Clog: A Declarative Language for C Static Code Checkers

Alexandru Dura
Lund University
Sweden
alexandru.dura@cs.lth.se

Christoph Reichenbach
Lund University
Sweden
christoph.reichenbach@cs.lth.se

Abstract

We present Clog, a declarative language for describing static code checkers for C. Unlike other extensible state-of-the-art checker frameworks, Clog enables powerful interprocedural checkers without exposing the underlying program representation: Clog checkers consist of Datalog-style recursive rules that access the program under analysis via syntactic pattern matching and control flow edges only. We have implemented Clog on top of Clang, using a custom Datalog evaluation strategy that piggy-backs on Clang's AST matching facilities while working around Clang's limitations to achieve our design goal of representation independence.

Our experiments demonstrate that Clog can concisely express a wide variety of checkers for different security vulnerabilities, with performance that is similar to Clang's own analyses and highly competitive on real-world programs.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; **Constraint and logic languages**; *Domain specific languages*; • **Theory of computation** → *Pattern matching*.

Keywords: Datalog, C, Syntactic Patterns, Static Analysis Frameworks

ACM Reference Format:
Alexandru Dura and Christoph Reichenbach. 2024. Clog: A Declarative Language for C Static Code Checkers. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3640537.3641579

1 Introduction

While the C programming language enforces certain correctness properties that all C programs must satisfy, C programmers have been utilizing supplementary static checkers to enforce additional constraints for most of the language's existence — the release of C in 1973 [26] was followed by the release of `lint` only 4 years later [15].

Since then, static program analysis has made substantial advances. Modern software engineers can draw from frameworks that offer interprocedural data flow analyses (e.g., *Phasar* [28]), complex call graph and points-to analyses with intricate interdependencies (e.g., *cclyzer* [4]) and efficient memory models via separation logic (*Infer* [25]). In practical software development, developers may also adopt checker frameworks that trade precision or soundness for speed, for easier integration into the build workflow, such as the *Clang Static Analyzer* [1], *clang-tidy*[2] and *CppCheck*[3].

However, most of these tools (*Phasar*, *Infer*, *Clang Static Analyzer*, *clang-tidy*, *CppCheck*) are not easily extensible: they only supply a fixed set of built-in *general-purpose* checkers. Thus, they are not designed to tackle the increasing reliance of modern software on external libraries, to support custom checks for internal APIs, or to incorporate custom, project-specific tweaks to existing analyses. Adding a new analysis or customizing an existing one depends on the internal representation of the program used by the tool (usually an abstract syntax tree (AST) or a 3-address intermediate representation (IR)), and these internal representations often evolve as the underlying tool itself evolves. For example, Clang offers a command-line interface for AST pattern matching (*clang-query*) that can find program locations corresponding to a user-defined pattern, but users must express these patterns in Clang's idiosyncratic pattern language.

Several recent tools, *Coccinelle* [18], *CodeQL* [3], and *cclyzer*, therefore explicitly offer domain-specific languages that enable software engineers to supply their own bug patterns or to tweak existing ones. We observe that these tools split bug detection into two *phases*:

1. *First phase*: moving from the concrete domain (AST or IR) to an abstract domain
2. *Second phase*: combine information in the abstract domain to derive the conclusions, computing fixpoints as needed

Both *cclyzer* and *CodeQL* provide a declarative, Datalog-style, analysis description that facilitates the development of custom analyses, but only in the second phase of the analyzer.

CC '24, March 2–3, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0507-6/24/03
https://doi.org/10.1145/3640537.3641579

[1] https://clang-analyzer.llvm.org/
[2] https://clang.llvm.org/extra/clang-tidy/
[3] https://cppcheck.sourceforge.io/

```c
void f(void) {
loop:
  if (1)
    goto loop;
}
```
(a)

```c
int g(int cond) {
  int exit_code = 0;
  if (cond) {
    exit_code = 1;
    goto error_exit;
  }
  return 0;
error_exit:
  cleanup();
  return exit_code;
}
```
(b)

```c
int h(int cond) {
  if (cond)
    return 0;
  else
    goto error_exit;
  error_exit:
  // cleanup();
  return 1;
}
```
(c)

Figure 1. Examples for the `goto` check. Case (a) is a back-jump. Case (b) is a forward jump to the single label in the function. Case (c) is a forward jump to a label followed by a return.

The analyses are still dependent on the internal representation of the analyzed program, be it an AST or an IR. *Coccinelle*'s code pattern language, meanwhile, allows developers to specify *syntactic patterns* that resemble C code, with various pattern extensions e.g. for intraprocedural control flow dependencies between patterns. More complex connections between patterns (e.g., those that require fixpoints) require custom Python or OCaml code.

In this paper, we introduce Clog, a declarative language that combines Datalog-style reasoning with *Coccinelle*-style syntactic pattern matching over the C language. For the first analysis phase, syntactic patterns allow us to describe C code patterns, without exposing the internal representation, unlike *cclyzer* and *CodeQL*. For the second analysis phase, Datalog-style reasoning allow us to freely combine syntactic patterns across arbitrary flow and dependency edges, avoiding the need to escape to scripting languages, as in *Coccinelle*, and enabling integration with the existing body of work on points-to analysis and context-sensitivity in Datalog.

To illustrate our approach, consider a common warning between static code checkers: the use of `goto` statement (Figure 1). Although there is variation in what the tools consider an admissible use of `goto`s, both *clang-tidy* and *CodeQL* agree on discouraging back-jumps (Figure 1a).

In Figure 2 we show a possible approach for implementing a back-jump checker in *CodeQL*. The checker collects all `goto` statements in a relation `GotoStmt` and all label statements in `LabelStmt` (line 1), from these relations selects the `goto` and label statements that refer to the same label (line 3), compares their source locations (line 4), and, if successful, generates a report (line 6). While this checker provides a concise description of the check, it still relies on the GotoStmt and LabelStmt relations, which are defined externally. The

```
1 from GotoStmt goto, LabelStmt label
2 where
3   goto.getTarget() = label and
4   label.getLocation().getStartLine()
5     < goto.getLocation().getStartLine()
6 select goto, "Goto jumps to a label that appears before the goto."
```

Figure 2. Implementation of a `goto` check in CodeQL

```
1 WarnBackwardGoto(g, l) :-
2   g (:goto $label;:),
3   l (:$label : $s:),
4   src_line_start(g) > src_line_start(l).
```

Figure 3. Implementation of a `goto` check in Clog

checker writer must be aware about how the `goto` and label statements are represented in the abstract syntax tree (AST) and what their attributes are (e.g., `.getTarget()`).

In contrast to *CodeQL*, Clog aims to hide implementation details internal to the analysis system such as the AST nodes and their attributes and introduces *syntactic patterns* to match arbitrary terms of the analyzed programs, including, but not limited to, terms representing single AST nodes.

