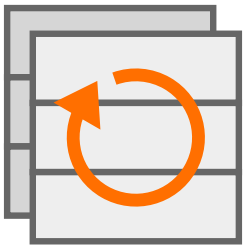


Operating System Group

TUHH

Technische Universität Hamburg

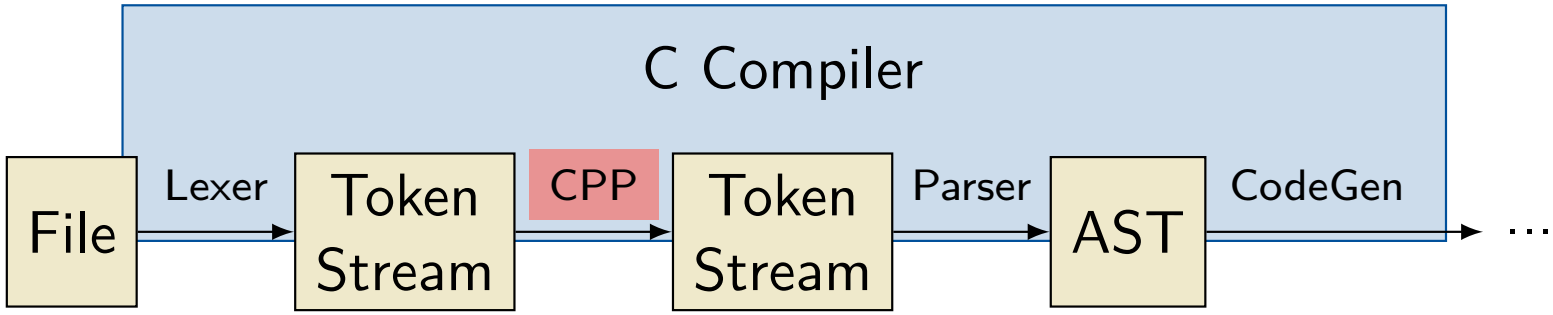
CppSig: Extracting Type Information for C-Preprocessor Macro Expansions

PLOS'21

Christian Dietrich

October 25, 2021

The C-Preprocessor (CPP) and its Macros



- CPP: A lexical preprocessor for the C/C++ parser
  - Features: file inclusion/conditional compilation/**macro expansion**
  - Method: insert/delete/replace elements in the token stream
  - Problem: Ignorant of the language's syntax rules

- CPP is **symbiotic** with the C/C++ language
  - Uses the C compiler for semantic analysis and type checking
  - Extends C by meta-programming flexibility and polymorphism

CPP in the Linux kernel

TUHH
Technische Universität Hamburg

```
#define raw_spin_is_locked(lock) \
        arch_spin_is_locked(&(lock)->raw_lock)
```

- Linux makes extensive use of CPP *(numbers for v5.12)*
  - Usage: modularization, static variability, (hardware) abstractions
  - Frequency: 1 `#ifdef` → 3 `#include` → 31 `#define` ($>$ 3 million)
  - Macros are wide-spread and are a challenge for readability: *(x86,def)*
    - Top-level, function-like: 7 519 macros → 142 861 expansions
    - Nesting of Macros: Up to 15 levels and 637 expansions

⇒ We have to understand CPP macros better!

CPP in the Linux kernel

TUHH
Technische Universität Hamburg

```
#define raw_spin_is_locked(lock) \
    arch_spin_is_locked(&(lock)->raw_lock)
```

- Linux makes extensive use of CPP *(numbers for v5.12)*
  - Usage: modularization, static variability, (hardware) abstractions
  - Frequency: 1 `#ifdef` → 3 `#include` → 31 `#define` ($>$ 3 million)
  - Macros are wide-spread and are a challenge for readability: *(x86,def)*
    - Top-level, function-like: 7 519 macros → 142 861 expansions
    - Nesting of Macros: Up to 15 levels and 637 expansions

⇒ We have to understand CPP macros better!

CppSig: What is the type signature of a macro *expansion*?

