instance, `PREPEND_LIST`, from the bash source [16], resembles a C function, but it assigns a value to an argument, `elist`, a side-effect not possible with C's call-by-value calling convention alone:

```
#define PREPEND_LIST(nlist, elist) \
  do { nlist->next = elist; elist = nlist; } while (0)
```

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3623323

A porting tool converting this macro to a C function, for instance, would need to simulate the macro's call-by-name behavior by changing the argument type to a pointer and adding a dereference in the body to match the behavior of the original macro.

The transformations for `MASK` and `PREPEND_LIST` are *interface-equivalent*, i.e., the abstract functional specification of the macro and its C-function equivalent have the same behavior in terms of inputs and outputs. But in general, macros can freely violate and modify C syntax [20, 21, 27]. For instance, there is no interface-equivalent C function for a macro that expands to only a switch statement's case label, as in this example, also from lua:

```
#define vmcase(l) case l:
```

Porting this macro to C or any typical programming language would require redesigning the function interface and refactoring the code that invokes it, a substantially more complicated transformation than for the interface-equivalent macros `MASK` and `PREPEND_LIST`. Interface-equivalent macros, in contrast, are *easy-to-port* macros, since they require simpler, largely syntactic transformations.

With an automated program analysis of macro usage, porting tools would be able to determine what transformations preserve behavior. But prior work on analyzing macro usage for porting is limited to narrow cases. Mennie and Clarke find and port only parameter-less macros (called object-like macros [45]) that are equivalent to C constant variables [34]. Our evaluation shows that such macros comprise only 19%, on average, out of all the 21 program we evaluated, leaving behind a substantial amount of easy-to-port, interface-equivalent macros. The Visual Studio IDE can also convert macros to `constexpr` variables and functions [37], but this transformation is purely syntactic, suffering similar limitations [38] to Mennie and Clarke's work. This means that developers must manually check that the conversion is correct, since the transformed declaration's definition will at best have the same syntax as the original macro definition, but not necessarily the same behavior.

We introduce the Macro Inspector Framework, the first comprehensive framework for analyzing the portability of macro usage in C programs. Our framework enables the automated understanding of how macros affect the C program, so that porting efforts can determine the needs for transforming each macro. The key insight is that our framework takes into account the macro definition and all its invocation sites, comparing the source code both before and after preprocessing to identify how the macro affects the C program. The challenge is that macros have great freedom to alter the C AST in myriad ways, making it difficult to determine what specific transformation of the macro would preserve behavior. Therefore, we decompose the changes macros cause into a set of 26 fine-grained properties and design a set of program analyses to discover which properties hold for each macro. Using our analytical framework, we study the combinations of properties that enable interface-equivalent transformations and which properties need more complicated refactorings before porting, such as the `vmbreak` macro above.

To evaluate the Macro Inspector Framework, we implement its program analyses in a tool called Maki, which automatically identifies the set of properties held by each macro definition. We use Maki to study macro behavior in 21 real-world C programs, including several taken from a classic study of preprocessor usage [12] and 2 more modern programs: the Linux kernel and lua, with a total of 86,199 macro definitions. We found that, surprisingly, macros in real-world code are much more portable than previously understood. More than a third (37%) are easy-to-port, interface-equivalent macro definitions that have a one-to-one mapping to a C function. Compared to prior work, we find, on average, 2x more and, counterintuitively, programs with the most complicated macro usage often have even more easy-to-port, interface-equivalent macros compared to prior work, up to 7x more in the best case. Lastly, we find that syntactic macros, those that respect C syntax, are usually interface-equivalent and consequently, projects with relatively more syntactic macros have more easy-to-port, interface-equivalent macros. On a statistically significant sample of the benchmark macro definitions, Maki has 94% accuracy in identifying properties compared to hand-checked ground truth.

To evaluate the utility of the Macro Inspector Framework, we conducted several case studies hand-porting macros guided by Maki's output on two complete programs from our benchmark (m4 and enscript) and two Linux modules. Macros identified as interface-equivalent took only minutes to port, while non-interface-equivalent macros took substantially longer. We also submitted patches for a case study of 11 interface-equivalent macros in Linux source and reported on the discussions that led to the acceptance of 9 of them by developers. This demonstrates that our framework is useful for helping developers move away from macro usage, even in large, mature C codebases with formal review processes.

In this paper, we make the following contributions:

- An analytical framework for preprocessor macros that identifies the portability of macro usage (Section 2).
- Maki, a Clang plugin and Python library that implements program analyses to detect the framework's properties (Section 3).
- An evaluation of the macro usage and portability in 21 medium-to-large, real-world C programs (Section 4).
- Case studies of how we used the framework to hand-port macros in several example programs (Section 5).

Maki's complete source code is available online as free and open-source software[1] as well as in our publicly-available artifact [3].

2 THE MACRO INSPECTOR FRAMEWORK

We detail the syntactic and semantic properties of macro usage defined in the Macro Inspector Framework, illustrating them with examples. Table 1 lists each property, including its name and formal and informal definitions, that together comprise our framework. All formal definitions take a macro m that contains three fields, tokens of the unpreprocessed macro invocation (*tokens*), the preprocessed macro as an AST if syntactically-valid C (*ast*), and the macro's preprocessed arguments as an AST if syntactically-valid (*args*).

We group properties into *portability categories* that provide guidance to porting tools about what language features are implicated in the macro usage and need to be supported in order to port the macro. *Calling-convention-adapting* properties concern macro behavior that can be ported by adapting the arguments of the function when ported to a C-like function. *Scope-adapting* properties involve the use of dynamic scope, and so require modifications to ensure

