# Compilation Techniques Final Project Report

"SQL Injection Detection"


ARVIN YUWONO (2502009721)
BERNARD CHOA (2502022414)
CHELLSHE LOVE SIMROCHELLE (2502043040)
IAN WIRAWAN (2502009596)
MUHAMMAD ALFIN RIZQULLAH (2502036842)

BINUS INTERNATIONAL

FACULTY OF COMPUTING AND MEDIA

COMPUTER SCIENCE

2023

# Introduction

## Background

There are a multitude of different types of attacks that can be done to a victim in the world of computer science. One of the most prominent of these attacks is called an “SQL injection attack”. The idea behind this attack is the possibility of inserting or injecting rogue code into a machine through SQL that can then be run on the victim machine. When it is in the machine it has access and capability to do anything the attacker wants. This is a massive security concern whether it be a personal device or enterprise grade business. In general, this is a massive concern of unauthorized access, because as stated above we do not know what the attacker wiil do with access to your machine.

The effect of this, can bring up concerns of data privacy from the breach of privacy. These attackers can steal your information and sell it to the black market impacting your daily life with spams and unwanted mails. These mails can contain phishing scams that can impact your personal life. In the enterprise area, this can cause massive data leaks causing the safety of your employee and client under threat.

## Problem Statement

The issue of security and privacy of anyone is always under constant threat with the possibility of this attack happening, this is why the creation of an SQL injection programme is paramount to reduce the possibility of this attack succeeding.

# Related Works

This research is titled “A comprehensive review of methods for SQL injection attack detection and prevention”. This paper discusses the general SQLIA, along with all the attacks that has been discovered at this point in time. There are 2 prevelant type of attack for SQL which are Tautology and Blind SQL injection. Tautology attacks are reliant on the the database accepting any input as *true even username and password. Blind SQL Injection meanwhile is the type of attack that we are attempting to counter. Since it will have code that would not appear malicious, but is dangerous (Wimukthi et al., 2022).

Titled “Dection and prevention of SQLI attack and developing compressive framework using machine learning and hybrid techniques”. It states “web application has existed for a long time” at this point, but the danger of vulnerability attack still goes more and more present. One of the most common attack to this

day, is still an SQL injection attack. This kind of attack are prevelant for website that were poorly design. There has been many attempts to eradicate but there hasn't been a fixed solution that can properly get rid of this problem. The research utilizes a hybrid approach (ANN and SVM) on a large dataset to detect SQL injection and evaluate the results using various performance metrics, with overall results being 98% and above (Demilie & Deriba, 2022). Although quite comprehensive already, there is still limited discussion on the reason of the approach and lack of comparison analysis with existing solutions. Furthermore, the data source details seems to be unknown mostly and there are missing details on the process of selecting features.

Web server contain loopholes that can be exploited by SQL injections. Ping-chen discusses this on his research paper. The research examines examples of ASP website platform system injection attack prevention technology, allowing SQL injection prevention technology, to provide a better, more effective role in the real-world application of web security systems against hackers and other malevolent damage (Ping-Chen, 2011). Overall, the paper is comprehensive as it provides technical details, practical examples, database identification and preventive measures. But, it is important to note that this paper was published in 2011 and may be considered outdated with the growing development of SQL injections. In addition, the paper's coverage is limited to traditional SQL injections.

Interestingly, there's a user-friendly Python module named Prōtegō that can be used to detect and prevent SQL injection threats by accurately identifying SQL Injections through the usage of new machine learning methods (Reeves, 2022). Compared to another Python module named RobotFileParser, Prōtegō has wildcard support and length-based precedence. However, in terms of performance, it pales in comparison by 40%.

# Implementation

## Formal Description of Computational Problem

### Input

- The characters in the alphabet Σ include special characters that are utilized in SQL queries.
- The input string "w" in * represents a SQL query.

### Output

Decision function f: \}* → {SQLInjection, NotSQLInjection} such that:

- The input string w displays patterns of potential SQL injection if f(w) = SQLInjection.
- The input string w is assumed to be safe from SQL injection if f(w) = NotSQLInjection.

### Problem

Assuming an input alphabet Σ and a set of legitimate SQL queries, the aim is to create a decision function *f* that, using syntactic patterns and structures linked to SQL injection vulnerabilities, can correctly categorize each input string w ∈ * as either an SQL Injection or not an SQLInjection.

