Intrusion detection by automatic extraction of the semantics of computer language grammars

Master of Science Thesis
University of Turku
Department of Computing
M.Sc. Tech - Cyber security
2023
QUETEL Grégor

UNIVERSITY OF TURKU
Department of Computing

QUETEL GRÉGOR: Intrusion detection by automatic extraction of the semantics of computer language grammars

Master of Science Thesis, 64 p., 7 app. p.
M.Sc. Tech - Cyber security
July 2023

Interactions between a user and information systems are based on an inescapable architectural pattern: user data is integrated into requests whose analysis is carried out by an interpreter that drives the system's activity. Attacks targeting this architecture (known as injection attacks) are very frequent and particularly severe. Most often, this detection is based only on the syntax of this data (e.g. the presence of keywords or sub-strings typical of attacks), with limited knowledge of their semantics (i.e. the effects of the query on the information system). The automatic extraction of these semantics is, therefore, a major challenge, as it would significantly improve the performance of Intrusion Detection Systems (IDS).

By leveraging the novel advancement in Natural Language Processing (NLP) it appears feasible to automatically and transparently infer the semantics of user inputs. This Master Thesis provides a framework centred on the instrumentalization of parsers. We focused on parsers for their pivotal role as the first layer of interaction with user inputs and their responsibility for the performed operation on an information system. Our research findings indicate the possibility of constructing an intrusion detection system based on this framework. Moreover, the focus on parser technologies demonstrates the potential for dynamically preventing the processing of malicious input (i.e. creating Intrusion Prevention Systems).

Keywords: Intrusion Detection, Natural Language Processing, Formal Language

Contents

1 Introduction

1.1 Motivation

In recent years, the escalating number of cyber-attacks has become a mounting concern for companies, organizations, and users. The landscape of cyber threats has witnessed exponential growth, both in terms of the sheer volume of attacks and the diversity of techniques employed by malicious actors. Despite the awareness of their existence, certain techniques continue to be commonly discovered even today. One such example is **injection attacks**, which occur when user input is inadequately validated or sanitized by applications allowing an attacker to perform unauthorized actions on a system (bypassing security measures, accessing sensitive data, deleting data, gaining control...).

Intrusion Detection Systems (IDS) are solutions designed to detect and alert against any anomalous behaviour occurring within a system or network. In the industrial sector, applicative intrusion detection systems primarily depend on the analysis of the syntactic structure of inputs to identify potential security breaches or unauthorized activities. Consequently, attackers constantly find new ways to craft their malicious payload. They evade security measures by mimicking the structure of benign inputs or introducing enough syntactic modification for the malware to become undetected. By only focusing on the syntactic structure of user actions, existing detection tools are lacking comprehensive access to the operations being

performed.

This Master's Thesis provides a methodology to automatically infer the semantics of user inputs thanks to the instrumentation of parsers. The long-term objective (not covered here) of this research is to develop an intrusion detection system using these semantics to effectively identify malicious activity.

The motivation behind focusing on the parser level stems from the fact that software source code contains valuable indicators regarding the semantic aspects of a query. Designers of programming languages usually name nonterminals in grammar according to their semantics. For instance, in MySQL, there are nonterminals named `drop_database_stmt` or `alter_table_stmt` which give clues about the performed operation for an input that matches this rule.

No other tool located outside the parser can access this knowledge. Furthermore, the parser is responsible for translating malicious input into operations that can potentially impact the system or manipulate data structures. Attempting to evade the parser becomes futile since no operations will be executed, rendering the malicious input ineffective.

Due to its inherent characteristics, our detection approach will not be capable of identifying unauthorized inputs without any impact on the system. For instance, in the case of a malicious user attempting to craft a SQL injection payload, our solution may not detect it until an unauthorized operation is performed on the system. However, the ability to detect intrusion attempts before a malicious payload is executed represents a valuable property that current IDS possess. Our proposed solution does not aim to replace current IDS but to improve their accuracy by adding correlation with a new source of data.

1.2 Contribution

This thesis presents a process to automatically produce a sequence of semantics for input processed by Bison-generated parsers. It will attempt to answer the following research questions:

- **RQ1**: What would be the semantic tags to consider for a query/command type language useful to an IDS?
- **RQ2**: What information is available in the grammar and high-level parser code to infer these tags?
- **RQ3.1**: How can we automate the grammar instrumentation?
- **RQ3.2**: How can we automate the generation of tags predictions?

1.3 Structure of the thesis

The thesis consists of the detailing of our approach to answering the research questions, Chapter 2 consists of the presentation of the concepts, mechanisms, and software used in this thesis. Chapter 3 details the intrusion detection systems and intrusion prevention systems which also uses parsing information to differentiate benign query from malicious one. In Chapter 4, we provide a description and justification of the comprehensive approach adopted to address the research questions. Additionally, Chapter 5 delves into the detailed account of our experiments. Lastly, in Chapter 6, summarize the thesis and explore options for future work.

2 Background

This chapter aims to provide the reader with a comprehensive overview of the technologies, mechanisms, and concepts that serve as the foundation for this thesis. The primary objective of this chapter is to establish the contextual framework in which the thesis operates.

2.1 Parser generation methods

In this section, we introduce various parser generator tools, conduct a comparative analysis among them, and argue for our decision to employ Bison as the parser generator tool for this thesis.

2.1.1 Parser inner-working

A **parser** is a software component designed to analyze input data, typically in text format, and construct a parse tree out of it. The parser ensures that the syntax is correct and is usually followed by other processing steps. For example, after parsing HTML a browser renders a web page. Nonetheless, some parsers are capable of interpreting inputs on the fly, as exemplified by the basic calculator programs provided by Bison. [1].

A **lexer** (often called a scanner) is responsible for dividing a sequence of characters in an input stream into individual tokens. **Tokens** are considered as a sequence of characters treated as a unit that cannot be further disassembled. For instance,

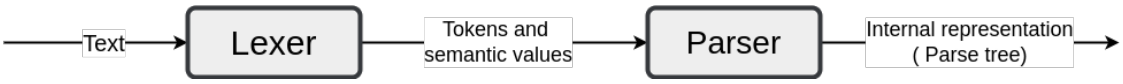


Figure 2.1: Detail of the parsing pipeline

in a calculator scanner, a token can be used to represent an integer `TOK_INTEGER`, symbols: `TOK_MULTIPLY`, `TOK_SUBSTRACT`.

Lexer and parser work in conjunction, the former generating tokens from a stream of characters, feeding them to the latter. The parser takes this sequence of tokens and checks that they form an allowable expression of the parser's language. An illustration of this pipeline is depicted in Figure 2.1.

The tokens returned by a lexer only give clues about their classification: if the lexer returns an `TOK_INTEGER` token, it could correspond to any existing integer. In many cases, the parser needs to assess the values associated with a token: in a networking application, a parser might want to check that port values are positives and inferior to 65,536. Therefore when a token is given to the parser, we can also provide its **value**. Bison defines this value as the **semantic value**.

It is common practice for lexers and parsers to be created using automated generation tools, rather than being manually constructed. To accomplish this, programmers define the lexical and syntactical specifications of a language using grammar files, and the generation tools automatically create code for the lexers and parsers.

2.1.2 Overview of parser generator software

Various parser generator tools are available. Nonetheless, the most employed ones are YACC/LEX and ANTLR. We used different criteria such as popularity, relevance for our use, and ease of use to choose which technology to work with.

YACC: Yacc possesses a bottom-up parsing approach, whereby a parse tree node is produced only after all its child nodes have been constructed. The parser is based

on an LALR(1) (Look-Ahead LR) parsing technique, which is a less complex version of typical LR(1) (Left-to-right, Rightmost derivation in reverse) bottom-up parsers. To determine whether the parser should consume another token from the input or perform a reduction, the algorithm uses a lookahead token. LALR parsers support by design **left-recursive** grammars which are the ones we tend to naturally use to describe a language. YACC follows the UNIX philosophy of one tool to achieve one task, hence we need to use an external lexer to provide tokens to the parser, typically LEX.

ANTLR: ANTLR follows a top-down parsing strategy, whereby the parent nodes of the parse tree are generated before their children. The parser is based on an adaptive LL(*) (Left-to-right, leftmost derivation) parsing algorithm that can have a finite but variable number of lookahead tokens [2]. ANTLR has added support for left-recursive rules in its latest version, which involves converting these rules into equivalent non-recursive ones dynamically. We will now highlight the different metrics and criteria we used to choose the parser generator tool to be utilized in this thesis.

To evaluate the **popularity** of both tools, we used GitHub Code search. Popularity serves as a metric that provides insight about the amount of documentation and support we can have with a given tool. Moreover, our objective is to support all grammar associated with a particular technology, choosing the most popular one involves being able to instrumentalize more software. The popularity of both tools cannot be examined by the number of stars or forks on their GitHub repository since YACC is not open-source. Moreover, Bison (the GNU version of YACC) is not hosted on GitHub either. We therefore looked at how many Bison and ANTLR grammar files are present on GitHub which gives us an estimation of their popularity. More than 182,000 Bison grammar files are found in Github [1] whereas around

[1] `https://github.com/search?q=language%3AYacc+&type=code`

34,000 ANTLR files are available [2] (as of in March 2023).

Regarding the **ease of use** of the two different parser generator tools, empirical studies conducted by Ortin, Quiroga, Rodriguez-Prieto, *et al.* [3] have indicated that ANTLR is more user-friendly and easier to comprehend. Nevertheless, it is important to note that maintaining, modifying, and debugging grammar can be a tedious task with both ANTLR and YACC/LEX and is a common occurrence in all parser generator tools.

Finally, the **relevance** of both technologies has been evaluated in the context of our research objectives. As previously stated, we aim to instrument grammar in such a way that we can identify system interactions whenever a reduction takes place with the final objective of being able to develop an IDS. With this purpose in mind, it is coherent to choose to work on the technology used in the majority of widely used database technologies [4]. YACC is the technology used to build MySQL and NoSQL-based DBMS. However, ANTLR seems more adequate for projects where inputs are dynamic, and is the parser generator used by SQL Developer IDE, NetBeans IDE...[5].

YACC demonstrates to be the most widespread technology, but most importantly is a lightweight tool. This means it can easily be modified whereas ANTLR is quite heavy. Understanding the behaviour of such software to modify it would be time-consuming. Furthermore, YACC appears to be the most relevant to our use case. We, therefore, choose to work with this parser generator tool.

2.1.3 Bison mechanism

Bison [6], being a freely available software implementation of YACC, is the technology that will be employed throughout this thesis. This section details how Bison and Flex work all together to generate a parser to give readers a better understanding

[2] `https://github.com/search?q=language%3AANTLR+&type=code`

of the coming choices and implementation details.

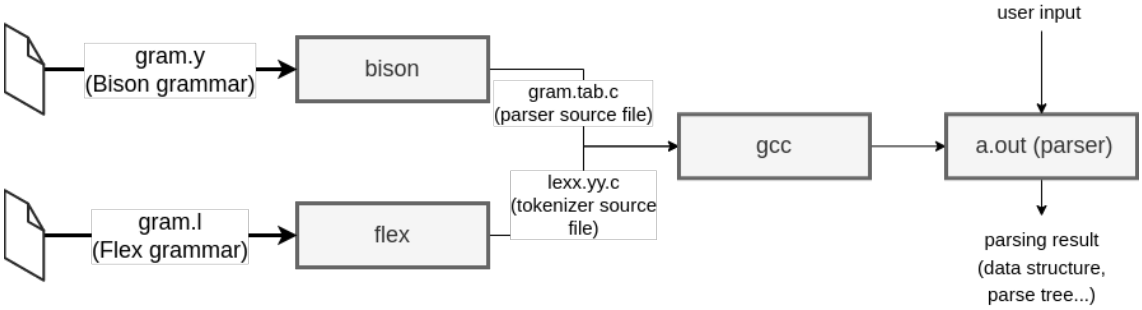


Figure 2.2: Overall Bison-Flex pipeline

Bison is a parser generator tool that constructs parsers from formal grammar descriptions of languages. The grammar file contains a set of rules that define which language is recognized. Flex is a tool that generates a lexical analyzer based on a set of regular expressions and actions expressed in another file. Using the tools on these grammar files results in the creation of a lexer and a parser in C code. The lexer takes a stream of characters as input, transforms it into a stream of tokens, and feeds it to the parser which can state if the input is a word of the language. The overall pipeline is depicted in Figure 2.2. Bison's grammar file is comprised of three distinct parts: the prologue, the grammar rules, and the epilogue.

The **prologue** is composed of C code and Bison-specific declarations. This is the place to initialize variables, define functions, and all necessary C code before the grammar rules are specified.

The **grammar rules** section consists of a sequence of rules which defines how input can be processed by the parser. A rule is made of a **left-hand side**: a nonterminal which identifies the rule, and a **right-hand side**, with none, one or more symbols (terminal or nonterminal). Additionally, the rule can possess **actions**. They are sections of code that will be executed whenever that rule is used, and the position of action within the rule will dictate when it will be called. Listing 2.1

illustrates a rule derived from the MySQL Bison grammar. In this listing, the left-hand side is highlighted in blue, the right-hand side is highlighted in red, and the action code is enclosed in brackets.

```
1   drop_user_stmt :  DROP USER if_exists user_list
2     {
3       LEX *lex=Lex;
4       lex->sql_command= SQLCOM_DROP_USER;
5       lex->drop_if_exists= $3;
6       lex->users_list= *$4;;
7     }
8 ;
```

Listing 2.1: Example of a rule in the MySQL Bison grammar

Finally, the **epilogue** allows for additional C code declaration required to conclude the Bison grammar file.