[1] https://github.com/appleseedlab/macro-analyzer

Property	Formal Definition	Description
	Interface-Equivalent	
Calling-Convention-Adapting		
Modified body	$\exists e \in \textsc{SideEffectedExprs}(ast), m.ast = e$	Expands to a side-effected expression
Modified arguments	$\exists a \in m.args, \exists e \in \textsc{SideEffectedExprs}(ast), a.ast = e$	Has a side-effected argument
Addressed body	$\exists e \in \textsc{AddressedExprs}(ast), m.ast = e$	Expands to an addressof (&) operand
Addressed arguments	$\exists a \in m.args, \exists e \in \textsc{AddressedExprs}(ast), a.ast = e$	Has an argument that expands to an addressof (&) operand
Unhygienic	$\exists d \in \textsc{LocalDeclRefs}(m.ast), \forall a \in m.args, \neg\textsc{InTree}(d, a.ast), \neg\textsc{DeclIn}(d, m.ast)$	Captures a declaration from macro m's caller's environment
Scope-Adapting		
Locally defined	$\neg\textsc{DefinedIn}(m, \textsc{GlobalScope}(ast))$	Defined in a local scope
Unordered declarations	$\exists d \in \textsc{DeclRefs}(m.ast), \textsc{DefinedBefore}(m, d)$	References a declaration defined after macro m
Unordered expansion type	$\textsc{DefinedBefore}(m, \textsc{Type}(m.ast))$	Expands to an expression whose type is defined after macro m
Unordered type declarations	$\exists t \in \textsc{TypeRefs}(m.ast), \textsc{DefinedBefore}(m, t)$	References a type defined after macro m
Unordered argument types	$\exists a \in m.args, \textsc{DefinedBefore}(m, \textsc{Type}(a.ast))$	Has an argument that expands to an expression whose type is defined after macro m
Unordered macros	$\exists n \in \textsc{NestedMacros}(m), \textsc{DefinedBefore}(m, n)$	Invokes a macro defined after macro m
Condition macro	$\exists c \in \text{CPP}\textsc{Conditionals}, \textsc{InCondition}(m.name, c)$	Is invoked in a CPP conditional
Anonymous type	$\exists e \in \textsc{AnonTypeExprs}(ast), m.ast = e$	Expands to an expression whose type is unnamed
Anonymous argument types	$\exists a \in m.args, \exists e \in \textsc{AnonTypeExprs}(ast), a.ast = e$	Has an argument that expands to an expression whose type is unnamed
Local argument types	$\exists a \in m.args, \exists e \in \textsc{LocalTypeExprs}(ast), a.ast = e$	Has an argument that expands to an expression whose type is defined in a local scope
Locally-typed subexpressions	$\exists e \in \textsc{LocalTypeExprs}(m.ast), e \in \textsc{SubExprs}(m.ast)$	Contains a subexpression whose type is defined in a local scope
Local type	$\exists e \in \textsc{LocalTypeExprs}(ast), m.ast = e, \neg\textsc{DeclIn}(\textsc{Type}(e), m.ast)$	Expands to an expression whose type is defined in a local scope
	Non-Interface-Equivalent	
Thunkizing		
Void arguments	$\exists a \in m.args, \textsc{Type}(a.ast) = \texttt{void}$	Has a void expression argument
Side-effecting arguments	$\exists a \in m.args, \exists e \in \textsc{SideEffectExprs}(m.ast), \textsc{InTree}(e, a.ast)$	Has an argument with side-effects
Callsite-context-altering		
Unaligned	$m.ast = null \vee \exists a \in m.args, a.ast = null$	Does not align with a single AST node
Conditional arguments	$\exists a \in m.args, \exists e \in \textsc{CondExprs}(m.ast), \textsc{InTree}(a.ast, e)$	Has an argument that is conditionally evaluated in the body of macro m
Nested		
Nested in body	$\exists n \in \textsc{AllInvocations}, m \in \textsc{NestedMacros}(n)$	Is invoked in the body of another macro
Nested in argument	$\exists n \in \textsc{AllInvocations}, \exists a \in n.args, m \in \textsc{NestedMacros}(a)$	Is invoked as an argument to another macro invocation
Metaprogramming and Generics		
Control flow	$m.ast \in \{\texttt{return}, \texttt{case}, \texttt{continue}, \texttt{break}, \texttt{goto}\}$	Alters caller's control flow
Non-expression arguments	$\exists a \in m.args, a.ast \notin \textsc{Exprs}(ast)$	Has argument that is not an expression
Stringizing / Token-pasting	$\# \in m.tokens \vee \#\# \in m.tokens$	Uses stringification or token-pasting

Table 1: Macro invocation properties.

Name	Description
AddressedExprs(a)	Set of expressions in AST a that are the operand of an addressof (&) expression.
AllInvocations	Set of all macro invocations.
AnonTypeExprs(a)	Set of expressions in AST a whose type is unnamed.
DeclIn(d, a)	Whether declaration d was declared in AST a.
DeclRefs(a)	Set of expressions in AST a that are references to declarations.
DefinedBefore(m, d)	Whether macro m is defined before declaration or macro d.
DefinedIn(d, s)	Whether declaration or macro d is defined in scope s.
CondExprs(a)	Set of short-circuiting expressions (e.g., the ternary operator, or logical and) in AST a.
CPPConditionals	Set of CPP static conditionals (e.g., `ifdef`, `defined`, etc.) in program.
GlobalScope(a)	Global scope of AST a.
InCondition(s, c)	Whether symbol s appears in CPP static conditional c.
InTree(a, b)	Whether AST a is a subtree of b.
LocalDeclRefs(a)	Set of expressions in AST a that are references to declarations declared in local scopes.
LocalTypeExprs(a)	Set of expressions in AST a whose type is defined at local scope.
NestedMacros(m)	Set of macro invocations or arguments in m's nested invocations.
SideEffectExprs(a)	Set of expressions in AST a with side-effects.
SideEffectedExprs(a)	Set of expressions in AST a that are modified by an assignment expression or the unary increment or decrement operator, or are passed to a function call.
Type(e)	Type of expression e.
TypeRefs(a)	Set of expressions in AST a that reference a type declaration.

Table 2: Helper functions used in formal definitions of properties.

static scope equivalence. *Definition-adapting* macros are those, like `MASK` (Section 1), that require only a syntactic change and involve no changes to calling convention or scoping.

Together, macros that are calling-convention-, scope-, or definition-adapting have a one-to-one equivalence with a C-like function and need only minimal changes to port away from preprocessor usage, i.e., they are *interface-equivalent*. Interface-equivalent macros are relatively easy to port, since they behave like C functions, so identifying macros that are interface-equivalent should help developers more easily port away from macro usage. In contrast, non-interface-equivalent macros require a redesign of the macro's functional interface, representing more complicated and difficult-to-port macro usage. We group properties of such macros into *Thunkizing* for those that require converting expressions to functions, i.e., thunks, *Call-site-context-altering* for those that alter the syntax of the macro call-site, *Nested* for nested macro usage, and *Metaprogramming* for macros that perform code generation.

2.1 Interface-Equivalent Properties

The `PREPEND_LIST` macro (from Section 1) causes side-effects on the value of its argument `elist`, which is not supported in C or other languages with call-by-value semantics but is with call-by-reference semantics. We call this behavior the *Modified arguments* property (or *Modified body* when it occurs in the macro body) and provide a formal definition in Table 1. A related macro behavior, also not supported by call-by-value, is the use of the C address-of operator (&). For example, the `linkgclist` macro below (from lua) takes the address of its `o` and `p` arguments:

```
1 #define linkgclist(o,p) \
2   linkgclist_(obj2gco(o), &(o)->gclist, &(p))
```

Argument values and addresses have function-local scope, so the address will not be the same as the parameter passed to the function call with call-by-value semantics.

The *Unhygienic* macro property stems from the same lack of macro scoping that affects address-of. For example, `ISSET` (from gawk [15]) expands to an expression that captures the local variable `sp` (highlighted in gray) from its caller's environment, not possible with statically-scoped functions.

```
1 #define ISSET(opt) (sp->fts_options & (opt))
2 void f() {
3   FTS *sp;
4   if (ISSET(FTS_LOGICAL))
5 }
```

We define unhygienic macros as those that capture symbols from the function-local scope and use them in the expanded macro invocation site. When the macro itself is defined inside of the function-local scope it is *locally-defined*. The formalization checks whether the macro has been defined in any scope but the global scope.