Outline

TUHH
Technische Universität Hamburg

- Motivation
- The CppSig Approach
  - Type signatures for macros?
  - Matching Expansion Tree and Abstract Syntax Tree
  - Challenging Macro Patterns
- Application to Linux kernel
- Conclusion

Outline

TUHH
Technische Universität Hamburg

- Motivation
- The CppSig Approach
  - Type signatures for macros?
  - Matching Expansion Tree and Abstract Syntax Tree
  - Challenging Macro Patterns
- Application to Linux kernel
- Conclusion

Macro Expansion Signatures

TUHH
Technische Universität Hamburg

Problem: CPP macros only have meaning within the expansion site!

```
#define add(a, b) ((a) + (b))

add(1,  2)   // (int, int) → int
add(1,  2.0) // (int, float) → float
add(1.0,2.0) // (float, float) → float

// Locks every struct with a field "nesting"
#define lock(lockable) ((lockable).nesting++)
```

⇒ Extract *Expansion* Signatures instead of *Definition* Signatures

or informal: “How is a macro used throughout the code-base?”

Macro Expansion Signatures

TUHH
Technische Universität Hamburg

Problem: CPP macros only have meaning within the expansion site!

```
#define add(a, b) ((a) + (b))

add(1,  2)   // (int, int) → int
add(1,  2.0) // (int, float) → float
add(1.0,2.0) // (float, float) → float

// Locks every struct with a field "nesting"
#define lock(lockable) ((lockable).nesting++)
```

⇒ Extract *Expansion* Signatures instead of *Definition* Signatures

or informal: “How is a macro used throughout the code-base?”

- Different Possible Use-Cases:
  - Code Understanding: How should I use this macro?
  - Type Checking: Is the macro used consistently throughout the code base?

The CppSig Approach

TUHH
Technische Universität Hamburg

1. Within CPP: Record expansion tree and track tokens
   - Macro arguments accumulate tokens from different locations
   - In Clang saves an expansion-location stack for each token.
     its parser propagates this location stack to the AST nodes.

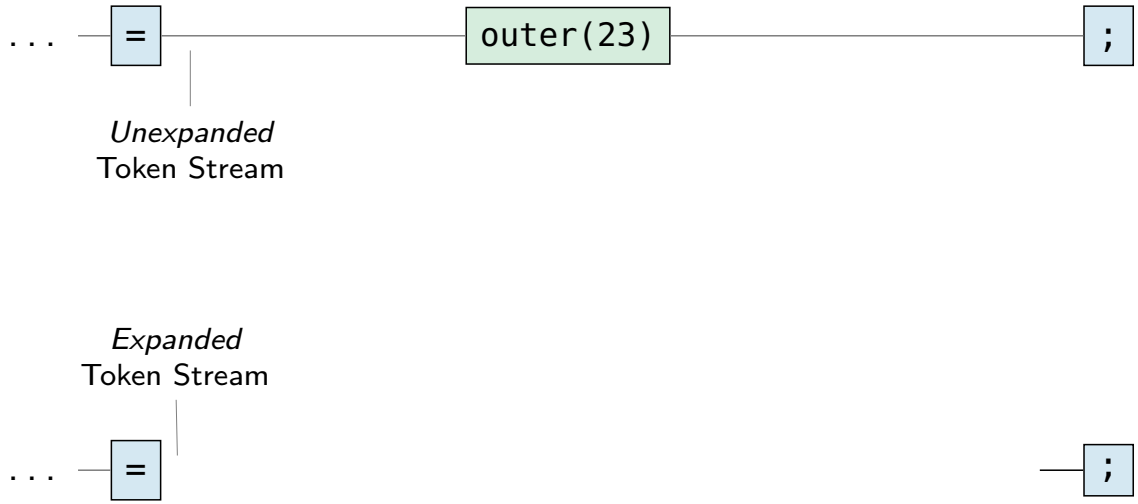
Expansion Tree and Abstract Syntax Tree

Source Code

```
1:  #define inner(I)  I / 100
2:  #define middle(M) inner(M * 1.0) - 20
3:  #define outer(O)  1 + middle(O)

5:  ....  = outer(23);
```