### Objective

Create a parser with minimal false negatives and false positives while optimizing accuracy in detecting malicious strings that lead to SQL injection. The program should appropriately detect safe strings without misclassifying them and classify strings with SQL injection patterns.

## Design

### Lexer

```
tokens = ('USERNAME', 'QUOTE', 'EQUAL', 'VALUE', 'OR', 'COMMENT')
```

Image 1.1 Tokens

To begin with, it is important to list the tokens that the lexer will use. The elements in a data are represented by tokens. For example,if a string of characters are determined to be a username by the lexer, the lexer will return a token of type 'USERNAME'. Each token are responsible to identify a certain pattern in a string.

```
def MyLexer():

 def t_OR(t):
  r'(?i)or'
  return t

 def t_VALUE(t):
  r'\d+'
   # Convert the matched text to an integer
  return t

 t_USERNAME = r'[a-zA-Z0-9_][a-zA-Z0-9_.]*'
 t_QUOTE = r"'"
 t_EQUAL = r'='
 t_COMMENT = r'--.*'
 t_ignore = ' \t'

 def t_newline(t):
  r'\n+'
  t.lexer.lineno += len(t.value)

 def t_error(t):
  print("Illegal character '%s'" % t.value[0])
  t.lexer.skip(1)

 # Build the Lexer
 return lex.lex()
```

Image 1.2 Defined functions in MyLexer class

The class named MyLexer is responsible for the content of the lexer. There are multiple rules are defined in the MyLexer class:

- **t_OR:** detecting the "or" keyword case-insensitively
- **t_VALUE:** matches one or more digit characters in the regular expression \d+ to determine a numeric value.
- **t_USERNAME, t_QUOTE, t_EQUAL, and t_COMMENT**: regular expressions that are used in order to pinpoint patterns within the input data.
- **t_newline:** increases the line number when encountering a newline character.
- **t_ignore:** ignore characters such as tabs and spaces.
- **t_error:** skips incorrect characters and prints the error message when facing an error
- **lex.lex**: build the lexer

## Parser

```
def p_sql_injection(p):
    '''
      sql_injection : username condition
                    | username COMMENT
      '''
    p[0] = p[1] + p[2]
```

Image 1.3 SQL Injection Function

The "p_sql_injection" function accepts a parameter "p" that stands for the tokens from the parsed input. The function combines the username with either a condition or a comment to produce a SQL injection, and it is used to define various types of SQL injection.

```
def p_username(p):
    '''
    username : USERNAME
             | empty
    '''
    p[0] = p[1]
```

Image 1.4 Username Function

It is specifying a rule for handling username parsing. It specifies that a username may be empty or the value of the token USERNAME. The value of the parsed username is kept in p[0] if a username is discovered. In the event that it is empty, p[0] is simply set to the value of p[1], which is likewise empty.

```
def p_condition(p):
  '''
  condition : simple_condition
            | complex_condition

  '''
  p[0] = " ".join(p[1:])
```

Image 1.5 Condition Function

A conditional parsing rule is defined in this code. Condition is defined into two types: simple condition and complex condition. The parsed condition's elements are combined into a string and assigned to the final parsed condition using **p[0] = " ".join(p[1:])**.

```
44  def p_simple_condition(p):
45      '''
46      simple_condition : QUOTE COMMENT
47      '''
48      p[0] = " ".join(p[1:])
49
50  def p_complex_condition(p):
51      '''
52      complex_condition :  QUOTE OR QUOTE VALUE QUOTE EQUAL QUOTE VALUE
53                         | QUOTE OR QUOTE QUOTE EQUAL QUOTE
54      '''
55      p[0] = " ".join(p[1:])
```

Image 1.6 Condition Types Function

A "simple_condition" is defined as a combination of a QUOTE and a COMMENT. Following parsing of the condition, the elements are combined into a single string and assigned to p[0].