The *unordered declarations, types, type declarations, argument types, and macros* properties all stem from the dynamic scoping of macros. Rather than requiring symbols to exist at macro define time, as in statically-scoped languages, macros need not check for other symbols being defined until they are invoked. For example, `open_spline` (line 1), from xfig [42] takes a symbol `s` that has type `F_spline`, which is declared on line 2 after `open_spline`:

```
1 #define open_spline(s) (!(s->type & 0x1))
2 typedef struct f_spline F_spline;
3 void update_spline(F_spline *spline) {
4   if (open_spline(spline));
5 }
```

A *condition macro* is one which is present in a preprocessor static conditional, e.g.., within an `#ifdef`'s condition. The preprocessor checks if macros are defined, but not functions, so a porting tool

handling the condition macro needs to also alter the original program so that all preprocessor conditionals relying on that macro will behave the same after the transformation.

Macros can expand to *anonymous types* and *argument types*. For example, the macro `TB_FLAGS` defined in fvwm [18] expands to an anonymously-typed expression:

```
1 typedef struct { struct { ... } flags; } TitleButton;
2 #define TB_FLAGS(tb)                ((tb).flags)
3 static void SetLayerButtonFlag(..., TitleButton *tb) {
4     TB_FLAGS(*tb).has_layer = 1; ...
5 }
```

The invocation of `TB_FLAGS` on line 4 expands to the expression `((*tb).flags)`, which is described by the anonymous struct type as highlighted in gray.

Related properties exist for invocations that expand to locally-typed expressions (*local type*), accept locally-typed expressions as arguments (*local argument types*), or contain *locally-typed subexpressions*, because macro invocations are untyped and can capture type symbols from the caller's scope. Functions, however, cannot, because function type signatures only have globally-defined types available in scope.

2.2 Non-Interface-Equivalent Properties

Non-interface equivalent macros do not have a one-to-one mapping to C-like function behavior and represent the most complicated uses of macros, requiring refactoring of the original use of the program before mapping them to functions. For instance, macros may take *void arguments*, i.e., expressions that have no return value, such as a call to a void function. But void arguments are not legal as parameters to C functions. Similarly, when an argument to a macro causes a side effect (*side-effecting arguments*), every use of that argument repeats the effect due to call-by-name semantics, whereas in C's call-by-value calling convention, the side effect is only computed once. For instance, macro `min`, from emacs [17], expands the same argument multiple times:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
```

If invoked with `min(x++, y++);`, the macro would increment either `x` or `y` twice, whereas a syntactically-similar C function would only trigger the increment once, when the arguments get evaluated.

Macro invocations operate at the raw token level and thus do not need to expand to or accept complete syntactic constructs, e.g. statements or expressions, as arguments. When a macro's expansion does not correspond to a complete syntactic construct, we identify it as *unaligned*. Take the following macro expansion:

```
1 #define ADD(a, b) a + b
2 4 * ADD(5, 6)
```

A function version of `ADD` would evaluate the addition first, leaving the multiplication, `4 * 11`, while the macro version results in an evaluation of the multiplication first, due to operator precedence on the expanded expression, `4 * 5 + 6`, a common pitfall in preprocessor usage [46]

If a macro invocation expands one of its arguments into a short-circuiting expression (e.g., a ternary expression or logical conjunction expression), then it has *conditional arguments* and may never evaluate that argument. For instance, the macro `AND` expands to `((0) && (*x))` which, due to short-circuiting, would never raise a null pointer fault in spite of the dereference of NULL with `*x`:

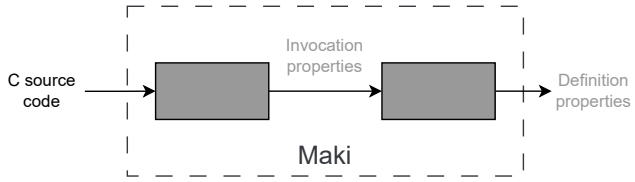

Figure 1: Maki architecture diagram.