CPP Expansion Tree



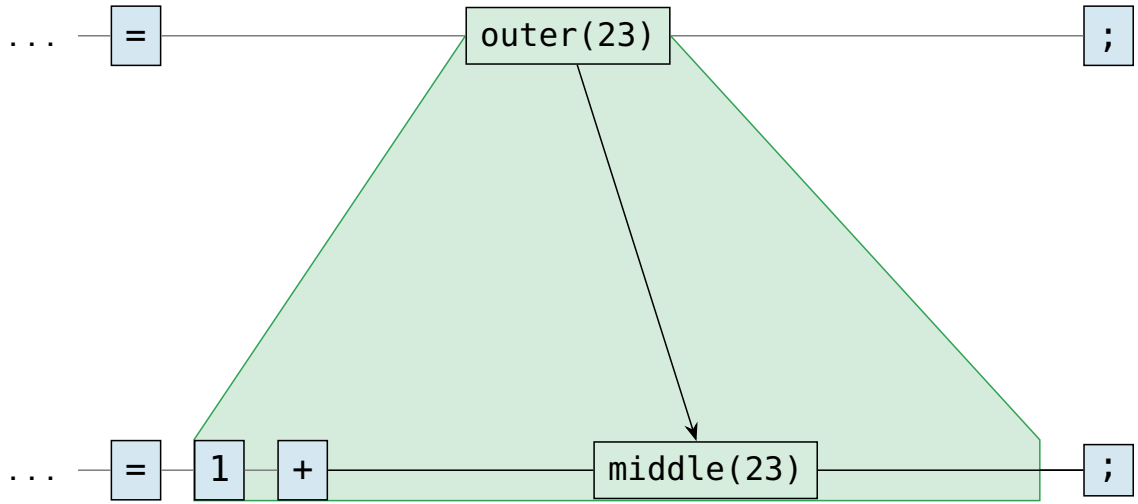
Expansion Tree and Abstract Syntax Tree

Source Code

```
1:  #define inner(I)  I / 100
2:  #define middle(M) inner(M * 1.0) - 20
3:  #define outer(O)  1 + middle(O)

5:  ....  = outer(23);
```

CPP Expansion Tree



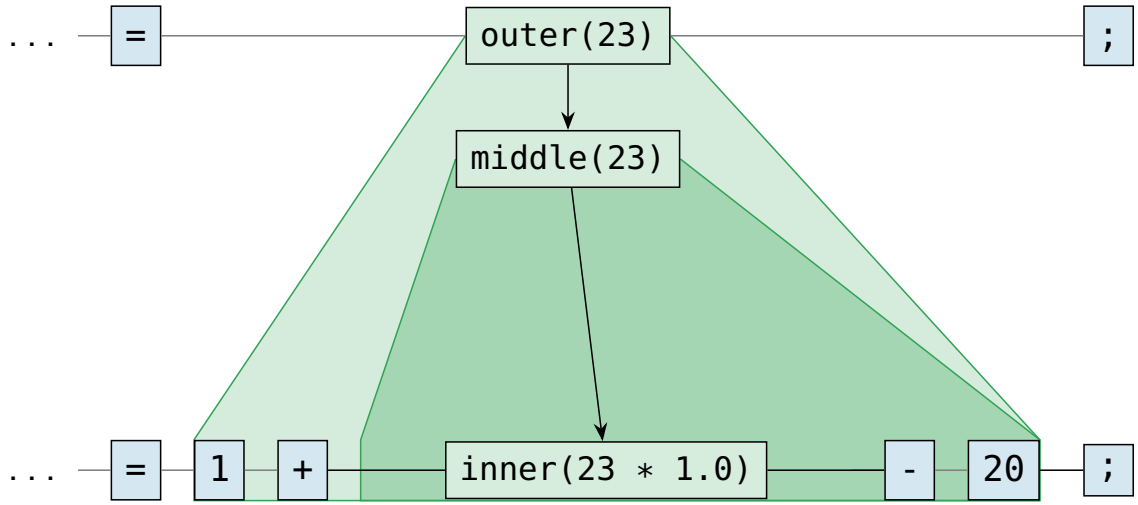
Expansion Tree and Abstract Syntax Tree

Source Code

```
1:  #define inner(I)  I / 100
2:  #define middle(M) inner(M * 1.0) - 20
3:  #define outer(O)  1 + middle(O)

5:  ....  = outer(23);
```

