

Customizing Static Analysis using Codesearch

Avi Hayoun
avi.hayoun@gmail.com
Snyk

Veselin Raychev
veselin.raychev@insait.ai
INSAIT
Sofia University
Bulgaria

Jack Hair
jack.hair@snyk.io
Snyk

April 22, 2024

Abstract

Static analysis is a growing application of software engineering, leading to a range of essential security tools, bug-finding tools, as well as software verification. Recent years show an increase of universal static analysis tools that validate a range of properties and allow customizing parts of the scanner to validate additional properties or “static analysis rules”. A commonly used language to describe a range of static analysis applications is Datalog. Unfortunately, the language is still non-trivial to use, leading to analysis that is difficult to implement in a precise but performant way. In this work, we aim to make building custom static analysis tools much easier for developers, while at the same time, providing a familiar framework for application security and static analysis experts. Our approach introduces a language called StarLang, a variant of Datalog which only includes programs with a fast runtime by the means of having low time complexity of its decision procedure.

1 Introduction

Static analysis is a growing application of software engineering, leading to a range of essential security tools [8, 17], bugfinding tools [7, 9] as well as software verification [1, 43]. The general idea of static analysis is that some properties of a program are validated without running the code, but instead by reasoning about its possible behaviors. In many cases, these tools are built from the ground up for verifying specific properties such as memory safety [1], concurrency properties, security properties [16], type safety [32] or others based on different techniques such as abstract interpretation [22], shape analysis [39], taint analysis [16] and others. Recent years show an increase of universal static analysis tools such as CodeQL [17], SemGrep [7], Snyk Code [8], and SonarSource [9] that validate a range of properties and allow customizing parts of the scanner to validate additional properties or “static analysis rules”.

This level of customization is usually achieved by unifying parts of the static analysis that are common, such as code parsing, value propagation [41], dataflow analysis [38], pointer analysis [14, 44], tpestate analysis [26], and taint analysis [48], and allowing the rules to be written using a domain-specific language or calls to an internal library that provide the base analyses.

A commonly used language to describe a range of static analysis applications is Datalog [13, 34, 42]. The language is not Turing complete, yet it was shown to be useful for expressing a range of analyses, such as points-to [44], security analyses [28] and others. Unfortunately, the language is still non-trivial to use, leading to analysis that is difficult to implement in a precise but performant way [12, 35, 47]. As a result, it is often not suitable for allowing end-users that are non-experts to write rules. This has led to some of the latest tools providing easier-to-use domain-specific languages to describe their static analysis rules [7, 8, 17].

This work In this work, we aim to make building custom static analysis tools much easier for developers, while at the same time, providing a familiar framework for application security and static analysis experts. Our approach introduces a language called StarLang, a variant of Datalog which only includes programs with a fast runtime by the means of having low time complexity of its decision procedure. This requirement limits the expressiveness of StarLang, but has the benefit that finding matches in code using StarLang (i.e. matching static analysis alarms) is always fast and allows users to see code matches in real-time as they author a rule. In addition to this, our system can provide autocompletion to help make writing rules even faster and easier.

An interesting observation is that despite StarLang being only as expressive as a subset of Datalog, the language still admits a wide range of useful real-world static analyses. A core reason why this is possible and easy is a *templates* language feature that we introduce. Templates allow hiding Datalog recursion while providing useful high-level abstractions, making the queries easier to read while enforcing the usage of the Datalog subset that is fast and efficient.

Example 1. We demonstrate the power of StarLang with a simple example in which we would like to discover read-after-close violations using static analysis, i.e., cases of reading from a file after closing it. Consider the following simplified piece of code containing three statements that manipulate a file in a JavaScript-like language:

```
1 let f = file(); f.close(); f.read();
```

In this case, we are looking for a call to `read` and need to follow the dataflow of the receiver object of this call (as a convention, the receiver object is sometimes called argument 0). In this case, one can write the following search query:

```
1 CallExpression<"read"> and
2 HasArg0<DataFlowAfter<Arg0In<CallExpression<"close">>>>
```

Following the logic above, the first line of the query matches the call expression to `read`. The second line is a sequence of templates that describe the other checked property — the receiver object has a call to `close` before that. To perform this query, Snyk Code builds a graph that represents the dataflow of the queried code snippet. This graph contains the nodes for the call expressions, as well as edges for the dataflow in the code. Then, each of the StarLang queries or subqueries returns nodes in this graph that match their described properties. For example, `CallExpression<"read">` will return every node in the graph that describes a call to `read`. The other templates `HasArg0`, `DataFlowAfter` and `Arg0In` traverse edges in the graph, but still always return graph nodes that satisfy the

given property, e.g., `HasArg0` matches on nodes that have receiver objects with the property given in the template. Some of these templates may also expand into recursive unary predicates, e.g., `DataFlowAfter` will match even if there are multiple dataflow steps (i.e., multiple edges) that need to be traversed in the dataflow graph.

Alternative approaches An alternative approach promoted by SemGrep [7], has been to perform syntactic pattern matching that would allow discovering the pattern above. For example, one can essentially use an expression like `$1.close(); ... $1.read()` to find if there is a variable (or other symbol) that had a sequence of `close` and `read` operations. Unfortunately, this approach is too syntactic in many cases. As opposed to StarLang, that operates on semantic relations such as dataflow, a syntactic matching approach requires that the specific code pattern matches. As a result, the user needs to modify the rule if they would want to match semantically equivalent, but syntactically different alternative code examples such as the following snippet:

```
1   function func(param) { param.close(); }
2   let f = file(); func(f); f.read();
```

On the other hand, Snyk Code correctly discovers the read-after-close violation in this snippet using its interprocedural analysis and the same, unmodified StarLang query.

Contributions The contributions of this work are:

- We define StarLang – a language as expressive as Datalog on unary predicates and show the time complexity of its decision procedures.
- We propose a system – Snyk Code – that computes dataflow, taint, points-to and other analyses and allows performing StarLang queries in realtime over large repositories. Figure 1 presents a schematic overview of the Snyk Code system.
- We demonstrate the expressiveness of StarLang on a range of useful queries that cover taint security properties, memory safety, tpestate safety and others.
- We present a frontend to StarLang called Codesearch that allows users to interactively built static analysis queries, significantly simplifying the job of security professionals and static analysis authors.

This paper focuses primarily on the parts of Figure 1 marked with a \star , and is structured as follows: In Section 2, we provide an short introduction to Datalog and Relational Algebra. Section 3 formally defines StarLang and the efficiency benefits of its limited expressiveness. In Section 4 we present Codesearch, a StarLang interface that allows users to define interesting custom static analysis queries. This is demonstrated in Section 5, in which we provide examples of such interesting static analysis queries that can be expressed concisely and simply using Codesearch. Finally, Section 6 covers related work.

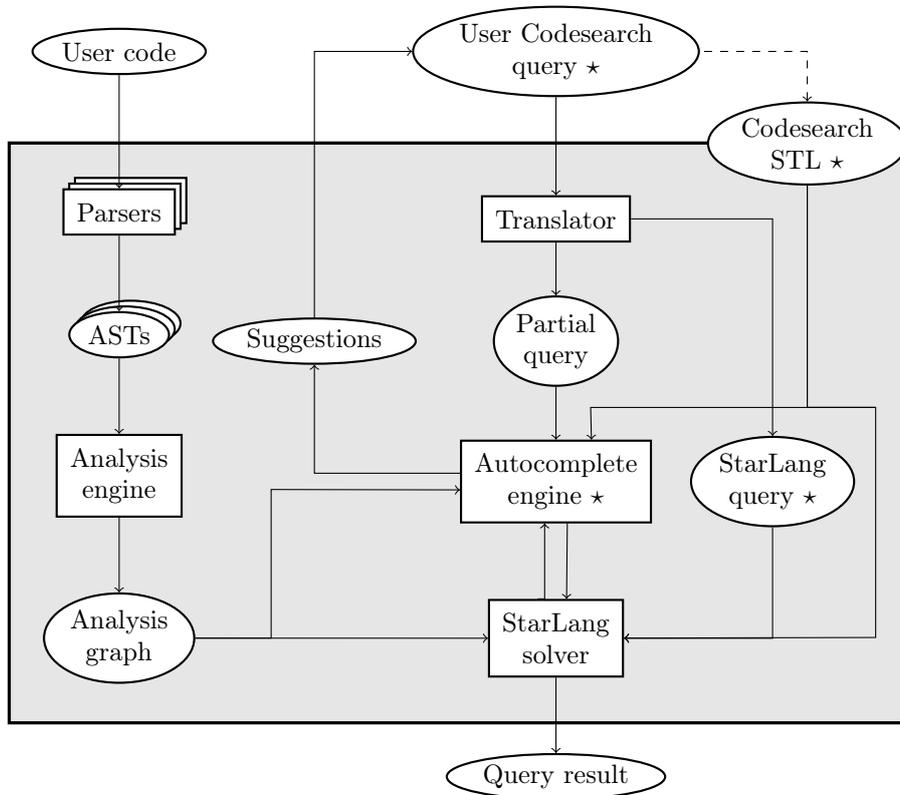


Figure 1: The Snyk Code Codesearch system. The components in the gray box comprise the system itself, with the inputs and outputs placed above and below respectively. Rectangles represent computations, while ellipses represent data. The “Codesearch STL” (STandard Library) node is on the border of the system, acting as an API to the underlying StarLang language.