```
1 #define AND(a, b) ((a) && (b))
2 int *x = NULL;
3 AND(0, *x);
```

If ported to a function with call-by-value semantics, in contrast, then the `*x` dereference would always raise the fault.

A macro can alter its caller's *control flow* directly by expanding to any of the `continue`, `break`, `goto`, or `return` keywords. An expansion to a `goto`, for instance, pierces the function abstraction by enabling the callee to return to any point in the caller, because the `goto` will be expanded by the macro into the body of its caller. *Nested macros* are those that are called by other macros. This may occur in the definition of the macro, for instance `lislalpha` in Section 1, or in a parameter at the macro's invocation site. Unlike functions, macros do not need to be passed expressions as arguments and can instead be passed any syntactic construct, such as statements, declarations, or even unaligned constructs. These *non-expression arguments* are not supported in C functions. Macro invocations can use the *stringification* and *token-pasting* operators to manipulate their arguments' tokens, which generate string literals and fresh language tokens at macro expansion time. This kind of code generation allows for metaprogramming, such as reflection to print error messages or system-dependent type names.

3 IMPLEMENTATION

We have implemented the Macro Inspector Framework in a new tool called Maki which performs automated program analysis to identify macro usage properties. Maki consists of a Clang plugin comprised of 2,180 lines of C++ code; and 1,519 lines of Python code. Figure 1 presents Maki's architecture. The Clang plugin analyzes macro invocations in individual C source files, so to analyze a complete program we first intercept its build system using Clang's `intercept-build` utility and use the resulting compile commands to analyze all files in the program. These results can then be passed to Maki's Python scripts to determine which properties apply to each of the program's macro definitions.

The Clang plugin works by hooking into the Clang preprocessor and AST to determine which macros are syntactically aligned. Maki has methods that implement each property's formalism by inspecting the C AST and a trace of preprocessor expansions. The Python scripts read these results into memory and use this information to determine which properties each macro definition satisfies. Based on these results, we can determine which portability category applies to each macro definition. Maki's source code is available both on GitHub [1] and in our public artifact [3].

4 EVALUATION

We use Maki to analyze real-world macro usage portability and answer the following research questions:

- **RQ1** (Portability) How much portability is there in real-world macro usage?
- **RQ2** (Comparison) How does Maki's ability to find portable macros compare to that of prior work?
- **RQ3** (Alignment and Portability) How does syntactic macro usage influence portability?
- **RQ4** (Runtime) How quickly does Maki analyze macro usage?
- **RQ5** (Accuracy) How accurately does Maki identify properties?

All summary data is in our free, publicly-available artifact [3].

4.1 Benchmark Program Selection

We draw 19 programs to analyze from Ernst et al.'s analysis of C preprocessor usage [12][2] but omit the remaining seven which either contain C++ code[3] or which we could not build due to missing dependencies[4]. To augment the benchmarks, we also add 2 very large, modern programs as well: the Linux kernel and Lua. Across all programs in our benchmark, there are 86,199 macro definitions.

4.2 Experimental Setup

We ran our experiments on a server with 2 AMD EPYC 7742 64-Core hyper-threaded processors, for a total of 256 available processes at one time, with 512GB RAM. We download and extract each program, and intercept its build system with Clang's `intercept-build` tool to obtain the specific compile commands used to compile each source file. We pass each of these compile commands to Maki's Clang plugin to determine which properties each macro invocation in the program satisfies. We run the Clang plugin on eight cores for all programs except for Linux, for which we use 32 cores due to the kernel's size. Finally, we run Maki's Python scripts to determine which properties apply to each macro definition. We only examine invocations of macros defined in the programs themselves and not invocations of macros defined in any system header or library files, since they are not part of the application. We reimplement the macro usage analysis from Mennie and Clarke, which ports only constant, object-like macros, in Maki, and measure how often they occur in each program.

4.3 RQ1: Portability

We use Maki to evaluate each of the macro definitions across all programs in our benchmark suite, recording which properties hold for each macro. For each macro, we use its properties to assign it to a portability category, as indicated in Table 1. Figure 2 presents our results. Each program has a segmented bar chart that represents the percentage distribution of macro definitions in each portability category. When there are multiple invocations of the same definition that have differing portability categories, we record them as having either multiple interface-equivalent properties or multiple non-interface equivalent. The interface-equivalent portability categories are color-coded by blue shades, whereas non-interface equivalent are yellow shades. The programs are sorted by the highest percentage of interface-equivalent macro definitions.

Programs have widely-varying macro portability, ranging from 12% to as much as 76% interface-equivalent macros. The largest program with the most (57,896) macro definitions is Linux with 41% being easily-portable. On average, over all 86,199 macros, 37% are easy-to-port, being interface-equivalent. The program analyses of macro semantics in the Macro Inspector Framework has enabled us to discover that macros in real-world code are much more portable than previously understood.

4.4 RQ2: Comparison

We measure how many macros are identified as portable according to our reimplementation of Mennie and Clarke's [34] macro usage analysis and compare the number of these macros to the number of interface-equivalent macros identified by our framework's properties implemented in Maki. Figure 3 presents the relative performance of Mennie and Clarke's tool against Maki, by dividing the number of interface-equivalent macros over the constant, object-like macros identified by Mennie and Clarke. We sort the results by most to least increase in identified, portable macros. Prior work transforms an average of 19%, a minimum of 3%, and a maximum of 61% of macro definitions across all the programs we study. On average, Maki finds twice as many portable macros, ranging from 1.21x to 7.04x more across all programs.