CPP Expansion Tree



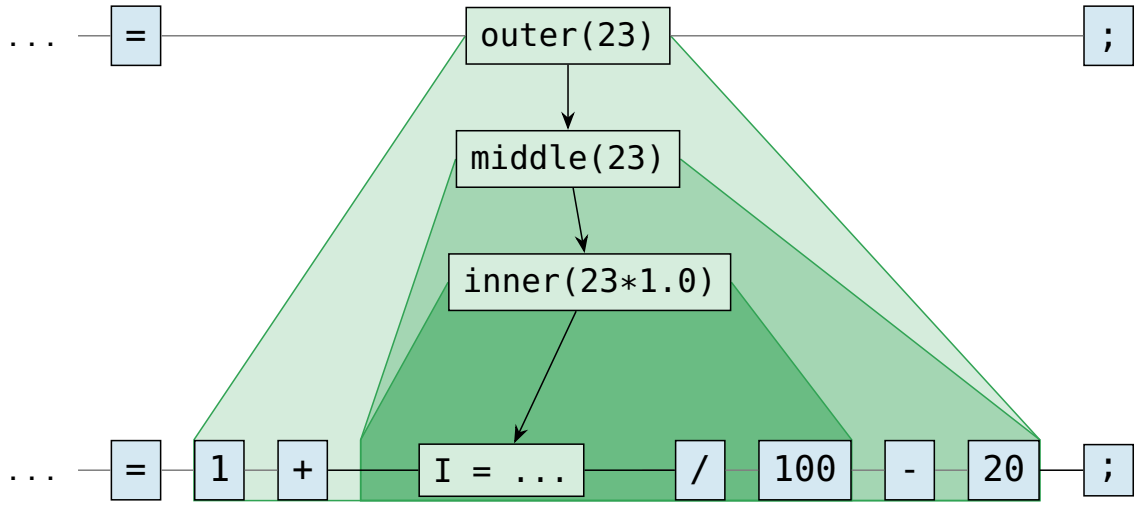
Expansion Tree and Abstract Syntax Tree

Source Code

```
1:  #define inner(I)  I / 100
2:  #define middle(M) inner(M * 1.0) - 20
3:  #define outer(O)  1 + middle(O)

5:  ....  = outer(23);
```

CPP Expansion Tree



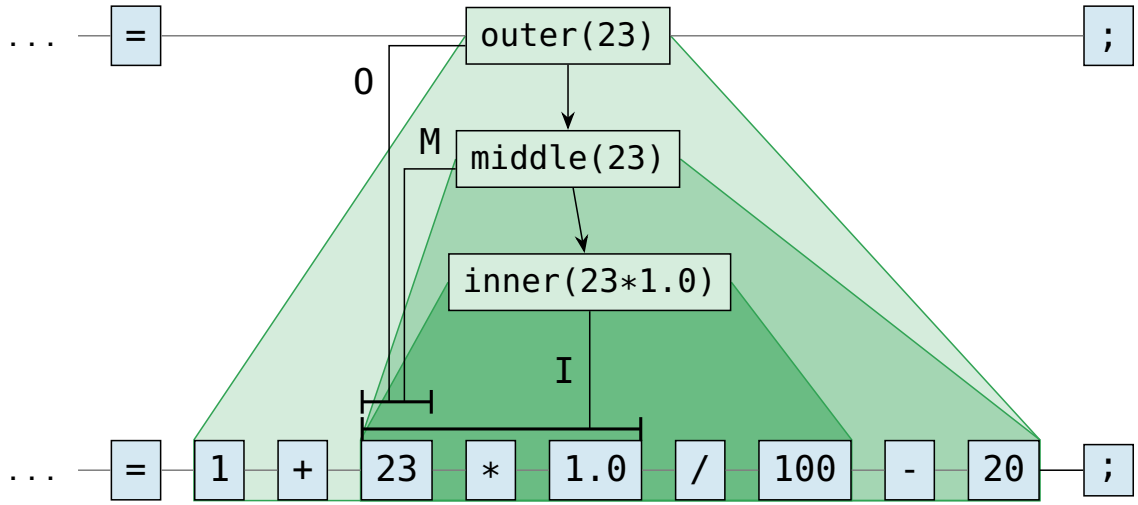
Expansion Tree and Abstract Syntax Tree

Source Code

```
1:  #define inner(I)  I / 100
2:  #define middle(M) inner(M * 1.0) - 20
3:  #define outer(O)  1 + middle(O)

5:  ....  = outer(23);
```