Figure 3 depicts the Clog implementation of the same `goto` check, where we use syntactic patterns instead of built-in relations. Following Datalog notation, `:-` represents logical right-to-left implication (⇐) and commas represents conjunction. The syntactic pattern in line 2 matches all `goto` statements in the program, binds them to variable g and their labels to `$label`. Analogously, the pattern in line 3 matches all label statements `l`, with the same label, `$label`. To distinguish between program names and Datalog variables, we prefix variables used inside patterns with the `$` sign, thus `$label` can bind to any label in the program and `$s` to any statement. Since these are variables of the analysis program, we call them *metavariables*. On line 4, the program compares source line numbers, and if, this succeeds, it adds the tuple `(g, l)` to the WarnBackwardGoto relation.

Syntactic patterns are not limited to single statements, but we can freely compose them as long as the result is a statement, expression, declaration or definition. For example, a program that detects labels before a `return` statement is:

```
LabeledReturn(l) :- l (:$label : return $r;:).
LabeledReturn(l) :- l (:$label : return;:).
```

Such a program may warn about missing cleanup code or hint that a return can be used directly instead of a `goto` (Figure 1c).

To enable our framework to handle industrial-quality code, including large code bases and C language extensions, we have implemented its frontend on top of Clang. This allows us to combine state-of-the-art Datalog evaluation techniques with a custom *pattern embedding* strategy that offloads parts of syntactic pattern matching and semantic analysis to Clang's own pattern matching and analysis facilities, on demand.

```
1  void entry0(arena *ma) {
2    int *p = aalloc(ma, sizeof(int));
3    int *q = malloc(sizeof(int));
4    // do work
5    free(p);
6    free(q);
7  }
8  void cleanup1(int *x, int *y) {
9    free(x);
10   free(y);
11 }
12 void entry1(arena *ma) {
13   int *p = aalloc(ma, sizeof(int));
14   int *q = malloc(sizeof(int));
15   // do work
16   cleanup1(p, q);
17 }
18 int * alloc21(arena *ma) {
19   int *p = aalloc(ma, sizeof(int));
20   return p;
21 }
22 int * alloc22(arena *ma) {
23   int *p = malloc(sizeof(int));
24   return p;
25 }
26 void entry2(arena *ma) {
27   int *p = alloc21(ma);
28   int *q = alloc22(ma);
29   // do work
30   free(p);
31   free(q);
32 }
```

Figure 4. Wrong uses of an arena allocator API

To summarize, our contributions are:

1. Clog, a declarative language that combines syntactic patterns and Datalog-style reasoning for program analysis of C programs;
2. A prototype implementation of Clog[4];
3. An execution strategy that allows us to automatically offload parts of most Clog analyses to Clang;
4. A comparison of speed and quality between 5 analyses implemented in Clog and the Clang Static Analyzer, with a validated artifact [11].

2 The Clog Language

We introduce the Clog language through two examples that illustrate both the language and the workflow that we use to develop static checkers in Clog. We first show how we can construct recursive inclusion-based analyses (Section 2.1) to catch misuses of a typical internal API, and then show how Clog's built-in knowledge about control flow (Section 2.2) can help identify violations of an API protocol for an external library. Sections 2.3-2.6 then give a full overview over the language.

2.1 Recursive Patterns: Arena Allocators

To illustrate how Clog can help developers find bugs related to internal APIs, we look at arena allocators [7]. These custom allocators can speed up memory allocation and (bulk) deallocation. Since arenas can be used to store temporary data structures that have a shorter lifetime than the program, pointers allocated in them may coexist along pointers allocated using `malloc`. However, calling `free` on an arena-allocated pointer is undefined behavior, so it is important that the two kinds of pointers are not confused in the program. In Figure 4 we provide three examples of such API misuse.

[4] https://github.com/lu-cs-sde/clog

```
1 ExprPointsToArena(e) :- e ⟨: aalloc(..) :⟩.
2 ExprPointsToArena(e) :- e ⟨: ($t) $f :⟩, ExprPointsToArena($f).
3 VarPointsToArena($p) :- ⟨: $t *$p = $e :⟩, ExprPointsToArena($e).
4 VarPointsToArena(p) :- ⟨: $p = $e :⟩, p = decl($p), p != undef,
5     ExprPointsToArena($e).
6 ExprPointsToArena($p) :- ⟨: $p :⟩, p = decl($p), p != undef,
7     VarPointsToArena(p).
8 FreeOfArenaPtr($p) :- ⟨: free($p) :⟩, ExprPointsToArena($p).
```

Figure 5. Intra-procedural check for `free`-ing arena-allocated memory.

In Figure 5, we give an analysis that can detect the first of these cases. Like all Clog programs, the analysis consists of a set of extended Horn clauses that conceptually take the form $P_0(\overline{x_0})$:- $P_1(\overline{x_1}), \ldots, P_n(\overline{x_n})$ where the P_i are the names of user-defined or built-in relations (or, equivalently, *predicates*). These rules follow the usual Datalog semantics: Let $\mathcal{X}$ be the set of all variables that occur in $\overline{x_0}, \ldots, \overline{x_n}$. Whenever we have a function ρ that maps each $x \in \mathcal{X}$ to a constant such that for all $i \in 1, \ldots n$, the relation P_i contains the tuple $\rho(\overline{x_i})$, then the relation P_0 also contains the tuple $\rho(\overline{x_0})$.

In Clog, $P_i(\overline{x_i})$ may also represent a syntactic pattern (for $i \neq 0$). For example, in Figure 5, line 1 states that if `e` is a specific element of the program under analysis that has the syntactic form ⟨: `aalloc(..)` :⟩ (where '`..`' is a wildcard that matches any sequence of elements), then we must conclude that ExprPointsToArena(`e`) is true. Syntactic patterns like `e` ⟨: `aalloc(..)` :⟩ behave like predicates quantified over the entire program under analysis, that is: a syntactic pattern matches if there exists a substitution of the pattern's metavariables with terms from the analyzed program such that the resulting term is present in the analyzed program.

We follow most Datalog dialects in further extending the language with comparison predicates, represented inline by the operators (e.g. `>`, `==`, etc.), pure functions (e.g. `src_line_start` which maps a program term to its source location) and stratified negation (using the '`!`' operator).

To catch the first misuse case (line 5), we describe an intra-procedural, flow-insensitive analysis in Figure 5.

In line 2, we also add the cast expressions to the same relation. Line 3 introduces a new relation, VarPointsToArena, which contains all the pointer variables pointing to arena-allocated memory. First, we add all variables that are initialized by an expression that points to an arena. In line 4, we handle variable assignments, which are matched by the ⟨: `$p = $e` :⟩ pattern, where `$p` is bound to an identifier and we use the built-in function `decl` to look-up the corresponding declaration, `p`. If the look-up is successful, and the right-hand side of the assignment points to an arena, then we deduce that the variable `p` also points to an arena. Line 6 contains the dual of the previous rule, stating that if variable points to an arena, then a reference to that variable, ⟨: `$p` :⟩, is an expression pointing to an arena. We observe that there is a circular dependency between the VarPointsToArena

and ExprPointsToArena relations - this is handled by the fix-point semantics of Datalog. Finally, in line 8, we collect all the problematic calls to `free` on an arena-allocated pointer in the relation FreeOfArenaPtr.