Comparing Figure 2 to the relative performance results, we see that some of the greatest improvement in portable macro identification occurs in programs with some of the most complex macros usage. For instance, perl, in which Maki identifies 7.04x more portable macros, has some of the most non-interface-equivalent macro definitions. This indicates that complex macro usage obscures macro portability, and Maki's analyses help tease out the semantic aspects of macro usage in order to identify easily-portable macros.

4.5 RQ3: Alignment and Portability

The C preprocessor is lexical and has no requirement to respect C syntax. This research question evaluates the portability of C macro usage when only considering syntactic macro use to see the impact on portability. To measure syntactic macro usage, we measure what C syntax the macro usage generates (statements, expressions, etc.) and whether it respects C syntax or violates it, which corresponds to the Unaligned macro property (Table 1).

Figure 4 presents the percentage of each program's macro definitions that align with the program's AST, shown in blue, compared with its percentage of interface-equivalent macro definitions, shown in gold. Across all programs, the percentage of aligned definitions necessarily exceeds the percentage of interface-equivalent definitions, because interface-equivalence depends on syntactic macro usage (the Unaligned property precludes interface-equivalence).

In general, the majority of syntactic macro definitions are easily-portable in most programs. Linux, for instance, has a very large portion that are interface-equivalent, likely because of the good macro usage coding guidelines encouraged by maintainers [49]. In some programs, however, only a minority of syntactic macros are easily-portable. For instance, gnuplot has many more aligned

[2] bash, bc, bison, cvs, emacs, enscript, flex, fvwm, gawk, gnuplot, gv, gzip, m4, mosaic, perl, rcs, remind, xfig, and zsh
[3] gcc, ghostscript, and gnuchess
[4] zephyr, workman, and RasMol

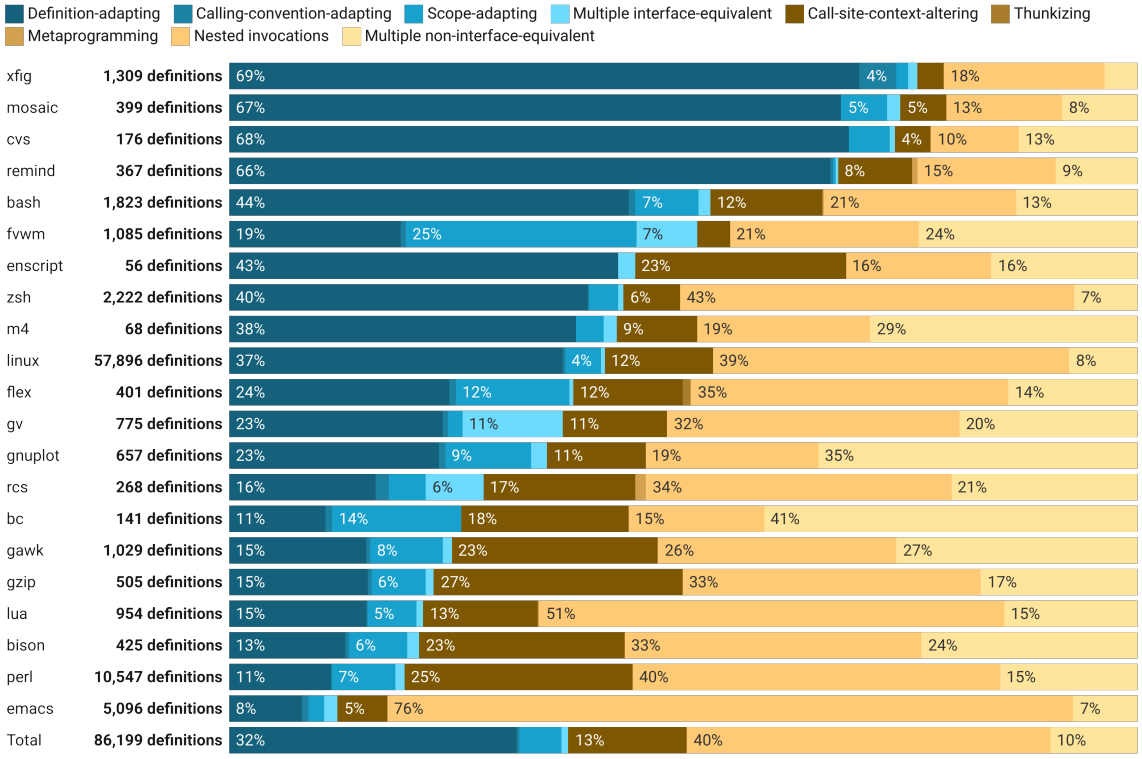

Figure 2: Percent of macro definition adaptation categories.

definitions than interface-equivalent ones because 8% of its definitions align with types (more than any other program), and due to limitations with Clang we cannot fully analyze the semantic properties of type-aligned invocations and conservatively report them as not interface-equivalent.

4.6 RQ4: Runtime

We measure the end-to-end runtime to analyze all definitions' properties and portability categories for each program. Across all 21 programs in our benchmark, the median time necessary to analyze all macro usage was about nine minutes and 20 seconds. The analysis of Linux took the longest, requiring about 17 days and 21 hours, although it was run with more threads than the rest of the programs. In total, it took about 19 days and eight hours to fully analyze all 86,199 macros across all programs.

4.7 RQ5: Accuracy

We measure Maki's accuracy by comparing it against ground truth for a statistically significant sample from the benchmark. A 383-macro sample from the 86,199 macros in our benchmark has a 5% margin of error and 95% confidence. We created ground truth for the sample by hand-checking all portability categories shown in Table 1 for each macro. We compute the precision ($\frac{\text{true positives}}{\text{true positives} + \text{false positives}}$) and recall ($\frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$), then we compute accuracy as the F_1 score (harmonic mean of precision and recall). Maki has a true positive when it reports the same portability categories as ground truth, a false positive when the portability categories differ. It has a true negative when the absence of portability categories matches ground truth, a false negative if it fails to report portability categories from ground truth.

Out of the 383-macro sample, Maki had 337 true positives, 17 false positives, and 26 false negatives, resulting in a precision of 95%, a recall of 93% and an F_1 accuracy of 94%.

Seven false positives were due to the macro expanding to a designated initializer [47] or a statement expression [48], GCC-specific C language extensions that Maki does not yet support. Maki appears to have aligned the ten remaining false positive macros with incorrect AST subtrees, leading to misclassifications. Maki failed to match 23 of the false negatives with any portability category at all, and failed to identify the remaining three as having nested invocations. By careful manual inspection we identified correct portability levels for all these macros, and therefore attribute these errors to limitations in Maki's implementation rather than the underlying framework. We plan on adding these misclassifications to Maki's test suite so that we may resolve them.

A spreadsheet of the sample detailing ground truth creation can be found in our publicly-available artifact [3].

5 CASE STUDIES

We present three case studies to demonstrate the capabilities of the Macro Inspector Framework and of Maki. First, we run Maki on Linux's driver staging directory, which contains drivers under development, and hand-transform several macros Maki identifies

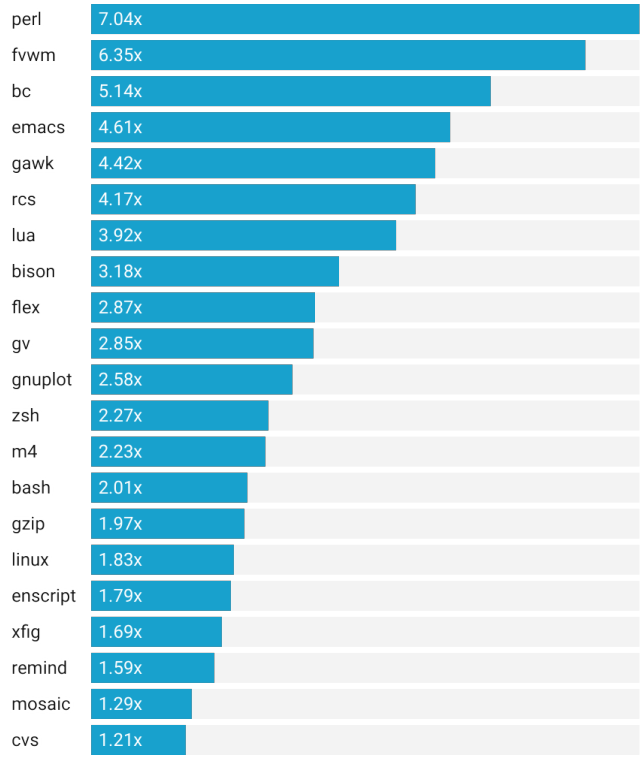

Figure 3: How many more easy-to-port, interface-equivalent macro definitions Maki finds over prior work.

as easy-to-port. We submit these changes as patches to the Linux Kernel Mailing List for approval [30] and report on the discussion that led to their acceptance or rejection by developers. This case study shows the viability of using Maki to help development on an actively-maintained, large-scale, critical C codebase that has a rigorous review process.

Second, using Maki's output as a guide, we transform by hand all macros in two programs from our benchmarks, enscript and m4, as well as two Linux modules, `ipc` and `sound/atmel`. This case study shows the utility of Maki to help quickly port away from macro usage.

5.1 Patching Linux Kernel Macro Usage

Modifying the Linux kernel requires negotiating a careful and sometimes lengthy patch review process with the Linux developers [52]. We therefore target a select set of macros in the `driver/staging/` directory [50] as candidates for patching in the Linux kernel. We selected the first 11 macros that Maki identifies as definition-adapting; definition-adapting are the simplest to port, requiring only a change of syntax from a macro definition to a C function with no further refactoring. For example, `GDM_TTY_READY` from `gdm724x/gdm_tty.c` evaluates a conditional expression for if-statements:

```
#define GDM_TTY_READY(gdm) \
  (gdm && gdm->tty_dev && gdm->port.count)
struct gdm *gdm = tty_dev->gdm[index];
if (!GDM_TTY_READY(gdm));
```

Porting this macro requires only copying the macro body to the new function's return statement. Since Maki confirms definition-equivalence, this transformation is safe for all invocations:

```
static inline bool gdm_tty_ready(struct gdm *gdm) {
  return gdm && gdm->tty_dev && gdm->port.count;
}
```

In total, we submitted eight patches transforming 11 macros. As of writing, Linux maintainers have accepted six of our patches which port nine macros. The remaining two macro transformations were rejected, because the maintainer preferred keeping the macro. In one case, the maintainer preferred permanently inlining it, making porting unnecessary.

Discussion with the maintainer on two of the ported macros led to safer code: First, `FPNTBL_BYTES` from `media/atomisp/pci/sh_css_params.c` contained unsafe multiplication susceptible to overflow. The maintainer requested using the `array3_size` helper, which computes size without overflowing [51]. Second, `irq_data_to_gpio_chip()` from `greybus/gpio.c` expanded to a `void *` expression. But the maintainer observed the macro's return value is only ever assigned to variables of type `struct gpio_chip *` and requested the new function have this return type. All patches we submitted, along with their mailing list histories, are included in the publicly-available artifact [3].

5.2 Porting All Macros in a Codebase

We hand-ported macros to C functions in two codebase from our benchmarks, m4 and enscript. For each codebase, we used Maki's output to produce a table of all macros it encountered, sorted by the portability categories listed in Table 1. We transformed each of these macros to C, one-at-a-time, starting with the interface-equivalent, which are the easiest to port. After each transformation, we ensured the program still built ('make') and passed tests ('make check').

As shown in Figure 2, m4 has 68 macro definitions and enscript has 56. Roughly half of macro definitions are of interface-equivalent macros in both codebases, 29 for m4 and 25 for enscript. These macros were fast and easy to transform by hand, taking one author only 20 minutes for enscript and 15 minutes for m4 to transform all the interface-equivalent macros. Maki enables this speed, because it rules out complex macro behavior for these macros, allowing for a simple syntactic conversion.

The non-interface-equivalent macros were more time-consuming. These macros (39 in m4 and 31 in enscript) took about 3:40 hours and 4 hours total to transform by hand, respectively. The challenge in porting is determining how to preserve the macro's non-functional behavior in C. For instance, the macro `__P`, defined in enscript, is conditionally defined to either expand to a given parameter list or an empty parameter list.