CPP Expansion Tree

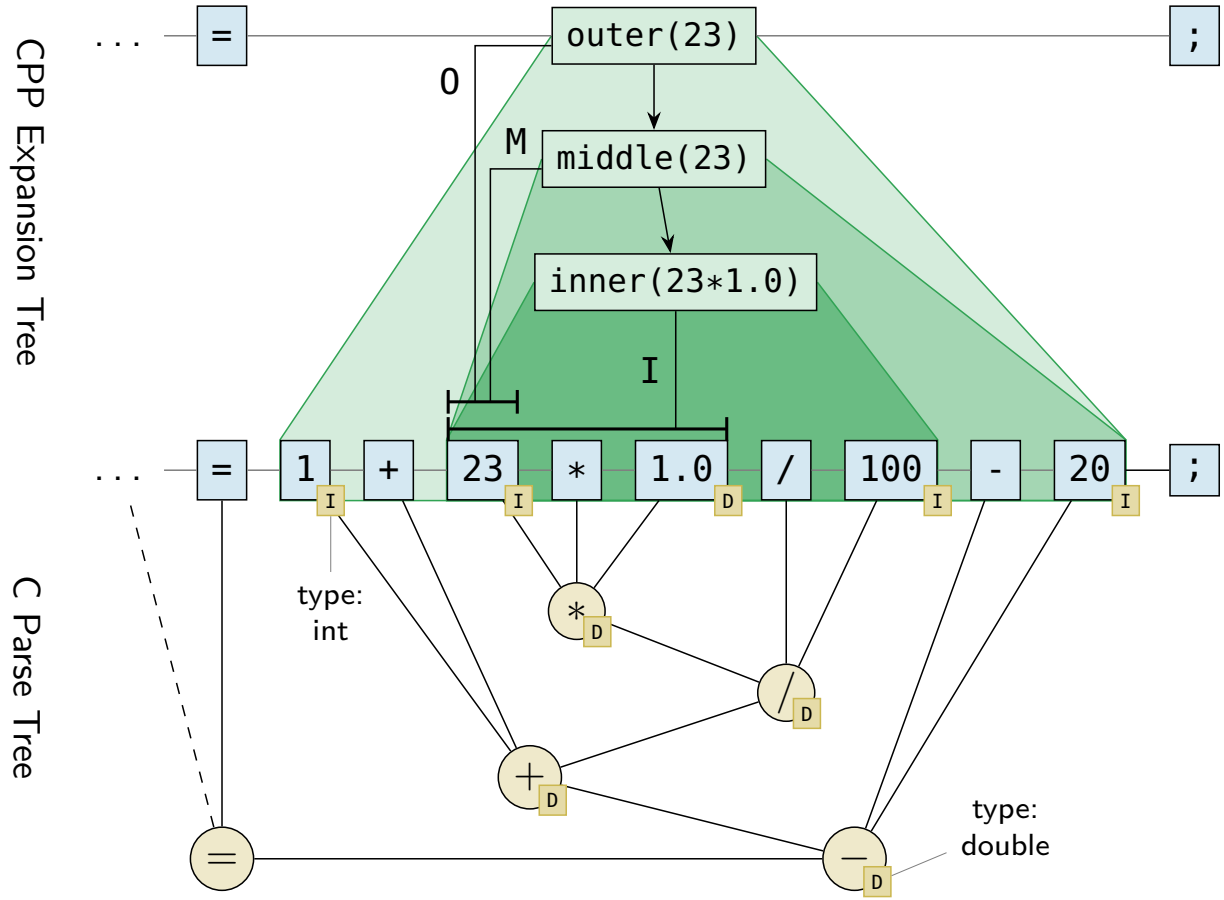


Expansion Tree and Abstract Syntax Tree

Source Code

```
1:  #define inner(I)  I / 100
2:  #define middle(M) inner(M * 1.0) - 20
3:  #define outer(O)  1 + middle(O)

5:  ....  = outer(23);
```