2 Preliminaries

2.1 Datalog

Datalog is a declarative logic programming language that is often used as a query language for deductive databases, and has been employed extensively for program analysis [13, 45, 12, 42].

A Datalog program consists of a set of Horn clauses [31], called *rules*. Rules are often encoded as implications: $u :- p, q, \dots, t.$, where $:-$ represents the implication arrow (\leftarrow), the commas ($,$) represent conjunctions (\wedge), $.$ is the expression delimiter, and $u, p, q,$ etc. are predicates. Logically, a rule is read as: “if p, q, \dots, t hold, then u holds”. u is commonly referred to as the *head* of the rule, while $p, q, \dots, t.$ is referred to as the rule *body*. An empty rule body is shorthand for a rule body that holds trivially (i.e., is simply *true*). Rules with empty bodies are referred to as *facts*, i.e. they always hold. As usual, predicates are claims of some n -ary relation holding for a given set of values, denoted using the standard notation: $r(X_0, \dots, X_{n-1})$, where each X_i

is either a concrete value in, or a variable over the universe of relation r . Specific predicate instances are sometimes referred to as *atoms*.

Most variants of Datalog support the use of an equality predicate ($=$) over values in the same universe in rule bodies.

A Datalog engine executes the specification for a set of input relations and produces an output relation for a query. The input relations of a Datalog program are referred to as the *Extensional Database* (EDB), and are treated as a set of facts. The rules encoding the logic program are referred to as the *Intensional Database* (IDB).

Example 2. The following Datalog program P computes the ancestor relations relative to an EDB:

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Consider the following EDB D :

```
father(john,mary).
father(joe,kurt).
mother(mary,joe).
father(tine,kurt).
```

Applying program P to D produces facts such as $\text{ancestor}(\text{mary}, \text{joe})$. and $\text{ancestor}(\text{john}, \text{joe})$.

The basic variant of Datalog does not allow for negating any terms in any rules. While queries of this variant can be solved very efficiently, it has limited use: Clearly, by disabling input negation altogether, one loses the capability of detecting the absence of database information, which restricts the expressivity of the formalism to queries satisfying some monotonicity property. Extending Datalog to include a limited form of negation, called *stratified negation* [20, 29] (DL^\neg) extends the expressiveness, at the cost of higher computational complexity [24]. Intuitively, a logic program is a valid program in DL^\neg if it contains no rule that is defined recursively in terms of its own negation (a *negative cycle*). More formally, a program is a valid program in DL^\neg if its IDB rules can be topologically sorted using a straightforward dependency-based ordering: let r , and p be two predicate symbols. $p \leq r$ if p appears in a rule body of r . If p appears negated in a rule body of r , then $p < r$.

Remark 1. Note that this definition admits non-negative recursion: $r :- r$. admits the valid ordering $r \leq r$, but $r :- \neg r$. admits the absurd ordering $r < r$.

Example 3. The rule $p(X,Y) :- p(X,Z), \neg q(Y, Z)$. is a valid DL^\neg rule (assuming q is not defined in terms of p).

A Datalog program belongs to monadic Datalog (MDL), if all of its rule heads are unary (i.e., have arity 1).

Example 4. The rule $p(X,Y) :- q(X,Y), r(Y)$. is not valid in MDL, since $p(X,Y)$ is not unary. However, $p(X) :- q(X,Y), r(Y)$. is a valid MDL rule, given that q is an extensional predicate.

2.1.1 Computational Complexity

There are several ways to define the complexity of queries or logic programs [19]. They differ in the parameters with respect to which the complexity is measured. The three common complexities in the context of Datalog are:

- **Data Complexity:** the complexity of evaluating a fixed query for variable database inputs.
- **Program Complexity:** the complexity of evaluating, on a fixed database instance, the various queries specifiable.
- **Combined Complexity:** the complexity of evaluating the various queries specifiable for variable database inputs.

In the setting of static program analysis, the logic program or query is often pre-determined based on the type of analysis being performed and alarms we would like to detect, and is thus usually considered constant. This means that Data Complexity is the relevant measure in our setting.

2.2 Relational Algebra

First introduced by Codd [21], relational algebra is a theory that uses algebraic structures for modeling data, and defining queries on it with a well founded semantics [27]. Relational algebra provides a theoretical foundation for relational databases, its main purpose being the definition of operators that transform one or more input relations to an output relation.

2.2.1 Operators

Since relations are sets, all of the usual set operations are applicable to them. Additionally, the following operations on relations are defined:

Selection Denoted with σ_p , where p is some predicate the domain of which are the elements of the relation to which the selection is being applied. It is used to derive a sub-relation R' from a given relation R such that $R' \subseteq R$ and $\forall r \in R', p(r)$.

Projection Denoted with π_i , with i being the index of an entry in the tuples of a given relation R . It is used to derive a unary relation R' from R such that $\forall r' \in R', \exists r \in R$ such that the value of r' , x , is the i th entry in r : $r = (y_0, \dots, y_i = x, \dots, y_n)$.

Natural Join Denoted with \bowtie . It is used to derive a relation from two given relations, which contains the catenation of tuples that have a matching element at their respective ends: Given relations R_1 of k -tuples and R_2 of m -tuples, $R_1 \bowtie R_2 = \{(x_0, \dots, x_{k-2}, y_0, \dots, y_{m-1}) \mid (x_0, \dots, x_{k-1}) \in R_1 \wedge (y_0, \dots, y_{m-1}) \in R_2 \wedge x_{k-1} = y_0\}$.

The relationship between Datalog and relational algebra is explored in great detail in [11]. In the context of this paper, we make use of the observation that each Datalog rule has an equivalent relational algebra expression. In some cases, it will be simpler and more intuitive to express certain concepts using relational algebraic terms, such as when discussing relation-joining.

3 StarLang

StarLang is a strict subset of Monadic Datalog with stratified negation (MDL^-). As *StarLang* is a monadic Datalog, every intensional rule defines a unary predicate, i.e., has a single variable in the head of the rule. One of the main restrictions in *StarLang* is that it does not include the standard explicit equality predicate. Instead, equality is implicit in the reuse of variable names, which is itself restricted.

StarLang rules have a very simple syntactic structure. This is not a semantic limitation. Rather, it is useful for the purposes of presentation, and to make the definition simpler.

Definition 3.1 (Valid starlang rules I). A valid *StarLang* rule r has the form

$$p(\mathbf{X}) \text{ :- } \mathcal{A}, \mathcal{B}, \mathcal{C}.$$

where \mathcal{A} , \mathcal{B} and \mathcal{C} are defined as follows:

\mathcal{A} is a sequence of zero or more predicate citations (either negative or positive) $q_1(\mathbf{X}), \dots, q_k(\mathbf{X})$, where \mathbf{X} is the variable in the head of r .

\mathcal{B} is an optional pair of conjoined citations. The conjunction, if it is present, has either the form $e(\mathbf{X}, \mathbf{Y}), t(\mathbf{Y})$ or the form $e(\mathbf{Y}, \mathbf{X}), t(\mathbf{Y})$, where \mathbf{X} is the variable in the head of r and $\mathbf{Y} \neq \mathbf{X}$. t may be either a positive or a negative citation.

\mathcal{C} is a sequence of zero or more predicate citations (either negative or positive) $d_1(\mathbf{Z}_1), \dots, d_j(\mathbf{Z}_j)$, where $\mathbf{Z}_i \neq \mathbf{X}$ for $1 \leq i \leq j$, \mathbf{X} being the variable in the head of r .

An additional limitation is imposed for negative citations which enforces that all negation is stratified: if q is cited negatively in the body of p , then q itself must not be defined (directly or transitively) in terms of p .

Example 5. Let p , e_1 and e_2 be extensional predicates. Then the following is a valid *StarLang* program:

$$\begin{aligned} r(\mathbf{X}) & \text{ :- } t_1(\mathbf{X}), p(\mathbf{X}), e_1(\mathbf{X}, \mathbf{Z}), q_1(\mathbf{Z}), d(\mathbf{U}). \\ t_1(\mathbf{X}) & \text{ :- } e_1(\mathbf{X}, \mathbf{Y}), t_2(\mathbf{Y}). \\ t_2(\mathbf{Y}) & \text{ :- } e_2(\mathbf{Y}, \mathbf{W}), r(\mathbf{W}). \\ q_1(\mathbf{Z}) & \text{ :- } e_2(\mathbf{V}, \mathbf{Z}), \neg p(\mathbf{V}). \end{aligned}$$

Remark 2. Note that Definition 3.1 does not allow for n -ary predicates for $n > 2$. This is due to the fact that *StarLang* is a Datalog aimed at static program analysis, and so the input databases would in most cases describe graphs (nodes and edges). Binary predicates naturally describe edges between nodes in a graph, while larger arity predicates are not commonly used to describe graph structures. *StarLang* could be extended to support hypergraphs (i.e., graphs with hyper-edges), but does not in its current form.