As hinted earlier, this checker does not catch cases where the allocation and the call to `free` are happening in different functions. To catch the error on line 9, we need to track the flow of values through function calls. We achieve this by appending two rules to the program.

```
9  Call(call, $actual, $formal) :- call ⟨: $c(.., $actual, ..) :⟩,
10     $callee = decl($c),
11     ⟨:$rt $callee(.., $pt $formal, ..) { .. }:⟩,
12     index($formal) == index($actual).
13 VarPointsToArena(f) :- Call(_, e, f), ExprPointsToArena(e).
```

The first rule (line 9) defines the Call predicate which contains the `formal` and `actual` argument pairs for each `call` expression. We use the `index` function to retrieve the index of a term in a list, in this case the index of the actual and formal argument. The second rule (line 13) states that if an actual argument is an expression pointing to an arena, then the formal argument points to an arena.

To catch the error on line 30, we also have to track values through function returns. Another two rules materialize this:

```
14 Return(call, $val) :- call ⟨:$c(..):⟩,
15     $callee = decl($c),
16     @$callee ⟨:return $val;:⟩.
17 ExprPointsToArena(call) :- Return(call, e), ExprPointsToArena(e).
```

The rule on line 14 defines the Return predicate which maps all call expressions to the expressions returned by the `$callee`. On line 16 we illustrate the use of restricted syntactic patterns, where the pattern `@$callee ⟨:return $val; :⟩` matches only the `return` statements from `$callee`.

We observe that we can develop the checker incrementally, by appending new rules, to transform it into an interprocedural analysis. Although the checker is far from being sound or complete, it already covers interesting cases and it is a starting point for incremental development, for example by adding call-site sensitivity. Adding various flavors of context-sensitivity to Datalog analyses has already been demonstrated by tools like *Doop* [6], thus we do not explore this direction as part of this work.

2.2 Control Flow: API Protocol for MPI

A common restriction is that the API functions need to be called in a certain order. For example, the OpenMPI library expects that each call to the non-blocking send function `MPI_Isend` is followed by a call to `MPI_Wait` (or similar). In Figure 6 we show 4 examples of API misuse, where the calls to `MPI_Isend` are not paired with calls to `MPI_Wait` with the same request handle, `req`.

In Figure 7 we implement a checker to detect these cases of API misuse. First, the checker identifies the involved API calls using syntactic patterns (lines 1 and 3) and defines two predicates, ISend and Wait which enumerate these call expressions (`c`) together with the variable that stores the request handle (`req`). In the next two rules (lines 5-7), we

```
1  void good(void) {
2    MPI_Request req;
3    MPI_Isend(ARGS, &req);
4    // do work
5    MPI_Wait(&req, 0);
6  }
7
8  void bad0(void) {
9    MPI_Request req;
10   MPI_Isend(ARGS, &req);
11   // do work
12 }
13
14 void bad1(void) {
15   MPI_Request req;
16   MPI_Isend(ARGS, &req);
17   // do work
18   MPI_Wait(&req, 0);
19   MPI_Isend(ARGS, &req);
20   // do work
21 }
```

```
22 void bad2(void) {
23   MPI_Request req;
24   MPI_Isend(ARGS, &req);
25   // do work
26   MPI_Isend(ARGS, &req);
27   // do work
28   MPI_Wait(&req, 0);
29 }
30
31 void bad3(void) {
32   MPI_Request req;
33   MPI_Isend(ARGS, &req);
34   if (cond) {
35     // do work
36     MPI_Wait(&req, 0);
37   }
38 }
```

Figure 6. Examples of non-blocking OpenMPI calls.

```
1  ISend(c, req) :- c⟨:MPI_Isend($a0, $a1, $a2, $a3, $a4, $a5, &$req):⟩,
2    req = decl($req), req != undef.
3  Wait(c, req) :- c⟨:MPI_Wait(&$req, $a1):⟩, req = decl($req),
4    req != undef.
5  ISendChain(c, c) :- ISend(c, _).
6  ISendChain(c, t) :- ISendChain(c, s), ISend(c, r), !Wait(s, r),
7    CFG_SUCC(s, t).
8
9  Warning(s) :- ISend(s, r), ISendChain(s, s1), s != s1, ISend(s1, r).
10 Warning(s) :- ISendChain(s, c), f = enclosing_function(s),
11   CFG_EXIT(f, exit), exit == c, ISend(s, r), !Wait(c, r).
```

Figure 7. Clog checker for mismatched OpenMPI calls.

inductively define the ISendChain relation, which maps an `Isend` call to its control-flow successors that are not `Iwait` instructions with the same request handle. For defining the ISendChain relation we rely on the CFG_SUCC built-in predicate, that maps `s` to all its control-flow successors `t`. On line 9, we emit a warning whenever there exists a control-flow path on which two `Isend` instructions with the same handle occur without a `Wait` in between (Figure 6, `bad2`). On line 10 we emit a warning if, starting from an `ISend` call, we can build a path that reaches a function exit, but does not contain a matching `Wait` call. To identify the function exits we rely on the `enclosing_function` built-in function, which maps a term to its enclosing function definition, and on the CFG_EXIT predicate which maps a function definition (`f`) to all its exits (`exit`).

2.3 Language Overview

As we have seen earlier, the Clog language is an extension of standard Datalog. We extend the syntax for body literals B to allow for pattern literals, T, and the syntax for terms, where we allow for function application ($f(\overline{e})$) and for a special

Program		::=	$\overline{K}$
Clause	K	::=	$\overline{H}$:- $\overline{B}$
Head literal	H	::=	$P\,(\overline{t})$
Body literal	B	::=	$H \mid !H \mid T \mid v = e \mid v > e \mid \ldots$
Pattern literal	T	::=	$[@t]\ [t]\ \langle{:}C{:}\rangle$
Term	t	::=	$e \mid \_ \mid k \mid 'P$
Expression	e	::=	$v \mid f\,(\overline{e}) \mid e + e \mid \ldots$
Variable	v	::=	$v_b \mid v_m$
Function	f	$\in$	*Functions*
Predicate symbol	P	$\in$	*Predicates*
Constant	k	$\in$	$\mathbb{Z} \cup$ *String* $\cup$ {undef}
Pattern	C	$\in$	*CPatternLanguage*
Ordinary variable	v_b	$\in$	*Var*
Metavariable	v_m	::=	$\$v_b$

Figure 8. The syntax of the Clog language

constant, `undef`. In Figure 8 we give the full syntax of the language.

The Clog language lacks explicit type declarations, but its predicates are statically typed. We rely instead on monomorphic type inference to deduce the type of predicates and variables. The possible types for variables are: *Integer*, *String*, *ASTNode* and *PredRef*. The *ASTNode* type represents program terms, thus all metavariables have this type. The *PredRef* is the type of predicate references, $'P$.

2.4 Pattern Literals

Pattern literals are predicates over the abstract syntax tree of the analyzed program. A pattern literal `@s r` ⟨:`C`:⟩ consists of a syntactic pattern, ⟨:`C`:⟩, and two optional nodes: the root, `r`, and the subterm restriction `@s`. The syntactic pattern matches terms in the analyzed program. When a match occurs, the metavariables in the pattern are bound and so is the optional root variable, `r`, which binds the whole matched term.

