CASTL: A Composable Source Code Query Language for Security and Vulnerability Analysis

Blake Johnson[a] and Rahul Simha[b]
The George Washington University, Washington, DC 20052, U.S.A.

Keywords: Source Code, Security, Static Analysis, Vulnerability, Query, Language, Composable, Parse, Syntax, Search.

Abstract: This paper describes CASTL (Composable Auditing and Security Tree-optimized Language), a new source code query language focused on security analysis. The widespread implementation of static analysis for vulnerability identification suggests the need for capable, approachable code query languages for security analysts. Languages customized for the unique properties of code can be more expressive and performant than generic solutions.
CASTL features a familiar SQL-style syntax, with inputs and outputs consisting of sets of abstract syntax trees (ASTs). This abstraction enables the advantages of (1) composability (the output of one query can become the input to another), (2) direct querying of the code's structure and metadata; (3) tree-specific language optimizations for performance; and (4) applicability to any AST-based language. Complex queries can be expressed in a compact, straightforward manner. Common vulnerabilities, including buffer overflows, ingestion, and server side request forgery (SSRF) (Christey and Martin, 2007) translate into simple, readable CASTL queries.
We describe CASTL and its capabilities, compare it to alternatives, finding potential advantages in clarity and compactness, discuss features and optimizations improving effectiveness and efficiency, and finally describe an example implementation applying CASTL to millions of Java source files.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Techniques for the static analysis of source code, for the purposes of security and quality analysis have comprised a fast growing area within computer security (Do, Wright, and Ali, 2020). While general-purpose source code analysers are useful, security analysts generally have more up-to-date domain and application-specific knowledge. Thus, an experienced analyst would benefit from a tool that enables easy construction of source code queries. Many current source code query languages contain limitations that reduce their suitability for this task. Urma and Mycroft (2012) surveyed query languages for Java and found shortcomings in most of the languages studied – some are character based and cannot query parse trees, some do not support all structures of the desired target language, some lack the ability to query bindings and expression types, some are proprietary, and others require overly complex or verbose queries. In the subfield of program security analysis, the limitations of the query language can directly affect the ease, or even the possibility, of detecting particular vulnerabilities.

In addition to practical requirements for performance, power and ease of use, a language that allows the full richness of source code to be analysed must incorporate two key characteristics of code – its syntactical organization, typically a tree structure based on a context-free grammar, and its metadata and post-compilation bindings. For example, when examining a reference to a variable, it is possible in many languages to know that variable's type, whether it was declared constant. When encountering a method call, it may be possible to know which method definition is being called. Here is the full set of requirements we identified for a code query language to assist security analysts:

- The language should offer specialized operators and optimizations to allow the user

[a] https://orcid.org/0009-0007-5533-9780
[b] https://orcid.org/0000-0002-0689-9411

Johnson, B. and Simha, R.
CASTL: A Composable Source Code Query Language for Security and Vulnerability Analysis.
DOI: 10.5220/0013176200003899
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 11th International Conference on Information Systems Security and Privacy (ICISSP 2025) - Volume 1*, pages 283-290
ISBN: 978-989-758-735-1; ISSN: 2184-4356
Proceedings Copyright © 2025 by SCITEPRESS – Science and Technology Publications, Lda.

to effectively manage the tree structure of code, since most programming languages are parsed into Abstract Syntax Trees (ASTs).

- The language should make all derivable metadata and bindings available for querying.
- The language should allow fine-grained queries down to individual syntax-token level.
- The language must enable optimized, high performance query execution that can scale to millions or billions of source files.
- The language should be composable – that is, the output of a query should be able to be used as the input to another. This allows complex queries to be built from smaller, simpler ones.
- The language should be extensible, and allow programmers to include imperative code inline for maximum flexibility and compactness.
- Finally, the language should have a familiar, easily-understood SQL-like syntax that will allow simple, compact construction of queries, with clear, meaningful results.

