CppSig: Extracting Type Information for C-Preprocessor Macro Expansions

Christian Dietrich
Technische Universität Hamburg
Germany
christian.dietrich@tuhh.de

Abstract

For decades, the C programming language proved to be a cornerstone of system-software ecosystems, leaving us with billion lines of existing source code. From today's perspective of object-oriented and functional languages, C itself seems rather limited in its expressiveness and abstractive power. However, with the C preprocessor (CPP) as its companion, macros, which operate on the raw token stream, allow for abstractions that are impossible to achieve within the language itself. While its flexibility and its ease of use make CPP attractive for programmers, its potential undisciplined usage makes it problematic for static source-code analysis and can slow down the on-boarding of new developers.

In this paper, we focus on a disciplined subclass of CPP macros: the statement-like and expression-like macros, which mimic regular C functions, with well-type C expressions as arguments and, in case, a return value. We show how to spot such macros and their arguments in the compiler's abstract syntax tree, whereby it becomes possible to deduct type signatures for individual macro expansions. With our CppSig prototype, implemented as a Clang plugin, we extract macro-type information from Linux 5.12, whereby it becomes easier to understand even deep macro-expansion nests. In the future, these expansion signatures could be used to statically enforce gradually-typed CPP macro definitions.

CCS Concepts: • **Software and its engineering → Preprocessors**; **Data types and structures**.

ACM Reference Format:
Christian Dietrich. 2021. CppSig: Extracting Type Information for C-Preprocessor Macro Expansions. In *11th Workshop on Programming Languages and Operating Systems (PLOS '21), October 25, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3477113.3487268

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLOS '21, October 25, 2021, Virtual Event, Germany
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8707-1/21/10…$15.00
https://doi.org/10.1145/3477113.3487268

1 Introduction

Due to its relative closeness to the hardware and its portability to new platforms, the C programming language is often chosen as substrate to grow system-software stacks on. And although the core language itself has a rather limited feature set (simplistic type system, no modules, no classes or objects, …), it comes in good company: The *C preprocessor (CPP)*. The CPP, which is used in almost all C (and C++) projects, brings a highly flexible meta-programming model to the table, which allows for modularization (`#include`), static configurability (`#ifdef`), and syntactic abstractions (`#define`). For this, the CPP manipulates the raw token stream before the actual parsing happens, whereby the view on the program can vastly diverge between developers and the compiler. While this attracted a lot of criticism from academia [5, 8, 16, 18, 20], and several disciplined replacements where proposed [7, 12, 15, 22], developers do not expect the CPP to disappear in the near future[13].

For operating-system developers, the CPP is especially attractive since its abstractions are static by nature and provoke no run-time overhead on their own. For example, the Linux developers sprinkled the v5.12 source base with over 104 000 `#ifdef`-directives[1] that, among other things, implement the chosen static Kconfig configuration on the fine-grained source-code level [3, 11]. And although conditional compilation, often known as the "`#ifdef`-hell", has already attracted a lot of attention [6, 8, 18, 20], it is by far not the most commonly used CPP feature in Linux 5.12: For every `#ifdef`, there are already three `#include`-directives, and even 31 macro definitions (> 3 million).

These macros impose an immense burden on the readability of the source code: Even if looking only at a single configuration (x86, `defconfig`) and if we ignore all macro expansions in header files, there are more than 360 000 top-level macro expansions with on average 5 nested expansions. Even worse, we identified *macro-expansion nests* below a single top-level macro expansion with up to 637 expansions and with a maximum nesting depth of 15 (both expansions are found in the i915 graphics driver). With this level of CPP usage, we should equip developers with tool support to better understand CPP macros and their expansions.