BISON-generated parsers utilize the **shift-reduce** parsing technique, which enables parsing an input text in a single forward pass by progressively constructing the parse tree. This construction is achieved through a combination of shift and reduce operations. A **shift** involves shifting by an input symbol (in our case a token) from the input buffer. This shifted symbol is treated as the node of a new parse tree and is stored in a stack. The **reduce** operation consists of using a grammar rule to merge the parse trees in the stack into a new one. We, therefore, simplify the stack by creating a higher-level representation of what has been processed so far.

2.1.4 Flex mechanism

Bison takes tokens provided by Flex as inputs. Flex reads grammar files and tokenizes words based on rules. Rules are constituted by a pattern and a related action. The pattern corresponds to a regular expression and the action usually returns a token as well as saves the semantic value of the token in `yylval` so it can be accessed

by Bison.

```
[0-9]+ {
      yylval = atoi(yytext);
      return INTEGER;
      }
```

Listing 2.2: Flex grammar rule example

The content of a file is not consistently tokenized using the same strategy. For instance, processing comments and code involve different strategies. Consequently, Flex implements **Start conditions**: features to limit the scope of rules. When processing a file, the lexer can transition into different states, determining the set of rules that can be applied at a given time. This feature proved to be highly valuable for us, as we could disregard the prologue and epilogue sections of a Bison grammar.

2.2 Word embedding methods

In this section, we offer a comprehensive introduction to the natural processing techniques that will be utilized throughout this thesis. Specifically, we are focusing on the word embedding mechanism, which plays a central role in this research.

2.2.1 Natural language processing background

Natural Language Processing (NLP), can be viewed as a data processing pipeline that begins with raw text (or corpus) which undergoes various transformation, normalization, and standardization techniques, including tokenization, and stop-word removal. **Tokenization** in natural language processing is the concept of splitting text into smaller chunks. A token can correspond to a sentence, a word, or even a subword. **Stop words** are frequently used words in a particular language that do not convey significant information for natural language processing tasks (*the, yours, there...*). By removing stop words, the focus can be directed toward the important

information present in a text. This process also helps in reducing the size of the dataset and improving processing time.

Using these techniques over corpus results in the creation of data structures tailored for various Machine Learning (ML) tasks. In this thesis, we aim to use NLP to compute **semantic similarity** between keywords parsed in source grammars and already defined tags that represent actions on a system.

We will use semantic similarity to predict the impact of user input on a given system by analyzing the **semantics** (in a linguistic sense) present in the source code. Semantic similarity computation involves being able to represent words, groups of words, sentences, or even concepts into a mathematical object on which we can perform operations in a realistic amount of time. We will now detail how **word embeddings** allows us to do so.

2.2.2 Overview of words embedding mechanism

Recent innovations in NLP take their origin from Word2Vec [7], presented by Mikolov, Chen, Corrado, *et al.* in "Efficient estimation of word representations in vector space". They managed to map words into a dimensional space with a relatively low number of dimensions, therefore allowing us to perform operations between words. However, work about representing words as vectors can be traced back to 1950 when features used to represent the meaning of words were manually selected. Word2Vec was a breakthrough in the NLP domain and allowed the appearance of many new mechanisms: subword analysis, [8], context sensitivity [9], [10]. We now present Word2Vec, BERT, and Sentence-BERT technologies that allow us to compute semantic similarity between tokens.

Word2Vec

Introduced in 2013, Word2Vec [7] is a word embedding technique that employs a neural network model to discover associations between words by analyzing a corpus. Using unsupervised machine learning, Word2Vec results in the creation of a vector space with several hundred dimensions and automatically maps words into vectors. **Unsupervised machine learning** corresponds to the usage of algorithms to train a model without labelling datasets(no human intervention). The use of low-dimensional vectors enables various operations to be performed between words, allowing for the inference of their semantic similarity.

In Machine Learning, a fundamental element in constructing models for any task is the identification of the important features that are essential and sufficient to obtain the highest performances possible. In the domain of natural language processing, the dimensionality of features can reach the size of the entire vocabulary. This is the case for the most basic way to encode a word: **one-hot encoding** where each word is represented by a binary vector containing a single 1 and the remaining cells as 0. With the increase in dimensional complexity though comes an exponential increase in computational resource requirements, a problem called the **curse of dimensionality**.

Word2Vec revolutionized the field for several reasons, as it offered an accurate and efficient solution to solve the curse of dimensionality by automatically and precisely selecting features to represent words within a given number of dimensions. But most importantly, it introduced the concept of **pre-training word embedding models**, allowing the distribution and offline usage of models and democratizing their widespread adoption and usage in the present day.

Despite being a major innovation in the field of NLP, Word2Vec had some limitations. Words with different meanings (**polysemy**) are not handled correctly. Additionally, the model cannot handle out-of-vocabulary inputs, an issue that will

later be solved through subword analysis by following models.

BERT

In 2018, Google introduce **transformers** a new embedding framework tailored for text translation tasks [11]. They present the mechanism of **attention**: even though the model focuses on a given token in a text, it has access to the entire input to aid in generating the next output.

A year later the same company released BERT [10], a transformers-based model that outperforms previous models on a wide range of NLP tasks. BERT also provided a way to manage polysemy, an embedding not only represents information about the token itself but also about the context it has been found in. Another notable feature of BERT is its capability to generate embeddings for groups of words, sentences, and even paragraphs whereas Word2Vec was limited to computing embeddings for individual words only.

Despite outstanding performances in question answering, natural language inference, or classification, the model's architecture leads to very high computation time for the task of semantic similarity (finding the most similar pair of sentences in a collection of 10,000 sentences takes about 65 hours).

Sentence-Transformers

Sentence-BERT [12] introduced in the paper *"Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks"* a few months after BERT, presents a modified BERT architecture, using the performances of the state-of-the-art model and tailored for large-scale semantic textual similarity computation. The model enables a significant reduction in computation time, decreasing it from 65 hours to just a few seconds.

For each token in a given input (sentence, text, group of words), the model

will compute contextualized word embeddings. Then the embeddings go through a pooling layer to get a single fixed-length embedding for the whole input. Different pooling strategies can be used such as computing the average or selecting the maximum values of embeddings.

Similar to the technologies mentioned earlier, Sentence-BERT enables the pre-training of models and their distribution. To utilize these models for general purposes, we can use the pre-computed models provided by the authors. They also published **Sentence-Transformers**, a Python framework to easily use and fine-tune models based on their architecture. In addition, they make use of the well-established HuggingFace model hub to distribute fine-tuned models [13].

3 Related work

3.1 Intrusion Detection Systems

In this short section, we review the literature on intrusion detection systems using parsing mechanisms and compare our proposed solution with existing approaches.

3.1.1 Diglossia

Diglossia [14] is a tool that aims to detect server-side web application injections. The tool relies on Ray and Ligatti's [15] definition of code to consistently detect any attempt of code injection and prevent false positives. They state that anything that is not a fully defined value is code. For example, in MySQL, values include strings, numbers, and reserved values (NULL, TRUE, FALSE...) and everything else is considered code. Diglossia employs a positive taint tracking methodology, wherein all trusted data are tracked and all user-provided inputs are untracked. Their taint implementation corresponds to the mapping of all application-generated characters into shadow characters. This results in the creation of a shadow query where only the user input is in original characters. The tool then uses a dual-parsing technique, both the original and shadow query are parsed. Subsequently, Diglossia checks if the two parse trees are syntactically isomorphic and that all code in the shadow query is in shadow characters.

Diglossia still has some limitations, as stated in their paper, it may have blind

spots when web applications rely on third-party PECL extensions. This reliance can lead to improper tainting, resulting in false positives or incorrect analysis results. Additionally, Diglossia can be bypassed through taint-inference attacks. [16].

Despite flaws in the implementation of the tainting mechanism, this paper highlights the importance of properly defining what are the intended behaviours, and which are the ones to prohibit to eliminate false positives and false negatives. Moreover, the paper also illustrates the ability to prevent a malicious query from being executed during the parsing, which is something we also aim to achieve, by placing ourselves within parsers. Diglossia's objective is to use tainting and parsing analysis to determine if a user input can be trusted. We differ from their solution by using parsing information to infer the semantics of a query.

3.1.2 Sqlcheck

SQLCheck is a tool introduced in 2006, by Su and Wassermann in the paper *"The essence of command injection attacks in web applications"* [17]. The base observation in this paper is that all user inputs that change the **syntactic structure** of the query are either malicious or invalid. Therefore, SQLCheck uses meta-characters to forbid input substrings from modifying the syntactic structure of a query.

SQLcheck is a tool that is unable to detect relatively complex attacks such as tautology, alternate encoding attacks, or stored procedures [18]. Moreover, it has the major drawback of needing configuration by the developer. Misconfiguration is still at the origin of roughly 10% data breaches each year [19] and in this case, setting too permissive rules will render the mechanism ineffective, and too restrictive ones will result in application failure. Moreover, as Diglossia, SQLCheck is not able to detect second-order injections.

We extend the definition they use to detect a malicious input, rather than checking whenever the syntactic structure of a query is modified. We aim to detect

whenever a not intended *operation* is about to be performed on a system.

Both Diglossia and SQLCheck are tools focused on injection detection and require some kind of configuration whereas our approach is transparent for the web programmers. Moreover, we aim to not only detect injections but all kinds of abnormal behaviour, for instance, an intruder that managed to have a foothold on a machine and access to MySQL shell, and create a new user will be detected through our approach. The semantics of an attacker query would differ from the semantics of normal/benign queries.

3.1.3 SEPTIC

Medeiros, Beatriz, Neves, *et al.* [12] noted a **semantic mismatch** between the assumptions by creators of detection tools and administrative systems regarding the behaviour of Database Management Systems when processing a query and the actual consequences of queries on the database. This disparity could result in lower effectiveness of security measures and potential vulnerabilities.

To tackle this issue they present SEPTIC, a runtime mechanism designed to prevent attacks within the DBMS. To identify SQL injection attacks, SEPTIC compares queries and authorized query models. A set of query models is computed by forcing calls to all benign queries within an application. It is updated after every new application release. If a query fails to match any of the established query models, it is put in quarantine.

Despite their emphasis on the syntactic structure of inputs the approach of Medeiros, Beatriz, Neves, *et al.* is similar to ours. We concur with the notion that vital information may be overlooked if the detection process is conducted outside the parser. Additionally, our objective aligns with theirs in terms of identifying system anomalies or vulnerabilities through the establishment of authorized models. In our case, models would be defined by sequences of semantics, and any deviant input

would be put in quarantine or discarded.

4 Framework

In this chapter, we elaborate on the methodologies employed in the thesis to address our research question, while also providing the rationale behind the decisions made during the entire research process.

The initial section of this chapter introduces the general structure and the formalization of the instrumentation component to address the first research questions: **What would be the semantic tags to consider for a query/command type language useful to an IDS?** Furthermore, we offer a response to the second research question: **What information is available in the grammar and high-level parser code to infer these tags?** This is accomplished by the presentation of the preliminary step of development: The construction of a tool for extracting semantic information in grammar. The final two sections outline our methodology to address the following question: **How can we automate grammar instrumentation and tag prediction?** Subsequently, we describe the utilization of word embedding techniques to extract the implicit semantics from the obtained data. Additionally, we give detail about the transparent modification of the Bison grammar to generate a trace of operations that are conducted for a given user input.

4.1 Instrumentation overview

The main objective of this master thesis was to find an automated way to instrumentalize parsers into probes while preserving the integrity of the recognized language

contrary to Diglossia and SQLCheck [14], [17]. We aim to have a program able to predict the potential operations, if any, that will be executed on the system triggered by user input. This could encompass various activities such as creating a new entry in a database, modifying a file, executing commands on the system, and similar operations. We focus on the semantics of a query: its impact on an information system to overcome the difficulties met by syntactic analysis (evasion, mutation...).

We provide in Figure 4.1 two MySQL queries syntactically different, the first one using a `DELETE` statement. In the second one, we store the hexadecimal representation of the first query ASCII code, we then execute the query using the `EXECUTE` statement. In the end, the outcome of both queries will be identical in the database, but the second query has the potential to bypass intrusion detection systems. By analyzing the semantics of queries within a parser, all risks of evasion can be eliminated: the parser serves as the code responsible for initiating operations on the information system. Consequently, attempting to evade the parser would be pointless as it would result in no operations being performed on the system.

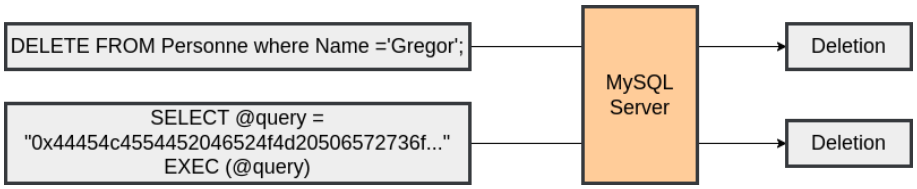


Figure 4.1: Two syntactically different MySQL queries performing the same operation on the system.