Remark 3. Definition 3.1 does not allow negated binary predicate citation. Since binary predicates in *StarLang* are assumed to represent directed graph edges, a negated binary predicate would represent a complementary set of edges in

the graph. While well-defined and not problematic in general, in the context of static program analysis, this is unlikely to be useful. While StarLang could easily be extended to support edge-predicate negation, it does not in its current form.

The citations in component \mathcal{C} may seem odd, at first glance. They can, in fact, be very useful, and are necessary for expressing certain queries in StarLang. Due to the limitations on variable-name reuse within a single rule body, without component \mathcal{C} , it would be impossible to express constraints on “disconnected”¹ parts of the input database in a single query. We refer to the citations in component \mathcal{C} as *disconnected citations*.

Definition 3.1 may seem overly restrictive. In fact, it does not encompass all valid StarLang rules. The astute reader might ask about deriving equivalent rules via inlining. And indeed, intuitively, any rule derived by inlining a set of StarLang rules must itself be a valid StarLang rule (as long as proper care is taken around variable naming), due to semantic equivalence. Thus an alternative, more complete definition may be formulated inductively:

Definition 3.2 (Valid StarLang rules II).

1. A monadic rule with an empty body, e.g., $p(X) :- .$, is a valid rule.
2. Let r be a valid rule with head h and body b , and let p be a unary predicate. Then $h :- b, p(X) .$ is a valid rule.
3. Let r be a valid rule with head h and body b , and let p be a unary predicate. Then $h :- b, \neg p(X) .$ is a valid rule, if the rule defining p does not depend on the predicate in h ².
4. Let r be a valid rule with head h and body b , and let e be an extensional binary predicate. Then $h :- b, e(X, Y) .$ and $h :- b, e(Y, X) .$ are valid rules, if X is not a fresh variable, but Y is (i.e., X already appears in r but Y does not).

Remark 4. Items 1, 2 and 3 of Definition 3.2 admit both components \mathcal{A} and \mathcal{C} of Definition 3.1, while items 1 and 4 admit component \mathcal{B} : Citation $t(Y)$ of component \mathcal{B} may always trivially be defined with an empty rule body.

Example 6. Let p_i be intensional unary predicates, let f_j be extensional unary predicates, and let e_k be extensional binary predicates.

The following are valid StarLang rules:

- $r(X) :- .$
- $r(X) :- f_1(X), p_1(X), f_2(X), p_2(X), p_3(X), f_3(Z) .$
- $r(X) :- \neg p_1(X), f_1(X), p_2(X) .$ (assuming p_1 does not rely on r , neither directly nor transitively.)

¹Recall that StarLang was designed to be used as part of a static-analysis engine, and thus the EDB relations are expected to represent forests of program analysis graphs.

²The stratifiability “check” here makes this inductive definition not strictly syntactic in its current form. However, one can easily extend this definition to a pair of program and rule $\langle \mathbb{P}, r \rangle$, and the syntactic check would be: make sure that if $p \in \mathbb{P}$ then r is not in the body of p or any of its dependencies.

- $r(X) :- \neg p_1(X), p_1(X)$. (trivially equivalent to `false`, but still a valid StarLang rule).
- $r(X) :- e_1(X, Y), e_1(X, Z), e_2(Y, W), r(W), e_2(V, Z), p_2(V)$.

The following are invalid StarLang rules:

- $r(X) :- e(Y, X), \neg r(Y)$. violates item (3) — r is negated in the body of r .
- $r(X) :- e(Y, Z)$. violates item (4) — both Y and Z are fresh.
- $r(X) :- e_1(X, Y), e_2(X, Y)$. violates item (4) — neither Y nor X is fresh in e_2 .

3.1 Equivalence of Definitions 3.1 and 3.2

One can use either of the definitions, 3.1 or 3.2, and derive the other. Intuitively, an MDL^\neg rule r is also a valid StarLang rule if it can be transformed into a semantically-equivalent set of rules, each having the “simple” $\mathcal{A}, \mathcal{B}, \mathcal{C}$ structure. What follows is the description of such a procedure. The idea is to construct an undirected graph with labeled edges from a given rule, based on the argument names used in the citations: all unary citations with the same argument will be grouped together in the same node, and binary predicates will define the labeled edges between these nodes (the predicate name being the label). Due to item 4 of Definition 3.2, the graph structure must be a tree. We then derive a separate rule for each node (structure \mathcal{A} of Definition 3.1) and for each edge (structure \mathcal{B} of Definition 3.1):

3.1.1 Procedure for transforming a Definition 3.2 rule into a Definition 3.1 rule.

Let r be the rule being transformed, let x be the variable in the head of r , and let B be the set of predicate citations in the body of r . Let $V = \{U(y) \mid U \subseteq B \wedge \forall u \in B : \text{if } u \text{ is unary with variable } y \text{ then } u \in U\}$. Let $E = \{b(y, z) \mid b \in B \wedge b \text{ is binary with variables } y, z\}$. Let $G = (V, E)$ be the graph induced by nodes V and edges E ³.

If V contains no node for x , then add a node $U(x) = \{\}$ to V .

Algorithm 1 DFS-traverses G starting at $U(x)$, the node for the variable x in the head of r . Nodes of G induce rules $t_u(u)$ for variable u and with fresh predicate name t_u . Labeled edges induce rules $s_v(v)$ for variable v and with fresh predicate name s_v .

Example 7. Consider the following MDL^\neg rule that matches Definition 3.2:

$$r(X) :- e_1(X, Y), e_1(X, Z), e_2(Y, W), r(W), e_2(V, Z), \neg p(V), \\ d(U), e_1(U, R).$$

³Note that E is not a multi-graph, since item (4) of Definition 3.2 prohibits the existence of two binary citations with the same two variables

It can be trivially transformed into a semantically-equivalent set of rules structured as described above as follows:

$$G = \begin{cases} V = & \{U(X) = \{\}, U(W) = \{r(W)\}, U(V) = \{\neg p(V)\}, U(U) = \{d(U)\}\} \\ E = & \{e_1(X, Y), e_1(X, Z), e_1(U, R), e_2(Y, W), e_2(V, Z)\} \end{cases}$$

$$\begin{aligned} t_1(X) & :- s_1(X), s_2(X), t_5(U). \\ s_1(X) & :- e_1(X, Y), s_3(Y). \\ s_2(X) & :- e_1(X, Z), s_4(Z). \\ s_3(Y) & :- e_2(Y, W), t_2(W). \\ s_4(Z) & :- e_2(V, Z), t_3(V). \\ t_2(W) & :- r(W). \\ t_3(V) & :- \neg p(V). \\ t_5(U) & :- d(U), s_5(U). \\ s_5(U) & :- e_1(U, R). \\ r(X) & :- t_1(X). \end{aligned}$$

Algorithm 1 Translate a rule r matching Definition 3.2 into a rule matching Definition 3.1 by DFS-traversing the graph induced by r

```

procedure TRANSLATE(Graph  $G = \{V, E\}$ , Entry point  $U(x)$ )
   $stack := V$  where  $U(x)$  is at the top of the stack
   $result := \emptyset$  ▷ Map of rule heads to rule bodies
  while  $stack$  is not empty do
     $top := \text{pop top of } stack$ 
    if  $top$  has not been visited then
       $y := \text{Var}(top)$ 
       $result[t_y] := \langle \rangle$ 
      if  $s_y$  is a rule in  $result$  then
         $result[s_y].add(t_y(y))$ 
      else if  $y \neq x$  then ▷ Handle disconnected rules
         $result[t_x].add(t_y(y))$ 
      end if
      for  $b(i, j)$  in edges of  $top$  do
        if  $b(i, j)$  is incoming (i.e.,  $i \neq y$ ) and  $i$  is already visited then
          skip ▷ Do not add incoming edge rules twice
        end if
         $z := (j \text{ if } i = y \text{ else } i)$ 
         $result[s_z] := \langle b(y, z) \rangle$ 
         $result[t_y].add(s_z(y))$ 
        push  $U(z)$  onto  $stack$ 
      end for
    end if
  end while
  return  $result$ 
end procedure

```

Remark 5. As the example demonstrates, Procedure 3.1.1 generates redundant rules in many cases. This is in order to keep the decision process of the procedure simpler for ease of explanation, not a requirement of StarLang.