[1] `git grep '#if\|#elif\|#else' '**/*.[ch]' | wc -l`

With the CppSig approach, we tackle the CPP problem by combining information from the preprocessing phase and the semantic-analysis phase: For every macro expansion, even the nested ones, we search the compiler's *abstract syntax tree (AST)* for nodes that can be traced back to the expansion. If those nodes form unambiguously-typed AST subtrees, we can infer the C-level types for the macro *expansion* and its arguments, and deduct a type signature. In the future, we want to summarize multiple expansions and deduct a common (possibly polymorphic) macro-*definition* signature. Furthermore, we also want to allow for gradual typing [17] of CPP macros, where strict type rules are enforced on some but not on all macro parameters and/or the return type.
The contributions of this paper are as follows:

1. With the CppSig approach, we identify statement-like and expression-like macro expansions in the abstract syntax tree and deduct (even nested) expansion signatures.
2. We provide a CppSig prototype as a Clang plugin, which can be applied to unmodified C source-code bases.
3. We demonstrate our approach on Linux v5.12.

The rest of the paper is structured as following: In Sec. 2, we briefly explain CPP macros, before we describe our approach in Sec. 3. After evaluating our prototypical implementation in Sec. 4 with Linux 5.12, we discuss the related work in Sec. 5, and conclude this paper in Sec. 6.

2 Background

The *C preprocessor (CPP)* is part of the C compilation process and provides three main features: file inclusion (`#include`), conditional compilation (`#ifdef`), and macros definition/expansion (`#define`). To perform its duties, the CPP filters and manipulates the raw token stream that is produced by the lexer. On the expanded token stream, the parser executes the syntactic analysis and conceptually builds up a parse tree, which covers all preprocessed tokens as leafs, to check the syntactic validity of the input program. Nowadays, instead of actually building a full-blown parse tree, modern compilers directly produce the condensed *abstract syntax tree (AST)*, which only contains the relevant program elements, which the following semantic analysis enriches with type information and cross-tree references (usage → declaration). While file inclusion [1] and conditional compilation [6, 8, 20, 21] are important topics on their own, this paper puts a focus on CPP macros.

Each macro definition begins with the keyword `#define` (see Fig. 1, line 1), followed by the *macro name* (e.g., `inner`). After the name, an optional parameter list of untyped identifiers enclosed in parentheses follows. Such parameterized macros are called *function-like* since their later usage looks like a function call. The rest of the definition contains the *macro body*, which is a list of regular tokens and parameter identifiers.

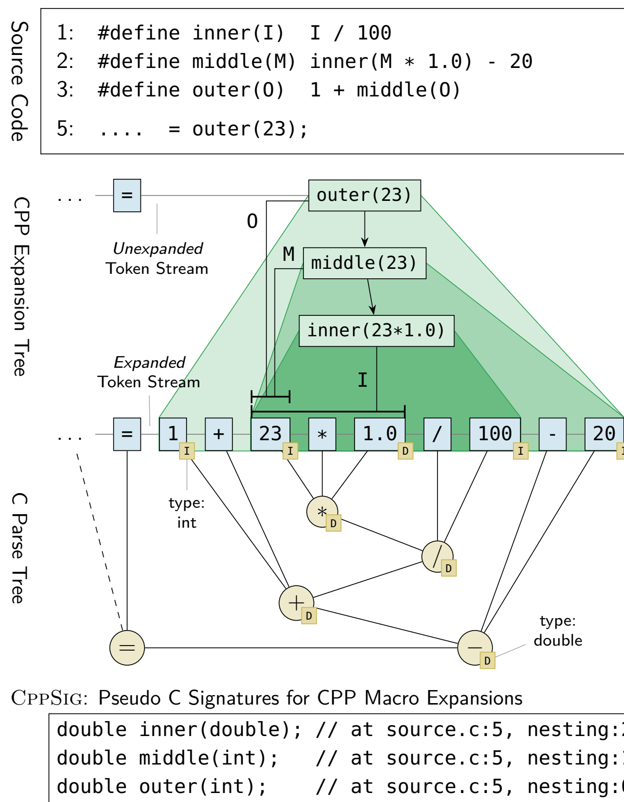

Figure 1. Example of Nested Macro Expansion. During macro expansion, the C preprocessor replaces tokens in the lexer's token stream, whereby argument tokens are directly inserted or passed on to nested macros. The parser builds up the parse tree from the *expanded* token stream, performs type and other semantic checks, and produces a binary. If expansion tree and parse tree align, CppSig can deduct type information for individual macro *expansions*.