This paper will present the Composable Auditing and Security Tree-optimized Language (CASTL), a new, flexible query language for source code. We will discuss the design of the language, how it meets the above criteria, its optimizations, some results of testing queries on Java source code, and our open source implementation, which may be further enhanced or integrated into other tools, and its performance. We also discuss a few of the surprises we found while implementing our language, such as discovering that the tree-pruning operations we introduced to optimize query performance were the same operations we came to use constantly to tailor our results correctly in our tree-based system.

2 RELATED WORK

The querying of source code is a relatively recent phenomenon, as publicly available codebases have grown in number and size, and a growing emphasis on security has increased interest in both automated and interactive evaluation of code quality. Historically, query writers used simple text searching tools like awk and grep, or relational database query languages like SQL. These tools are not well-suited for source code because they do not capture the rich structure and semantics of code. However, they have nonetheless been used. Google Code Search offered a huge archive of code to search, regular expression queries, and improved special character handling, a step above most other options at the time (Cox, 2012). Natural language query interpretation, correlated against the linguistic information present in source comments and identifier names, can also avoid the limitations of flat searches. Haiduc, Bavota, Marcus, Oliveto, De Lucia, and Menzies (2013) developed a system to automatically detect low-quality queries and rewrite them for more relevant results. Hill (2010) proposed a hybrid system combining natural language and program structure that used the natural language query to prune poor results, allowing the query to return more promising results.

Not all static analysis needs to query deeply within source code at all. Robles and Merelo (2006) described how non-code artifacts in a project may be as rich a source of information as the code itself. CQL (Code Query Language) and its successor, CQLinq, which underpin the popular NDepend static analysis tool, are SQL-like languages for .NET projects that look primarily at high level .NET assembly metadata, representing programs as simple relations (Smacchia, 2008). They have limited capability to search below the class and member level. PQL (Martin, Livshits, and Lam, 2005) is a fascinating query language focused on the identification of object event patterns, analyzing sequences of method calls. Mcmillan, Poshyvanyk, Grechanik, Xie, and Fu (2013) identify chains of function calls as the key search result developers require and their Portfolio system thus models the code as a directed graph of function calls.

Query languages focused on general tree or graph structures also have relevance to code querying. XQuery (Chamberlin, 2003) is a W3C standard language based on XPath for querying XML documents, which particularly influenced our work due to its natural implementation of paths through the XML nodes and its straightforward and powerful FLOWR expression which allows the output to be filtered, tweaked, and output into a customized html format. PMD (PMD Introduction, 2018), a source code analysis tool also based on XPath, with many simple and useful rules predefined, allows efficient querying of the entire AST, but lacks support for bindings and metadata. Gremlin (Rodriguez. 2015) is a graph traversal and query language that allows a mixture of imperative and declarative queries, that we have also drawn inspiration from.

The BOA language and infrastructure (Dyer, Nguyen, Rajan, and Nguyen, 2013) is a comprehensive system with high performance for source code mining. It provides streamlined ASTs for querying. However, the visitor-based query language, derived from Google Sawzall, has a steep

learning curve and can result in complex queries. Some similar issues affect ASTLOG (Crew, 1997), a Prolog variant for analysis of ASTs – the queries can be long and complicated, even for simple questions.

srcQL, by Bartman, Newman, Collard and Maletic (2017), features a SQL-like query syntax down to the token level, and pattern matching within tokens, however it lacks exposure of bindings and metadata. CodeQL, by Moor et al. (2007) is based on Datalog and similar to CASTL in many ways, but its somewhat counterintuitive modelling of object-oriented relationships, and nondeterministic expressions may be an obstacle to wide adoption.

The use of Machine Learning (ML) for vulnerability detection has seen increasing interest in recent years. Marjanov, Pashchenko, Massacci (2022) present a useful survey of developments in this area. Significant challenges remain, particularly related to providing sufficiently large and diverse training data, and the opacity of many ML techniques limiting our ability to fully understand the reasoning behind their outputs or clearly perceive their limits. Still, the potential for AI in this space is high, especially if combined with more traditional techniques. One middle ground that may be productive is the use of Generative AI to produce or enhance queries, in a manner similar to that of Troy, Sturley, Alcaraz-Calero and Wang (2023), in languages like CASTL, realizing the creative potential of ML while preserving traceability and oversight and allowing for human customization.