The Procedure 3.1.1 described above will not produce correct results when starting from an MDL^\top rule that is not a valid StarLang rule according to Definition 3.2, as the following example demonstrates.

Example 8. Consider the MDL^\top rule that does not match Definition 3.2: $r(X) :- e_1(X, Y), e_2(X, Y)$. It cannot be transformed into a semantically-equivalent set of rules structured as described above; if we apply Procedure 3.1.1 to this rule, we get the following set of rules:

$$\begin{aligned} t_1(X) &:- s_1(X), s_2(X). \\ s_1(X) &:- e_1(X, Y). \\ s_2(X) &:- e_2(X, Y). \\ r(X) &:- t_1(X). \end{aligned}$$

This set of rules is not equivalent to the original MDL^\top query, since the two instances of Y in s_1 and s_2 are independent, which may lead different instantiations of X in the query result when evaluating the transformed program compared to the original. For example, consider the EDB $\{e_1(1,2), e_1(3,2), e_2(1,2), e_2(3,7)\}$.

Evaluating the original program on this input results in $X = \{1\}$, while evaluating the transformed program results in $X = \{1, 3\}$.

Claim 3.1. *Definitions 3.1 and 3.2 are equivalent.*

Proof sketch.

(3.1) \subseteq (3.2). Let r be a rule as described in Definition 3.1. It is easy to check that r adheres to the structural rules laid out in Definition 3.2.

(3.1) \supseteq (3.2). Let r be a rule as defined in Definition 3.2. Using Procedure 3.1.1, r can be transformed into an equivalent set of rules that adheres to the structure described in Definition 3.1. \square

The proof of Claim 3.1 relies on the correctness of Procedure 3.1.1, i.e., that if the input is input is a valid StarLang rule according to Definition 3.2, then it outputs a semantically-equivalent set of valid StarLang rules according to Definition 3.1. We now prove this holds.

Claim 3.2. *Given a valid StarLang rule according to Definition 3.2, Procedure 3.1.1 outputs a semantically-equivalent set of valid StarLang rules according to Definition 3.1.*

Proof sketch. There are three ways in which Procedure 3.1.1 could violate the claim: (1) making dependant variable instances independent (as in Example 8), (2) changing the way disconnected rules constrain the query, (3) incorrectly handling variable names, and (4) generating a disjunctive rule.

Since the procedure does not change any variables, Item (3) is trivially not an issue. Similarly, Item (4) is not an issue, since Procedure 3.1.1 cannot generate disjunctions (see Algorithm 1).

For Item (1), observe that due to Item 4 of Definition 3.2, graph G of the procedure must be a forest of trees; nodes in the graph represent variables, and by Item 4 of Definition 3.2, exactly one of the variables in a binary predicate (i.e., edge) citation must be fresh. This means that it would be impossible for G to contain two distinct paths between any two nodes.

Finally, for Item (2), observe that disconnected rules constraint the entire rule (i.e., are global constraints), and thus evaluating them at the root of the rule sequence is correct. Actually, since Procedure 3.1.1 does not generate disjunctive rules, adding the disconnected rule constraints at any “level” of the output rule sequence would be valid. \square

3.2 Expressiveness

As we saw in Example 8, StarLang is strictly less expressive than MDL^\neg :

Claim 3.3. *MDL^\neg queries that establish equality between non-head variable operands of n -ary predicates ($n > 1$) are not expressible in StarLang.*

Example 9. The following MDL^\neg query cannot be expressed in StarLang: $Q \equiv p(X) :- q(X, Y), r(X, Y) .$, where q and r are EDB predicates.

Q is an invalid StarLang query, as it violates restriction 4.

A corollary of claim 3.3 is that queries such as “Is there a method m of object O that is called both behind a lock and not behind a lock?”⁴ cannot be expressed in StarLang. Despite this, StarLang is still quite expressive. For example, since it supports stratified negation, there are many StarLang queries that are not expressible in Semi-positive Datalog — namely, any query that includes negations of intensional predicates.

3.2.1 Templates

StarLang includes a syntactic feature called “Templates”. These are somewhat similar to Lisp macros — each template invocation represents a predicate that the compiler auto-generates with the “holes” filled by the arguments of the invocation. Unbounded recursion and nested citation are both allowed in templates.

Example 10. Given two predicates p and q , the following template generates a logic program equivalent to the statement $p \vee (\neg p \wedge q)$:

TEMPLATE $\text{tmpl}(p, q) \equiv t(X) :- p(X). t(X) :- \neg p(X), q(X).$

where t is a fresh predicate name.

Example 11. Given a binary predicate e and a predicate p , the following template generates a logic program that computes the transitive closure of e limited to paths consisting only of nodes for which p holds:

TEMPLATE $\text{tmpl}(p) \equiv t(X) :- p(X), e(X, Y), \text{tmpl}(p).$

where t is a fresh predicate name.

⁴This query could be useful to find potential concurrency bugs.

Templates do not add expressiveness to StarLang; compilation of a templated logic program terminates if and only if the expanded program is finite, in which case it could have been constructed manually. This is the case even for template recursion, as in Example 11. This is due to the memoization in the implementation of the expansion procedure: once a hole has been “filled” with a given predicate, other instantiations of the same template with the same predicate do not need to be re-expanded.

Even though no expressiveness is added by them, templates significantly improve the usability of the language by allowing users to define abstractions that lead to less code-duplication, thus reducing program size and simplifying program-writing.

3.3 Efficiency

StarLang evaluation can be accomplished in time polynomial in the size of the EDB (as is the case for any stratified Datalog with negation [24]), for example by using standard bottom-up, semi-naïve evaluation [18]. However, the structure of StarLang rules can be leveraged to achieve a more efficient evaluation process.

Recall the requirement in item 4 of Definition 3.2, namely that one of the variables in any binary citation must be fresh. This means that constraints expressed using binary predicates are limited to the existence of elements (or “paths”) in the (cumulative) natural joins of relations, without the use of selection or projection operators.

Example 12. Consider the input EDB $I = \{e_1(1,2) ., e_1(4,3) ., e_1(2,4) ., e_2(1,3) ., e_2(4,2) ., e_2(2,4) .\}$.

It is possible in StarLang to encode the constraint that there must be a “path” $e_1;e_2$ between two elements (i.e., there exists a tuple in the join of e_1 and e_2). For example, the tuple $(1,2,4)$ would be an element in such a join in I . In relational algebraic terms: $\exists x : x \in (e_1 \bowtie e_2)$.

However, it is impossible to encode the constraint that there are two elements that are connected by both e_1 and e_2 (i.e., that there exists a tuple in the natural join of e_1 and the inverse relation of e_2 such that the first and last elements are equal). For example, StarLang can not express a query predicated on the existence of the pair $(2,4)$, which are connected by both e_1 and e_2 in I . In relational algebraic terms: $\exists x : x \in \sigma_{1=3}(e_1 \bowtie e_2^{-1})$.

The constraints on joins of binary relations expressible in StarLang effectively amount to the existence (or lack thereof) of transitive “paths”. However, answering transitivity queries correctly does not require actually computing the entire join: by the definition of transitivity, path can be computed cumulatively, “one step at a time”. This results in there being no need to fully compute joins at all when solving StarLang queries; checking transitivity of relations can be thought of as a sequence of selection and projection operations and single, independent joins between unary and binary relations. Thus, while more general Datalog variants require complex runtime join optimizations or manual annotation of join order by users [15] in order to achieve remotely-efficient query-solving [36], StarLang queries are efficient to compute by design.

Example 13. Consider the StarLang query

$$r(X) := e_1(X, Y), e_2(Y, Z), e_3(Z, W), p(W).$$

In relational algebraic terms, it could be expressed as $\sigma_{\mathbf{p}}(\mathbf{e}_1 \bowtie \mathbf{e}_2 \bowtie \mathbf{e}_3)$. Alternatively the same query could be expressed as $\pi_1(\mathbf{e}_1 \bowtie \pi_1(\mathbf{e}_2 \bowtie \pi_1(\mathbf{e}_3 \bowtie \mathbf{p})))$. Note that in the second form, joins only occur between a binary relation and a unary relation (the result of a projection or the citation \mathbf{p}). As a result, the nesting is not necessary — the projection effectively “forgets” the joined relation at each intermediate step, and so the steps can be thought of as a sequence

$$R_1 := \pi_1(\mathbf{e}_3 \bowtie p); R_2 := \pi_1(\mathbf{e}_2 \bowtie R_1); R_3 := \pi_1(\mathbf{e}_1 \bowtie R_2)$$

where R_3 is the result of the query.