Later on, if the CPP encounters an identifier in the token stream, it consults an internal table of currently active macro definitions and starts, in case, the *macro-expansion* process: For function-like macros, the CPP consumes the arguments as token lists and replaces all parameter references in the macro body with the corresponding argument token list. In the pre-expanded body, it furthermore searches for further macro identifiers and performs *nested* macro expansions until the *top-level macro* is fully expanded. Please note that the CPP does not support recursive macro definitions and that arguments are not necessarily used in the expansion.

In Fig. 1, the macro `outer()` has `middle()` as a nested macro, which by itself uses `inner()` within its macro body. We also see that `outer()` passes on its parameter `O` to `middle()`, which then passes it on, after appending "∗ 1.0", to `inner()`. It is noteworthy that CPP macros can only pre- or append tokens to their arguments if passed on to nested expansion, but regular argument-token lists cannot be split

up again. Therefore, the arguments of deeper-nested macro expansions can only cover more tokens in the *expanded token stream* than their parent expansions (see `M` vs. `I` in Fig. 1).

Another important aspect for the analysis of macro expansions is the distinction between *spelling location* and *expansion location* as tokens are *copied* during the expansion process. The spelling location of a token identifies where the lexer encountered a token originally. If a token is copied during an expansion, we attach the invocation point of the macro as another expansion location. Therefore, each token has one spelling location and a stack of expansion locations whose height corresponds to the macro-nesting depth. For example, in Fig. 1, the token "`/`" in the expanded token stream has its spelling location in line 1, and three expansion locations in line 2, line 3 and line 5.

3 Macro-Expansion Types

The goal of CppSig is to deduct return and parameter types for CPP macros to foster the understanding of macro-heavy source bases. Furthermore, inferring these types is a necessary precondition for gradually typing [17] CPP macros, which would increase the confidence in complex macros constructs.

First, it is important to understand that CPP macros *definitions* have no type on their own, but they only gain typing information in an expansion context for a specific set of arguments. One could even say that the CPP language is symbiotic with the C language as it enriches the later one with flexibility but relies on the C parser to validate its output. Therefore, we can only look at individual macro *expansions* and infer their typing information. In a later step, one could automatically combine the inferred *expansion signatures* to a more general definition signature. But this is a point for further research.

The second important observation is that CPP is a generator in the language-theoretical sense, while the C parser is an acceptor. As both languages, CPP and C are context free[2], preprocessor and parser result in derivation trees: the *macro-expansion tree* and the *parse tree*. As both operate on the same linear token stream, we can attach those trees on opposite sides of the token stream (Fig. 1 and Fig. 2). By folding one side over, we can match both tree structures and infer which AST nodes and subtrees stems from which macro expansion.

For the type extraction, we have to look at the AST's typing information: For *expression nodes*, the semantic analysis calculates and propagates types from the AST leaves upwards to inner nodes (see `I` and `D` in Fig. 1). Therefore, if the root of a subtree is an expression node, the whole subtree has a "result" type, which we assign to its counterpart in the macro-expansion tree.

[2] If we ignore such ugly details like `typedef` for the moment, which requires the usage of a context-sensitive lexer.

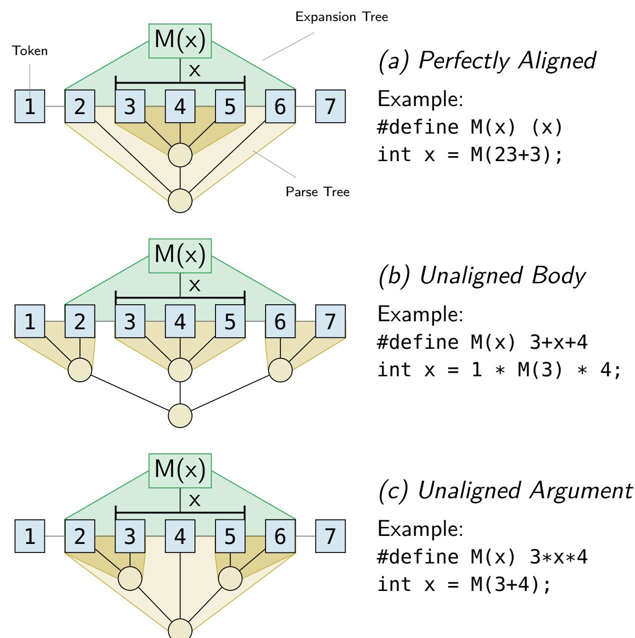

Figure 2. (Mis-)Alignment of Macro-Expansion and Parse Tree. We deliberately use latent buggy macro definitions here to briefly demonstrate misaligned tree structures.