Table 1: Source Code Query Language Feature Comparison.

	CASTL	srcQL	Boa	CQLinq	PQL	Astlog
Queries Tree Structure	X	X	X[1]			X
Metadata/ Bindings	X			X	X	
Token Level Queries	X	X				X
Composable	X			X		X
Imperative Queries	X		X[2]	X		
Familiar, SQL-like	X	X		X	X	

[1] Boa queries operate on a tree that is slightly abstracted from the native language tree

[2] Boa queries are imperative but must be structured according to Boa's visitor paradigm.

3 LANGUAGE DESCRIPTION

3.1 Overview

In CASTL, the primary input and output data of each query are sets of Abstract Syntax Trees, along with user defined variables. The language combines a familiar SQL-style declarative syntax with imperative, C-like code for filtering, refinement, and post-processing of results. Much of the additional information that source code contains beyond regular text (hierarchies, metadata, and bindings) is available to query. It also contains several keywords and operators for query optimization. CASTL is programming-language agnostic, but our initial implementation operates on Java source code. The broad structure of a query is familiar, and modeled on SQL and the relational algebra – selecting a subset of entities (in his case, ASTs) from a set, often winnowed by a where clause.

```
1:  // query to detect potential improper unicode handling
2:  // based on "IMPROPER_UNICODE" pattern from find-sec-
bugs
3:  // -----------------------------------------------------
4:  select ({Block} b) // Select all blocks in the program
5:  {
6:      // See if this method contains a unicode upper/lowercase
7:      // conversion
8:      select ({MethodInvocation} i) directly in b
9:      where i.expression.typebinding() == "java.lang.String" &&
10:         (i.name == "toUpperCase" || i.name == "toLowerCase")
11:     {
12:         // Now see if the method contains a string equality
13:         // comparison on the converted string
14:         select ({MethodInvocation} i2) in i.parent()
15:         where i.name == "equals" &&
16:                 i2.position() >= i.position() {
17:             print(i.filename() + ":" + i.linenumber() +
18:               " may contains improper unicode handling.";
19:             num_improper_unicode_results++;
20:         }
20:     }
```

Listing 1: An example CASTL query.

3.2 Guided Tour of a CASTL Query

Listing 1 shows an example CASTL query to identify a potential improper Unicode handling vulnerability, based on the IMPROPER_UNICODE pattern from find-sec-bugs plugin. The vulnerability arises when an attacker inputs a crafted

Unicode string that is subjected to an upper-or-lowercase conversion that may allow the evasion of subsequent text comparisons intended to enforce input restrictions. The overall approach is to find all code blocks, then identify string case conversions in each that are followed by comparisons.

We first select all Blocks (or more precisely, the set of subtrees with a Block as the root node) from our source project. Identifiers enclosed in curly braces (like "{Block}" on line 4) represent node types in the AST. The "in" keyword on line 8 directs the set of output trees from the outer query to the input of the inner query. The result set of Block subtrees becomes the input to the inner query for MethodInvocations. The "directly" keyword, on line 7, is a tree pruning operation, discussed in section 3.4, which eliminates invocations within sub-blocks of each result, as they are duplicative for this query.

The where expression on lines 9-10 identifies the criteria the MethodInvocations will be selected on (specifically, toUpper or toLower invocations on Java Strings. For each of these results, beginning at line 14 we select another MethodInvocation (using the parent() function to limit the search to the same scope) that follows the first and executes the equality comparison function on a String.

On line 17 we switch briefly to imperative code to output the result and increment a counter to log the number of potential vulnerabilities found. We also use the CASTL "position()" function, which obtains the character position within the source file of the first character in the text comprising the AST node, to see which statement precedes the other.

```
1: // Listing 2, snippet 1: contains() using an AST
2: // node type -- find catches that throw
3: select ({CatchClause} c) {
4:    if (c.contains({ThrowStatement})) {
...
5: // snippet 2: contains() using a bound result tree
6: select ({TypeDeclaration} t1) {
7:    select ({TypeDeclaration} t2) {
8:       if (t1.contains(t2)) {
...
9: // snippet 3: isparent() property
10: select ({Block} b) {
11:    select ({Statement} s) in s
12:             where b.isparent(s) {
...
```

Listing 2: CASTL query snippets demonstrating the contains() and isparent() properties.