Let B be a binary relation and U a unary relation, and let α, β be sorts. If $U : \alpha \times \beta$ and $B : \alpha$, then joining B with U can be done in time $O(\max(|U|, |B|))$, given that B is indexed by its α values (e.g., if B is represented using a hash table). Note that indexing in this case could be done once, as a pre-processing step, since the binary relation being joined is always extensional, i.e., an input relation. Compare that with performing standard computation, with the full joins between two relations B_1, B_2 instead, which would be $O(|B_1| \cdot |B_2|)$ in the worst case (and could not generally be efficiently computed ahead of time, e.g. in the case of joining temporary relations). Thus, StarLang programs can be evaluated more efficiently than other MDL⁷ programs.

3.3.1 Data Complexity of StarLang

Let P be a StarLang program, let I be a finite input database, and let T be the number of elements in the active domain of I . Given that StarLang programs are stratifiable, the computation of P over I using a fixed-point approach will take at most T iterations: the process terminates once no new tuple is added to the output relation, and there are at most T tuples which can be added⁵. In each iteration of the fixed-point computation, at most $|P|$ joins may be evaluated (exactly $|P|$ if the entire program consists of citations of binary EDB relations). Querying a unary relation U can be done in time $|U|$, and there are at most $|P|$ such queries.

Let $m = \max\{|R| \mid R \in I\}$ and the size of the largest EDB relation. Indexing any EDB relation can be done in $O(m)$ time. Let k be the number of EDB relations. Then the data complexity of a StarLang program is bound from above by $O(km + m|P|T)$ — km being the time to index the input, and $m|P|T$ being the upper bound on the time to compute the solution, given said index. Since $|P|$ is considered to be constant when considering data complexity we get the upper bound $O(m(k + T))$. Thus, the data complexity of StarLang has a linear dependency on three dimensions of the EDB: (1) the size of the largest relation; (2) the number of relations; and (3) the number of elements in the active domain.

4 Codesearch

Codesearch is a simplified interface for using StarLang in the Snyk Code system provided to non-expert users for the definition of custom semantic analysis rules

⁵This is because a StarLang query must be unary, and so the resulting output relation may only contain singleton tuples, i.e., at most the entire active domain T of I

supported by the Snyk Code analysis engine. The syntax of Codesearch is much more limited than that of StarLang: Users may only define unnamed, stand-alone rules in the form of what is essentially a single disjunction-of-conjunctions query. This query may only contain invocations of pre-defined templates and citations of pre-defined predicates. These predicates and templates are supplied in the form of the StarLang standard library.

As Codesearch is essentially a StarLang “API”, the abstraction power of templates makes them perfect to act as the primary building blocks of the standard library. The formal grammar of Codesearch can be found in Appendix A, while the description of the definitions in the standard library can be found in Appendix B.

Despite the additional syntactic limitations, Codesearch is still very expressive, due to the expressiveness of StarLang in combination with the extensiveness of the standard-library. In Section 5 we present various case studies, demonstrating some interesting queries expressible in Codesearch.

The Codesearch language and the standard library are designed to be as language-agnostic as possible, aiming to make it easy to express abstract semantic concepts, alleviating the need to think about syntactic structures and concrete code patterns.

Example 14. Recall the following example presented in Section 1, in which the following Codesearch query matches two semantically-equivalent, but syntactically different code snippets.

```
1 CallExpression<"read"> and
2 HasArg0<DataFlowAfter<Arg0In<CallExpression<"close">>>>
```

Snippet #1:

```
1 let f = file(); f.close(); f.read();
```

Snippet #2:

```
1 function func(param) { param.close(); }
2 let f = file(); func(f); f.read();
```

This works by Codesearch enabling users to focus on semantic concepts, such as *dataflow* or *call receivers*, moving structural and syntactic concepts, such as *variable names* or *scoping*, to the backseat.

4.1 Auto-completion

To improve the discoverability of standard library definitions, and to assist users in making correct use of said definitions, the Snyk Code Codesearch editor includes an auto-completion mechanism. This mechanism is context-sensitive, and aims to suggest the most relevant literals, predicates and templates, as demonstrated in Figure 2. This context sensitivity is achieved by performing some limited analysis of the code repository being queried, in order to approximate which parts of the code are likely to match for a given template, predicate or literal.

Despite being context sensitive, the auto-completion suggestion system is quite fast. This is done by performing the partial analysis up front and caching the results. This analysis and cache are associated with a single code-base, and not shared among customers. This is both better for customer privacy and data



Figure 2: Auto-completion for the argument of the `CallExpression` template in the context of the code snippet from Example 14. The most highly-ranked suggestions are all names of functions that are called in the code snippet.

security, as well as better for usability: populating the suggestions based on the analysis of unrelated code repositories would just introduce irrelevant noise to the list of suggestions.

5 Case Studies

We present a few cases studies, in various programming languages, to demonstrate the expressiveness, conciseness and simplicity of Codesearch.

5.1 Finding Taint Vulnerabilities

Taint-flow analysis covers a very large category of security-related vulnerabilities. The following C# code snippet demonstrates a SQL injection vulnerability, where a string coming from an HTTP request is used to insecurely build a query that is executed on a SQL server.

```

1 using Microsoft.AspNetCore.Mvc;
2 using System.Data.SqlClient;
3
4 public class DbHandler {
5     private const String CONNECTION_STRING = "Server=
myServerAddress;Database=myDataBase;User Id=myUsername;
Password=myPassword;";
6
7     public static String DoQuery(string query) {
8         using (SqlConnection connection = new SqlConnection(
CONNECTION_STRING)) {

```

```

9         SqlCommand cmd = new SqlCommand(query,
connection);
10         return (string)cmd.ExecuteScalar();
11     }
12 }
13 }
14
15 public class Product { public string? Name { get; set; } }
16
17 public class ProductFactory {
18     private static String GenQuery(string id) {
19         return "SELECT name FROM products WHERE id = " + id;
20     }
21
22     public static Product GetProduct(string id) {
23         return new Product {
24             Name = DbHandler.DoQuery(GenQuery(id))
25         };
26     }
27 }
28
29 public class VulnerableController : Controller {
30     [HttpGet]
31     [Route("/product/{id}")]
32     public IActionResult ProductInfo(string id) {
33         Product product = ProductFactory.GetProduct(id);
34         return new JsonResult(product);
35     }
36 }

```

The Codesearch standard library includes a dedicated `Taint<>` template that allows users to specify a set of sources, sanitizers and sinks to apply to the taint-flow analysis results coming from the analysis engine.

A Codesearch query to match the vulnerability on line 9 is `Taint<PRED:AnySource, PRED:SqliSanitizer, PRED:SqliSink>`. The convenience predicates `AnySource`, `SqliSanitizer` and `SqliSink` are included in the Codesearch standard library as well, along with a multitude of additional taint-flow focused predicates (see Appendix B).

5.2 General Dataflow as Taint

The Codesearch `Taint<>` template provides access to more data-flow analysis results than simply taint-flow. As demonstrated in the following example, the `Taint<>` template can be used to query non-taint data-flow.

```

1 package org.example.app;
2
3 import java.lang.annotation.*;
4 import java.lang.String;
5
6 @Target(ElementType.FIELD)
7 @interface Sensitive { }
8
9 class User {

```

```

10     @Sensitive
11     private String username;
12
13     User(String username) {
14         this.username = username;
15     }
16
17     public String getUsername() {
18         return username;
19     }
20 }
21
22 public class Main {
23     public static void main() {
24         User user = new User("JohnDoe");
25         System.out.println(user.getUsername());
26     }
27 }

```

A Codesearch query that matches on the leak of the `Sensitive`-annotated field `username` to `stdout` on line 25 is `Taint<HasAnnotation<"Sensitive">,PRED:None,Arg1In<CallExpression<"java.lang.System.out.println">>>`.

5.3 Utilizing Typestate Analysis

Codesearch can be used to detect various typestate analysis-based patterns. The following is an example of use-after-free bugs taken from [30]. This example is a simplification of actual bugs in Internet Explorer (CVE-2010-0249) and the Linux kernel (commit c3aabf0).

```

1 #include <stdlib.h>
2
3 typedef struct{ int ref;} A; A* gpA;
4 typedef struct{ A* _pA;} B; B* gpB;
5
6 void dec_ref(A* a) { if (--(a->ref) == 0) free(a); }
7
8 void clone(B* b1, B* b2){ b1->_pA = b2->_pA; }6
9
10 void filename_lookup(A* lpA){ dec_ref(lpA); }
11
12 void demo_code(){
13     A* pA = (A*)malloc(sizeof(A));
14     pA->ref = 1;
15     gpA = pA;
16     B* pB = (B*)malloc(sizeof(B));
17     pB->_pA = pA;
18     gpB = (B*)malloc(sizeof(B));
19     clone(gpB, pB);
20     pB->_pA = NULL;
21     free(pB);
22     filename_lookup(pA);
23 }
24

```

```

25 void main(){
26     demo_code();
27     dec_ref(gpA);
28     printf(gpB->_pA->ref);
29     free(gpB);
30 }

```

The two bugs are triggered on lines 27 and 28 (see [30] for a detailed discussion of the bugs). A Codesearch query to match them is `DataFlowAfter<Arg1In<CallExpression<free>>>`.