The optional subterm restriction, `@s`, restricts the matching to a strict subterm of the term `s`. For example,

```
1  ⟨:while ($cond) $body:⟩, @$body ret ⟨:return $e;:⟩
```

matches all return statements occurring in the body of a `while` loop, and binds them to variable `ret` and their respective return expression to `$e`. The pattern, ⟨:`C`:⟩, can have any of the following syntactic categories: *expression*, *statement*, *declaration* or *function definition*. Terms in the pattern are either concrete, following the C grammar, or they are left abstract and replaced with a metavariable. Metavariables can be used in place of concrete terms from the following syntactic categories: *identifier*, *init-declarator*, *parameter-declaration*, *expression*, *statement*. In places where a list of these is required, but the list elements are not relevant, a *gap* (`..`) can be used. For example, ⟨:`printf(.., $e, ..)`:⟩ matches any function call to `printf`, and binds its arguments, in turn, to `$e`. The occurrence of the call expression `printf("Hello %s!", name)` results in two pattern matches, one when `$e` binds the string literal `"Hello %s!"` and another, when it binds the identifier `name`.

2.5 Built-in Predicates

Our implementation assigns special semantics to a set of predicates. These can be grouped into three categories: control-flow predicates, I/O and infinite predicates.

Control-flow predicates expose the intra-procedural control-flow graph to the Clog program. While traversing the control-flow graph is achievable by using only syntactic patterns, this increases the verbosity of the code, therefore we expose the following predicates:

- CFG_SUCC (n, n_s) maps the term n to its successors in the control-flow graph, n_s. Since the C language leaves unspecified the order of evaluation in some cases (e.g. subexpressions, function arguments), the CFG_SUCC relation represents only one of the possible orderings. The variable n must be bound by other literals in the same clause.
- CFG_EXIT (f, n_e) maps the function definition f to all its exits, n_e. The variable f must be bound by other literals in the same clause.

The I/O predicates enable the Clog program to read or write relations to a tabular format (CSV or SQLite3) and they are most frequently used to read analysis parameters or to output analysis results. As the I/O predicates, the infinite predicates are identical to the ones used by *JavaDL*, and therefore we refer the reader to [12]. As syntactic sugar, Clog provides infix notations for comparison (==, <=, etc.) and variable binding (=).

2.6 Built-in Functions

Clog provides a set of predefined functions. These functions are free of side-effects and their use is allowed inside the operands of the comparison predicates or as the right operand of the = predicate. Clog requires that the arguments to the functions are either other expressions (i.e. arithmetic or function application) or that they are bound variables. The purpose of these functions is to expose properties of the analyzed program that are not expressible through syntactic patterns.

2.6.1 Type and Name Analysis Functions.

While type and name analysis for C can be expressed as a Datalog program, doing so bloats the analysis program and hinders readability. Therefore, Clog exposes these semantic properties through predefined functions[5].

- `type`(e) - the type of the expression e.
- `decl`(n) - the declaration of the the identifier n.

Because not all C constructs have a type and, for incomplete C programs, not all identifiers have a declaration, the `type`

[5] In the Clog implementation, these functions are prefixed by `c_`, e.g. `c_decl`, `c_name`, etc.

and `name` functions are partial. In such cases, they evaluate to a special value, `undef`.

2.6.2 Program Structure Functions. Clog provides convenience functions that enable the traversal of the program structure:

- `parent`(n) retrieves the parent term of n, if it exists.
- `enclosing_function`(n) retrieves the enclosing function of term n, if it exists.

2.6.3 Names. The Clog programs are general over the set of names, and thus identifiers can be replaced with metavariables inside syntactic patterns. However, metavariables can bind terms that may or may not have a name, for example the pattern ⟨: `$l + $r` :⟩ matches the expression `a + 1` and only the metavariable `$l` binds an identifier. To retrieve the variable name, we introduce a function `name`(n) that maps the term n to its name if it is an identifier or a named declaration, or to the empty string otherwise.

2.6.4 Control-Flow Functions. In addition to the control-flow predicates, Clog defines the `cfg_entry`(f) function that maps a term f to the entry node of its control-flow graph. Terms that have a CFG are function definitions, statements and expressions.

2.6.5 Source Location Functions. To support report generation, Clog defines a set of functions that retrieve the source location of a program term n: `src_line_start`(n), `src_col_start`(n), `src_file`(n), etc.

3 Implementation

3.1 Overview

We built the Clog prototype using two major components:

1. A Datalog implementation that we extended with support for syntactic patterns, functions and built-in predicates.
2. A Clang library, (*Clang-Clog*), which provides support for pattern matching, CFG predicates and built-in functions.

We chose to use Clang as the parser for the analyzed program over using our own C parser because we wanted Clog to be able to analyze real-world C programs that may use language extensions beyond the standard C grammar used by our parser. Moreover, by choosing a mature compiler such as Clang, we also gain access to other standard compiler analyses, such as name and type analysis and CFG construction.

In its implementation, Clog reuses infrastructure from previous systems that combine syntactic pattern matching and Datalog: *MetaDL* [10] and *JavaDL* [12]. Both these tools implement syntactic pattern matching in Datalog and translate the AST of the entire analyzed program to Datalog relations. We have experimented with the same approach in an early Clog prototype, but it proved impractical, because for C programs the AST contains nodes for all included files (transitively) and serializing the AST into Datalog relations dominated the running time of Clog. Instead, we opted for using Clang's own AST matching infrastructure, provided by the *LibASTMatchers*[6] library. This way, Clog could perform pattern matching directly on the Clang AST and only serialize to relations the AST fragments that match.

The *LibASTMatchers* provides a domain-specific language (DSL) for building up a tree of matchers from a predefined set of base matchers. For example, an exact matcher for the `int x;` variable declaration is

```
varDecl(hasName("x"), hasType(qualType(isInteger())),
        unless(hasInitializer(anything()))).bind("$m")
```

where `varDecl` matches a `VarDecl` AST node, `hasName` and `hasType` are predicates on its name and type; `unless` is a matcher that succeeds when its inner matcher fails (in this case, `hasInitializer`). The `bind` attribute specifies a name for the matching AST nodes.

To use the *LibASTMatchers* library, Clog translates the syntactic patterns to AST matchers. However, the Clog pattern grammar and the Clang abstract grammar have been designed for different purposes. On one hand, the goal for the Clog pattern grammar is to be close to the C grammar even after adding metavariables. On the other hand, the Clang grammar is optimized for compiler analyses.