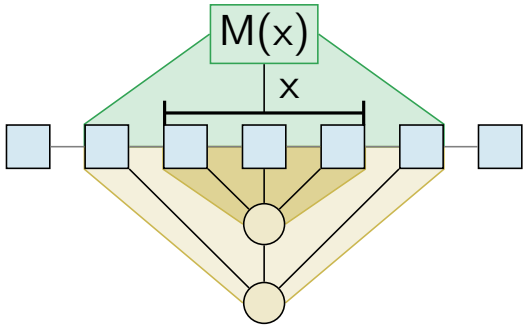
The CppSig Approach

TUHH
Technische Universität Hamburg

1. Within CPP: Record expansion tree and track tokens
   - Macro arguments accumulate tokens from different locations
   - In Clang saves an expansion-location stack for each token.
     its parser propagates this location stack to the AST nodes.

2. Match expansion-tree and AST nodes
   - Find AST nodes that stem from a (nested) expansion
   - They come together as one or multiple AST subtrees

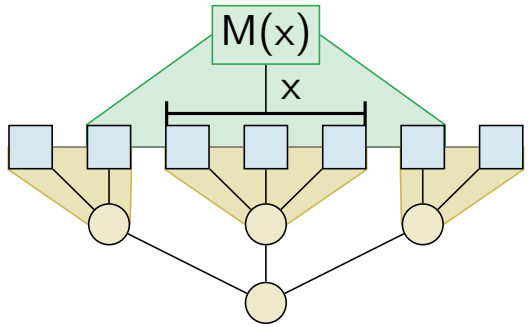
Challenging Tree Alignments



(a) Perfectly Aligned

Example:

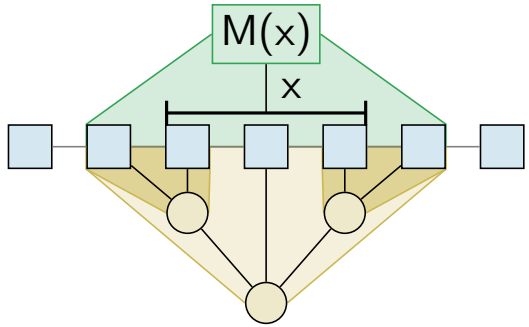
```
#define M(x) (x)
int x = M(23+3);
```



(b) Unaligned Body

Example:

```
#define M(x) 3+x+4
int x = 1 * M(3) * 4;
```



(c) Unaligned Argument

Example:

```
#define M(x) 3*x*4
int x = M(3+4);
```

Unaligned expansion are considered a bad code smell.

⇒ CppSig handles them gracefully!

The CppSig Approach

TUHH
Technische Universität Hamburg

1. Within CPP: Record expansion tree and track tokens
   - Macro arguments accumulate tokens from different locations
   - In Clang saves an expansion-location stack for each token.
     its parser propagates this location stack to the AST nodes.

2. Match expansion-tree and AST nodes
   - Find AST nodes that stem from a (nested) expansion
   - They come together as one or multiple AST subtrees

3. Find macro-arguments in the expansion subtree(s)
   - Select subtree-nodes that are located within a
   - Again: one or multiple AST subtrees

The CppSig Approach

TUHH
Technische Universität Hamburg

1. Within CPP: Record expansion tree and track tokens
   - Macro arguments accumulate tokens from different locations
   - In Clang saves an expansion-location stack for each token.
     its parser propagates this location stack to the AST nodes.

2. Match expansion-tree and AST nodes
   - Find AST nodes that stem from a (nested) expansion
   - They come together as one or multiple AST subtrees

3. Find macro-arguments in the expansion subtree(s)
   - Select subtree-nodes that are located within a
   - Again: one or multiple AST subtrees

4. Derive macro return and argument types from AST nodes
   - Exactly one subtree: Root type is unambiguous type
   - Multiple subtrees: Statement-level macro or ambiguous argument type

Outline

TUHH
Technische Universität Hamburg

- Motivation
- The CppSig Approach
  - Type signatures for macros?
  - Matching Expansion Tree and Abstract Syntax Tree
  - Challenging Macro Patterns
- Application to Linux kernel
- Conclusion

CppSig on the Linux kernel

TUHH
Technische Universität Hamburg

- Run CppSig as a Clang plugin on Linux 5.12, x86, defconfig
  - Matching both trees took 366ms (median)
  - Longest running file: 53 minutes (`net/mac80211/airtime.c`)
  - Problem: At-least quadratic run-time of prototypical implementation due to Clang's AST Matcher Interface.