5.4 Utilizing Points-to Analysis and Negation

The next example is constructed based on a commit⁷ from the Apache Lucene project, that fixes a `FileInputStream` resource leak. We demonstrate the use of Codesearch to detect points-to analysis patterns.

```

1 package org.apache.lucene.ant;
2
3 import org.w3c.dom.Element;
4 import org.w3c.tidy.Tidy;
5
6 import java.io.FileInputStream;
7 import java.io.IOException;
8
9 public class HtmlDocument {
10     private Element rawDoc;
11
12     public HtmlDocumentLeaky(File file) throws IOException {
13         Tidy tidy = new Tidy();
14         tidy.setQuiet(true);
15         tidy.setShowWarnings(false);
16         org.w3c.dom.Document root =
17             tidy.parseDOM(new FileInputStream(file), null);
18         rawDoc = root.getDocumentElement();
19     }
20
21     public HtmlDocumentFixed(File file) throws IOException {
22         Tidy tidy = new Tidy();
23         tidy.setQuiet(true);
24         tidy.setShowWarnings(false);
25         org.w3c.dom.Document root = null;
26         InputStream is = new FileInputStream(file);
27         try {
28             root = tidy.parseDOM(is, null);
29         } finally {
30             is.close();
31         }
32         rawDoc = root.getDocumentElement();

```

⁶The definition of `clone()` on line 8 is slightly different than that in [30]. This is due to a gap in the tpestate analysis used performed by the Snyk Code static analysis engine. Given the necessary improvement in the engine, the same Codesearch query would find the bugs with the original definition of `clone()`.

⁷<https://github.com/apache/lucene-solr/commit/85f5a9c>

```
33     }
34 }
```

A Codesearch query that matches on the resource initialization on line 17 but not on 26 is `CallExpression<"java.io.FileInputStream">` and not `ForSomeObject<Arg0In<"close">>`. This is also an example of the usefulness of negation in StarLang, which is available in Codesearch.

5.5 Linting using Syntactic Structures

The previous examples focus on querying semantic structures based on the results of static analysis. However, as the Codesearch standard library contains templates targeting common syntactic structures (in addition to the many semantically-focused templates), it can be used for simpler tasks as well, such as linting, as demonstrated by the following example.

Default arguments of a function in Python are initialized only once, when the function definition is evaluated. This means that using mutable objects or non-pure expressions as default argument values is probably incorrect. The snippet below demonstrates this with a non-pure function call.

```
1 from datetime import datetime
2 import secrets
3
4 def gen_filename():
5     return f"{datetime.utcnow().timestamp()}_{secrets.
6         token_hex(16)}.txt"
7
8 """
9 A possible correct implementation is
10 def dump_data(data, filename=None):
11     if filename is None:
12         filename = gen_filename()
13     ...
14 """
15 def dump_data(data, filename=gen_filename()):
16     with open(filename, "w") as f:
17         f.write(data)
18 dump_data("foo")
19 dump_data("bar")
```

The function `gen_filename` (defined on line 4) generates a different string each time it is called. However, `gen_filename()` is the default value of the argument `filename` (line 14) so the function is only called once. The two calls on lines 18 and 19 will write to the same file, with the data from the second call overwriting the data from the first, which is unlikely to be the intended behavior.

A Codesearch query to match on such likely-incorrect default argument definitions is `AnyParamIn<DataFlowAfter<CallExpression<*>>>`.

6 Related Work

In this section, we review works closely related to ours.

Restricting Datalog for Efficient Computation Similar to StarLang, other restricted subsets of Datalog have been found to be useful for efficiently solving a specialized set of problems. Linear Datalog, a restricted form of Datalog that is computable in NL-time, was shown to be equivalent to a class of constraint-satisfaction problems [23]. Symmetric Datalog, a further-restricted form of Linear Datalog that is computable in Logspace, was also utilized in the context of efficiently-computable constraint-satisfaction problems [25].

Customizing static analysis Lint static analysis tools have recently become important components of popular programming languages. While many of these tools start as discovering syntactic stylistic code issues, many expanded into semantic properties and discovering performance, safety and security bugs. For example ESLint [4], PyLint [6], FindBugs [5], CppCheck [3] and StaticCheck [10] are tools for JavaScript, Python, Java, C++ and Go respectively, that currently include dataflow and other analysis as part of their rules. These tools are typically built on top of a common library of analysis, but they still require the rule author to write the code that performs the actual check, only benefiting from reusing the code of the underlying analysis. StarLang is sufficiently expressible to encode most rules of these tools and has large portion of their checks already encoded as Snyk Code quality checks.

Domain specific languages (DSLs) for rules have a number of advantages over manually implemented analyzers, including sharing computation between different rules, better validation of the rules, testing tools and others. Many of these DSL tools, similar to our work, operate on top of graphs to enable easier debugging and visualization. IncA [46] and CxQL by Checkmarx [2] perform matching based on graph patterns [40]. PidginQL [33] is a query language matching on graphs focusing on enforcing security properties, claiming to perform matching in real-world programs within 15 seconds. Other examples of such tools are SemGrep [7] or CodeQL [17]. According to the user study in [37], none of these systems is user-friendly for describing complex security properties such as taint analysis, whereas they present *fluent*TQL [37] that is focused strictly on such taint rules. While all of these tools provide the capability to perform various types of matching on code, none offers time complexity guarantees.

7 Summary

We presented StarLang, restricted subset of MDL^\neg , which includes only logic programs that can be computed very efficiently, in terms of their data complexity. Based on its efficiency and simplicity characteristics, we constructed an easy-to-use wrapper around StarLang named Codesearch, which allows users to interactively construct static analysis queries, significantly simplifying the job of security professionals and static analysis authors. Finally, we demonstrated the usefulness of Codesearch for its purpose, despite the intentionally-limited expressiveness of both StarLang and Codesearch.

References

- [1] Astrée static analyzer. <https://www.absint.com/astree/>, 2023. [Online;

- accessed 2023-11-07].
- [2] Checkmarx. <https://checkmarx.com/>, 2023. [Online; accessed 2023-11-03].
 - [3] CppCheck. <https://cppcheck.sourceforge.io/>, 2023. [Online; accessed 2023-12-15].
 - [4] ESLint. <https://eslint.org/>, 2023. [Online; accessed 2023-12-15].
 - [5] FindBugs. <https://findbugs.sourceforge.net/>, 2023. [Online; accessed 2023-12-15].
 - [6] PyLint. <https://pypi.org/project/pylint/>, 2023. [Online; accessed 2023-12-15].
 - [7] SemGrep. <https://semgrep.dev/>, 2023. [Online; accessed 2023-11-03].
 - [8] Snyk Code. <https://snyk.io/product/snyk-code/>, 2023. [Online; accessed 2023-11-03].
 - [9] SonarSource. <https://www.sonarsource.com/>, 2023. [Online; accessed 2023-11-03].
 - [10] StaticCheck. <https://staticcheck.dev/>, 2023. [Online; accessed 2023-12-15].
 - [11] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
 - [12] N. Allen, B. Scholz, and P. Krishnan. Staged points-to analysis for large code bases. In B. Franke, editor, *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9031 of *Lecture Notes in Computer Science*, pages 131–150. Springer, 2015.
 - [13] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. Datalog-based program analysis with BES and RWL. In O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers, editors, *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2010.
 - [14] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, 1994. DIKU Research Report 94/19.
 - [15] S. Arch, X. Hu, D. Zhao, P. Subotic, and B. Scholz. Building a join optimizer for soufflé. In A. Villanueva, editor, *Logic-Based Program Synthesis and Transformation - 32nd International Symposium, LOPSTR 2022, Tbilisi, Georgia, September 21-23, 2022, Proceedings*, volume 13474 of *Lecture Notes in Computer Science*, pages 83–102. Springer, 2022.