```
#if PROTOTYPES
#define ___P(protos) protos
#else /* no PROTOTYPES */
#define ___P(protos) ()
extern char *strerror ___P ((int));
\\ 49 more function declarations.
```

It is used to remove the parameter list for older versions of C. We converted each unique function signature into a conditionally-declared typedef to include or exclude parameters.

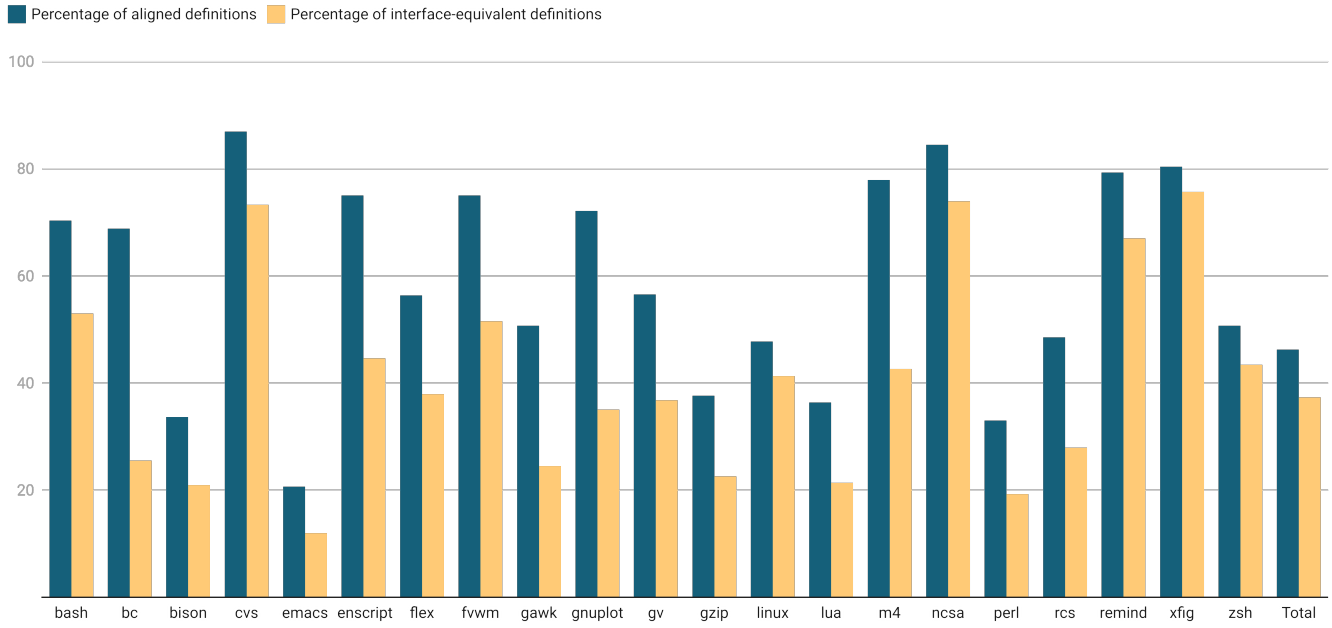

Figure 4: Percentage of aligned and interface-equivalent definitions in each program.

We found that Maki's results had two false negatives in enscript and 12 in m4. The two in enscript and eight in m4 were due to Maki's lack of support for ad-hoc polymorphism and variadic arguments in macro invocations. The remaining four Maki identified as calling-convention-adapting because they modified a bit-field argument. The C standard prohibits taking the addresses of bit-fields [26], a subtlety of C that Maki does not take into account.

We also hand-ported two whole Linux modules, `ipc` and `sound/atmel`, each containing 49 and 55 macros, respectively, taking three hours total. 91 macros were interface-equivalent while the remaining 13 were non-interface-equivalent. We transformed 102 out of the 104 macros defined in both modules; the two macros we did not transform were header guard macros. We checked the changes by compiling the modules and running Linux's checker, `sparse`, with `make C=2`. We have not submitted patches for these changes to Linux maintainers because they are significantly larger in scope than the patches discussed in Section 5.1, and Linux maintainers prefer to review series of small patches rather than large patches [41].

The changes made to all transformed programs and detailed notes can be found in our publicly-available artifact [3].

6 MAKI FOR CODE GENERATION

In addition to guidance for hand-porting, Maki can help inform mechanical transformation approaches by providing guarantees about macro usage. When function-like macros are definition-adapting, the semantics of the macro are identical to a C function, requiring only simple syntactic changes to place the macro body in a return statement and to replace the `#define` with a C function prototype, as shown in the Linux Case Study (Section 5.1). All other interface-equivalent macros are either calling-convention-adapting or scope-adapting, i.e., they differ from functions in calling-convention (call-by-name vs. call-by-value) or scoping.

When macros are calling-convention-adapting, i.e., they exploit macros' call-by-name semantics, generating C code versions of the macros only requires simulating call-by-name with call-by-value. For instance, when there are side-effects on a macro's argument, such as `PREPEND_LIST` (Section 1), the call-by-name convention means the updated value of the argument is reflected in the caller. Such behavior can be simulated by passing references to the arguments instead of their values, e.g., passing a pointer in C, passing-by-reference in C++, or passing an object in Java.

Scope-adapting macros exploit the lack of any scope, besides global, and the dynamic scoping of macro definitions. Macros' lack of scoping enables them to capture symbols from their callers' scope as seen in the use of the function-local `sp` variable in the `ISSET` macro (Section 2) without explicitly passing a parameter. Maki's ability to identify scope-adapting properties would enable automated code generation tools to identify what refactorings are needed to produce equivalent functions. For macros that exploit the lack of scoping, the code generation tool need only identify any undeclared symbols in the macro body that are in the caller's scope, then add these as parameters, akin to a function extraction refactoring [32]. Macros that exploit dynamic scoping only need reorderings of declarations by the code generation tool to ensure any macros called in the body are available in the static scope.

As the hand-ported macro case study shows (Section 5.2), non-interface-equivalent macros are more complicated to port, because they are not guaranteed to correspond one-to-one with a function. They instead require first refactoring the functional interface to

match the language's function semantics before transforming the macro. For instance, when macro arguments have side-effects, e.g., a macro takes `x++` as an argument, this side-effect is repeated every time the argument is used. A code generation tool can simulate this behavior by refactoring the argument into a thunk, which preserves the side-effect throughout the execution of the macro body and reflects the final value in the caller's scope.

Even though functions take expression parameters and function calls themselves are expressions, macros may take and return any grammar construct, including declarations and control-flow altering statements such as `goto` or `break`. Barring language-specific extensions to functions such as C's *longjmp*, a code generation tool will likely not be able to simulate these behaviors with functions in general. For nested macros, however, there is more hope for code generation tools. When the outer macro is interface-equivalent, it may be possible to first convert the outer macro definition, thereby unnesting the inner macro. After porting, the inner macro is now an outer macro that can have its properties checked for portability.

7 THREATS TO VALIDITY

Internal validity. The properties are intended cover all macro usage. If any one macro is not analyzed, we might miss portability. To ensure completeness, we designed the properties so that when a macro meets none of the properties, it is interface-equivalent (the definition-adapting portability category). To ensure that the implementation of our macro analysis framework in Maki matches the formal properties, we developed a test suite to exercise each property, made from hand-crafted examples and tests adapted from benchmarks. In total, our test suite comprises 63 files and 895 source lines of code. We used this test suite to make Maki highly accurate, as Section 4.7 demonstrates. We intend to add the misclassified macros we found while hand-checking Maki's output against ground truth in Section 4.7 to Maki's test suite so that we may address them as well. Finally, due to a limitation with Clang, Maki is unable to check if type-aligned macro invocations satisfy any scope-adapting properties, so we conservatively assume all type-aligned invocations are scope-adapting. Since scope-adapting macros are interface-equivalent, even if this issue were resolved Maki would still find the same quantity of interface-equivalent macros.

External validity. The results about macro usage and portability depend on the set of benchmark programs chosen. To achieve a wide variety of program types, we started with Ernst et al.'s preprocessor metrics benchmarks, which has programs from many domains, including languages, utilities, shells, etc. To broaden the range and size of programs, and to compensate for older programs no longer under development, we added new benchmarks. Our framework is geared towards C preprocessor macros, but other macro systems, such as for Rust [28] and Lisp [23] have different semantics. Some properties from our framework would apply, such as scoping differences, but applying our framework to other macro systems would require adjusting and adding properties, which we leave as future work. Moreover, porting tools that target high-level language constructs may have one-to-one mappings for non-interface equivalent macros; for instance, languages with first-class functions could map thunkizing macros using anonymous functions. Similarly, we do not support analyzing macros used in C++ programs, which could have different properties, e.g., related to its additional language constructs that would affect how often macros are easily-portable. On the other hand, porting to object-oriented languages could also open the door to more types of macro transformations, which we leave to future work.

8 RELATED WORK

The most recent and only related macro property analyzer is Mennie and Clarke's [34] automated tool for transforming certain object-like macros to C variables. Their tool collects facts about macros, classifies them based on these facts, and then generates plans for transforming each macro based on its facts and classifications. It employs sophisticated rules for inserting transformed code, which enable it to automatically transform certain object-like macros into correctly-scoped local variables. Maki's design is similar to theirs in that it first collects properties about macros, and then categorizes macros based on the properties they satisfy. Maki differs from Mennie and Clarke's work in that it does not automatically transform macros, and analyzes a wider array of macro properties to find many more easily portable macros.

SugarC [1] transforms preprocessor usage to C by targeting the preprocessor static conditionals and convering them into runtime C conditions. SugarC improves variability-aware analyses of programs that use preprocessor static conditionals for configuration management, but does not try to maintain developer abstractions in the transformed code it produces. Hercules [19] is another tool that transforms CPP compile time conditionals to C runtime conditionals. Unlike SugarC, it uses an AST generated by Typechef [27] to perform its transformation. C Reconfigurator's [29] transformation rules were proven to be sound, but only for a theoretical language that is a subset of what it actually transforms.

McCloskey and Brewer [33] developed Macroscope to transform CPP macros to new a macro preprocessor language, called ASTEC. ASTEC has advantages over CPP, but presents all the same issues to porting tools as CPP code since it is still a preprocessor language. Moreover, while Macroscope can merge duplicate transformed definitions into single definitions, their rules for placing transformed definitions are not as rich as those of Mennie and Clarke's tool [34].