To a C programmer, the pattern ⟨: `$t $f, *$g` :⟩ matches two variable declarations, a variable `$f` with type `$t` and another variable, `$g`, with type pointer-to-`$t`, without any initializer. In terms of the C grammar, this means that the type of `$g` is split between two non-terminals: the *type specifier* `$t` and the *declarator*, `*$g`. However, the Clang AST represents this using two disjoint entities: a `VarDecl` which has a `QualType` as its type attribute. This is reflected in the LibASTMatchers DSL as the matcher:

```
varDecl(hasType(qualType(pointsTo(qualType().bind("$t")))),
  unless(hasInitializer(anything())).bind("$g")
```

In the AST matcher, as in the Clang abstract grammar, the `qualType` matcher is an attribute of `varDecl`, while in the C grammar these nodes reside in different subtrees of the *declaration* non-terminal. This means that building the Clang AST matcher in the semantic actions is not feasible, thus Clog uses three intermediate translation steps. First, it parses the syntactic patterns to an *internal* AST, which contains the same non-terminals as the C11 grammar extended with metavariables. Secondly, it translates the *internal* AST to a *pseudo-Clang* AST, which contains the same AST nodes as the Clang AST, but it also allows concrete nodes from abstract grammatical categories such as *expression* (`clang::Expr`), *statement* (`clang::Stmt`) *declaration* (`clang::Decl`) or *qualified type* (`clang::QualType`), which are marked as being a metavariable or a gap. Thirdly, Clog traverses the *pseudo-Clang* AST and generates the AST matchers.

[6] https://clang.llvm.org/docs/LibASTMatchers.html

3.1.1 Clog Extensions to Clang AST Matching.

We have extended the set of AST matchers to cover all *pseudo-Clang* nodes generated by Clog.

In Clog, the gap construct `(..)` in syntactic patterns allows for parts of a list to remain unspecified, while still preserving the matching order of the list members that are specified. To support the semantics of gaps, we implemented a family of matchers that match a particular children list of a node against a list of sub-matchers. The sub-matchers must match the list elements in order, but the matched elements are not necessarily consecutive. For the pattern `⟨: $f($a, $b) :⟩`, Clog generates the matcher

```
callExpr(callee(expr().bind("$f")), argumentCountIs(2),
         hasArgument(0, expr().bind("$a")),
         hasArgument(1, expr().bind("$b")))
```

which matches function call expressions with precisely two arguments, while for `⟨: $f(..,$a,.., $b,..) :⟩`, it generates:

```
callExpr(callee(expr().bind("$f")),
    argumentDistinct(expr().bind("$a"), expr().bind("$b")))
```

The `argumentDistinct` matcher ensures that its inner matchers match distinct arguments of the call expression.

3.1.2 Limitations.

A fundamental limitation of Clog is that it matches the syntactic patterns against Clang ASTs, which means the metavariables always bind Clang AST nodes. Thus, we configured the pattern grammar generation to allow metavariables only for the terminals and non-terminals that have a corresponding Clang AST node. For example, Clog rejects patterns that have metavariables in place of qualifiers, such as `⟨: $q int *$f :⟩` because Clang represents qualifier as fields of the `clang::QualType` node. However, the pattern `⟨: const int *$f :⟩` is valid, because Clog can check that the AST node bound by `$f` is a pointer to a `const`-qualified integer.

While the use of Clang AST can enable future extensions of Clog to analyze C++ programs, this comes with challenges. The most significant challenge is that the language must introduce a distinction between terms existing in the source and terms arising from template instantiation, `auto` type deduction and default methods.

3.2 Clog at Runtime

Figure 9 provides a runtime view of the Clog implementation. The evaluation of a Clog program proceeds with the parsing of the analysis code. The parsing of syntactic patterns is deferred to the *Pattern parser*. Then the semantic analysis phase ensures that the program is well formed. This is followed by a plan generation phase, where Clog generates an evaluation plan according to the semi-naive evaluation strategy [31]. In the evaluation phase, Clog executes the evaluation plan to produce the results.

3.2.1 Clang-Clog Interface.

Clog interfaces with Clang through a library built using the Clang *LibTooling* support library[7]. This *Clang-Clog* library handles the parsing of the analyzed sources and builds the ASTs for all sources. In addition to building the ASTs, it supports pattern matching through Clang's *LibASTMatchers* library, the evaluation of built-in functions and CFG queries.

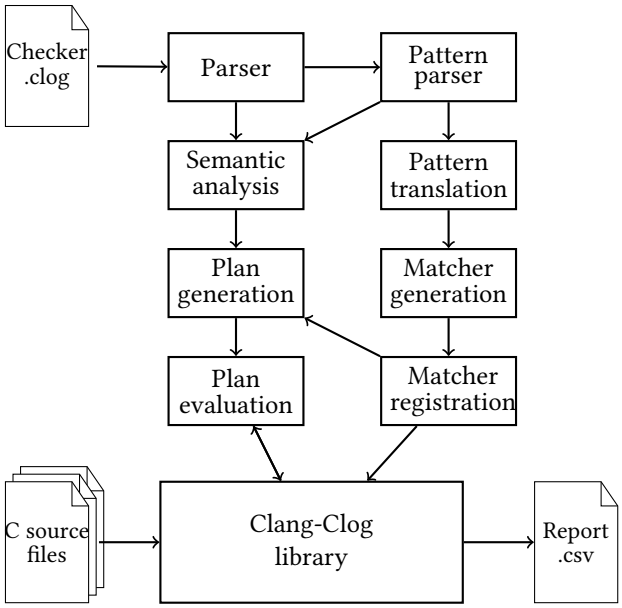

Figure 9. Phases of a Clog checker evaluation

3.2.2 Pattern Parser.

The Clog parser defers the parsing of patterns to an Earley parser [29], which is capable of handling general context-free grammars and ambiguity. Support for ambiguity is necessary, since the patterns lack context to disambiguate cases such as `t * a;` or `f(x);`[8]. The pattern grammar follows the C grammar given in the Annex A of the C11 language standard [1]. This grammar is automatically extended when Clog is built, to accept metavariables and gaps for a configurable set of non-terminals. The result of the pattern parsing phase is an AST with nodes from the *internal grammar*.

3.2.3 Pattern Translation.

During the pattern translation phase, Clog translates the pattern ASTs from the *internal* grammar to a *pseudo-Clang* grammar.

3.2.4 Matcher Generation.

In this phase, Clog traverses the *pseudo-Clang* AST and generates matchers in the LibASTMatchers DSL.

The translation from the pattern ASTs to the AST matchers is not always one-to-one. Due to ambiguities in the C language, Clog generates two AST matchers for the pattern `⟨: $t *$p :⟩`, one for the pointer declaration and another for the multiplication. This is reflected in the evaluation plan as a disjunction of two literals.

[7] https://clang.llvm.org/docs/LibTooling.html
[8] A variable `x` of typedefed type `f` or a function call?

3.2.5 Matcher Registration.

In this phase, Clog registers the matchers with the Clang-Clog library. The Clang-Clog library parses the matcher DSL, builds the AST matchers and returns a unique identifier for each matcher, that the Datalog engine uses during the plan evaluation phase for retrieving the results of the match. Clog registers a matcher a single time to avoid the cost of parsing the matcher DSL each time it evaluates a pattern literal.

The Clang-Clog library supports three kinds of pattern matchers:

- *node matcher* - attempts to match one given node;
- *subtree matcher* - finds all matches in all descendants of one given node;
- *global matcher* - finds all matches across all ASTs.