A "complex_condition" is defined as a set of token combinations that include QUOTE, OR, VALUE, and EQUAL as written on the code s.nippet. It then combines the components into a single string and assigns it to p[0], similar to "simple_condition".

```
38  def p_empty(p):
39      '''
40      empty :
41      '''
42      p[0] = ""
```

Image 1.7 Empty Function

The rule **p_empty** handles empty production. Basically, the parser will set the value of p[0] to an empty string in the event that it comes across an empty production. This rule exist so that the parser is able to manage situations in which there isn't anything to parse.

# Complexity Analysis for Algorithms & Data Structures

## Lexer

### Time Complexity

**Lexer Tokenization**

Tokenizing a string of length n usually has an O(n) time complexity.
Token definitions using regular expressions are frequently processed in linear time

.

**Matching Regular Expressions**

The time complexity of the regular expression patterns (such as r'(?i)or', r'\d+', and r'--.*') that your lexer uses is often proportional to the length of the input text. Being proportional to length means the complexity is O(n).

**Loop for Tokenization**

The temporal complexity of the loop that tokenizes the input string by iterating over it is proportional to its length. Since its based on length, then the complexity is O(n).Space Complexity

### Space Complexity

**Storage of Tokens:**

Each token must have information about its type, value, and potentially position stored by the lexer.
The quantity of tokens created during the lexing process directly correlates with the space complexity required for token storage. The space complexity at this part is O(n).

**Compiling Regular Expressions:**

Regular expressions in their compiled form could use more RAM.
The quantity and complexity of regular expressions utilized have an impact on the space complexity.
Thus, it is O(n).

**Lexer State**

Certain internal states of the lexer, like the line and column numbers that are currently in use, might need to be maintained.
To keep this condition in place, the space complexity remains constant, O(1).

**Error Resolution:**

In the event of an error, the lexer may need to keep some error-related data.
Error handling space complexity is often constant, O(1).

## Parser

### Time Complexity

**Parse Table Construction**

This construction has a time complexity of O(n^3), which is really slow, however the construction does not have to be repeated, which justifies the cubic time complexity.

.

**Shift/Reduce Actions**

Each token is iterated one at time, resulting in linear time complexity.

### Space Complexity

**Parsing Table**

The table consists of terminals on one of the dimensions (typically the row), and nonterminals on the other dimension (typically the column), and within the table itself are production rules corresponding to the appropriate terminal-nonterminal pair. The space complexity at this part is O(n^2).

**Parsing Stack**

The stack will temporarily and dynamically store tokens as the shift/reduce operations are executed, and is generally proportional to the input string length in the worst case (O(n)).

**Input String Representation**

The input string supplies tokens to the parsing stack one at a time, since it is a linear data structure like the parsing stack, the input string is also O(n).

**Abstract Syntax Tree Representation**

The size of the AST varies greatly between different input strings, however it has a near-linear space complexity in most practical scenarios.

# Evaluation

# Implementation Details

## Lexer

```
from ply import lex
import re
```

Image 2.1 Imported Libraries in Lexer

The **from ply import lex** is where we import the Lex module. The group utilized **ply** as it is a popular library that can be used for lexical analysis. This library is helpful in tokenizing input text, which is a vital part of the lexical analysis and understanding of computer languages.

The built-in Python module **re** is imported with the **import re statement**. The regular expressions can be used to match and manipulate string patterns. Therefore, it is commonly used for tasks like pattern matching and string management.

```
lexer = MyLexer()
```

Image 2.2 Instantiate a lexer object

Create a lexer object based on the MyLexer class.

```
lexer.input(data)
```

Image 2.3 Input data to lexer

The input data is sent into the lexer using the function lexer.input(data), where the input data will be tokenized by the lexer, which will divide it up into discrete parts known as tokens.

```
for token in lexer:
 print(token)
```

Image 2.5 Print each token in lexer

This code iterates over each token generated by the lexer using a loop. It prints the token to the console for every token.

```
Run                                    118ms on 17:20:58, 01/07
LexToken(USERNAME,'admin',1,0)
LexToken(QUOTE,"'",1,5)
LexToken(OR,'or',1,7)
LexToken(QUOTE,"'",1,10)
LexToken(VALUE,'1',1,11)
LexToken(QUOTE,"'",1,12)
LexToken(EQUAL,'=',1,13)
LexToken(QUOTE,"'",1,14)
LexToken(VALUE,'1',1,15)
```