3.1 Expansion-Tree Alignment

However, due to the lexical nature of CPP, the matching between both trees can be incomplete in the sense that an expansion corresponds to multiple (neighboring) AST subtrees (see Fig. 2). While the literature calls this undisciplined [5, 10, 16] CPP usage, we argue that such mismatches might be intentional[3] and we should gracefully handle such misalignments and extract as much useful typing information as possible: For example, in Fig. 2 b, we still can match the macro argument unambiguously with a subtree, even if the rest of the expansion is unaligned. Fig. 2 c shows the inverse situation, where the expansion as a whole aligns but its argument expansion remains unaligned.

For the matching, we extend the preprocessor to record the expansion tree with one node for each macro-expansion. We also save the invocation location for each macro and for each macro arguments we record the spelling location of all passed tokens.

First, we find AST subtrees that correspond to a macro expansion (see Fig. 3): For each AST node, we walk the expansion-location stack from the outermost macro invocation location to the innermost one and thereby narrow down the forest of macro-expansion trees to a single expansion node. At this expansion node, we collect all matched nodes in a subtree collector (`collect_subtrees()`), which only records the parent node if both the parent and the child

[3] For example, Protothreads [4] use undisciplined CPP macros to extend C with rudimentary co-routine support.

```
def match_expansions(ast : AST, exp_roots : List[Expansion]):
    for node in ast.nodes:
        expansion = None
        exp_nodes = exp_roots
        # Narrow down to an expansion node
        for loc in node.expansion_loc_stack
            for exp in exp_nodes:
                if exp.invocation_loc == loc:
                    expansion = exp
                    break
            if not expansion:
                break
            exp_nodes = expansion.children
        # Collect the AST nodes at the found expansion nodes
        # and arrange them in AST subtrees
        if expansion is not None:
            collect_subtrees(expansion, node)
```

Figure 3. Matching of AST with the Expansion-Tree Forest

node are collected. As a result we have one (or multiple) AST subtrees per macro expansion.

In order to identify the macro arguments for an expansion, we visit the found AST nodes and match them with their spelling location against the list of recorded argument-token spelling locations. Again, we arrange the matching nodes in one or multiple AST subtrees with a subtree collection. Thereby, we collect one (or multiple) subtrees for each macro-expansion argument.

By relying on source-location information, instead of tracking the actual lexer tokens, CppSig not only works with a parse tree, where all tokens are actual leaf nodes, but also on the condensed *abstract syntax tree (AST)*. As long as the compiler tracks not only the spelling location of an AST node, but also an expansion-location stack, our matching method can be applied. Since this information is useful to provide macro-aware error messages, it should already be available in modern compilers.

Result of the matching process is a forest of AST subtrees for each macro expansion and for each of their arguments. From these subtrees, we infer language-level types for the expansion itself (return type) and its arguments: If the forest is a single *expression* subtree or if all subtree roots have the same type, we infer an unambiguous macro type. If we find multiple subtrees with different root-node types, the macro type was ambiguous (see Fig. 4, example 1). In situations, where we have found one (or multiple) statement nodes, we infer `void` as a return type. If no subtrees were found or if we only found declarations, we cannot deduct type information on the C level. In Fig. 4, we show three challenging macro expansions that are already supported by CppSig.