3.3 CASTL Language Features

3.3.1 Tree Node Relationships

Listing 2, snippet 1 is part of a CASTL query designed to answer "how often does a try-catch actually throw an exception?" Line 4 demonstrates the contains() function, which is true if a tree contains the given node type. This can help avoid an additional nested query, in cases where the analyst needs to know only the existence of a particular node type within a tree.

Snippet 2 shows a second form of contains(), where it is passed a sub-tree, rather than a node type, and returns true if the parent tree contains the given subtree. This form is less often useful, as it is usually more efficient to flow output from one query to another using the "in" keyword as in Listing 1, rather than determine the relationship between trees after the fact using contains().

Snippet 3 is a query that returns statements paired with their enclosing block. isparent() returns true when a direct child to the root matches the node type or given subtree.

3.3.2 Tree Traversal

```
1:  // Figure 3: Finding getters
2:  select ({MethodDeclaration} m) where
3:      m.parent().isnodetype({ClassDeclaration})
4:        && !m.{parameters}
5:        && m.{body}.{statements} == 1) {
6:     select outmost ({Statements} s) in m where
7:     s.isnodetype({ReturnStatement}) &&
8:     s.{Expression}.isnodetype({Name})) {
9:        print(m + " is a getter");
10:    }
11: }
```

Listing 3: A CASTL query snippet demonstrating the contains() and isparent() properties.

The query in Listing 3 is designed to find getters, here defined as methods with no parameters and a single statement that directly returns a single variable. This query shows some ways to traverse the returned subtrees. First, we select methods that have no parameters and a single statement. Then a subquery retrieves the statements from within the methods, filtering out those that are not return statements or do not directly return a variable.

The parent() function on line 3 is used to ascend to the parent of the given node, in this case to affirm that this method is within a class (rather than an

interface). To descend the tree, as on lines 4, 5 and 8, the user may specify either the name of the subtree or subtree list that they wish to traverse to (as in {parameters}, {body}, and {statements}), or, as a shortcut in cases where the subnode can be unambiguously determined, by specifying the node type of the desired child node (as in {Expression}). These names are determined by the grammar of the target language. m.{body}.{statements} on line 5 refers to a list of ASTs. CASTL automatically casts the list to an integer (the number of nodes in the list) when it is compared against an integer.

3.3.3 Incorporating External Queries

```
1: // Listing 4: Query Calls - counting loop depths
2: q1 : select outmost ({ForStatement} f) {
3:    nested_for_count ++;
4:    callquery(q1) directly in f;
5: ...
```

Listing 4: A CASTL query snippet demonstrating a recursive query invocation.

Listing 4 contains a snippet from a query designed to find the most deeply nested for loops. For this we must recursively descend through the parse tree through all of the nested loops. Line 4 shows how the "callquery()" function can be used to dynamically incorporate any other query (or the same query again), using the output set from the current query as input. In this case the query passes its results to itself to count the nested loops.

This feature is useful in cases where a query is searching for two or more associated node types, to dispatch execution to the appropriate inner query based on the types actually encountered. This feature can also be used to implement query "functions," centralizing the implementation of a complex filter or traversal needed by multiple queries.

3.3.4 Type and Method Bindings

```
1:  // Listing 5, snippet 1: Method bindings
2:  select ({MethodDeclaration} m) {
3:       select ({MethodInvocation} i)
4:            where  m  ==  i.methodbinding()
```

```
1:  // Listing 5, snippet 2:  Type bindings
2:  select ({TypeDeclaration} t) {
3:       select ({Expression} e) in t
4:            where e.typebinding() == t {
```

Listing 5: CASTL query snippets demonstrating method and type bindings.

The snippets in Listing 5 demonstrate how the method and type bindings implicit within the source code may be queried. The first snippet contains two nested queries which find method declarations paired with method calls that invoke them. The methodbinding() function returns the method declaration that a method invocation is calling.