Image 2.6 Console results

## Parser

```
1  from lexer import tokens
2  from ply import yacc
```

Image 2.7 Imported libraries in parser

The tokens are obtained from the lexer that have already been created previously.
One tool used in the compilation process to create parsers for grammars is called Yacc. This enables us to specify a language's grammar and apply it to the development of a parser that comprehends and translates code written in that language.

```
62  parser = yacc.yacc()
```

Image 2.8 Instantiate parser object

Build the parser using the yacc library.

```
64 with open('parserinjection.txt', 'r') as file:
65     lines = file.readlines()
66     for line in lines:
67         result = parser.parse(line)
68         if result:
69             print(f" SQL Injection: {line}")
70         else:
71             print(f"Not SQL Injection: {line}")
72
73 print('---NOT SQL INJECTION---')
74 with open('parsersafe.txt', 'r') as file:
75     lines = file.readlines()
76     for line in lines:
77         result = parser.parse(line)
78         if result:
79             print(f" SQL Injection: {line}")
80         else:
81             print(f"Not SQL Injection: {line}")
```

Image 2.9 Looping through lines and determining sql injection

This section of the code iterates through each line on the designated text file and parses the data on each line then print on console whether it is sql injection or not.

```
84  snapshot = tracemalloc.take_snapshot()
85  top_stats = snapshot.statistics('lineno')
86
87  print("[ Top 10 ]")
88  for stat in top_stats[:10]:
89      print(stat)
```

Image 2.10 Taking snapshots of memory usage

Tracemalloc is used to record the memory usage and later print out the top 10 usage based on the statistics obtained.

## Data Sets

The datasets are the ‘parserinjection.txt’ and ‘parsersafe.txt’ files. Each text file contains 15 lines of words. They will be used as input for the parsing.

parserinjection.txt

```
1  'or''='
2  'or'1'='1
3  user'or'1'='1
4  user' or ''='
5  admin' or '1'='1
6  admin' or ''='
7  '--
8  user'--
9  admin'--
10 vsauce34659'or ''='
11 vsauce34659' or '3'='3
12 vsauce34659'--
13 test' or '2'='2
14 user' or '3'='3
15 example' or '4'='4
```

Image 2.11 Content of ‘parserinjection.txt’

The ‘parsersafe.txt’ file contains 15 lines. Each line are common words typically typed in the string when conducting a SQL injection attack.

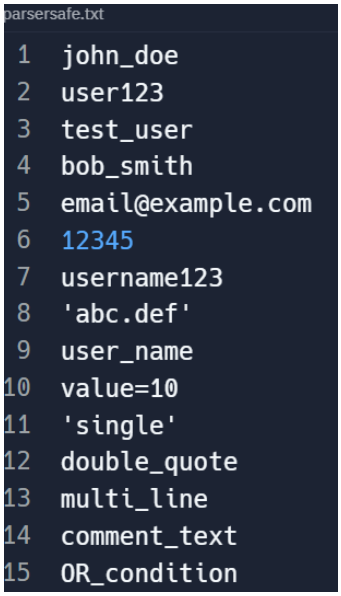

Image 2.12 Content of ‘parsersafe.txt’

The ‘parsersafe.txt’ file contains 15 lines. Each line are common words typically typed in the string when conducting a SQL query.

## Experimental Analysis and Results

The program is relatively fast, finishing execution in as little as 200ms, and there are no significant indications that the program is performing slowly. The recorded time data below is provided by Replit in the terminal.

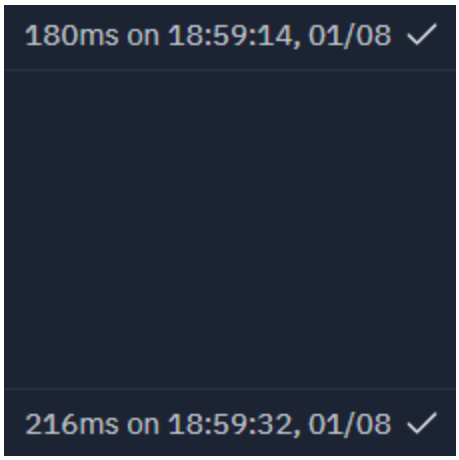

Image 2.13 Program runtime details on replit.com to calculate time consumption

Using tracemalloc, we were able to discover that memory usage is also minimal, mostly in the kilobyte range, as seen in the screenshot below. This means the scanning and parsing process itself is actually quite lightweight.