All pattern literals without a subtree restriction or a root node, of the shape ⟨: C :⟩, correspond to *global matchers*. For the ones that have a root literal, but not a subtree restriction, `r` ⟨: C :⟩, Clog generates a *node matcher* if the root variable `r` is bound by other literals in the clause. Otherwise, it generates a global matcher. Patterns with a subtree restriction, `@s r` ⟨: C :⟩, result in *subtree matchers*.

3.2.6 Plan Generation and Evaluation.

In this phase, Clog translates the Datalog rules to an evaluation plan. The operations of this plan are similar to the Relational Algebra Machine used by the *Soufflé* Datalog engine [27].

In contrast to the approach in [12], where the entire AST of the analyzed program is materialized as Datalog relations, in Clog we have implemented an *on-demand* approach, where we only materialize the AST nodes that are matched by the syntactic patterns and the ones that are exposed through the CFG predicates and built-in functions.

In addition, for pattern literals, we optimize for cases where the root variable of the pattern is bound by a literal occurring earlier in the clause. In this case, Clog runs a *node matcher* on the root node, instead of a *global matcher* on the entire AST. In the current implementation, the plan generator preserves the order of literals in the clause, with the exception of the infinite and binding predicates, which it may reorder.

For the CFG_SUCC and CFG_EXIT predicates, the Clang-Clog library lazily computes the CFGs only for functions for which these predicates are queried. In effect, this means that the CFG is computed only for functions relevant to the analysis.

The evaluation of the analysis proceeds with running the global matchers. This is handled by the Clang-Clog library using Clang's AST matching API, which runs all the global matchers in a single traversal over the whole AST. This stage is followed by the execution of the Datalog plan. In the final stage, Clog writes the output relations to disk.

Table 1. A mapping from common weaknesses (CWEs) to Juliet test sets. We omit CWEs that are not relevant to C programs.

#	CWE	Juliet tests
1	CWE-787 Out-of-bounds Write	CWE121_Stack_Based_Buffer_Overflow CWE122_Heap_Based_Buffer_Overflow CWE123_Write_What_Where_Condition CWE124_Buffer_Underwrite
4	CWE-416 Use After Free	CWE416_Use_After_Free
5	CWE-78 Improper Neutralization of Special Elements Used in an OS Command	CWE78_OS_Command_Injection
6	CWE-20 Improper Input Validation	CWE134_Uncontrolled_Format_String CWE114_Process_Control CWE190_Integer_Overflow CWE785_Path_Manipulation_Function ...
7	CWE-125 Out-of-bounds Read	CWE126_Buffer_Overread CWE127_Buffer_Underread
8	CWE-22 Improper Limitation of a Pathname to a Restricted Directory	CWE23_Relative_Path_Traversal CWE36_Absolute_Path_Traversal
12	CWE-476 Null Pointer Dereference	CWE476_NULL_Pointer_Dereference
14	CWE-190 Integer Overflow or Wraparound	CWE190_Integer_Overflow

4 Evaluation

In this section we ask the following questions:

1. ***RQ1*** How capable is Clog to express code checkers with good precision and recall rates?
2. ***RQ2*** How does the execution time of Clog compare to other tools?

To answer these question we implemented several checkers in Clog and ran them on synthetic benchmarks and on real programs. We compared our results against the Clang Static Analyzer (CSA). We chose CSA because both itself and Clog have access to the same underlying AST and we wanted to assess the analysis expressivity and speed, without comparing the underlying AST representations. To compare with CSA[9], we ran the relevant checkers using the *clang-tidy* frontend. We used an AMD EPYC 7713P 64-Core Processor with 504 GB RAM for our evaluation.

4.1 Synthetic Benchmarks

In line with earlier work [8, 32], we adopted the Juliet 1.3[10] test suite in our evaluation. Juliet is a collection of synthetic tests aimed at assessing static analysis tools.

4.1.1 Experimental Setup.

To evaluate the expressivity of Clog we attempted to implement checkers for the first 15 weaknesses listed in the *2023 CWE Top 25 Most Dangerous Software Weaknesses* published by MITRE [11] and run these checkers on Juliet. For the cases where the Juliet suite did not contain tests for a specific weakness, we exploited the hierarchical organization of the CWE database and we used the tests for the weakness' direct descendants. We present our mapping from listed CWEs to Juliet test sets in Table 1.

[9] Built from https://github.com/llvm/llvm-project@98acd74683
[10] https://samate.nist.gov/SARD/test-suites/112
[11] https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

For each Juliet test set corresponding to a CWE we implemented a checker in Clog. We explicitly list the enabled checkers in Table 2. In our analysis development, we followed an iterative process in which we aimed for increasing the checker's recall, without having a precision that is lower than the precision of the Clang checker.

4.1.2 Discussion. In Table 2 we present a comparison regarding precision, recall and running times of the CSA and Clog checkers.

To detect ***CWE-78 OS Command Injection*** we implemented an inter-procedural data-flow analysis. CSA did not produce any warnings, even though we configured the taint analysis to use the same propagation rules as Clog. We adapted the same checker for ***CWE-134 Uncontrolled Format String***, with a different set of taint sources and sinks.

To achieve high recall on the test sets ***CWE-121 Stack Based Buffer Overflow***, ***CWE-122 Heap Based Buffer Overflow***, ***CWE-124 Buffer Underwrite***, ***CWE-126 Buffer Overread***, ***CWE-127 Buffer Underread*** the checkers need to perform constant propagation or numerical domain analysis. Both these analyses are difficult to encode in Datalog without general lattice support, which is a limitation of our implementation.

These checkers also brought to light a mismatch between the pattern grammar and the Clang AST grammar. The pattern for matching array declarations ⟨`:$t $a[$n]:`⟩ allows an expression for the array length, `$n`. However, the Clang AST contains two nodes for arrays: one for fixed-length and another for variable-length arrays. The fixed-length array node does not have a child node for encoding the length, but it encodes it explicitly, as an integer field. In effect, this means that there is no AST node that the variable `$n` can bind, and this is not compatible with our assumption, that each metavariable binds an AST node. Fortunately, this is not a fundamental limitation of our translation scheme, since it can be circumvented by adding a custom AST matcher which instantiates a constant integer AST node when matching the size of an array.

In the ***CWE-416 Use after Free*** checker we implemented an intra-procedural data-flow analysis. An intra-procedural analysis proved sufficient, because freed pointers are reported when passed as arguments or return values.

To detect ***CWE-476 Null Pointer Dereference*** we implemented an inter-procedural data-flow analysis. To reduce false positives, we added rules that exclude paths dominated by a null test of a variable. One of these rules is

```
NotNullPath(s, d) :- ⟨:if ($v) $t else $f:⟩,
    d = decl($v), d != undef, s = cfg_entry($t).
```

where we define a predicate, NotNullPath, to mark that a variable `d` is not null on paths starting from `s`.

To achieve good precision and recall on the ***CWE-190 Integer Overflow*** test set, the analysis needs to perform numerical domain analysis, a Clog limitation we have also seen earlier, on the tests sets for ***CWE-121,122,124,126,127***. Detecting cases of ***CWE-123 Write What Where Condition*** requires heap modeling, which out of the scope of this work. CSA also fails to report any warning on this test set. The test sets ***CWE-114***, ***CWE-785*** use the Windows API, while ***CWE-23*** and ***CWE-36*** contain only C++ sources. We did not implement checkers for these test sets.