3.2 Prototypical Implementation in Clang

We have implemented the described approach as a plugin for the Clang compiler [9], which already tracks spelling

```
// 1. Ambiguous Macro-Argument Type
#define with_ctx(var) (var + ctx.var)
struct ctx_t { double a; } ctx;
int a;
with_ctx(a)
// CppSig: double with_ctx({int, double} a)

// 2. Macros Names as Arguments
#define outer(macro) { macro(0x00), macro(0x80) }
#define inner(i) (i ^ 0x7f)
int table[] = outer(inner);
// CppSig: int[2] outer(? macro)
// CppSig: int    inner(int i)

// 3. Polymorphic Macro Parameter
#define lock(lockable) do { (lockable)->nesting++; } while(0)
struct foo_t { int nesting; } foo_obj;
struct bar_t { int nesting; } bar_obj;
lock(&foo_obj);
lock(&bar_obj);
// CppSig: void lock(struct foo_t * lockable)
// CppSig: void lock(struct bar_t * lockable)
```

Figure 4. Examples of Challenging Macro Expansions

locations and stacks of expansion locations at its AST nodes. We use the callback interface of Clang's preprocessor to record a macro-expansion tree and search for AST nodes with the `ASTMatcher` interface. Our plugin is available[4] and we provide a Docker image for easy exploration of CPP code.

4 Evaluation

We ran our approach against the Linux kernel in version 5.12[5]. We chose Linux as this source base makes heavy use of macros to support variability and portability to different architectures. We used the `defconfig` configuration for the `x86` architecture but had to disable a few features to make the kernel compile with Clang. Our evaluation machine was equipped with an AMD Ryzen 7 PRO 5850U octa-core CPU with 16 hardware threads and 48 GiB of RAM. We only looked at macros expansions in source files and skipped all expansions in header files, as these are potentially included by multiple source files.

4.1 Time Overhead for the AST Matching

We recorded the time required to match the AST against expansion tree, which is the main source of run-time overhead that the CppSig prototype induced the for the 2554 compilation units. On average, the matching took 2.7 s with a standard deviation of 63 s. As the median run time, we observed 366 ms per compilation unit.

[4]https://collaborating.tuhh.de/e-exk4/projects/cpp-macro-types

[5]Git Hash: `9f4ad9e425a1d3b6a34617b8ea226d56a119a717`

The large mean matching time and the large standard deviations stems from some files that took very long to process. 82 percent of the processed files took less than 1 second to complete, while the longest-running file (`net/mac80211/airtime.c`) took 53 minutes. After inspection, we can attribute this to deeply nested expansion trees that cover a large number of AST nodes. Since our prototype relies on Clang's `ASTMatcher`, which does not allow cutting off subtrees in the depth-first search, our current implementation has at least a quadratic run time.

In the future, we plan to improve CppSig's overhead by implementing a specialized AST-matching visitor that avoids inspecting subtrees if they certainly stem from the same expansion. Together with optimized source-location–indexing data-structures, this should bring CppSig's overhead closer to linear run time.

4.2 Example: `raw_spin`-Macros

As an exemplary output of CppSig, Tab. 1 shows the inferred types for all expansions (top-level and nested) of macros that begin with "raw_spin". We see that CppSig inferred the same type signature for most macro definitions. For example, all 1422 expansions of `raw_spin_lock_irqsave` were detected to have the same signature. This shows that our existing prototype provides insightful and unambiguous type signatures for most macro expansions.

However, in a few cases, CppSig reported different signatures for the same definition: For `raw_spin_{un,}lock_rcu_node`, expansions with and without `const` were found, which is an example of (valid) polymorphic macro usage. The other anomaly was detected for `raw_spin_lock_init`, where CppSig reported an ambiguous signature for a single expansion site. Manual inspection revealed that this happened in a nested expansion, where our argument matching algorithm failed to identify the correct subtree. Nevertheless, grounded in our experience with applying CppSig to Linux, we believe that this is only a problem with our prototype and not inherent to the CppSig approach.

Another interesting inference happened for the parameter `subclass` of `raw_spin_lock_irqsave_nested`, where CppSig did not report a type. In our kernel configuration, this argument is not used in the expansion, whereby it cannot be found in the AST and it actually has no C type.