CppSig on the Linux kernel

TUHH
Technische Universität Hamburg

- Run CppSig as a Clang plugin on Linux 5.12, x86, defconfig
  - Matching both trees took 366ms (median)
  - Longest running file: 53 minutes (`net/mac80211/airtime.c`)
  - Problem: At-least quadratic run-time of prototypical implementation due to Clang's AST Matcher Interface.

- 142 861 function-like top-level macro expansions
  - 58 % ⇒ single expression AST subtrees
  - 32 % ⇒ multiple subtrees
  - 10 % ⇒ match failed, type expansion or, expansion became "" (?)

CppSig on the Linux kernel

TUHH
Technische Universität Hamburg

- Run CppSig as a Clang plugin on Linux 5.12, x86, defconfig
  - Matching both trees took 366ms (median)
  - Longest running file: 53 minutes (`net/mac80211/airtime.c`)
  - Problem: At-least quadratic run-time of prototypical implementation due to Clang's AST Matcher Interface.

- 142 861 function-like top-level macro expansions
  - 58 % ⇒ single expression AST subtrees
  - 32 % ⇒ multiple subtrees
  - 10 % ⇒ match failed, type expansion or, expansion became "" (?)

- 7 519 function-like definitions used as top-level expansion
  - 55 % ⇒ unambigous parameter type (expression param)
  - 53 % ⇒ unambigous return type (expression macro)
  - 31 % ⇒ one or multiple void-typed nodes (statement macro)

CppSig on the Linux kernel

TUHH
Technische Universität Hamburg

- Run CppSig as a Clang plugin on Linux 5.12, x86, defconfig
  - Matching both trees took 366ms (median)
  - Longest running file: 53 minutes (`net/mac80211/airtime.c`)

Top-10 Macro-Paramter Types

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

- 7 519 function-like definitions used as top-level expansion
  - 55 % ⇒ unambigous parameter type (expression param)
  - 53 % ⇒ unambigous return type (expression macro)
  - 31 % ⇒ one or multiple void-typed nodes (statement macro)

Would you have guessed it?

TUHH
Technische Universität Hamburg

- `shm_ids(ns)`: 34 occurences
- `disk_to_dev(disk)`: 35 occurences
- `wake_up(x)`: 178 occurences
- `ext4_journal_stop(handle)`: 91 occurences
- `fw_domain_init(uncore, id, set, ack)`: 12 occurences

Would you have guessed it?

TUHH
Technische Universität Hamburg

- `shm_ids(ns)`: 34 occurences
  ⇒ `struct ipc_ids * (struct ipc_namespace *)`
- `disk_to_dev(disk)`: 35 occurences
  ⇒ `struct device* (struct gendisk *)`
- `wake_up(x)`: 178 occurences
  ⇒ `void (struct wait_queue_head *)`
- `ext4_journal_stop(handle)`: 91 occurences
  ⇒ `void (struct jbd2_journal_handle *)`
- `fw_domain_init(uncore, id, set, ack)`: 12 occurences
  ⇒ `int (intel_uncore *, int, i915_reg_t, i915_reg_t)`

Outline

TUHH
Technische Universität Hamburg

- Motivation
- The CppSig Approach
  - Type signatures for macros?
  - Matching Expansion Tree and Abstract Syntax Tree
  - Challenging Macro Patterns
- Application to Linux kernel
- Conclusion

Conclusion

TUHH
Technische Universität Hamburg

- CPP is a **symbiotic** but **problematic companion** language
  - In-kernel usage: abstractions, variability, modularization
  - Complex constructs (nesting, involved macros) hinder readability

- CppSig: Extract expansion types from the C-AST nodes
  1. Record macro expansion tree and argument token lists
  2. Match AST Nodes against expansion tree
  3. Derive (un)ambiguous macro types from matched subtrees

- The Linux kernel is a heavy user of CPP macros
  - Monomorphism: 84% return- and 55% argument types are unambigious
  - `struct drm_i915_private *` is the most frequent non-int param type

Source Code and Docker Image are available at:
https://collaborating.tuhh.de/e-exk4/projects/cpp-macro-types