In Table 3 we list the sizes of Clog programs, counted as number of rules. We were able to implement all checkers concisely, with no more than 35 rules and most of them with 20-30 rules. We also note that, for all checkers, the number of syntactic patterns is close to the number of rules, which shows that they are a well-utilized language feature.

Looking back at ***RQ1***, we conclude that Clog is expressive enough to encode high precision checkers for typical data-flow analyses, even inter-procedural. However, we encountered difficulties in achieving good recall on the checkers for buffer accesses. This class of vulnerabilities highlighted a limitation of Clog: it can't express analyses that use lattices other than the power set. Fortunately this is a known limitation of Datalog and is addressed by approaches orthogonal to ours [22, 30].

In reply to ***RQ2***, in Table 2, we observe that the running times of Clog are about 2-3 times slower than CSA, but this is not surprising if we consider that we implemented Clog partly as a Java application, partly as a native library and the two do not share a heap, so all the results of pattern matching must be copied from a native heap to the Java heap. In spite of this, the fact that Clog is fully declarative enables other optimization approaches, such as parallelizing the evaluation engine or incrementalization, in the style of *IncA* [30] or *JavaDL* [12].

4.2 Realistic Workloads

Inspired by a study on the effectiveness of static C code analyzers [21], we have reused programs from the Magma v1.1 fuzzing benchmark [14] as targets for our testing. Magma contains a set of programs with known vulnerabilities and their respective fixes. Conveniently, these fixes can be enabled or disabled through a preprocessor symbol. To avoid assumptions about how the location of a fix corresponds to the location of a report, we adopt a *differential approach*: we compare the reports between the fixed and the faulty versions of a program.

From the checkers we have implemented for the Juliet benchmark, we selected the ones with good recall and full precision that also have corresponding vulnerabilities in the Magma programs: ***CWE-416*** and ***CWE-476***.

From the Magma programs, we selected *openssl*, *sqlite*, *libxml2* and *libpng*. We were not able to properly extract the compilation commands for *libtiff* and *php* and we discarded *poppler* for being mostly a C++ project.

Table 2. CSA and Clog results on Juliet test sets. *ArrayBoundsConfig* represents the checkers: alpha.security.ArrayBound, alpha.security.ArrayBoundV2, alpha.security.taint.*, alpha.unix.cstring.OutOfBounds, security.insecureAPI.*. GT is the number of ground-truth reports.

CWE	CSA checkers	GT	True positive		False positive		Precision		Recall		Time (s)	
			CSA	Clog	CSA	Clog	CSA	Clog	CSA	Clog	CSA	Clog
CWE-78	alpha.security.taint.*	1520	0	1008	0	0		100.00	0	66.31	35.82	119.39
CWE-121	*ArrayBoundsConfig*	4036	2132	240	6810	20	23.84	92.30	52.82	5.94	80.14	171.09
CWE-122	*ArrayBoundsConfig*	2606	1174	240	3542	20	24.89	92.30	45.04	9.20	50.84	128.46
CWE-124	*ArrayBoundsConfig*	1288	720	300	3593	0	16.69	100.00	55.90	23.29	27.22	73.01
CWE-126	*ArrayBoundsConfig*	972	449	160	3180	0	12.37	100.00	46.19	16.46	20.11	46.40
CWE-127	*ArrayBoundsConfig*	1288	720	300	3593	0	16.69	100.00	55.90	23.29	27.12	73.15
CWE-134	alpha.security.taint.*	1900	570	1278	780	0	42.22	100.00	30.00	67.26	48.98	312.61
CWE-416	unix.Malloc	138	36	108	0	0	100.00	100.00	26.08	78.26	2.27	2.65
CWE-476	core.NullDereference	270	174	150	16	0	91.57	100.00	64.44	55.55	4.73	6.27

Table 3. Predicate, rule and pattern literal counts for Clog programs

CWE	Predicates	Pattern literals	Rules
CWE-78	19	18	24
CWE-121	8	23	25
CWE-122	8	23	25
CWE-124	7	18	19
CWE-126	8	22	24
CWE-127	9	17	20
CWE-134	20	18	25
CWE-416	13	23	28
CWE-476	14	31	35

Table 4. CSA and Clog report numbers and running times on Magma test programs. *V*-columns refer to the vulnerable version, *F*-columns refer to the fixed version.

CWE	Program	CSA		Clog		Time (s)	
		V	F	V	F	CSA	Clog
CWE -476	libpng	0	0	0	0	29.87	3.73
	openssl	25	24	217	217	452.78	191.14
	sqlite3	18	17	387	383	1109.19	155.86
	libxml2	33	33	425	425	349.69	23.47
CWE -416	libpng	0	0	0	0	31.80	1.31
	openssl	0	0	0	0	492.01	61.55
	sqlite3	0	0	0	0	1183.19	4.59
	libxml2	0	0	0	0	372.68	3.40

In Table 4 we present the results of running Clog and CSA analyses on the selected Magma programs. For ***CWE-476***, both checkers discovered a real issue in *sqlite*, while for ***openssl*** only CSA succeeded. For the *sqlite* issue, CSA generates one report, while Clog generates four. The difference is that CSA reports only the first dereference of a null pointer, while Clog reports all. In the ***openssl*** case, our implementation of the null pointer dereference checker does not handle uninitialized variables, while CSA does. This is a limitation of our analysis set, but not of Clog itself, since an uninitialized variable checker is another instance of dataflow analysis, a class of analyses we have showed that Clog can express. For ***CWE-416***, both CSA and Clog fail to find any cases of use-after-free vulnerabilities, even though one vulnerability is present in each of the *libxml2*, *sqlite3* and *openssl* programs. Contrary to results we have seen on the Juliet test suite, the running times for Clog are significantly better than for CSA.

5 Related Work

Declarative approaches to the static analysis of program semantics have a rich history based on a variety of techniques, including attribute grammars [17], pattern-matching on algebraic data types [2], term rewriting [16], logical queries [23], flow-sensitive types [13], and combinations such as Cobalt's use of integrated modal logic and term rewriting [20].

Most approaches operate on some representation of the AST and/or CFG, often involving intermediate code; custom extensions must then follow these (generally tool-specific) abstractions [3, 24, 28]. This is a well-understood challenge in the field of API protocol checking [5, 13], which attempts to identify API-specific bugs. Some tools try to address this challenge by encoding analysis rules in an internal DSLs (e.g., embedding them inside API code via Java annotations [5]). However, these approaches are limited by the host language's annotation facilities.

External DSLs based on syntactic pattern matching therefore offer an appealing alternative. For C, *Coccinelle* [18] offers facilities for code matching and transformation, based on syntactic patterns. In contrast to Clog, the *Coccinelle* DSL that describes the syntactic patterns requires the user to explicitly declare metavariables and their syntactic categories. *Coccinelle* provides a scripting interface (Python and OCaml) that the users can use to combine pattern matching with their custom analysis and build bug finding tools [19]. The scripting interface serves approximately the same purpose as the Datalog language in Clog.