4.1.1 GAUR Architecture

The first step of this master thesis was to think about the overall architecture that would allow us to produce a sequence of *operations* while parsing. An *operation* corresponds to a keyword describing an actual operation we aim to monitor on a system (reading a file, creating a new database entry...).

Our initial idea involved transmitting to an intrusion detection system the identifier of the left-hand side nonterminal associated with a rule while parsing. The intrusion detection system would **dynamically** compute embeddings and decide which *operation* is associated with a sequence of nonterminals (corresponding to all the rules used to process an entry). Through anomaly detection, the IDS would allow or discard queries based on the generated sequence of *operation*. However, this approach introduces a subsequent overhead. Furthermore, for benign traffic, we would consistently compute identical embeddings, as benign queries always follow the same parse tree structure established by the developer.

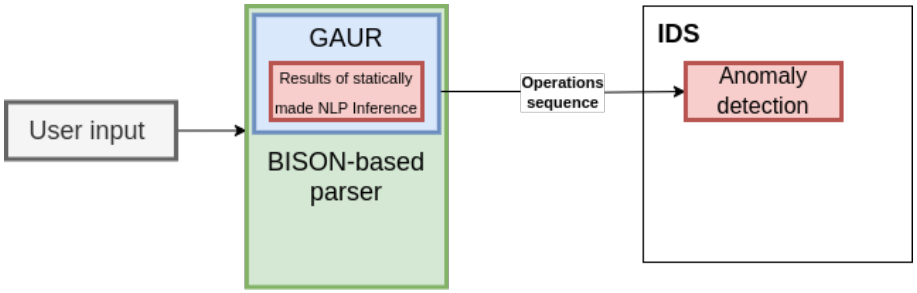


Figure 4.2: Architecture using statically computed embeddings

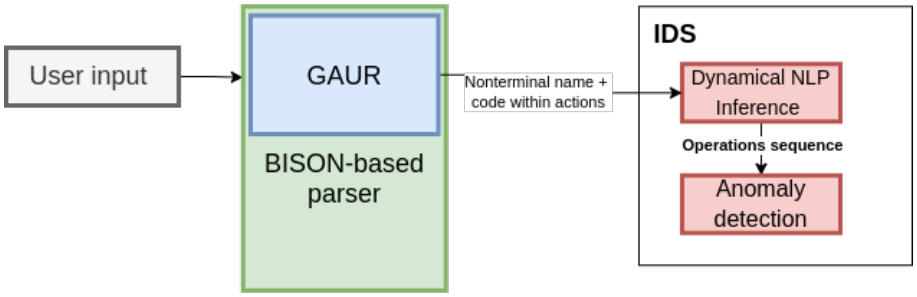


Figure 4.3: Architecture using dynamically computed embeddings

Hence, we adopted an alternative strategy wherein embeddings for each left-hand side nonterminal are **statically** computed. Subsequently, we perform an inference process to assess their semantic proximity to the *operations* we aim to monitor on the system. Finally, we use these pre-computed values within the code of the parser

at each reduction. When a user input has been fully processed, we can produce a sequence of semantics corresponding to an approximation of the *operations* performed on the system triggered by the input. The overview provided in Figure 4.4 illustrates a three-phase process. The initial phase involves extracting data from Bison grammars, followed by a semantic classification in the second step. The final step focuses on injecting code and the results of the classification process into the grammar to dynamically generate semantic traces.

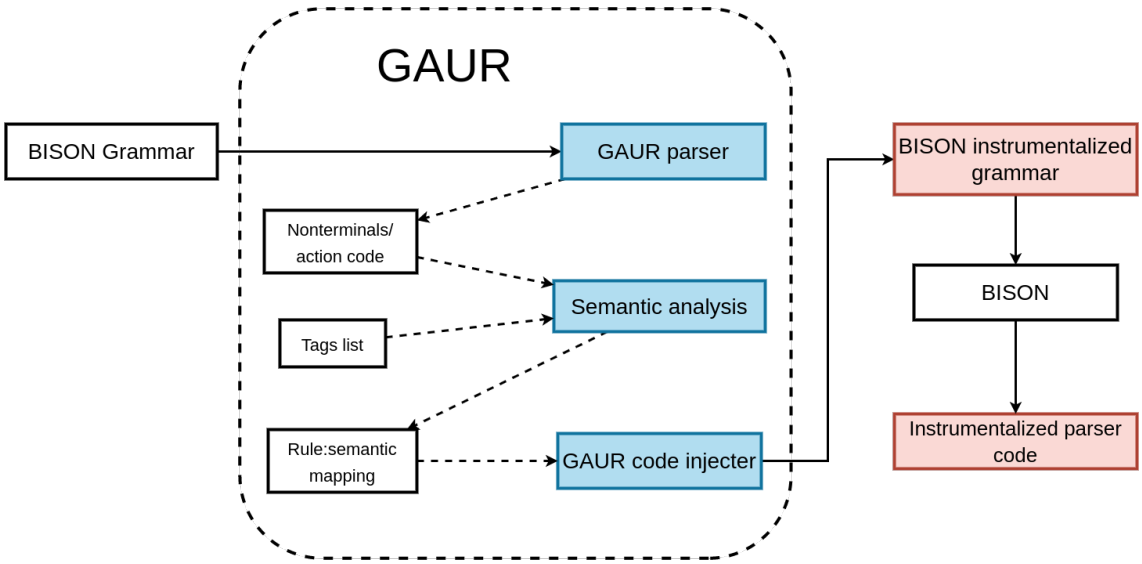


Figure 4.4: Overview of GAUR architecture

This thesis lead to the development of two programs: The first program is a Python script designed to classify documents containing nonterminals and data extracted from their action code. The second one is the GAUR executable, able to parse Bison grammars and extract relevant information for the classification. Another feature of the executable is to modify a given Bison grammar given the output of the Python program to instrument this grammar.

4.1.2 Formalization

We now formally define concepts that will be used throughout this thesis. We define an *operation* in the following way:

$$operation = \left\{ \begin{array}{l} create \\ \mid delete \\ \mid execute \\ \mid modify \\ \mid read \\ \mid seq(operation) = (operation, operation) \\ \mid \emptyset \end{array} \right\}$$

We justify our tag selection for their relevance in terms of access control. In the context of Linux permissions, three permission types are defined: read, write, and execute. To introduce more granularity, we differentiate writing operations from creation, modification, and deletion operations. Note that we consider creation and deletion as a modification.

We now define the function $markN$ as a function that returns the *operation* associated with the name of a nonterminal N given as input. Nonterminal does not always have a semantic, hence $markN$ can return $\emptyset$.

A grammar rule is not limited to a single operation, sometimes a single rule can perform a sequence of operations on a system. We define $Seq(a)$ in algorithm 1 that takes as input an array of *operation*. If the array of operations is empty, the function returns $\emptyset$, if the array contains only an element the function returns this element, and if it contains more than one, we return a **sequence** of the first element and the rest of the array.

With these defined data structures and functions we now want to get a sequence of *operation* given an abstract syntax tree, we do so through the $markParseTree$ function defined in algorithm 2.

Listing 1 Seq

```
1: function SEQ(a)
2:     if len(a) == 0 then
3:         return ∅
4:     end if
5:     if len(a) == 1 then
6:         return a[0]
7:     end if
8:     return seq(a[0], Seq(a[1 :]))
9: end function
```

The *markParseTree* functions expect a single parameter: tree, corresponding to the parse tree of user input. It works as follows: it gets the *operation* associated with the tree's root with a call to $markN$. If no *operation* is assigned to the root, we iterate over its children and get their corresponding operation with $markN$. We filter out the $\emptyset$ contained in the resulting array (line 4). If no *operation* has been assigned for all of the children nonterminal present in the tree, the resulting array will be empty and we return $\emptyset$. If exactly one *operation* has been found for all the children we return it. If we found more we return a sequence using our previously defined $Seq(a)$ function (line 11).

In the case where the root nonterminal possesses an *operation*: we calculate the corresponding *operation* of the root's children, if the resulting array is empty we only return the root's *operation*, for the case where the array consists of a single element we return a sequence of *mroot*, and this element using the $Seq(a)$ function. Otherwise, we also use the $Seq(a)$ function to build the sequence of *operation* from *mroot* and the multiple elements array *mchildren*.

Listing 2 markParseTree

```
1: function MARKPARSETREE(tree)
2:     mroot ← markN(root(tree))
3:     if mroot == ∅ then
4:         mchildren = filter(∅, map(markN, children(tree)))
5:         if len(mchildren) == 0 then
6:             return ∅
7:         end if
8:         if len(mchildren) == 1 then
9:             return mchildren[0]
10:        end if
11:        return Seq(mchildren)
12:    else
13:        mchildren = filter(∅, map(markN, children(tree)))
14:        if len(mchildren) == 0 then
15:            return mroot
16:        end if
17:        if len(mchildren) == 1 then
18:            return seq(mroot, mchildren[0])
19:        end if
20:        return seq(mroot, Seq(mchildren))
21:    end if
22: end function
```

4.2 GAUR: Data extraction

This section covers the first instrumentation step: the extraction of semantically important data from Bison grammars. We aim to collect all information giving clues about operations executed on the system upon using a grammar rule. We had different locations where we could have extracted this information. We now detail why we chose to use the input grammar file.

Extracting from grammar file

This method involves creating a parser for Bison grammar files, we can use Bison source code as a baseline. Grammar files are human-readable and relatively small which improves comprehension and debugging. Nonterminal name and action code are all together, facilitating data extraction as shown in listing 4.1.

```
1 drop_user_stmt: DROP USER if_exists user_list
2 {
3       LEX *lex=Lex;
4       lex->sql_command= SQLCOM_DROP_USER;
5       lex->drop_if_exists= $3;
6       lex->users_list= *$4;;
7 }
8 ;
```

Listing 4.1: Example of a rule in a Mysql query grammar file.

Extracting from output C file

Files generated by Bison are C files by default. Parsing C from scratch can be tedious but as stated earlier Bison is widely used and therefore we can find implementations of Bison grammar that parses C code. Although Bison generates files with a fixed structure, the resulting content is often difficult for humans to read and comprehend. Information about nonterminals and their corresponding *operation* can be found

within code and comments across the file as shown in listing 4.2.

```
case 1945: /* drop_user_stmt: DROP USER if_exists user_list  */
#line 11962 "/home/mysql/sql/sql_yacc.yy"
{
      LEX *lex=Lex;
      lex->sql_command= SQLCOM_DROP_USER;
      lex->drop_if_exists= (yyvsp[-1].num);
      lex->users_list= *(yyvsp[0].user_list);
}
```

Listing 4.2: Rule in C file produced by Bison.

We conclude that the first approach was the best choice for its simplicity (which is important since Bison is a new tool to us) and efficiency. Both files have a predetermined structure that facilitates parsing and data extraction, but due to the complexity of C, the Bison file appears to be more amenable to process. Furthermore, in a standard project compilation pipeline, build automation tools transparently invoke Bison and then compile the produced C code. Hijacking the compilation pipeline would be challenging, and requires analyzing each project compilation pipeline and figuring out when to intervene. On the contrary, intervening in the Bison grammar can be accomplished with relative ease and transparency before all project compilation. We will now elaborate on the data that can be extracted from the Bison grammar file.

Extracted information

Grammar files contain a variety of information that gives clues about the operations that will result from user input parsing. We first present all of the observed sources of information:

- **Left-hand side:** In grammar rules, the left-hand side corresponds to the name of a nonterminal. By convention, this name is an indication of the

resulting operations following the usage of the given rule. The left-hand sides are considered our main source of information for classification.

- **Terminals and nonterminals:** Positioned on the right-hand side of a rule, both terminals and nonterminals can give insights into the semantics of a set of rules. However, relying on nonterminals to classify isn't possible as we would be out of scope: their name describes actions that will be performed within their associated rule, and not in the one where they are children.

- **Code within actions:** The code present within action is also interesting to analyze, as it is the one responsible for transforming parsed data into an abstract syntax tree or directly interpreting it. We choose to extract content from the action and also provide it to the model for classification.

- **Comments:** Comments in grammar files can also give clues about the role of rules. However no convention exists on where to place a comment in a source file, it would be complex to accurately define to what rule a comment is related to.

- **Nonterminal relationships analysis:** Finally analyzing the relations between nonterminals, it becomes possible to identify clusters. This analysis could improve our classification task and is for future work.

Because of the complexity of processing all these different kinds of sources of information, we choose to focus on the extraction of the left-hand side and alphabetic strings contained in the action code. Further analysis of the relevance of this information is provided in section 5.2.2. We now detail the characteristics of extracted data in grammar.

In most grammars, the semantic of the nonterminal holds information about the processed input. For example, in listing 4.1, the nonterminal `drop_user_stmt` is a clear description of the database operation which will be performed after parsing

the user input. Additionally, more information can be found in grammar files, especially within the **action code**. Code contained in the action of a given rule will be responsible for the performed operations, it will usually initialize variables and call functions to operate on the system. The name of the variables and functions also do hold valuable semantic information. Moreover, variable type gives hints about the type of data that will be manipulated and therefore interesting to extract as well.