The C preprocessor analysis tool most relevant to this project is Dietrich's CppSig [11], which collects macro invocations into tree structures, and infers function signatures for them. Maki relies on insights akin to those behind CppSig to find AST-aligned macro invocations and infer types for invocations' expansions and arguments. The seminal work in CPP analysis is that of Badros and Notkin [2], which outlines a method of C source code analysis that offers both preprocessing and C parsing "actions" that are analogous to Clang's preprocessor callbacks [43] and AST [44]. This is similar in spirit to the approach taken by Maki, since it uses both Clang preprocessor callbacks and the Clang-generated AST to perform its analysis. The first large study of macro usage was conducted by Ernst et al. [12], from which we drew many of our benchmarks programs. CScout [40] enables the analysis of "program families" [40], i.e., workspaces comprised of interdependent programming languages. CScout can perform simple refactorings on C program families such as identifier renaming, but cannot identify all the easily portable macros that Maki does.

Favre's [13] is the only work we know of to fully formalize the preprocessor itself. He outlined a denotational semantics for CPP, taking into account subtle features such as stringification and tokenization [45]. Since Favre does not consider macro's interaction with C code, it is unrelated to the AST-oriented properties of our the Macro Inspector Framework.

There have been several proposals for hygienic alternatives to preprocessor macros for C [22, 33, 54], including the aforementioned ASTEC [33]. These macro languages are syntactic, and require that all macro invocations align with the program's AST; however they are still preprocessor languages that similarly hinder porting. Rust [39] offers syntactic macro where arguments are annotated with their AST-node type. Syntactic macro systems have a home in the Lisp family of programming languages [14, 53]. Pombrio and Krishnamurthi [36] demonstrated the feasibility of reconstructing abstractions in Lisp-like languages.

3C [6] ports C programs to Checked C [35], a pointer-safe dialect of C designed to prevent memory bugs. Macro usage has impaired 3C porting efforts [7–9]. Maki could be used to help develop porting tools to mitigate the challenge. c2rust [25] translates C code to unsafe Rust code. Like 3C, c2rust faces problems with preprocessor macros [24] and preprocesses first, losing macro abstractions.

9 CONCLUSION

In this work we present the Macro Inspector Framework and its embodiment in the Maki analyzer. Compared to prior work, our implementation finds an average of twice as many easily portable macros, and up to 7x more for programs with more complex macro usage. Using our framework as a guide, we hand-patched 11 linux kernel macros, nine of which have been accepted by kernel maintainers. In future work, we will study the application of our framework's properties when used to port macros to other languages besides C. Ultimately, we plan to leverage our framework to create an automated tool for porting easy-to-port macros, so that developers can focus their porting efforts on more complex definitions.

ACKNOWLEDGMENTS

We would like to thank Mike Hicks, Elaine Weyuker, and all the reviewers for their valuable feedback. This work was supported in part by NSF grants CCF-1840934 and CCF-1941816.

REFERENCES

[1] Appleseed Lab. Sugarc. https://github.com/appleseedlab/superc/tree/master/src/superc/cdesugarer, 2022.
[2] Badros, G., and Notkin, D. A framework for preprocessor-aware c source code analyses. *Software Practice and Experience 30* (07 2000).
[3] Brent Pappas, P. G. Artifact for semantic analysis of macro usage for portability. https://zenodo.org/record/8326488, 2023.
[4] Chance, E. c2go. https://github.com/elliotchance/c2go, 2021.
[5] CMU SEI SERT Team. Pre00-c. prefer inline or static functions to function-like macros. https://wiki.sei.cmu.edu/confluence/display/c/PRE00-C.+Prefer+inline+or+static+functions+to+function-like+macros, Apr 2022.
[6] Correct Computation Inc. 3c. https://github.com/correctcomputation/checkedc-clang/, 2021.
[7] Correct Computation Inc. 3c. https://github.com/correctcomputation/checkedc-clang/issues/400, 2021.
[8] Correct Computation Inc. 3c. https://github.com/correctcomputation/checkedc-clang/issues/40, 2021.
[9] Correct Computation Inc. 3c. https://github.com/correctcomputation/checkedc-clang/issues/439, 2021.
[10] Datawrapper GmbH. Datawrapper: Create charts, maps, and tables. https://www.datawrapper.de/, 2023. Used to create charts.
[11] Dietrich, C. *CppSig: Extracting Type Information for C-Preprocessor Macro Expansions.* Association for Computing Machinery, New York, NY, USA, 2021, p. 62–68.
[12] Ernst, M. D., Badros, G. J., and Notkin, D. An empirical analysis of c preprocessor use. *IEEE Trans. Softw. Eng. 28*, 12 (dec 2002), 1146–1170.
[13] Favre, J.-M. Cpp denotational semantics. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation* (2003), pp. 22–31.
[14] Flatt, M. Composable and compilable macros: You want it when? In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2002), ICFP '02, Association for Computing Machinery, p. 72–83.
[15] Free Software Foundation. Gawk v5.1.1. https://www.gnu.org/software/gawk/, 2021.
[16] Free Software Foundation. Bash v5.2 rc1. https://www.gnu.org/software/bash/, 2022.
[17] Free Software Foundation. Emacs v28.1. https://www.gnu.org/software/emacs/, 2022.
[18] FVWM Team. Fvwm v2.6.9. https://www.fvwm.org/, 2019.
[19] Garbe, F. Performance measurement of c software product lines. Master's thesis, University of Passau, 2017.
[20] Garrido, A., and Johnson, R. E. Analyzing multiple configurations of a c program. *21st IEEE International Conference on Software Maintenance (ICSM'05)* (2005), 379–388.
[21] Gazzillo, P., and Grimm, R. Superc: Parsing all of c by taming the preprocessor. *SIGPLAN Not. 47*, 6 (jun 2012), 323–334.
[22] Gosling, J. Ace: a syntax-driven c preprocessor. *Australian Unix Users Group* (1989).
[23] Graham, P. *On Lisp.* Prentice Hall, 1993.
[24] Immunant. c2rust. https://github.com/immunant/c2rust/issues/16, 2018.
[25] Immunant. c2rust. https://github.com/immunant/c2rust, 2022.
[26] ISO Technical Committee ISO/IEC JTC 1/SC 22. ISO-IEC-9899-2011. Standard, International Organization for Standardization, Dec. 2011.
[27] Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. 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* (New York, NY, USA, 2011), OOPSLA '11, Association for Computing Machinery, p. 805–824.
[28] Klabnik, S., and Nichols, C. Macros - the rust programming language. https://doc.rust-lang.org/book/ch19-06-macros.html, Nov 2022.
[29] Lazar, A., and Melo, J. C reconfigurator. https://github.com/itu-square/c-reconfigurator, 2017.
[30] Linux Kernel Maintainers. Linux kernel mailing list. https://lore.kernel.org/.
[31] Lua Team. lua. https://github.com/lua/lua, 2022.
[32] Martin Fowler, K. B. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., USA, 1999.
[33] McCloskey, B., and Brewer, E. Astec: A new approach to refactoring c. *SIGSOFT Softw. Eng. Notes 30*, 5 (sep 2005), 21–30.
[34] Mennie, C., and Clarke, C. Giving meaning to macros. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* (2004), pp. 79–85.
[35] Microsoft Research. 3c. https://github.com/Microsoft/checkedc/, 2021.
[36] Pombrio, J., and Krishnamurthi, S. Resugaring: Lifting evaluation sequences through syntactic sugar. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, Association for Computing Machinery, p. 361–371.
[37] Popa, A. Convert macros to constexpr. https://devblogs.microsoft.com/cppblog/convert-macros-to-constexpr/, Jun 2018.
[38] Reddy, S. 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, Oct 2018.
[39] Rust Team. rust. https://github.com/rust-lang/rust, 2022.
[40] Spinellis, D. Global analysis and transformations in preprocessed languages. *IEEE Trans. Softw. Eng. 29*, 11 (nov 2003), 1019–1030.
[41] Sterba, D. Re: [patch] fs: Ntfs read-write driver gpl implementation by paragon software. https://lore.kernel.org/linux-fsdevel/20200815190642.GZ2026@twin.jikos.cz/.
[42] Sutanthavibul, S., Yap, K., Smith, B. V., King, P., Boyter, B., Sato, T., and Loimer, T. Xfig v3.2.8b. https://mcj.sourceforge.net/, 2021.
[43] The Clang Team. clang::ppcallbacks class reference — clang 16.0.0git documentation. https://clang.llvm.org/doxygen/classclang_1_1PPCallbacks.html, Nov 2022.
[44] The Clang Team. Introduction to the clang ast — clang 16.0.0git documentation. https://clang.llvm.org/docs/IntroductionToTheClangAST.html, 2022.
[45] The GCC Team. The c preprocessor. https://gcc.gnu.org/onlinedocs/cpp/, 2022.
[46] The GCC Team. Operator precedence problems. https://gcc.gnu.org/onlinedocs/cpp/Operator-Precedence-Problems.html#Operator-Precedence-Problems, Nov 2022.

[47] The GCC Team. Designated inits. https://gcc.gnu.org/onlinedocs/gcc/Designated-Inits.html, 2023.
[48] The GCC Team. Statement exprs. https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html, 2023.
[49] The Linux Kernel. Linux kernel coding style. https://github.com/torvalds/linux/blob/master/Documentation/process/coding-style.rst, 2023. Section 12, "Macros, Enums and RTL".
[50] The Linux Kernel. Linux Kernel drivers/staging, 2023.
[51] The Linux Kernel. Memory Allocation Guide. https://www.kernel.org/doc/html/latest//core-api/memory-allocation.html, 2023.
[52] The Linux Kernel. Submitting patches: the essential guide to getting your code into the kernel. https://github.com/torvalds/linux/blob/master/Documentation/process/submitting-patches.rst, 2023.
[53] The Racket Team. Racket. https://racket-lang.org/, 2022.
[54] Weise, D., and Crew, R. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1993), PLDI '93, Association for Computing Machinery, p. 156–165.