- [16] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In M. F. P. O’Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM, 2014.
- [17] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer. QL: object-oriented queries on relational data. In S. Krishnamurthi and B. S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPICs*, pages 2:1–2:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.
- [18] F. Bancilhon. Naive evaluation of recursively defined relations. In M. L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies, Book resulting from the Islamorada Workshop 1985 (Islamorada, FL, USA)*, Topics in Information Systems, pages 165–178. Springer, 1985.
- [19] A. K. Chandra and D. Harel. Structure and complexity of relational queries. *J. Comput. Syst. Sci.*, 25(1):99–128, 1982.
- [20] A. K. Chandra and D. Harel. Horn clauses queries and generalizations. *J. Log. Program.*, 2(1):1–15, 1985.
- [21] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
- [22] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fix-points. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.
- [23] V. Dalmau. Linear datalog and bounded path duality of relational structures. *Log. Methods Comput. Sci.*, 1(1), 2005.
- [24] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.
- [25] L. Egri, B. Larose, and P. Tesson. Symmetric datalog and constraint satisfaction problems in logspace. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 193–202. IEEE Computer Society, 2007.
- [26] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective tpestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2):9:1–9:34, 2008.
- [27] A. V. Gelder, K. A. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In C. Edmondson-Yurkanan

- and M. Yannakakis, editors, *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 21-23, 1988, Austin, Texas, USA*, pages 221–230. ACM, 1988.
- [28] N. Grech and Y. Smaragdakis. P/taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):102:1–102:28, 2017.
- [29] S. Greco, D. Saccà, and C. Zaniolo. DATALOG queries with stratified negation and choice: from P to dP. In G. Gottlob and M. Y. Vardi, editors, *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*, volume 893 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1995.
- [30] L. He, H. Hu, P. Su, Y. Cai, and Z. Liang. Freewill: Automatically diagnosing use-after-free bugs via reference miscounting detection on binaries. In K. R. B. Butler and K. Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 2497–2512. USENIX Association, 2022.
- [31] A. Horn. On sentences which are true of direct unions of algebras. *J. Symb. Log.*, 16(1):14–21, 1951.
- [32] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In J. Palsberg and Z. Su, editors, *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009.
- [33] A. Johnson, L. Wayne, S. Moore, and S. Chong. Exploring and enforcing security guarantees via program dependence graphs. In D. Grove and S. M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 291–302. ACM, 2015.
- [34] H. Jordan, B. Scholz, and P. Subotic. Soufflé: On synthesis of program analyzers. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 422–430. Springer, 2016.
- [35] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In H. Boehm and C. Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 423–434. ACM, 2013.
- [36] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *VLDB J.*, 27(5):643–668, 2018.
- [37] G. Piskachev, J. Späth, I. Budde, and E. Bodden. Fluently specifying taint-flow queries with fluent tq. *Empirical Software Engineering*, 27(5):104, 2022.

- [38] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In R. K. Cytron and P. Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61. ACM Press, 1995.
- [39] T. W. Reps, S. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In R. Alur and D. A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 15–30. Springer, 2004.
- [40] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
- [41] S. Sagiv, T. W. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, volume 915 of *Lecture Notes in Computer Science*, pages 651–665. Springer, 1995.
- [42] B. Scholz, H. Jordan, P. Subotic, and T. Westmann. On fast large-scale program analysis in datalog. In A. Zaks and M. V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 196–206. ACM, 2016.
- [43] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song. Learning loop invariants for program verification. In S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 7762–7773, 2018.
- [44] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, 2015.
- [45] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: context-sensitivity, across the board. In M. F. P. O’Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 485–495. ACM, 2014.
- [46] T. Szabó, S. Erdweg, and M. Voelter. Inca: a dsl for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 320–331, 2016.
- [47] T. Tan, Y. Li, X. Ma, C. Xu, and Y. Smaragdakis. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021.

- [48] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In M. Hind and A. Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 87–97. ACM, 2009.

A Formal Codesearch Grammar Description

$\langle Query \rangle ::= \langle Citation \rangle (\langle LogicalConnective \rangle \langle Citation \rangle)^*$

$\langle Citation \rangle ::= \langle Negation \rangle (\langle Predicate \rangle | \langle Template \rangle | \langle Literal \rangle)$

$\langle Predicate \rangle ::= \text{'PRED:'} \langle ValidName \rangle$

$\langle Template \rangle ::= \langle ValidName \rangle \langle \text{'<' } \langle Query \rangle \text{'>' } \langle Query \rangle^* \text{'>'}$

$\langle ValidName \rangle ::=$ Standard rules for naming variables

$\langle LogicalConnective \rangle ::= \text{'and' } | \text{'or'}$

$\langle Negation \rangle ::= \text{'not' } | \varepsilon$

$\langle Literal \rangle ::=$ Any UTF-8 character sequence | $\text{'~"'} \text{ RE2 regex ' "}$

In the interest of keeping the grammar description clean, optional grouping of $\langle Citation \rangle$ s using parenthesis has been omitted.

Operator precedence is standard, i.e., **not** has a higher precedence than **and**, which has a higher precedence than **or**.

B Codesearch Template and Predicate Library

Predicate Any: A “catchall” rule. Matches on anything.

Predicate AnySink: Matches on a range of potential data sinks, including server responses, file systems, database writes, external APIs, logging mechanisms, and other forms of data export or display.

Predicate AnySource: Matches on various types of potentially user controlled data sources, both servers (e.g., HTTP parameters/header/body, URLs, cookies, etc.) or indirect ones such as database fields, local files, I/O or environment variables.

Predicate ApexPageReferenceSource: Matches on potential XSS sources.

Predicate CleartextCookieStorageSanitizer: Matches on cleartext cookie storage sanitizers.

Predicate CleartextCookieStorageSink: Matches on cleartext cookie storage sinks.

Predicate CleartextTransmissionSanitizer: Matches on cleartext transmission sanitizers.

Predicate CleartextTransmissionSink: Matches on cleartext transmission sinks.

Predicate ClientXssSanitizer: Matches on client XSS (e.g., DOMXSS) sanitizers.

Predicate ClientXssSink: Matches on client XSS (e.g., DOMXSS) sinks.

Predicate CodeInjectionSanitizer: Matches on code injection sanitizers.

Predicate CodeInjectionSink: Matches on code injection sinks.

Predicate CommandInjectionSanitizer: Matches on command injection sanitizers.

Predicate CommandInjectionSink: Matches on command injection sinks.

Predicate DeserializationSanitizer: Matches on deserialization sanitizers.

Predicate DeserializationSink: Matches on deserialization sinks.

Predicate EmailContentInjectionSanitizer: Matches on email content injection sanitizers.

Predicate EmailContentInjectionSink: Matches on email content injection sinks.

Predicate ErrorMessageOutput: Matches on error message outputs (e.g., stack-traces).

Predicate ErrorMessageOutputSanitizer: Matches on error message output sanitizers.

Predicate ErrorMessageOutputSink: Matches on error message output sinks.

Predicate FileInclusionSanitizer: Matches on file inclusion sanitizers.

Predicate FileInclusionSink: Matches on file inclusion sinks.

Predicate InformationDisclosureSanitizer: Matches on information disclosure sanitizers.

Predicate InformationDisclosureSink: Matches on information disclosure sinks.

Predicate JndiInjectionSanitizer: Matches on JNDI injection sanitizers.

Predicate JndiInjectionSink: Matches on JNDI injection sinks.

Predicate LdapInjectionSanitizer: Matches on LDAP injection sanitizers.

Predicate LdapInjectionSink: Matches on LDAP injection sinks.

Predicate LogsForgingSanitizer: Matches on log-forging sanitizers.

Predicate LogsForgingSink: Matches on log-forging sinks.

Predicate MemoryCorruptionSanitizer: Matches on prototype memory corruption sanitizers.

Predicate NoSqliSanitizer: Matches on NoSQL sanitizers.

Predicate NoSqliSink: Matches on NoSQL sinks.

Predicate None: An “anti-catchall” rule. Matches on nothing.

Predicate OpenRedirectSanitizer: Matches on open-redirect sanitizers.

Predicate OpenRedirectSink: Matches on open-redirect sinks.

Predicate PointerOperationSink: Matches on prototype memory operation sinks.

Predicate PotentialXssSink: Matches on potential XSS sinks.

Predicate PrototypePollutionAssignmentSanitizer: Matches on prototype pollution assignment sanitizers.

Predicate PrototypePollutionAssignmentSink: Matches on prototype pollution assignment sinks.

Predicate PtSanitizer: Matches on path-traversal sanitizers.

Predicate PtSink: Matches on path-traversal sinks.

Predicate RedosSanitizer: Matches on regular-expression denial-of-service sanitizers.

Predicate RedosSink: Matches on regular-expression denial-of-service sinks.

Predicate ReflectionSanitizer: Matches on reflection sanitizers.

Predicate ReflectionSink: Matches on reflection sinks.

Predicate SoqliSanitizer: Matches on soqli sanitizers.

Predicate SoqliSink: Matches on soqli sinks.

Predicate SosliSanitizer: Matches on sosli sanitizers.

Predicate SosliSink: Matches on sosli sinks.

Predicate SourceArchive: Matches on reading values that are coming from zip, tar or other archives.

Predicate SourceCLI: Matches on reading command line arguments.

Predicate SourceClientFramework: Matches on reading values that are coming from a client-side framework such as Android, SwiftUI, UIKit, the DOM of an HTML page.

Predicate SourceContainsSensitiveData: Matches on reading sensitive data.

Predicate SourceCookie: Matches on reading values of cookies in an http server. These values are of security interest, because they can be fully controlled by malicious users.