Currently, our analysis operates at the granularity level of each left-hand side nonterminal. For each LHS, we will inspect all of its right-hand side components, and collect every alphabetic string within the action code. We chose to ignore string constants, most of them are error messages that would pollute extracted data. In the future, we could shift the granularity level towards rules, as associated actions may vary for rules within the same set.

To give an example of the data we are extracting from a Bison rule, we took the set of rules associated with the left-hand side nonterminal `create_user` from the MySQL Bison grammar in listing 4.3. We highlighted in red the data we extract and will feed it to the next step of our pipeline. We now give some implementation details of the extraction step.

```
create_user : user identification opt_create_user_with_mfa {
              $$ = $1;
              $$-> first_factor_auth_info  = *$2;
               if   ($$-> add_mfa_identifications ($3.  mfa2 , $3. mfa3))
               MYSQL_YYABORT ;
              }
|user identified_with_plugin opt_initial_auth {
              $$= $1;
              $3-> nth_factor  = 1;
              $3-> passwordless  =  false ;
              $$-> first_factor_auth_info  = *$3;
              $2-> nth_factor  = 2;
              $2-> passwordless  =  true ;
               if  ($$-> mfa_list . push_back ($2))
               MYSQL_YYABORT ;
                     $$-> with_initial_auth  =   true ;
              }
|user opt_create_user_with_mfa {
              $$ = $1;
               if   ($$-> add_mfa_identifications ($3.  mfa2 , $3. mfa3))
               MYSQL_YYABORT ;
              }
;
```

Listing 4.3: Example of group rule from the MySQL Bison grammar and extracted data (in red).

4.2.1 Implementation details

To extract this information from every possible Bison grammar, we recreated a modified Bison parser to process Bison grammar: GAUR. We took the source code of Bison as a baseline. Since the data we extract is only within the grammar rules section we got rid of the code that parses the prologue and epilogue to reduce com-

plexity. We now detail the principal pieces of code responsible for data extraction.

Since our objective is to extract the nonterminal identifier and the alphabetic words within actions, it is necessary to augment the grammar rules with additional code that identifies and captures these elements during parsing. First, every time GAUR parses a new rule it outputs the name of the left-hand side nonterminal as seen in listing 4.4. `$1` references the semantic value associated to the terminal `ID_COLON` token which represents the left-hand side. We use our custom function `print_nterm` to print this value in an output file.

```
1 rules: ID_COLON {
2         latest_id_colon = strdup($1);
3         print_nterm(latest_id_colon);
4         free($1);}
5   named_ref COLON   }
6   rhses.1  {free(latest_id_colon);end_group_rule();}
7 ;
```

Listing 4.4: Rule recognizing the LHS of a rule in GAUR

The second phase involves extracting relevant words within the action code. To achieve this we use the **start conditions**: when parsing an action our lexer will undergo a new state and will signal every sequence of alphabetic symbols within the code. To accomplish this, the lexer returns the token `STRING_CODE` to our parser, enabling it to output its associated semantic value. Listing 4.5 provides the Bison grammar code responsible for this: the semantic value of the `STRING_CODE` token is referenced by `$2` allowing us to output it in a file.

```
code:  %empty
| code NEWLINE
| code L_BRACKET code R_BRACKET
| code STRING_CODE {print_nterm_action($2);free($2);}
| code SYMBOL_CODE
| code STRING
| code CHAR_LITERAL
| code DOLLAR_DOLLAR
;
```

Listing 4.5: Rule recognizing the code within a rule in GAUR

By incorporating these two strategically placed function calls, we can retrieve all of the desired information about a group of rules. In the future, the extraction of sequences of alphabetical characters could be replaced by a C parser, offering a finer level of granularity and enabling the extraction of more pertinent data. Calling the GAUR executable on a Bison grammar will result in the creation of a file with one nonterminal per line. On the same line will be appended the words extracted from the action code as seen in listing 4.6 which is a subset of GAUR output on the MySQL Bison grammar.

Upon analyzing the file generated by GAUR, we observe the presence of noise and words lacking semantic meaning. This underscores the necessity to use pre-processing steps, which we will now elaborate upon in detail.

```
1    create_user first_factor_auth_info if add_mfa_identifications
     mfa mfa MYSQL_YYABORT nth_factor passwordless false
     first_factor_auth_info nth_factor passwordless true if mfa_list
     push_back MYSQL_YYABORT with_initial_auth true if
     add_mfa_identifications mfa mfa MYSQL_YYABORT
2    opt_create_user_with_mfa nth_factor nullptr nth_factor
     nth_factor
3    identification
4    identified_by_password LEX_MFA m NEW_PTN LEX_MFA if m nullptr
     MYSQL_YYABORT m auth to_lex_cstring m uses_identified_by_clause
     true m Lex contains_plaintext_password true
```

Listing 4.6: Subset of extracted data from the MySQL grammar.

4.3 GAUR: Semantic similarity computation

This section presents an overview of the natural language processing techniques we applied to predict whether an operation is executed on the system when a reduction rule is employed. This analysis is conducted using the data that was previously extracted.

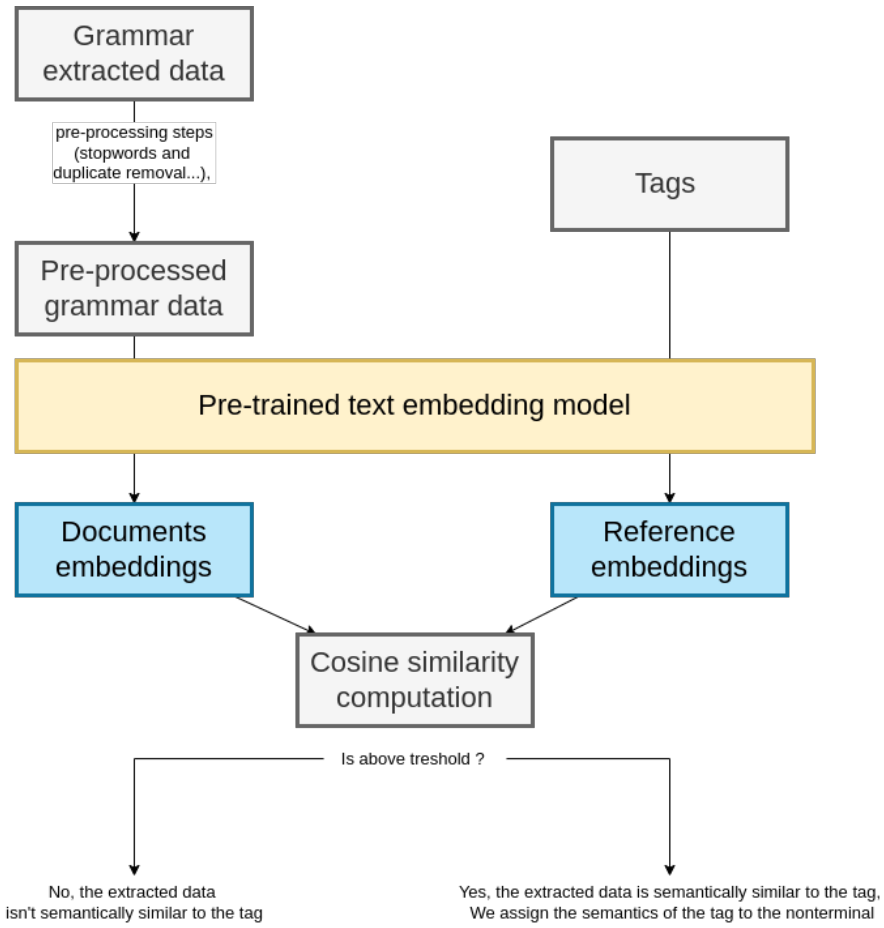


Figure 4.5: Overview of our NLP pipeline.

The overall pipeline involves a pre-processing step applied to the data extracted by GAUR. Subsequently, we utilize a pre-trained model tailored for semantic textual similarity calculation to compute the embeddings of our tags. Furthermore, we also compute embeddings for our **documents**, where each document corresponds to a nonterminal along with its associated extracted data. Finally, we compute the

cosine similarity between both embeddings: if the score is superior to a given threshold we attribute the tag semantic to the rule related to the nonterminal. The overall pipeline is depicted in figure 4.5.

4.3.1 Pre-processing steps

The extracted data consists of the nonterminal name and all alphabetic words found within the action. Pre-processing data is a common practice in various natural language processing tasks to enhance accuracy and efficiency before feeding them into models.

The initial pre-processing step involves converting all extracted data to lowercase and removing duplicate words. However, despite these measures, noise may still be present due to the inclusion of words that do not contribute any semantic information. To address this, we choose to construct a list of stopwords to eliminate such irrelevant words from the extracted data. **Stopword removal** is a standard pre-processing step in all NLP pipelines. However, in many cases, the focus of the study revolves around generic English language texts. In our case, we are dealing with a domain-specific language related to software engineering, with a particular emphasis on parsing. We were unable to find a stopwords list on such a niche domain. Consequently, we choose to construct our own.

In their work titled *"Stopwords in technical language processing"* [20], Sarica and Luo defined a methodology for constructing a domain-specific collection of stopwords in the field of technical language processing. We have decided to adopt their approach for our research. The overall methodology consists of collecting Bison grammar and extracting the nonterminals and the code from their rules. Out of this parser-related corpus, we compute ranked lists using NLP statistical metrics to identify potential candidate stopwords. Subsequently, a human evaluation of these ranked terms is conducted to validate their lack of significance. The entire procedure

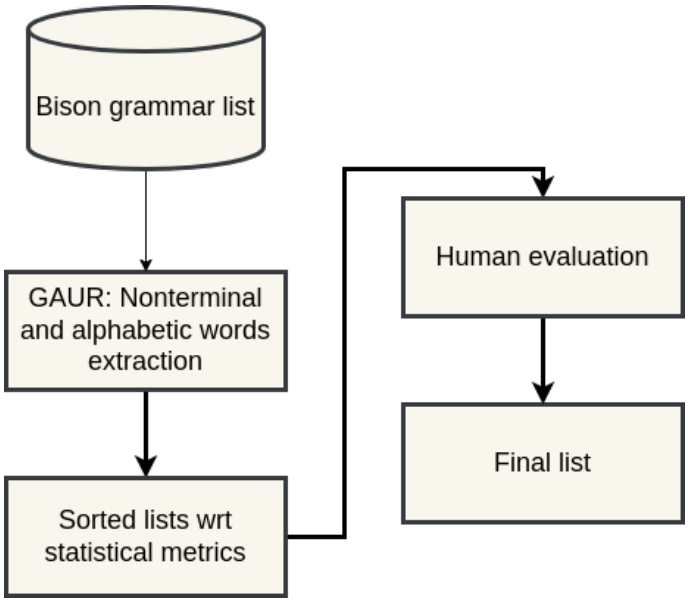


Figure 4.6: Overall stopwords removal procedure based on Sarica and Luo's work [20].

is illustrated in Figure 4.6.

The first step of the process consists of the creation of a corpus of documents extracted by GAUR. One document corresponds to a nonterminal name and the alphabetic words within its action(s).

We selected a total of 11 Grammar from widely used GitHub repositories (more than 200 stars) and with various provenance (DBMS, CLI interpreter, log aggregation system...). We applied the GAUR extractor to them to build a corpus of nonterminals and code contained in actions. We obtained a corpus of more than 50,000 tokens. The specific grammar used to construct this corpus can be found in table 4.3.1.

Each document in this corpus corresponds to a nonterminal and its associated alphabetic words from actions and is considered a document. From this corpus, we compute the four following metrics:

- **Term frequency:** This metric denotes the frequency of a term across a cor-

Grammar	Repository
arangod	https://github.com/arangodb/arangodb/blob/devel/arangod/Aql/grammar.y
bash	http://git.savannah.gnu.org/cgit/bash.git/tree/parse.y
bison	http://git.savannah.gnu.org/cgit/bison.git/tree/src/parse-gram.y
cockroach	https://github.com/yuzefovich/cockroach/blob/foobar/pkg/sql/parser/sql.y
jq	https://github.com/stedolan/jq/blob/master/src/parser.y
loki	https://github.com/grafana/loki/blob/main/pkg/logql/syntax/expr.y
mongodb	https://github.com/mongodb/mongo/blob/master/src/mongo/db/cst/grammar.yy
mysql	https://github.com/mysql/mysql-server/blob/8.0/sql/sql_yacc.yy
nebula	https://github.com/vesoft-inc/nebula/blob/master/src/parser/parser.yy
postgres	https://github.com/postgres/postgres/blob/master/src/backend/parser/gram.y
zeek	https://github.com/zeek/zeek/blob/master/src/parse.y

Table 4.1: Bison grammar used to build our corpus

pus. High-term frequency is a characteristic trait commonly observed in stopwords.

$$TF = \frac{\text{term frequency in corpus}}{\text{total words in corpus}}$$

- **Inverse Document Frequency:** This metric minimize terms appearing in a majority of the documents, implying that such terms possess limited semantic information and therefore are potential candidates for our stopword list.

$$IDF = \log\left(\frac{\text{total documents in corpus}}{|\text{ documents with term }|}\right)$$

- **Term-Frequency Inverse-Document-Frequency:** This metric gives preference to terms that exhibit a high frequency in a limited number of documents. We calculate the TF-IDF for each term within each document. Subsequently, we compute the cumulative TF-IDF score for each term across all documents. Terms with the most significant TF-IDF values emerge as potential candidates for inclusion in our stopwords list. TF-IDF is computed by multiplying the inverse document frequency of a term in a corpus of documents

by the term frequency in the whole corpus of that same term.

$$TFIDF = \log\left(\frac{\text{total documents in corpus}}{\mid \text{documents with term} \mid}\right) * \frac{\text{term frequency in corpus}}{\text{total words in corpus}}$$

- **Entropy:** This metric evaluates the distribution uniformity of a term within a corpus. Words having the highest entropy values indicate a less significant amount of semantic information compared to words with lower entropy values.