Snippet 2 has a similar design, and finds type declarations paired with expressions that resolve to the type. The typebinding() function returns the type declaration that an expression returns. Thus, a method invocation may have both a methodbinding and a typebinding, which would be the return type of the method being called. Note that only bindings that may be statically determined at compile time can be queried in this manner.

```
1: // Listing 6: Counting number of in/out calls
2: select ({TypeDeclaration} t) {
3:   num_incoming = 0;
4:   mum_outgoing = 0;
5:   // find methods calls out of this class
6:   select ({MethodInvocation} m) in t
7:      where !t.isparent(m.methodbinding()) {
8:    num_outgoing++;
9:  }
10:  // now find method calls into this class
11:  select ({MethodInvocation} m2)
12:      where !t.contains(m2) &&
13:           t.isparent(m2.methodbinding()) {
14:    num_incoming++;
15:  }
```

Listing 6: A CASTL query for measuring the number of incoming/outgoing method calls per class.

Listing 6 contains an appealingly compact query to count, for each class, the number of method calls into and out of that class' methods. It is a component of a simple, static-analysis version of the "Hubs and Authorities" webmining-based class identification method developed by Zaidman and Demeyer (2008). The query proceeds by selecting all classes, then selects all method invocations that originate within each class but conclude outside of it. Finally, it selects all method invocations that originate outside each class but conclude inside it. These operations are only possible because of the method binding information exposed by the methodbinding() function, and they demonstrate the powerful and concise queries that it enables.

3.4 Tree Optimizations

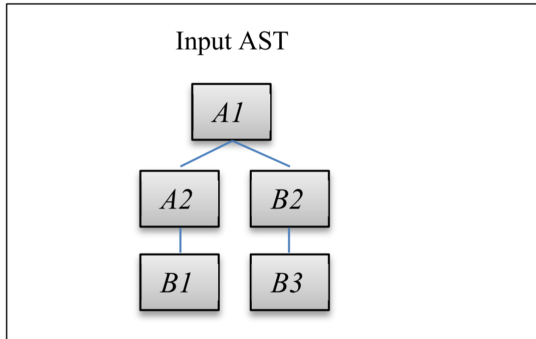

Figure 1: An example AST for demonstrating tree-pruning keywords.

```
Query1 : select ({A} a) {
  select ({B} b) in a {}}
// Query1 returns B1, B2, B3

Query2 : select ({A} a) {
  select outmost ({B} b) in a {}}
// Query2 returns B1, B2

Query3 : select ({A} a) {
  select inmost ({B} b) in a {}}
// Query3 returns B1, B3

Query4 : select ({A} a) {
  select ({B} b) directly in a {}}
// Query4 returns B2, B3

Query5 : select ({A} a) {
  select outmost ({B} b) directly in a {}}
// Query5 returns B2

Query6 : select ({A} a) {
  select inmost ({B} b) directly in a {}}
// Query6 returns B3
```

Listing 7: Queries demonstrating the inmost, outmost, and directly keywords operating on the AST from Figure 1.

Listing 7 shows three of the tree-pruning keywords added to CASTL which may be used to tailor results and optimize performance. The listing demonstrates the effect of these modifiers on query results generated from the input AST pictured in Figure 1.

The "outmost" keyword causes the query to only return subtrees where there are no other nodes of the queried type interposed between the root of the query input and the root of the subtree being returned. The inverse modifier "inmost," is also available, which has the effect of excluding subtrees where the returned subtree contains additional nodes of the queried type. An associated optimization is the "directly in" syntax causes the query to return only subtrees where there are no other nodes of the type of the root node of each input tree interposed.

We initially designed these modifiers to filter extraneous or duplicative subtrees and thus increase performance, but they have also surprised us by being very useful functionally, as well, and often are necessary for strictly correct results. For example, the situation arises commonly (as in our initial example in Figure 2) where the query author wishes to find only the top-level blocks or statements within a method, and exclude those within nested loops or inner classes. The use of "outmost" and "directly" make this condition easy to implement, saving us a significant amount of complex filtering.