4.3 Top-Level Macro Expansions

In the following, we will only look at top-level macro expansions as these are of special importance for the vast majority of kernel developers. For Linux 5.12, we found 363 269 top-level expansions that originate from 43 018 macro definitions. Of these expansions, 142 861 stem from 7519 function-like macro definitions. 58 percent of the function-like macro expansions matched with a single expression subtree. 32 percent matched with multiple nodes, which can, for example,

#Exp.	Expansion Signature	
`#def`	`raw_spin_lock(lock)`	(*172* exps.)
172	void macro(struct raw_spinlock *)	
`#def`	`raw_spin_unlock(lock)`	(*201* exps.)
201	void macro(struct raw_spinlock *)	
`#def`	`raw_spin_trylock(lock)`	(*18* exps.)
18	int macro(struct raw_spinlock *)	
`#def`	`raw_spin_lock_irq(lock)`	(*138* exps.)
138	void macro(struct raw_spinlock *)	
`#def`	`raw_spin_is_locked(lock)`	(*67* exps.)
67	int macro(struct raw_spinlock *)	
`#def`	`raw_spin_lock_init(lock)`	(*47* exps.)
46	void macro(struct raw_spinlock *)	
1	void macro({struct raw_spinlock *|struct raw_spinlock **})	
`#def`	`raw_spin_unlock_irq(lock)`	(*168* exps.)
168	void macro(struct raw_spinlock *)	
`#def`	`raw_spin_lock_nested(lock, subclass)`	(*39* exps.)
39	void macro(struct raw_spinlock *, int)	
`#def`	`raw_spin_lock_irqsave(lock, flags)`	(*1422* exps.)
1422	void macro(struct raw_spinlock *, unsigned long)	
`#def`	`raw_spin_lock_rcu_node(p)`	(*8* exps.)
7	void macro(struct rcu_node *)	
1	void macro(struct rcu_node *const)	
`#def`	`raw_spin_trylock_irqsave(lock, flags)`	(*11* exps.)
11	int macro(struct raw_spinlock *, unsigned long)	
`#def`	`raw_spin_unlock_rcu_node(p)`	(*11* exps.)
10	void macro(struct rcu_node *)	
1	void macro(struct rcu_node *const)	
`#def`	`raw_spin_trylock_rcu_node(p)`	(*2* exps.)
2	void macro(struct rcu_node *)	
`#def`	`raw_spin_unlock_irqrestore(lock, flags)`	(*306* exps.)
306	void macro(struct raw_spinlock *, unsigned long)	
`#def`	`raw_spin_lock_irq_rcu_node(p)`	(*6* exps.)
6	void macro(struct rcu_node *)	
`#def`	`raw_spin_lock_irqsave_nested(lock, flags, subclass)`	(*2* exps.)
2	void macro(struct raw_spinlock *, unsigned long, ?)	
`#def`	`raw_spin_unlock_irq_rcu_node(p)`	(*10* exps.)
10	void macro(struct rcu_node *)	
`#def`	`raw_spin_lock_irqsave_rcu_node(p, flags)`	(*12* exps.)
12	void macro(struct rcu_node *, unsigned long)	
`#def`	`raw_spin_unlock_irqrestore_rcu_node(p, flags)`	(*19* exps.)
19	void macro(struct rcu_node *, unsigned long)	

Table 1. Expansion Signatures for all "`raw_spin`" macros. Signatures are noted as "ret-type macro(arg-types)", `void` indicates that the macro expanded to one or multiple untyped statements, ambiguously inferred types are noted as `{T1|T2}`, and missing argument subtrees as "?".

`Parameter Type`	#Parms.	Parameter Type	#Params.
`int`	1412	unsigned char	143
`unsigned int`	712	struct device *	102
`unsigned long`	320	unsigned short	88
`unsigned long long`	279	void *	71
`struct drm_i915_private *`	165	struct tty_struct *	64

Table 2. Top-10 Inferred Macro Parameter Types

happen, if the macro expands to more than one statement.[6] For the remaining 10 percent, we cannot infer whether they have an empty expansion, they expanded to a (type) declaration, or if they are an indicator for a problem with our prototype. This uncertainty is rooted in our prototype, which, due to limitations of the ASTMatcher, only searches for expression and statement nodes, but not declaration or type nodes.

Of the 7519 function-like top-level definitions, 53 percent were shown to have a unique C-type as a return type and 31 percent resulted in one or multiple untyped C statements (`void`). For 55 percent of the 11 368 formal macro parameters, we could infer an unambiguous C type; the top-10 of those are shown in Tab. 2.