The whole pre-processing step is implemented through the Python programming language. Specifically, the computation of these metrics for our corpus involved utilizing the `scikit-learn` machine learning library [21]. Specifically, we employed the results of the `CountVectorizer` and `TfidfVectorizer` functions provided by the library.

We considered the top 50 terms for each metric and construct a union set from these terms. As expected by the result observed in Sarica and Luo 's paper, we obtain a notable overlap among the four metrics. The complete stopwords list can be found in appendix A. These 61 potential candidates then need to be evaluated by a human to validate their lack of significance. We, therefore, choose to eliminate words like `append`, `makenode`, or `new` and obtain a final list of 55 subwords. The eliminated words were words that add valuable information about a nonterminal's semantics.

4.3.2 Tags and embeddings computation

The next step in our pipeline is the computation of the embeddings for our **tags**. Tags are the kind of operations we want to monitor on a system as we previously defined in section 4.1.2. To be able to capture every nuance of a tag, we have expanded its scope by incorporating additional keywords:

- **Create**: Put, write, produce, create

- **Execute**: Invoke, run, execute
- **Delete**: Erase, remove, drop, delete
- **Modify**: Update, change, modify, alter
- **Read**: Show, display, read

4.3.3 Semantic similarity computation

We defined tags that will behave as references for comparing them with our pre-processed documents. To infer how semantically similar a document and a tag are, we use a pre-trained model tailored for semantic textual similarity. The model choice is an important step that is detailed in the Experimentations chapter.

Using the pre-trained model, we compute embeddings for each keyword and document and compute the **cosine-similarity** for each document-tag pair. Cosine similarity is the measurement of the angle between the two vectors. Whenever the angle between the two vectors is very small, the cosine of this value will be close to 1 meaning they are **semantically similar**. Once the computation of the semantic similarity value for each pair tag-document, we compare them to established thresholds to dictate if we classify a document with the tag.

Cutoff selection

Cutoff selection is the establishment of a threshold that will determine the classification of an input and in our case our document. In our work, we opted to utilize the **Youden's index** to determine this threshold.

When predicting a binary or two-class classification problem, two types of errors could occur: **False Positive** and **False Negative**. From these values, we can deduce two other metrics very useful for accuracy computation: the false positive rate (FPR) and the true positive rate (TPR). The true positive rate, also known as

sensitivity or **recall**, signifies the proportion of positive examples that are correctly classified as positive. The TPR is computed as follows:

$$TPR = \frac{\text{Number of True Positive}}{\text{Number of True Positive} + \text{Number of False Negatives}}$$

The false positive rate denotes the proportion of negative instances within the sample that have been erroneously classified as positive by the model. The FPR is computed as follows:

$$FPR = \frac{\text{Number of False Positive}}{\text{Number of False Positive} + \text{Number of True Negatives}}$$

.

Youden's index serves to optimize the true positive rate while concurrently minimizing the false negative rate. A Youden's index value of 1 signifies an absence of false positive and false negative errors, thereby indicating a flawless classification performance. The formula to compute Youden's J statistic is the following:

$$J = \text{True Positive Rate} + \text{True Negative Rate} - 1$$

Which can be simplified to:

$$J = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negatives}} - \frac{\text{False Positive}}{\text{False Positive} + \text{True Negatives}}$$

To ensure an optimal threshold selection process that accounts for both false positive and false negative rates, we compute Youden's index for various threshold values (between 0 and 1) and choose the threshold that yields the highest accuracy. It is worth noting that this approach assigns equal weight to both false positive and false negative errors, which may not be desirable in certain scenarios where minimizing false positives is important.

4.3.4 Output

We have currently acquired a semantic representation for every nonterminal. The semantics of each nonterminal can be: absence of *operation*, singular *operation*, and

multiple *operation*. The last step is to output it to a format usable by GAUR to easily modify the Bison Grammar.

We, therefore, choose to represent these semantics as flags. If the rule associated with a nonterminal possesses a semantic, the flag value is set to 1, 0 if not. For a given nonterminal we will have 5 flags, if none of them has been set it means that no *operation* has been associated with that rule. From the most significant bit to the least significant one, the flags represent the following *operations*: read, modify, execute, delete, create.

Table 4.3.4 illustrates how rules from MySQL represented by their nonterminal are flagged by our process. As an example, the rule responsible for parsing a query that results in the removal of a user `drop_user_stmt` would have attributed the semantics modify and delete. The semantics for `create_user_entry` would be: create and modify since this rule is used to treat queries to create a new user in the MySQL server. However, the nonterminal `start_entry` used as a baseline to parse every query is not given any semantics because no *operation* is performed on the system upon its usage.

	Read	Modify	Execute	Delete	Create
drop_user_stmt	0	1	0	1	0
create_user_stmt	0	1	0	0	1
start_entry	0	0	0	0	0

Table 4.2: Semantic flags attributed to a subset of MySQL grammar nonterminals.

```
1    0b00000 create_table_stmt
2    0b01010 drop_user_stmt
3    0b01001 create
4    0b01000 alter_procedure_stmt
5    0b01000 alter_function_stmt
6    0b01010 drop_table_stmt
```

```
7     0b00000 start_entry
```

Listing 4.7: Output example for the classification process.

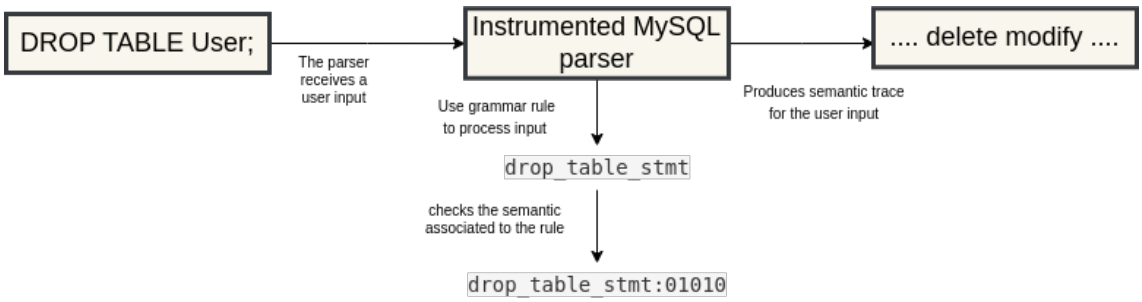
Listing 4.7 provides an example of an output of our NLP pipeline. The order in which the nonterminal and their action code have been parsed remains the same in the output produced as it is important for the next phase of our process. We are now going to detail this final step of the instrumentation of Bison grammar.

4.4 GAUR: Transparent code injection

The last phase of this thesis entails the use of the output generated from the classification process to modify the Bison grammar. In this section, we describe the methodology employed to dynamically generate a sequence of semantics by modifying Bison grammar.

4.4.1 Injecting flag values in grammars

By reading the file produced by the classification phase, we possess the semantics of each set of rules (denoted by the nonterminal name associated with the rule). We now need to use this information whenever the parser performs a reduction while processing user input to produce a semantic trace associated with the input. This process is depicted in Figure 4.4.1.



To achieve this objective we re-use the GAUR executable. It takes as input the Bison grammar to the instrument, the list of nonterminal-semantic pair, and a modified Bison skeleton. From these components, it will output an instrumented grammar file. This instrumented grammar file possesses a mapping of nonterminal towards their semantic, and function definitions to dynamically produce traces.

Upon the usage of a reduction rule, the program checks for the semantics of this rule and concatenate it with the semantics of the previously used reduction rules (its children) as described in algorithm 2. Subsequently, when using the last reduction to process a user input we possess a data structure with the semantics associated with the user input.

We now detail the different steps that take place during instrumentation.

Array injection

Re-using GAUR, we parse the Bison grammar a second time to produce an instrumented copy with code added within the prologue. To be able to dynamically utilize the result of the classification we integrate an array of semantics associated with each left-hand side nonterminal in the grammar prologue named `ggntsem`.

As stated earlier the order in which the element within the array is given is important as the Nth element in the array corresponds to the Nth declared group of rules in the grammar. For example, in listing 4.8 the semantic of the rule `start_entry` is the first one defined as it is the first rule to be defined in the MySQL grammar.

```
1    static const  int32_t ggntsem[] = {
2        0b00000, /* start_entry */
3        0b00000, /* sql_statement */
4        0b00000, /* opt_end_of_input */
5        0b00000, /* simple_statement_or_begin */
6        ...
```

```
7     }
```

Listing 4.8: Definition of the ggntsem array for the instrumented MySQL grammar.

We now develop the process by which we successfully incorporated functions that leverage the array containing semantics, allowing for the transparent generation of sequences of said semantics.

Bison skeletons

Bison implements routines for the parser to execute custom code upon performing a **reduce operation**. These routines are used to instantiate, modify or reset data structures. We choose to use Bison's already implemented piece of code to execute our custom functions when a rule is used. Hacking Bison itself would require high knowledge of the mechanism behind the tool and would be too much time-consuming for this thesis, we therefore choose to use a Bison feature: **skeletons**.

To transform a Bison grammar into C code able to recognize the described language, Bison uses **skeletons**. They are C files which dictate the structure of the resulting C file and it defines the macros, functions, and data structures. Bison provides a default skeleton, and as an option, we can modify the skeleton Bison will use by specifying a Bison declaration within the prologue of the grammar.

```
1     %skeleton "my_custom_skeleton.c"
```

Listing 4.9: Bison directive to provide a custom skeleton within a grammar prologue.

Consequently, we include the skeleton declaration in the prologue of the instrumented grammar to enable the utilization of our modified Bison skeleton. Within this customized skeleton, we have incorporated four strategically placed GAUR macro calls, as follows:

- `GAUR_PARSE_BEGIN` is called at the beginning of the parsing to initialize data structures, the main one being `gaur_sem` storing the symbols semantic.

- `GAUR_SHIFT` is positioned so it's executed whenever a shift occurs. And is used to initialize data structures.

- `GAUR_REDUCE` is called upon each reduction, and executed code corresponds to the $markParseTree$ (algorithm 2). We retrieve the semantics of the left-side nonterminal associated with the current rule. We append it to the whole query semantics buffer.

- Finally `GAUR_PARSE_END` is used to output the semantic of the whole query which so far was in a buffer. It also clears and resets any used data structure during parsing.

Our application, therefore, provides a modified skeleton that will transparently be used by Bison to generate the parser source code. We will now detail the mechanism taking place during parsing to produce traces of semantics.

Bison data structures

To gain a thorough comprehension of the adopted methodology, we further detail the different data structures manipulated by Bison [6]. In the prologue of the produced C code, Bison attributes an integer value for each symbol (terminal or nonterminal) used in the grammar. This value is denoted as **symbol kind** and is defined through an `enum` named `yysymbol_kind_t`. As a rule, Bison attributes a kind for Bison-defined symbols first, then for terminals, and finally for nonterminals. The token kind follows the order of the definition of rules within the grammar. The nonterminal appearing as a left-hand side in the first declared rule will have the lowest kind value. This is the reason why we also preserved this order in `ggntsem`: we needed to have a correspondence.

Additionally, Bison provides the macro `YYNTOKENS` which indicates the number of terminals defined in the grammar +1. We present an illustration of this mecha-

nism in Figure 4.4.1 which represents the `yysymbol_kind_t` enum for a small Bison calculator [1] and the correspondence it has with the `ggntsem` generated by Bison. We choose this grammar as an example for its small size which allows us to present the mechanisms in place in this thesis. We wouldn't instrumentalize this grammar outside of this context since no operation is triggered on the system by its parser - and therefore no semantics is appearing in the `ggntsem` array.