Predicate SourceDatabase: Matches on reading values that are coming from a database.

Predicate SourceEnvironmentVariable: Matches on reading environment variables of a process.

Predicate SourceFile: Matches on reading values that are coming from files.

- Predicate SourceHttpBody:** Matches on reading http request body in an http server. These values are of security interest, because they may be fully controlled by malicious actors.
- Predicate SourceHttpFileUpload:** Matches on the name and content of file uploaded to an http server. These values are of security interest, because they may be fully controlled by malicious actors.
- Predicate SourceHttpHeader:** Matches on reading values of http headers in a server. These values are of security interest, because they may be fully controlled by malicious actors.
- Predicate SourceHttpParam:** Matches on reading values of http parameters in an http server. These values are of security interest, because they may be fully controlled by malicious actors.
- Predicate SourceLocalEnv:** Matches on reading values from the local environment of the running process. This includes command line arguments, standard input or environment variables.
- Predicate SourceNetworkRequest:** Matches on reading values that are coming from a remote resource through network requests.
- Predicate SourceNonServer:** Matches on reading values that may be controlled by an adversary, but not directly by sending requests to a server. E.g. if an application fetches a value from a URL, an adversary in control of that URL may use it to control its content.
- Predicate SourceRequestUrl:** Matches on reading request URLs in a server. The URLs are of security interest, because they may be fully controlled by malicious actors.
- Predicate SourceResourceAccess:** Matches on reading values that may be controlled by an adversary if they gain access to a resource. The resources this matches are remote URLs, files, database fields or other framework-specific cases such as Android intents.
- Predicate SourceRpcApiParam:** Matches on parameters of RPCs implemented in an RPC server. These values are of security interest, because they may be fully controlled by malicious actors.
- Predicate SourceServer:** Matches on reading values that an attacker can send to a server. Examples are HTTP parameters/header/body, URLs or cookies. Since these may be directly controllable by attacker, these sources are of significant security interest.
- Predicate SourceStdin:** Matches on reading input from the standard input of a process.
- Predicate SourceUnrestrictedArchiveFilePath:** Matches on zipslip sources.
- Predicate SourceWebForm:** Matches on reading values of web forms in a web server. These values are of security interest, because they may be fully controlled by malicious actors.

Predicate SqliSanitizer: Matches on SQL injection sanitizers.

Predicate SqliSink: Matches on SQL injection sinks.

Predicate SsrfSanitizer: Matches on SSRF sanitizers.

Predicate SsrfSink: Matches on SSRF sinks.

Predicate SstiSanitizer: Matches on SSTI sanitizers.

Predicate SstiSink: Matches on SSTI sinks.

Predicate UnsafeSoqliConcatSource: Matches on unsafe sosli/soqli concatenations.

Predicate UnsafeSosliConcatSource: Matches on unsafe sosli/soqli concatenations.

Predicate XPathInjectionSanitizer: Matches on XPath injection sanitizers.

Predicate XPathInjectionSink: Matches on XPath injection sinks.

Predicate XamlInjectionSanitizer: Matches on XAML injection sanitizers.

Predicate XamlInjectionSink: Matches on XAML injection sinks.

Predicate XmlInjectionSanitizer: Matches on XML injection sanitizers.

Predicate XmlInjectionSink: Matches on XML injection sinks.

Predicate XssSanitizer: Matches on XSS sanitizers.

Predicate XssSink: Matches on XSS sinks.

Predicate XxeSanitizer: Matches on XXE sanitizers.

Predicate XxeSink: Matches on XXE sinks.

Predicate ZipSlipSanitizer: Matches on zipslip sanitizers.

Predicate ZipSlipSink: Matches on zipslip sinks.

Template And: A binary conjunction. Matches only if both arguments match.
Has 2 arguments: conjunct, conjunct.

Template AnyParamIn: Matches on all parameters of the provided method or function declaration/signature.
Has 1 argument: Function.

Template Arg0In: Matches on the 0th index argument (i.e. the receiver object for method calls) for the provided method or function.
Has 1 argument: Function.

Template Arg1In: Matches on the 1st index argument for the provided method or function.
Has 1 argument: Function.

- Template Arg2In:** Matches on the 2nd index argument for the provided method or function.
Has 1 argument: Function.
- Template Arg3In:** Matches on the 3rd index argument for the provided method or function.
Has 1 argument: Function.
- Template Arg4In:** Matches on the 4th index argument for the provided method or function.
Has 1 argument: Function.
- Template Arg5In:** Matches on the 5th index argument for the provided method or function.
Has 1 argument: Function.
- Template Arg6In:** Matches on the 6th index argument for the provided method or function.
Has 1 argument: Function.
- Template Arg7In:** Matches on the 7th index argument for the provided method or function.
Has 1 argument: Function.
- Template BooleanLiteral:** Matches on boolean type literals.
Has 1 argument: Value.
- Template CallExpression:** Matches when a given name is called.
Has 1 argument: Function, method or constructor to call.
- Template DataFlowAfter:** Matches on entities that happen after in the dataflow of its parameter.
Has 1 argument: The previous action executed.
- Template DataFlowsFrom:** Matches on places which a taint data can flow from.
Has 1 argument: Source.
- Template DataFlowsInto:** Matches on places which a taint data can flow into.
Has 1 argument: Sink.
- Template ExplicitSelfParamIn:** Matches on the explicit receiver parameter (e.g., `self` in Python and Rust) for the provided method or function declaration.
Has 1 argument: Function.
- Template ForSameObject:** Matches on entities that happen on the same object as its parameter.
Has 1 argument: The action that happens on the object.
- Template HasAnnotation:** Matches on entities annotated by a given annotation.
Has 1 argument: The annotation with which the entity is annotated.

- Template HasAnyArg:** Matches on entities that take any argument with the provided value.
Has 1 argument: Value.
- Template HasArg0:** Matches on entities that take an argument in the 0th index (i.e. receiver object for method calls) with the provided value.
Has 1 argument: Value.
- Template HasArg1:** Matches on entities that take an argument in the 1st index with the provided value.
Has 1 argument: Value.
- Template HasArg2:** Matches on entities that take an argument in the 2nd index with the provided value.
Has 1 argument: Value.
- Template HasArg3:** Matches on entities that take an argument in the 3rd index with the provided value.
Has 1 argument: Value.
- Template HasArg4:** Matches on entities that take an argument in the 4th index with the provided value.
Has 1 argument: Value.
- Template HasArg5:** Matches on entities that take an argument in the 5th index with the provided value.
Has 1 argument: Value.
- Template HasArg6:** Matches on entities that take an argument in the 6th index with the provided value.
Has 1 argument: Value.
- Template HasArg7:** Matches on entities that take an argument in the 7th index with the provided value.
Has 1 argument: Value.
- Template HasNamedArg:** Matches on entities that take a named argument with the provided value.
Has 2 arguments: The name of the argument., The value the named argument should have.
- Template Identifier:** Matches on an identifier.
Has 1 argument: The entity that should be an identifier.
- Template InPath:** Matches on entities in the source file with the provided path.
Has 1 argument: The path of the file in which to match entities.
- Template Literal:** Matches on string/boolean or number type literals.
Has 1 argument: Value.
- Template NamedArgIn:** Matches on the named argument for the provided method or function.
Has 2 arguments: The name of the argument., The provided method or function.

- Template Not:** A negation. Matches only if the argument does not match.
Has 1 argument: property.
- Template NumberLiteral:** Matches on numeric type literals.
Has 1 argument: Value.
- Template Or:** A binary disjunction. Matches if either (or both) arguments match.
Has 2 arguments: disjunct, disjunct.
- Template Param1In:** Matches on the 1st parameter for the provided method or function declaration.
Has 1 argument: Function.
- Template Param2In:** Matches on the 2nd parameter for the provided method or function declaration.
Has 1 argument: Function.
- Template Param3In:** Matches on the 3rd parameter for the provided method or function declaration.
Has 1 argument: Function.
- Template Param4In:** Matches on the 4th parameter for the provided method or function declaration.
Has 1 argument: Function.
- Template Param5In:** Matches on the 5th parameter for the provided method or function declaration.
Has 1 argument: Function.
- Template Param6In:** Matches on the 6th parameter for the provided method or function declaration.
Has 1 argument: Function.
- Template Param7In:** Matches on the 7th parameter for the provided method or function declaration.
Has 1 argument: Function.
- Template ReturnedBy:** Matches on the returned entity.
Has 1 argument: The entity that returns.
- Template Returns:** Matches on the entity (e.g. a function or a method) that returns the value provided as argument.
Has 1 argument: What is returned.
- Template StringLiteral:** Matches on string type literals.
Has 1 argument: Value.
- Template Taint:** Identify data propagation flows that start at the specified source(s) and reach the designated destination sinks (like vulnerable methods) without going through the specified sanitizer(s).
Has 3 arguments: Source, Sanitizer, Sink.