Other tools that combine syntactic pattern matching with logical queries include *SOUL* [9], which combines AST pattern matching with Prolog-style logic programs to analyze Java programs, though *SOUL* restricts patterns to five predefined syntactic categories, and *JavaDL* [12], which is closest to our work in spirit in that it offers Datalog-style rules and syntactic pattern matching, on Java programs. Clog reuses parts of the *JavaDL* infrastructure, specifically the grammar transformation mechanism and *JavaDL*'s Datalog implementation. Unlike Clog, *JavaDL* offers no built-in predicates for CFG traversal, which limits its ability to express flow-sensitive analyses. Internally, *JavaDL* performs pattern matching by encoding ASTs as tables of Datalog facts and relying on an external Datalog engine, while Clog translates syntactic patterns to queries for the Clang AST matching mechanism and materializes only the results of these queries as Datalog tuples.

Clog is not the first Datalog-style tool for C: *cclyzer* [4] implements declarative points-to analyses for C and C++ in Datalog. Compared to Clog, which works on the AST, the *cclyzer* analyses are implemented in terms of LLVM IR instructions. Like *JavaDL*, *cclyzer* relies on the high-performance Datalog engine *Soufflé* [27]. Despite its wide adoption for program analysis tasks [4, 6], *Soufflé* itself does not extend the Datalog language with features particularly geared towards program analysis.

6 Conclusions

We have shown that the Clog language can express powerful custom checkers for C code without exposing program representation internals. While our experiences suggest that implementing a language like Clog on top of Clang may require nontrivial internal plumbing to align program structure and control flow, we find that the cost for this abstraction is modest: Clog-based checkers may even run faster than Clang's own checkers, despite delivering competitive results. Like most Datalog-based approaches, Clog is limited to boolean (product) lattices but scales to a wide variety of practical analyses, including interprocedural and dataflow analyses. Overall, we argue that Clog's combination of language simplicity, expressivity, and performance make it uniquely suited for building custom C code checkers.

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

The authors thank Magnus Templing and Patrik Åberg for encouraging this work and for their constructive feedback. We also thank Dániel Krupp for insightful comments on an early version of Clog. The initial prototype of Clog was developed during the first author's internship at Telefonaktiebolaget LM Ericsson.

References

[1] [n. d.]. *ISO/IEC 9899:2011 – Information technology – Programming languages - C*. Standard. International Organization for Standardization.

[2] Andrew W. Appel. 1998. *Modern Compiler Implementation: In ML* (1st ed.). Cambridge University Press, USA.

[3] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[4] George Balatsouras and Yannis Smaragdakis. 2016. Structure-sensitive points-to analysis for C and C++. In *International Static Analysis Symposium*. Springer, 84–104. https://doi.org/10.1007/978-3-662-53413-7_5

[5] Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. 2009. Practical API protocol checking with access permissions. In *ECOOP 2009–Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings 23*. Springer, 195–219.

[6] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of OOPSLA '09* (Orlando, Florida, USA). ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[7] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R Newton. 2017. Instruction punning: Lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 320–332.

[8] Xi Cheng. 2016. RABIEF: range analysis based integer error fixing. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 1094–1096.

[9] Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. 2011. The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java* (Kongens Lyngby, Denmark) *(PPPJ '11)*. Association for Computing Machinery, New York, NY, USA, 71–80. https://doi.org/10.1145/2093157.2093168

[10] Alexandru Dura, Hampus Balldin, and Christoph Reichenbach. 2019. MetaDL: Analysing Datalog in Datalog. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM, 38–43. https://doi.org/10.1145/3315568.3329970

[11] Alexandru Dura and Christoph Reichenbach. 2024. *Clog: A Declarative Language for C Static Code Checkers*. https://doi.org/10.5281/zenodo.10525151

[12] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. 2021. JavaDL: automatically incrementalizing Java bug pattern detection. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–31. https://doi.org/10.1145/3485542

[13] Jeffrey S Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 1–12.

[14] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2022. Magma: A Ground-Truth Fuzzing Benchmark. *SIGMETRICS Perform. Eval. Rev.* 49, 1 (jun 2022), 81–82. https://doi.org/10.1145/3543516.3456276

[15] Stephen C Johnson. 1977. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill.

[16] Lennart C.L. Kats, Martin Bravenboer, and Eelco Visser. 2008. Mixing Source and Bytecode: A Case for Compilation by Normalization. *SIGPLAN Not.* 43, 10 (Oct. 2008), 91–108. https://doi.org/10.1145/1449955.1449772

[17] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (1968), 127–145.

[18] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 years of automated evolution in the Linux kernel. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 601–614.

[19] Julia L Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. 2009. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 43–52. https://doi.org/10.1109/DSN.2009.5270354

[20] Sorin Lerner, Todd Millstein, and Craig Chambers. 2005. Cobalt: A Language for Writing Provably-Sound Compiler Optimizations. *Electronic Notes in Theoretical Computer Science* 132, 1 (2005), 5–17. https://doi.org/10.1016/j.entcs.2005.03.022 Proceedings of the 3rd International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2004).

[21] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 544–555. https://doi.org/10.1145/3533767.3534380

[22] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: a Declarative Language for Fixed Points on Lattices. *ACM SIGPLAN Notices* 51, 6 (2016), 194–208. https://doi.org/10.1145/2980983.2908096

[23] Michael Martin, Benjamin Livshits, and Monica S. Lam. 2005. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 365–383. https://doi.org/10.1145/1094811.1094840

[24] Krishna Narasimhan, Christoph Reichenbach, and Julia Lawall. 2017. Interactive Data Representation Migration: Exploiting Program Dependence to Aid Program Transformation. In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Paris, France) *(PEPM 2017)*. ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/3018882.3018890

[25] Peter O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95.

[26] Dennis M. Ritchie. 1993. The Development of the C Language. In *The Second ACM SIGPLAN Conference on History of Programming Languages* (Cambridge, Massachusetts, USA) *(HOPL-II)*. Association for Computing Machinery, New York, NY, USA, 201–208. https://doi.org/10.1145/154766.155580

[27] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-scale Program Analysis in Datalog. In *Proceedings of the 25th Int. Conf. on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. ACM, New York, NY, USA, 196–206. https://doi.org/10.1145/2892208.2892226

[28] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 393–410. https://doi.org/10.1007/978-3-030-17465-1_22

[29] Elizabeth Scott. 2008. SPPF-style parsing from Earley recognisers. *Electronic Notes in Theoretical Computer Science* 203, 2 (2008), 53–67. https://doi.org/10.1016/j.entcs.2008.03.044

[30] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing Lattice-Based Program Analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 139 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276509

[31] J. D. Ullman. 1989. Bottom-up Beats Top-down for Datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Philadelphia, Pennsylvania, USA) *(PODS '89)*. Association for Computing Machinery, New York, NY, USA, 140–149.

[32] Andreas Wagner and Johannes Sametinger. 2014. Using the juliet test suite to compare static security scanners. In *2014 11th International Conference on Security and Cryptography (SECRYPT)*. IEEE, 1–9.

Received 13-NOV-2023; accepted 2023-12-23