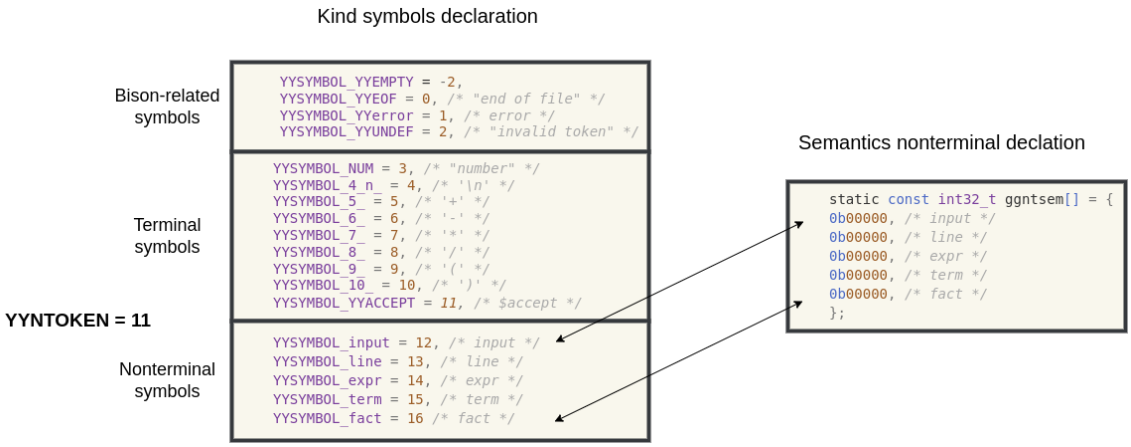


Figure 4.7: Example of the kind value attribution by Bison for a calculator grammar and the corresponding array of semantics generated by GAUR

Symbol kind is an important concept to grasp as it is how we manage to retrieve which rule has been used to perform a reduction. We previously placed our macro call `GAUR_REDUCE` in the `yyreduce` routine where we can access the symbol kind of the left-hand side of a rule. This value is therefore given to `GAUR_REDUCE` to retrieve the semantics of the rule.

Since both of our semantics array `ggntsem` and the enum `yysymbol_kind_t` are ordered the same way, we can retrieve the index of `ggntsem` $N_{ggntsem}$ to look for, given a kind using the formula:

$$N_{ggntsem} = \text{Kind - YYNTOKENS - 1}$$

4.4.2 Macro definition

We can now give details about the implementation of the macros that we placed in our custom Bison skeleton. We define these macros in the prologue of the instrumented Bison grammar after the injection of the array of semantics.

First, we define the macro `MARK_N` which returns the semantic of a rule, given the kind of its left-hand side, and implementation of the algorithm 2. For `GAUR_PARSE_BEGIN` we define the data structure responsible for storing the semantics of the different parse trees while parsing, and other variables.

```
1 #define MARK_N(i) (ggntsem[i - YYNTOKENS - 1])
```

Listing 4.10: Definition of the MARK_N macro

```
1     #define GAUR_PARSE_BEGIN(size, state_stack) \
2     char ggsem[YYINITDEPTH][MAX_SIZE_SEM];  \
3     int ggi = -1; /* Index for ggsem */     \
4     long ggid = (long)&state_stack[0];
5
```

Listing 4.11: Definition of the GAUR_PARSE_BEGIN macro

In `GAUR_SHIFT`, we increment index values to indicate that we are evaluating the semantics of a new symbol. And perform variables initialization to get the semantics of the next rule which will be used.

```
#define GAUR_SHIFT(yytoken)
do
{
    ggi++;
    if (yytoken <= YYNTOKENS)
        strcpy(ggsem[ggi], "N"); /* Shift terminal */
    else
        ggsem[ggi][0] = '\0'; /* Shift nonterminal */
} while (0)
```

Listing 4.12: Definition of the GAUR_SHIFT macro

We also defined the macro `GAUR_REDUCE` to correspond to the implementation of the *markParseTree* algorithm (algorithm 2) detailed earlier. The code for this macro is available in appendix B.

Finally, in `GAUR_PARSE_END` which is after parsing a word. We retrieve the semantics of the user query from the array `ggsem` and print it to a log file. Shortly, it will be sent to an intrusion detection system rather than simply being printed in a log file.

```
#define GAUR_PARSE_END()
do {
    FILE *f_logs;
    const char *env_fn = getenv(LOG_ENV);
    if (env_fn)
        output_name = strdup(env_fn);
    f_logs = fopen(output_name, "a");
    if (f_logs == NULL)
        perror("Gaur: cannot open file to output semantics logs");
    else {
        fprintf(f_logs, " %ld - %s\n", ggid, ggsem[ggi]);
        fclose(f_logs);
    }
```

```
14 } while(0);
```

Listing 4.13: Definition of the GAUR_PARSE_END macro

To conclude, the instrumentation of the Bison grammar is performed through the injection of different components within the prologue: an array of semantics, custom macro, and function definition, and the usage of a custom Bison skeleton. The resulting Bison grammar will therefore be able to dynamically generate traces of semantics upon parsing inputs.

We will now proceed to present the various experiments we designed to address the research questions.

5 Experimentations

5.1 Experimentations objectives

The purpose of this chapter is to provide the detail of the experimental investigations conducted following the defined methodology. The ultimate goal of these experiments is to address the various research questions posed in the study and obtain answers or insights.

To begin with, we will introduce our approach to **select an efficient embedding model** to compute semantic textual similarity. Furthermore, we will explore the potential application of **graph theory to enhance the classification** process. Lastly, we will discuss the utilization of **keywords to augment each tag** and capture nuanced semantics, aiming to enhance the accuracy of the classification process.

5.2 Experimentation results

5.2.1 Which embedding model is the most efficient to compute semantic similarity?

We now detail our approach to determine, which model is the best suited for the semantic textual similarity computation task.

Word2Vec

To accurately represent the semantics of a document in a vector, the selection of an appropriate model is crucial. Initially, we opted for the simple yet efficient Word2Vec [7] approach. Word2Vec cannot take a group of words as input. We therefore had to use a **pooling** method which is the process of transforming a sequence of word embeddings into a single sentence embedding. Consequently, we first divided every document into sequences of single words and use the max function as a way to aggregate each word embeddings into a single sentence embedding.

Our methodology involved feeding each word from a document into a pre-trained Word2Vec model and calculating the cosine similarity with each of our predefined tags. subsequently, for each tag, we would select the highest similarity score among all the words and assign this value as the semantic similarity of the document. We managed to find a Word2Vec model trained on posts from the Stack Overflow website, therefore being software-oriented [22]. As an example the most similar words to *virus*, in the base Word2Vec would be avian flu virus, viruses, flue virus, bird flu virus, and swine flu virus, whereas in the fine-tuned model the most similars are: Malwarebytes, malware, McAfee, anti-malware, and viruses[22].

Nevertheless, this approach had some limitations as it was founded on the assumption that words within a nonterminal are independent and not correlated. This assumption appeared to be false, as an example `show_create_user_stmt` would receive the semantics *read* and *create*. This classification is false: this nonterminal is seen when a `SHOW CREATE USER` query is performed. The output of such a query is the statement that creates a given user: no user creation is performed. Therefore, we need to use a model which takes into account the input and understands the relations between words in it.

Sentence-BERT

Therefore, we opted to use models using the state-of-the-art **Sentence-BERT** [12] architecture. They grasp information that relates to the sequence as a whole rather than the individual constituents. The paper's authors have released a library called `sentence-transformers`, which facilitates the utilization of pre-trained models hosted on the HuggingFace hub [13]. Among the recently published models available on the HuggingFace hub, a notable number demonstrate very high performance in generating sentence embeddings [1]. Consequently, we choose to compare models based on the same architecture but pre-trained on different datasets and different dataset weights. We focus our comparison on the small models (less than one gigabyte). To conduct our comparison, we manually labelled a subset (n = 135) of the MySQL Bison grammar rules according to the *operation* performed when the rule is used.

For model evaluation, we compute an embedding for each document and subsequently calculate the semantic similarity between each document and tag. By comparing the predicted similarity scores with the corresponding labels, we can generate Receiver Operating Characteristic curves.

Receiver Operating Characteristic curves serve as a valuable instrument for visualizing and comparing the binary classification accuracy of various models simultaneously. Essentially, a ROC curve illustrates the relationship between the false positive rate and the true positive rate, accounting for a range of thresholds. A model with good performance will have a curve that will bow up to the top left of the plot: the model manages to classify with a high TPR while concurrently maintaining a near-zero FPR. A model classifying at random will be represented as a diagonal from the bottom left to the top right. Conversely, a model operating under a random classification strategy will be represented by a diagonal line originating

[1] `https://www.sbert.net/docs/pretrained_models.html`

from the bottom-left corner towards the top-right corner of the plot. ROC curves additionally allow the computation of the **Area Under Curve** score, which serves as a metric to assess a model's accuracy across all threshold values. A high AUC score, approaching one, signifies a proficient model that minimizes false negatives even when utilizing a low threshold. Whereas an AUC score approaching zero indicates a model that produces a high number of false positives while generating only a limited number of true positives.

Table 5.2.1 displays the AUC scores corresponding to the models featured in the library documentation. Each model's ROC curve score is computed for individual classification tasks, where each task involves predicting the semantic similarity between a document and one of our predefined tags (create, delete, execute, modify, read). For each model, the average AUC score is presented which has been computed by aggregating each tag classification AUC score. AUC scores are very similar for most of the models. We have some outliers with slightly better scores or slightly worse, but they mostly all have the same accuracy.

We chose to work with the `multi-qa-MiniLM-L6-cos-v1` model because of its high score. Additionally, it's also a model that is fast. This isn't as important as if embeddings are dynamically computed but interesting nonetheless. Presented in "Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers" [23], MiniLM is a distilled version of UniLM [24]. **Knowledge distillation** refers to the process of transferring knowledge from a teacher model to a student model. This approach allows the creation of a smaller, faster, and more easily fine-tuned student model, albeit with a minor trade-off in terms of accuracy for NLP tasks. This particular version of MiniLM has been obtained by retaining only 6 layers from the teacher UniLM model. The result is a small model of size 80 Megabytes with the same accuracy. The UniLM model has not been fined tuned with the same dataset to compare performances, however for comparison, knowl-

edge distillation has been used on `stsb-roberta-base`, and the trade-off between performance decrease/speed on the STSBenchmark dataset is very interesting. The teacher model has 12 Layers by default, possess a STSbenchmark performance of 85.44, and can embed 2300 sentences per second on a V100 GPU. By reducing the layers to 6, we only lose 0.20 points of accuracy but we double the number of sentences processed by a second.

Model	AUC average
multi-qa-distilbert-cos-v1	0.8509483847390324
multi-qa-distilbert-dot-v1	0.8229308676160875
multi-qa-mpnet-base-cos-v1	0.8263801311859907
multi-qa-mpnet-base-dot-v1	0.8260692792732023
multi-qa-MiniLM-L6-cos-v1	0.8561244801764554
multi-qa-MiniLM-L6-dot-v1	0.8188262851717344
all-MiniLM-L6-v2	0.8297332102556874
all-MiniLM-L12-v2	0.8182517384338406
all-mpnet-base-v2	0.7711062189166938
paraphrase-multilingual-mpnet-base-v2	0.7637361880435583
paraphrase-albert-small-v2	0.7720677916575838
paraphrase-multilingual-MiniLM-L12-v2	0.7731578009831429
distiluse-base-multilingual-cased-v1	0.830088070090207
all-distilroberta-v1	0.7600088979704358

Table 5.1: ROC Area under curve for our dataset and small Sentence-BERT models

We now compare the accuracy performance for both models: the Word2Vec model built from Stack overflow posts and the Sentence-Transformers model. The curves for the prediction of each tag using the `multi-qa-MiniLM-L6-cos-v1` model is presented in figure 5.1 and figure 5.2. For each line, the value associated with the optimal threshold is represented as a dot.

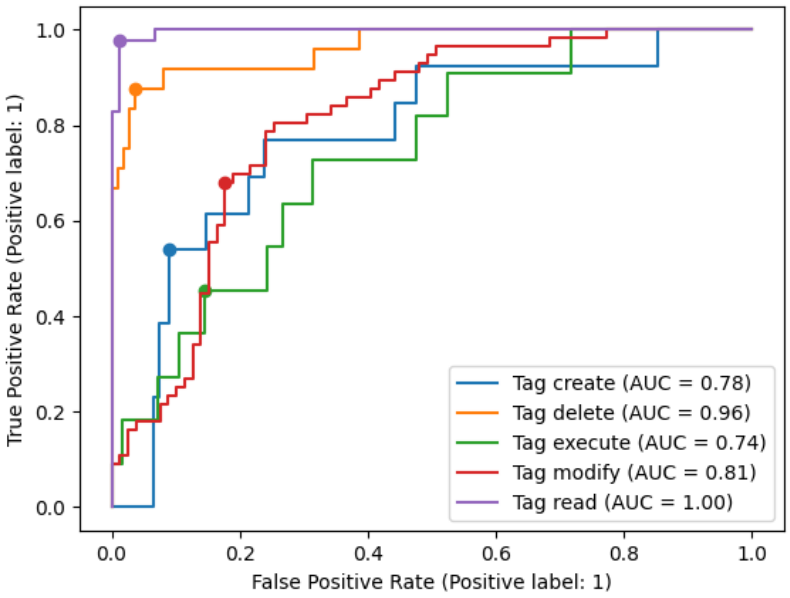


Figure 5.1: ROC curves for multi-qa-MiniLM-L6-cos-v1 model and our tags

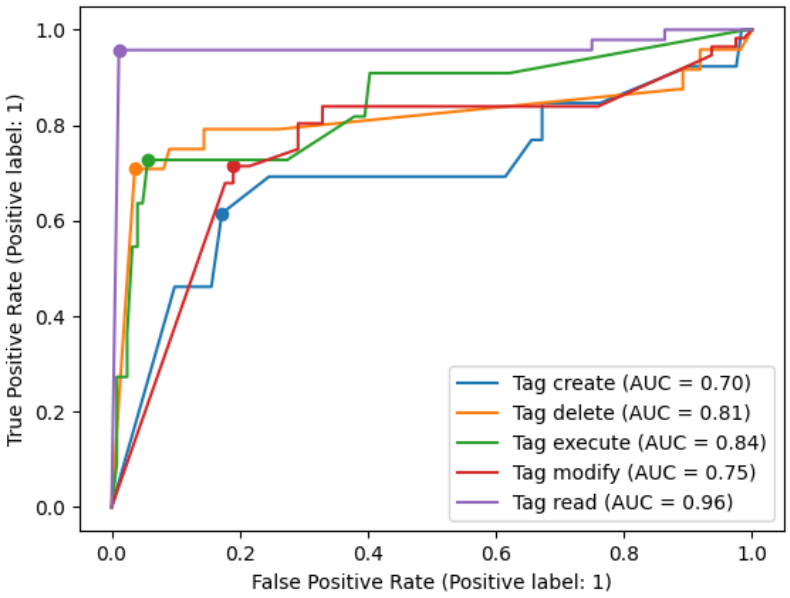


Figure 5.2: ROC curves for Word2Vec model and our tags

We provide the mean AUC for each model in table 5.2.1. The superior classi-

fication accuracy exhibited by MiniLM compared to Word2Vec can be attributed to its enhanced capability in capturing the semantic meaning of groups of words. Additionally, we have included accuracy scores for the models without performing stopwords removal and without incorporating the action code. Surprisingly, we observed that the inclusion of alphabetic words from the rule code resulted in a decrease in accuracy for the MiniLM model. Conversely, for the Word2Vec model, there was a minimal increase in accuracy. The ROC curves associated with this score are available in appendix C.