4 SAMPLE IMPLEMENTATION

To explore the power and performance of CASTL, we built an initial open-source implementation of the parser and query processor supporting Java-language source code. For the Java language parsing and metadata inference, we used the Eclipse Java Development Tools (JDT) libraries interfaced with our own code for parsing and executing CASTL. The system supports initializing sets of project-specific variables that will flow into the queries, may be used or modified, and are output along with the result subtrees and any other output variables created within the queries. After query execution, the output variables for all projects are gathered up and built into a single spreadsheet to aid in analysis.

To test our implementation, we also wrote several dozen queries to find common vulnerabilities and investigate areas of interest in the use of the Java language. We executed these queries on over 2 million Java source files, with well over 100 million lines of code, taken from the top 3000 Java projects on github spanning two decades of Java usage. The analysis of our query results will be presented in a companion paper. Here we consider the performance characteristics of this large scale execution.

Our measured performance on this dataset was reasonable for a lightly optimized, single-threaded implementation. CASTL performance is highly dependent on the number and content of the queries and projects, as well as the hardware platform, but when executing our reasonably broad sample of about 50 queries of varying sizes on a modest PC with a spinning hard drive, the system worked through about 25 million lines of code per hour. On

a single simple query, representing the maximum performance scenario, the system processed about 100 million lines of code per hour. The relative lack of speedup indicates that a significant portion of the total time in a single-query scenario is taken up by parsing, which is relatively less important in the 50 query scenario (as each project is parsed only once regardless of how many queries are executed on it).

There are many potential improvements we intend to make to this early implementation. Some of the smaller convenience features of the language have yet to be implemented, and there is scope for additional performance optimizations such as multithreading. It should also be possible to add support for C++ code by integrating the Eclipse C/C++ Development Tools (CDT) libraries, which share a common interface with the JDT libraries we currently use, as our CASTL execution code is written to be largely independent of the target language (though existing queries would need to be modified for the different node types in a C++ AST).

5 CONCLUSION

Our results demonstrate the benefits of a query language combining the composability and familiar syntax of traditional relational query languages with an input/output model adapted for ASTs, and operations and optimizations tailored for source code. Many queries that are possible in CASTL are simply impossible in some other languages. In other cases it may be possible to design queries that are more compact, clear, or efficient, when compared against their equivalents in other languages.

```
1: select ({Block} b) {
2:    select outmost ({Statement} s1) directly in b
3:    {
4:       select outmost ({Statement} s2) directly in b
5:          where s1.position() < s2.position() &&
6:             (s1.isnodetype({BreakStatement}) ||
7:             s1.isnodetype({ReturnStatement}) ||
8:             s1.isnodetype({ThrowStatement}) ||
9:          s1.isnodetype({ContinueStatement}))
10:      {
11:         print(s2.filename() + " - " +
12:                  s2.linenumber());
13:      }
14:   }
15:}
```

Listing 8: A CASTL query for locating instances of unreachable code.

For example, Listing 8 shows a query to identify certain instances of unreachable code, based on an equivalent query from the reference documentation of the Boa language. The query selects all blocks, then attempts to find two statements directly within each block where the statement that precedes the other is of a type that aborts execution of the block code (specifically, the break, return, throw, and continue statements). The CASTL query used 15 lines and 418 characters, compared with the equivalent query in the other language, which used 47 lines and 1,210 characters. Users may also find the CASTL example to be clearer and easier to read, due to its SQL-like semantics and because it is structured and ordered the way a human would perform the same search, versus the powerful but unintuitive visitor-type query required by Boa. Our initial implementation of a CASTL system demonstrates that these benefits are realizable without sacrificing high performance.

Further work is needed to refine our implementation and explore additional improvements to the language itself. For example, the “directly” and “outmost” keywords, which have proved immensely useful in our query set, feel like two special cases of a more general purpose operator, which would prune subtrees in which any arbitrary, user-specified AST node type (or set of node types) is interposed between the input and result tree roots. We would like to find a way to implement such a general operator without sacrificing the clarity and compactness of our query code. We could also boost performance with a new pruning keyword replacing the isparent() function.

We would also like to build on our example implementation to support other languages beyond Java. A clear first step would be to add C/C++ support via the Eclipse C/C++ Development Tools library. These additions would inform and improve our efforts to make CASTL a valuable tool for querying any AST-based language.