Over all *top-level expansions*, CppSig reports an ambiguous argument type in 2.77 percent of the cases. Whether these are valid ambiguities (as shown in Fig. 4) or whether these instances of the described argument-matching problem (see Sec. 4.2) is unclear at this point.

5 Related Work

The main academic criticism with the CPP is its potential to produce not only syntactic invalid code but its ability to result in unexpected behavior (see Fig. 2). Ernst et.al [5] analyzed the usage of preprocessor macros in 26 software packages and they estimate that the average C program contains 0.28 macros per line of code and that 23 percent of all macro definitions contain latent bugs, which could be activated with adversely-chosen arguments or expansion contexts. Sæbjørnsen et.al. [16] propose a method to detect such expansion bugs: For each top-level macro expansion, they extract the corresponding subtree from the parse tree and produce a *normalized syntax tree* by stripping all nodes that stem from expanded macro arguments. For each macro definition, they compare the normalized syntax trees for similarity and, thereby, detect macros that result in syntactically different expansions. In contrast to CppSig, they ignore nested expansions by folding them into the enclosing top-level expansion and they only look at the tree structure, but ignore the type information. Liebig et.al [10] investigated on the usage of conditional compilation in 30 million lines of code and find that 84% of all `#ifdefs` are *disciplined* and respect boundaries between declarations, statements, and/or expressions. From the extent of the CPP problem, which these empirical studies document, and the persistence of developers to use the CPP [13], it is important to provide easy-to-use tools, like CppSig, that analyze CPP usage.

As another take on the CPP problem, researchers have proposed different *syntactic* preprocessors as replacements for the CPP, which respect the AST structure and cannot lead to syntax errors or unexpected behavior: With Ace [7], the developer uses directives to describe AST transformations to request code optimizations, like loop hoisting or time/space tradeoffs. Inspired by LISP macros [19], Weise et.al.[22] proposed a syntactic macro system, where all macros are "well-typed" with regard to the underlying parse tree. While their macros define their syntactic return and parameter types (e.g., statement, expression), they do not constraint the used language-level types (e.g., int, pointer). In contrast, ASTEC [12] supports syntactic macros with language-level type annotations and enforces the expressed constraints in the semantic analysis. As a transition path from CPP to ASTEC, they presented the semi-automatic MacroScope source-to-source transformation tool: Similar to CppSig, MacroScope inspects macro expansions, matches them with the respective parse-tree nodes, and finally translates them to ASTEC macros. Besides having a different goal than CppSig, MacroScope operates on the parse tree, where the availability of all lexer tokens eases the matching but makes extraction of expansion signatures impossible as typing information is not yet available. Németh et.al. [15] proposed another replacement strategy for CPP: they translate a limited subset of Haskell λ-Expressions to C-preprocessor macros, which makes the definition of disciplined macros less cumbersome but results in an immense increase in preprocessing time.

Another aspect of CPP, which was not the focus of this work, is its capability to express variability in terms of conditional compilation. In our own work [14, 21], we searched for `#ifdef`-blocks that are never seen by the parser and used a sampling-based approach to compile all other blocks at least once. PCp3 [2] is an early integrated preprocessor and parser, which provides various hooks and a scripting facility, that could be used make CppSig at least heuristically variability aware. A sound variant is variability-aware parsing [6, 8], where all conditionally-compiled blocks are combined into a single, variability-aware AST. In contrast to CppSig, these techniques parse C in the presence of the CPP, while we see CPP more as a language on its own that "rides" on top of C. Furthermore, the overheads of variability-aware parsing, which either involves SAT solving [8] or handling of binary-decision diagrams [6], hinder its adoption in everyday use.

6 Conclusion

We have presented CppSig, an approach to extract type signatures for C preprocessor macros from their respective expansion sites. For this, we match the macro-expansion tree against the abstract syntax tree by using source-location information that modern compilers already track to provide good error messages. We provide a prototypical implementation of our approach as a Clang plugin, which is publicly available, and we applied it to the Linux 5.12 source base. In this evaluation, we could infer unambiguous return types for 53 percent of the macro definitions that were used as top-level expansions. For 55 percent of their parameters, we could provide an unambiguous C type.