	Word2Vec	MiniLM
Nonterminals	0.8121	0.8811
Nonterminals and actions	0.8127	0.8417
Nonterminal, action and stop word removal	0.8133	0.8561

Table 5.2: Mean AUC for the classification task

The conducted experiments have demonstrated that MiniLM, as a more advanced language model, seems to understand the contextual relationships and nuances within word combinations and possess better performances at our classification task. Thus, based on these results, we have opted to incorporate MiniLM into our pipeline.

5.2.2 Which source of information in grammar file is the most relevant for classification ?

We presented the information that is extracted by GAUR from Bison grammar files and given to our classification pipeline. We provide a comparison of the classification based on the provided data. We compare the classification using both the Word2Vec model fine-tuned for software engineering [22] and the MiniLM model. We perform the classification for both models using different inputs: left-hand side nonterminals,

alphabetic words in actions, and both. We present the results of the classification in table 5.3

	Word2Vec	MiniLM
Nonterminals only	0.8121	0.8811
Actions only	0.8170	0.8145
Nonterminals and actions	0.8127	0.8417

Table 5.3: Different sources of information and their AUC scores for the classification task

The results indicate that, for the Word2Vec-based model, the action code itself can serve as a valuable source of information, as using only this information yields slightly improved classification performance compared to when we use nonterminals. However, the area under curve scores are overall very similar with improvements being on the order of hundredths.

The MiniLM-based model demonstrates reduced classification performance when provided with only the actions as input, resulting in an AUC of 0.8145. When supplied with only the nonterminals, the model achieves a better classification as we obtain an area under the curve of 0.8811. Finally, when providing both information the classification accuracy diminishes and reaches 0.8417 AUC. One possible explanation for these results could be the presence of data redundancy, wherein certain information may mislead the model when both actions and nonterminals are provided together. However, this is purely speculation, as we currently lack a concrete explanation for our results.

The results indicate that the information contained within both the actions and nonterminals' names is roughly equivalent, suggesting that there is no clear primary source of information between the two. However, to validate these findings and ensure their generalizability we would like to validate these results with a bigger dataset coming from different bison grammars. The creation of such a dataset is

time-consuming as we need to evaluate the semantics of each set of rules by either analyzing executed action code or the application's documentation.

5.2.3 How to utilize relations between nonterminals to improve tagging ?

The relations between all Bison grammar rules can be represented through a directed graph: a left-hand side nonterminal could be linked to all of its right-hand side nonterminals. To facilitate the manual labeling we added a feature to GAUR to produce a DOT file upon parsing a Bison grammar. DOT is a graph description language and allows the representation of the directed graph. The syntax is quite simple, and we use GraphViz to generate a Scalable Vector Graphics representation of the graph [25]. We present an example of a generated DOT file for the basic calculator grammar provided by Bison [1] in listing 5.1. The resulting graph generated with GraphViz is depicted in Figure 5.3.

```
digraph D { concentrate=true
    "input" -> "input";
    "input" -> "line";
    "line" -> "expr";
    "line" -> "error";
    "line" -> "\n";
    "expr" -> "expr";
    "expr" -> "term";
    "term" -> "term";
    "term" -> "fact";
    "fact" -> "number";
    "fact" -> "expr";
}
```

Listing 5.1: DOT file for a basic calculator grammar.

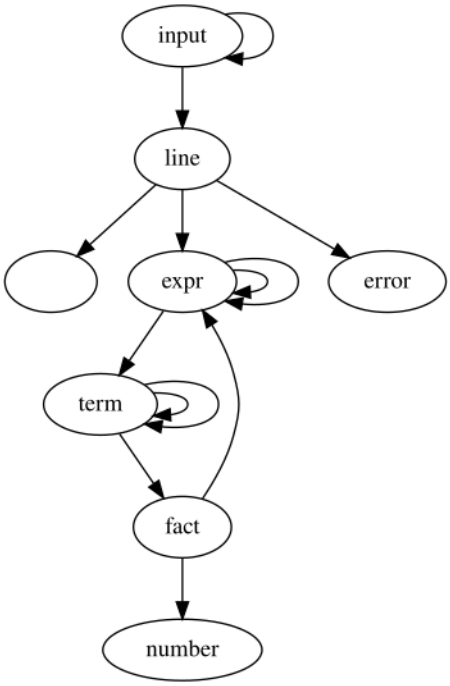


Figure 5.3: Visual representation of the calc Bison grammar with Graphviz

The visual representation of the relationships between nonterminals helped us during the labelling phase of MySQL. Moreover, the analysis of clusters within the created graph appears as a promising source of improvements for classification.

5.2.4 How to construct a keyword list to capture every nuance of a semantic

By analyzing Bison grammar, we realized that grammar developers use descriptive variables and function names but different vocabulary is used across all grammars. To tackle this challenge, we chose to augment our tag list by adding keywords for each one of them. Keywords have been added after empirical analysis of widely used projects containing a Bison grammar (4.3.1). We concluded our analysis by creating the following list of keywords for each tag:

- **Create**: Put, write, produce, create

- **Execute**: Invoke, run, execute
- **Delete**: Erase, remove, drop, delete
- **Modify**: Update, change, modify, alter
- **Read**: show, display, read

We present the impact of using a group of keywords for each tag in listing 5.2.4. The usage of keywords improves the accuracy prediction for the transformer-based model by 6 points, whereas it improves the classification by nearly 15 points for the Word2Vec-based model. The ROC curves related to these results are available in appendix C.

	Word2Vec	MiniLM
Tags alone	0.6682	0.7958
Tags and keywords	0.8133	0.8561

Table 5.4: Average AUC score for Word2Vec and MiniLM model with and without the usage of keywords for the tags

6 Conclusion and future work

6.1 Summary and conclusion

In conclusion, this master thesis describes a novel approach for inferring the semantics of user inputs in an information system. Additionally, we provide a proof of concept of the process using Bison grammar. We now provide answers to the various research questions previously defined:

What would be the semantic tags to consider for a query/command type language useful to an IDS? We showed that for our use case, using tags representing operations monitored by access control applications was relevant. We, therefore, choose to build the following list of tags: **create, delete, execute, modify, read**.

What information is available in the grammar and high-level parser code to infer these tags? Through the analysis of Bison grammar, we have identified various sources of semantical information to automatically infer the impact of a query on an information system. By examining the semantics contained in nonterminal names we achieved good classification performances. While it was initially hypothesized that incorporating variable names, function names, and other content from grammar actions would further enhance accuracy, our findings indicate that such additions introduce noise and marginally reduce prediction performance. Consequently, it appears that nonterminal names are the main source of information to

automatically infer these tags.

How can we automate the grammar instrumentation and tags prediction? We introduced GAUR, a parsing tool designed to extract valuable data from Bison grammars. Using GAUR we extracted essential information from the grammars and employed state-of-the-art Sentence Transformers to automatically infer the semantics associated with each grammar rule. GAUR is further utilized to generate an instrumented version of the input grammar. This last process involves the injection of an array containing the semantics of each rule, the incorporation of custom functions, and the utilization of a customized Bison code skeleton.

6.2 Future work

The experiments conducted during this thesis are part of a broader objective: detecting attacks using the semantics contained in the parser generator source code. To construct an effective intrusion detection system, it is imperative to enhance the accuracy of our classification, which is our primary area of focus for improvement.

6.2.1 Improving classification

False positive classifications can result in erroneous interpretations during the intrusion detection step. In light of this concern, our current objective is to enhance the accuracy of our classification approach. We present the potential improvements we identified thus far.

Improving data extraction

Transformers models are often perceived as black boxes with outcomes that are challenging to explain. Throughout our experiments we had the intuition that furnishing more content to the model would improve its classification, we however observed the

opposite. We hypothesize that adding too much data extracted from the code engenders confusion for the model: The input we provide deviates significantly from the natural language on which the model has been trained. Consequently, we aim to conduct experiments involving a more fine-grained extraction of data from the Bison grammar source code to enhance the classification process. Additionally, other sources of data, such as comments, code documentation, and online documentation could be used.

Fine-tuning transformers model

Another key improvement would be the usage of a Sentence-BERT model fine-tuned with software embedding language as we could find with Word2Vec [22]. Moreover, it appears that passing code as a sequence of tokens to a model is not optimal [26], [27]. A tree representation of code snippets leads to better classification and analysis by models. Therefore finding a more suitable data structure for the action code could also lead to higher classification scores.

Family detection through graph

The graph generation conducted using GAUR and GraphViz highlighted the potential of employing graph theory to further enhance classification. By analyzing the relations between nonterminal, it becomes possible to identify clusters based on their semantics. Incorporating a graph analysis phase into our methodology could reduce the rates of false positives and false negatives, thus improving the accuracy of our classification.

6.2.2 Creation of an intrusion detection system

We aim to create an intrusion detection system based on **anomaly detection**. By setting, or automatically crafting a list of authorized sequences of actions, we would

be able to detect any deviant user input, and subsequently raise a warning to the system administrator.

6.2.3 Upgrading to an intrusion prevention system

The main advantage of the instrumenting parser is our ability to interact with it, if malicious user input is detected we could dynamically stop the parsing/processing improving our solution to a **Intrusion Prevention System**. This is where reside the difference between an IDS and IPS: the ability for an intrusion prevention system to interact with user inputs and to block them if they are judged malicious.

References

[1] P. GNU. "Simple bison calculator grammar". (2023), [Online]. Available: https://git.savannah.gnu.org/cgit/bison.git/tree/examples/c/calc/calc.y.

[2] T. Parr and K. Fisher, "Ll (*) the foundation of the antlr parser generator", *ACM Sigplan Notices*, vol. 46, no. 6, pp. 425–436, 2011.

[3] F. Ortin, J. Quiroga, O. Rodriguez-Prieto, and M. Garcia, "An empirical evaluation of lex/yacc and antlr parser generation tools", *Plos one*, vol. 17, no. 3, e0264326, 2022.

[4] S. IT. "Db-engines ranking". (Jun. 2023), [Online]. Available: https://db-engines.com/en/.

[5] T. Parr. "About the antlr parser generator". (2023), [Online]. Available: https://www.antlr.org/about.html (visited on 02/27/2023).

[6] P. GNU. "Gnu bison". (2023), [Online]. Available: https://www.gnu.org/software/bison/.

[7] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space", *arXiv preprint arXiv:1301.3781*, 2013.

[8] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information", *Transactions of the association for computational linguistics*, vol. 5, pp. 135–146, 2017.

[9] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation", in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding", *arXiv preprint arXiv:1810.04805*, 2018.

[11] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need", *Advances in neural information processing systems*, vol. 30, 2017.

[12] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks", in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, Nov. 2019. [Online]. Available: `https://arxiv.org/abs/1908.10084`.

[13] HuggingFace. "Huggingface model hub: Sentence-transformers". (2023), [Online]. Available: `https://huggingface.co/models?library=sentence-transformers`.

[14] S. Son, K. S. McKinley, and V. Shmatikov, "Diglossia: Detecting code injection attacks with precision and efficiency", in *Proceedings of the 2013 ACM SIGSAC conference on computer & communications security*, 2013, pp. 1181–1192.

[15] D. Ray and J. Ligatti, "Defining code-injection attacks", *Acm Sigplan Notices*, vol. 47, no. 1, pp. 179–190, 2012.

[16] A. Naderi, M. Bagheri, and S. Ramezany, "Taintless: Defeating taint-powered protection tachniques", *Black Hat USA*, 2014.

[17] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications", *Acm Sigplan Notices*, vol. 41, no. 1, pp. 372–382, 2006.

[18] I. Medeiros, M. Beatriz, N. Neves, and M. Correia, "Septic: Detecting injection attacks and vulnerabilities inside the dbms", *IEEE Transactions on Reliability*, vol. 68, no. 3, pp. 1168–1188, 2019.