ACKNOWLEDGEMENTS

The authors would like to thank Sarah Casay and Loc Nguyen, whose senior design projects contributed to this paper.

REFERENCES

Christey, Steve and Martin, Robert A. *Vulnerability Type Distributions in CVS.* (May 2007). Retrieved January

14, 2024 from https://cve.mitre.org/docs/vuln-trends/index.html

L. Nguyen Quang Do, J. R. Wright, and K. Ali, “Why do software developers use static analysis tools? a user-centered study of developer needs and motivations.” In *Proceedings of the Sixteenth Symposium on Usable Privacy and Security*, 2020.

Raoul-Gabriel Urma and Alan Mycroft. 2012. Programming language evolution via source code query languages. In *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools* (PLATEAU '12). ACM, New York, NY, USA, 35-38.

Cox, Russ (2012). *Regular Expression Matching with a Trigram Index*. Retrieved from https://swtch.com/~rsc/regexp/regexp4.html

Smacchia, Patrick A (2008). *Code Query Language 1.8 Specification.* Retrieved from https://www.javadepend.com/CQL.htm

Michael Martin, Benjamin Livshits, and Monica S. Lam. 2005. *Finding application errors and security flaws using PQL: a program query language.* SIGPLAN Not. 40, 10 (October 2005), 365-383.

Don Chamberlin. 2003. XQuery: a query language for XML. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (SIGMOD '03). ACM, New York, NY, USA, 682-682. DOI=http://dx.doi.org/10.1145/872757.872877

PMD Introduction (2018, January 21). PMD Source Code Analyzer Project. Retrieved from https://pmd.github.io/pmd-6.0.1/

Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages* (DBPL 2015). ACM, New York, NY, USA, 1-10. DOI=http://dx.doi.org/10.1145/2815072.2815073

Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering* (ICSE '13). IEEE Press, Piscataway, NJ, USA, 422-431.

Roger F. Crew. 1997. ASTLOG: a language for examining abstract syntax trees. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages* (DSL), 1997 (DSL'97). USENIX Association, Berkeley, CA, USA, 18-18.

Emily Hill. 2010. *Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration*. Ph.D. Dissertation. University of Delaware, Newark, DE, USA.

Gregorio Robles, Jesus M. Gonzalez-Barahona, and Juan Julian Merelo. 2006. *Beyond source code: the importance of other artifacts in software development (a case study)*. J. Syst. Softw. 79, 9 (September 2006), 1233-1248.

Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering* (ICSE '13). IEEE Press, Piscataway, NJ, USA, 842-851.

Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. 2013. *Portfolio: Searching for relevant functions and their usages in millions of lines of code*. ACM Trans. Softw. Eng. Methodol. 22, 4, Article 37 (October 2013), 30 pages.

Andy Zaidman, Serge Demeyer. Automatic Identification of Key Classes in a Software System Using Webmining Techniques. In *Journal of Software Maintenance and Evolution: Research and Practice* 20(6): 387-417, Wiley, November/December 2008.

B. Bartman, C. D. Newman, M. L. Collard and J. I. Maletic, "srcQL: A syntax-aware query language for source code," *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering* (SANER), Klagenfurt, 2017, pp. 467-471.

O. d. Moor et al., "Keynote Address: .QL for Source Code Analysis," Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), Paris, France, 2007, pp. 3-16, doi: 10.1109/SCAM.2007.31.

T. Marjanov, I. Pashchenko and F. Massacci, "Machine Learning for Source Code Vulnerability Detection: What Works and What Isn’t There Yet" in *IEEE Security & Privacy*, vol. 20, no. 05, pp. 60-76, Sept.-Oct. 2022, doi: 10.1109/MSEC.2022.3176058.

C. Troy, S. Sturley, J. M. Alcaraz-Calero and Q. Wang, "Enabling Generative AI to Produce SQL Statements: A Framework for the Auto- Generation of Knowledge Based on EBNF Context-Free Grammars," in *IEEE Access*, vol. 11, pp. 123543-123564, 2023, doi: 10.1109/ACCESS.2023.3329071.