[6]For example: `#define locked(seq) lock(); seq; unlocked()`

References

[1] Bence Babati and Norbert Pataki. Analysis of include dependencies in c++ source code. In *FedCSIS (Communication Papers)*, pages 149–156, 2017.

[2] Greg J Badros and David Notkin. A framework for preprocessor-aware C source code analyses. *Software: Practice and Experience*, 30(8):907–924, 2000. ISSN 0038-0644. doi:10.1002/(SICI)1097-024X(20000710)30:8<907::AID-SPE324>3.3.CO;2-9.

[3] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Understanding Linux feature distribution. In Christoph Borchert, Michael Haupt, and Daniel Lohmann, editors, *Proceedings of the 2nd AOSD Workshop on Modularity in Systems Software (AOSD-MISS '12)*, New York, NY, USA, 2012. ACM Press. ISBN 978-1-4503-1217-2. doi:10.1145/2162024.2162030.

[4] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, November 2006. URL http://dunkels.com/adam/dunkels06protothreads.pdf.

[5] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28 (12):1146–1170, 2002. ISSN 0098-5589. doi:10.1109/TSE.2002.1158288.

[6] Paul Gazzillo and Robert Grimm. Superc: parsing all of c by taming the preprocessor. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*, pages 323–334, New York, NY, USA, 2012. ACM Press. ISBN 978-1-4503-1205-9. doi:10.1145/2254064.2254103.

[7] James Gosling. Ace: a syntax-driven c preprocessor. *Australian Unix Users Group*, 1989.

[8] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*, New York, NY, USA, October 2011. ACM Press. doi:10.1145/2048066.2048128.

[9] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5, 2008.

[10] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In Shigeru Chiba, editor, *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD '11)*, pages 191–202, New York, NY, USA, 2011. ACM Press. ISBN 978-1-4503-0605-8. doi:10.1145/1960275.1960299.

[11] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. Evolution of the Linux kernel variability model. In Kyo Kang, editor, *Proceedings of the 14th Software Product Line Conference (SPLC '10)*, volume 6287 of *Lecture Notes in Computer Science*, pages 136–150, Heidelberg, Germany, September 2010. Springer-Verlag. ISBN 978-3-642-15578-9.

[12] Bill McCloskey and Eric Brewer. Astec: a new approach to refactoring c. *ACM SIGSOFT Software Engineering Notes*, 30(5):21–30, 2005.

[13] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 495–518, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN 978-3-939897-86-6. doi:10.4230/LIPIcs.ECOOP.2015.495.

[14] Sarah Nadi, Christian Dietrich, Reinhard Tartler, Ric Holt, and Daniel Lohmann. Linux variability anomalies: What causes them and how do they get fixed? In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*, pages 111–120, Washington, DC, USA, May 2013. IEEE Computer Society Press. ISBN 978-1-4799-0345-0. doi:10.1109/MSR.2013.6624017.

[15] Boldizsár Németh, Máté Karácsony, Zoltán Kelemen, and Máté Tejfel. Defining c preprocessor macro libraries with functional programs. *Computing & Informatics*, 35(4), 2016.

[16] Andreas Sæbjørnsen, Lingxiao Jiang, Daniel Quinlan, and Zhendong Su. Static validation of c preprocessor macros. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 149–160. IEEE, 2009.

[17] Jeremy G. Siek. Gradual typing for functional languages. In *In Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[18] Henry Spencer and Gehoff Collyer. #ifdef considered harmful, or portability experience with C News. In *Proceedings of the 1992 USENIX Annual Technical Conference*, Berkeley, CA, USA, June 1992. USENIX Association.

[19] Guy Steele. *Common LISP: the language*. Elsevier, 1990.

[20] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In Christoph M. Kirsch and Gernot Heiser, editors, *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys '11)*, pages 47–60, New York, NY, USA, April 2011. ACM Press. ISBN 978-1-4503-0634-8. doi:10.1145/1966445.1966451.

[21] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Static analysis of variability in system software: The 90,000 #ifdefs issue. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX '14)*, pages 421–432, Berkeley, CA, USA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/tartler.

[22] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 156–165, 1993.