[19] Verizon. "2022 data breach investigations report". (2022), [Online]. Available: `https://www.verizon.com/business/resources/reports/dbir/`.

[20] S. Sarica and J. Luo, "Stopwords in technical language processing", *Plos one*, vol. 16, no. 8, e0254937, 2021.

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python", *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[22] V. Efstathiou, C. Chatzilenas, and D. Spinellis, "Word embeddings for the software engineering domain", in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 38–41.

[23] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers", *Advances in Neural Information Processing Systems*, vol. 33, pp. 5776–5788, 2020.

[24] H. Bao, L. Dong, F. Wei, *et al.*, "Unilmv2: Pseudo-masked language models for unified language model pre-training", in *International conference on machine learning*, PMLR, 2020, pp. 642–652.

[25] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering", *Software: practice and experience*, vol. 30, no. 11, pp. 1203–1233, 2000.

[26] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code", *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[27] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs", *arXiv preprint arXiv:1711.00740*, 2017.

Appendix A Stopwords list

```
thd
false
str
ast
expression
sqllex
userfieldname
auto
yymem_root
tree
val
sql_command
error
else
err
null
lappend
nullptr
object
expr
push_deprecated_warn
set_location
list_make
new
mysql_yyabort
keyfieldname
return
append
length
makestring
exprs
my_error
makedefelem
type
namelist
std
new_ptn
makenode
myf
yythd
if
options
missing_ok
name
nil
to_lex_cstring
unresolvedobjectname
location
value
int
node
true
push_back
adoptref
altertablecmd
pctx
objectchildren
sp
cnode
parser
lex
```

Listing A.1: List of stopwords generated through the usage of TF-IDF, IDF, TF, entropy metrics

Appendix B Macro code definition

```
#define GAUR_REDUCE(yysymbkind, yylen)
do
{
    printf("Begin: %d index: %d\n", yysymbkind, yysymbkind - YYNTOKENS - 1);          \
    for (int i = 0; i <= ggi; i++)
    {
        printf(">> ggsem[%d]: %s\n", i, ggsem[i]);
    }
    int32_t isem_root = MARK_N(yysymbkind);
    char sem_ast[MAX_SIZE_SEM];
    strcpy(sem_ast, "");
    if (!isem_root)
    {
        /* Root has no semantic */
        int is_empty = 1;
        for (int i = ggi - (yylen - 1); i <= ggi; i++)
        { /* Append semantic of children is sem <> 'N' */
            if (strcmp("N", ggsem[i]))
            {
                concat(sem_ast, ggsem[i]);
                is_empty = 0;
            }
        }
        if (is_empty) /* All children has 'N' semantic */
            strcpy(sem_ast, "N");
    }
    else
    { /* Root  has a semantic*/
        char *ssem_root = seq(isem_root);
        if (!ssem_root)
            break;
        for (int i = ggi - (yylen - 1); i <= ggi; i++)
        { /* Append semantic of children is sem <> 'N' */
            if (strcmp("N", ggsem[i]))
                concat(sem_ast, ggsem[i]);
        }
        concat(sem_ast, ssem_root);
        free(ssem_root);
    }
    ggi -= yylen - 1;
    strcpy(ggsem[ggi], sem_ast); /* Save sem(AST)*/
} while (0)
;
```

Listing B.1: Definition of the GAUR_REDUCE macro

Appendix C ROC Curves

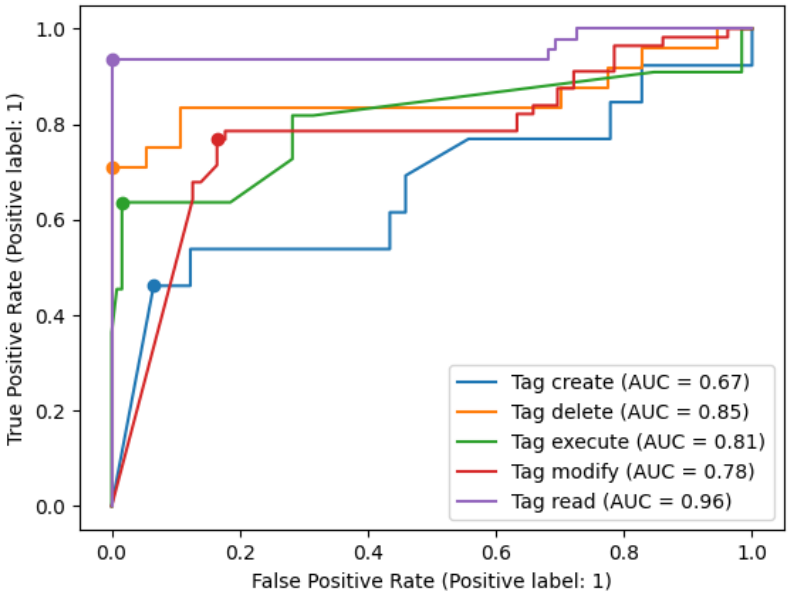


Figure C.1: ROC curves for Word2Vec model with terminal name as input

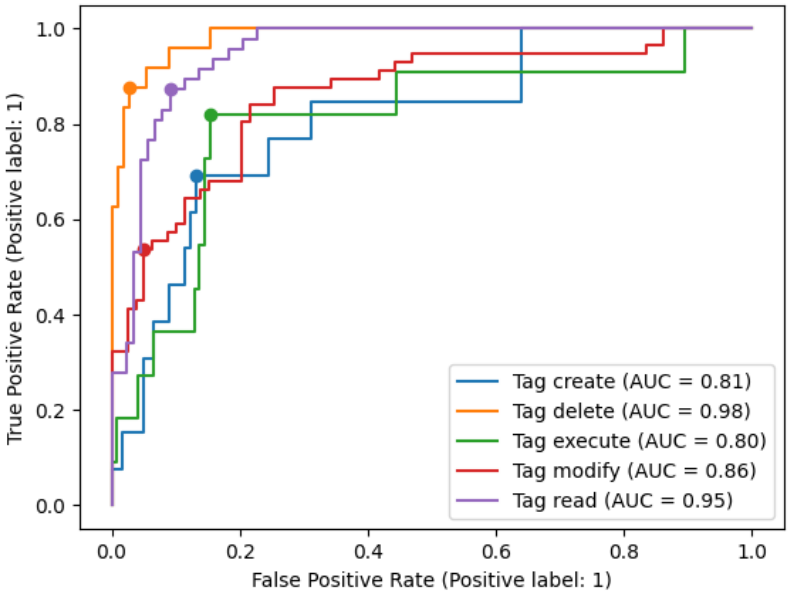


Figure C.2: ROC curves for MiniLM model with terminal name as input

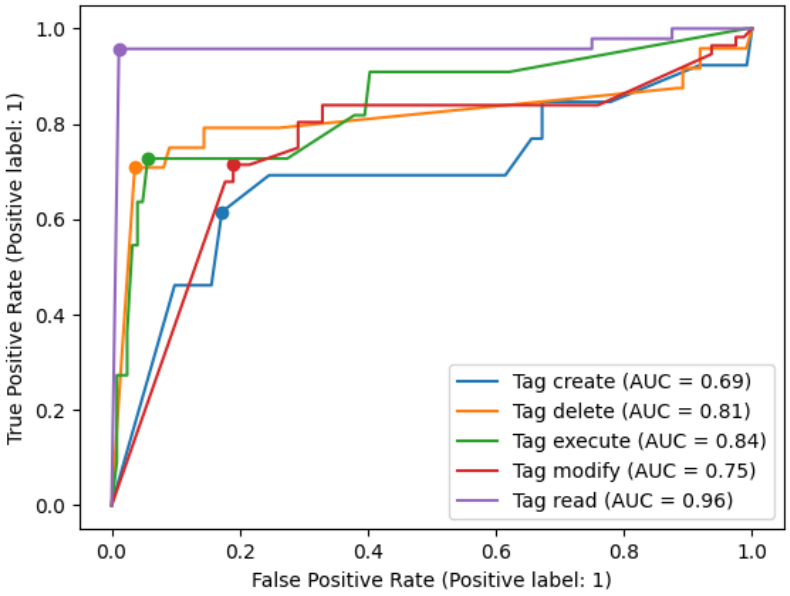


Figure C.3: ROC curves for Word2Vec model with terminal name and action code as input

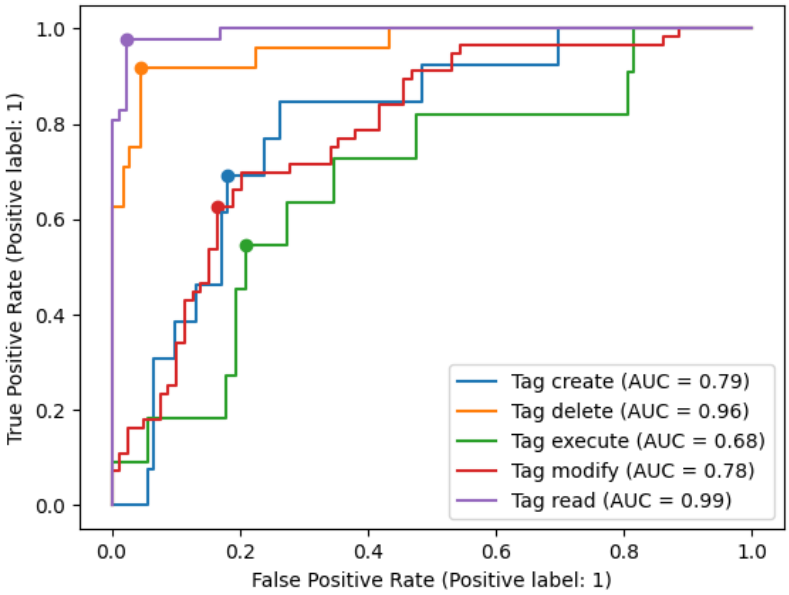


Figure C.4: ROC curves for MiniLM model with terminal name and action code as input

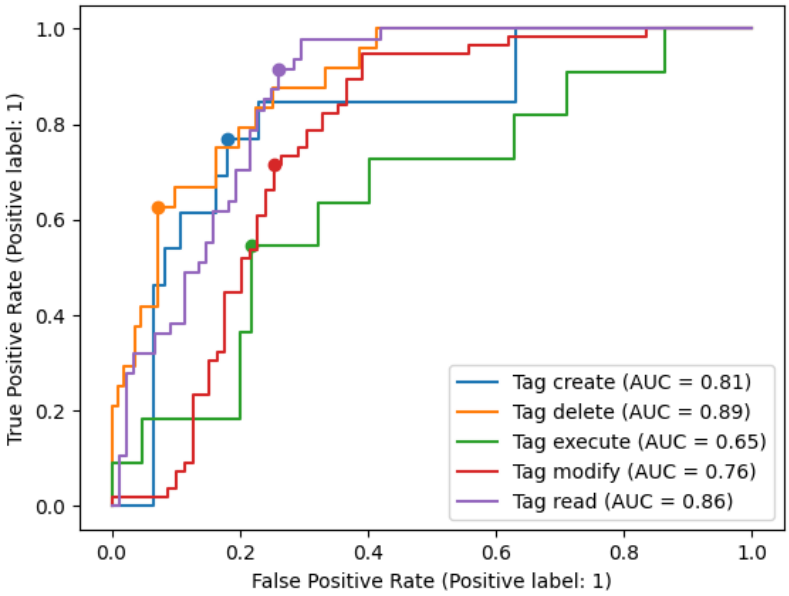


Figure C.5: ROC curves for MiniLM model without tag augmentation

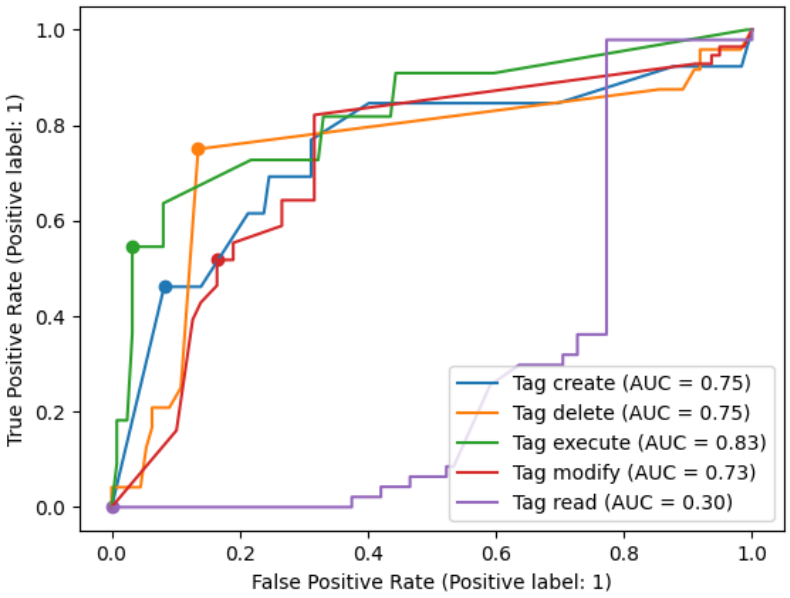


Figure C.6: ROC curves for Word2Vec model without tag augmentation