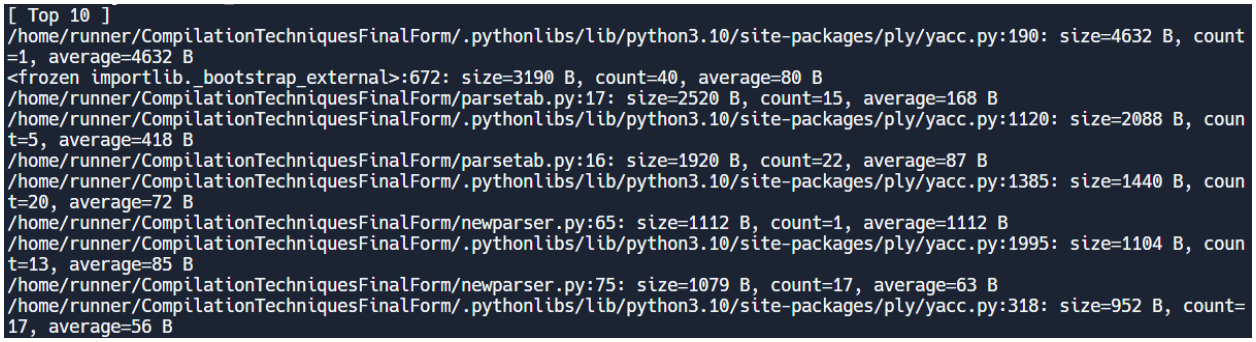

Image 2.14 Program runtime details to calculate memory consumption

The parser was able to correctly identify 15 out of 15 positives and 15 out of 15 negatives from the dataset. This shows that given the current limited range of defined rules, the parser was able to successfully classify the SQL queries correctly.

| | **SQL Injection (‘parserinjection.txt’)** | **Not SQL Injection (‘parsersafe.txt’)** |
|---|---|---|
| **True** | 15 | 15 |
| **False** | 0 | 0 |

Table 1.1 Correct and Incorrect SQL Injection Detection using Parser

# Discussion

The program's goals have been met, and it represents a big step forward in strengthening security protocols. The ability to identify malicious strings without unnecessarily interfering with legitimate operations demonstrates the parser's efficacy. But there's always room for improvement, even in the face of success. The development process should include frequent updates and improvements to accommodate new threats and changing attack methods. Sustaining the parser's robustness across various environments also requires ongoing testing using a variety of datasets and real-world scenarios. Working together with the cybersecurity community can yield insightful comments and insights that will help developers stay ahead of potential security holes and improve the program's ability to fend off SQL injection attacks. Maintaining the effectiveness of the parser requires proactive steps and constant attention to detail.

# Conclusion and Recommendation

The program works well at detecting malicious string inputs based on some of the most common SQL injection types. However, there are several other types of SQL injections which the program has not been designed to detect. The program itself is also not implemented in any

specific application, perhaps the program can be embedded within a web application in the future to assist with data security and sanitization.

# Program Manual

Using GitHub

1. Open https://github.com/Gitroars/CompilationTechniques_FinalProject on your browser
2. Download the code as ZIP
3. Open the extracted folder in your IDE like Visual Studio Code
4. Install all required libraries such as lexer and ply by using the following in the console:
   pip install lexer
   pip install ply
5. Open ‘newparser.py’
6. Run the program

# References


Demilie, W. B., & Deriba, F. G. (2022). Detection and prevention of SQLI attacks and developing compressive framework using machine learning and hybrid techniques. *Journal of Big Data*, *9*(1). https://doi.org/10.1186/s40537-022-00678-0

HR, Y. W., Kottegoda, H., Andaraweera, D., & Palihena, P. (2022). A comprehensive review of methods for SQL injection attack detection and prevention. *ResearchGate*. https://www.researchgate.net/publication/364935556_A_comprehensive_review_of_methods_for_SQL_injection_attack_detection_and_prevention

Ping-Chen, X. (2011). SQL injection attack and guard technical research. *Procedia Engineering*, *15*, 4131–4135. https://doi.org/10.1016/j.proeng.2011.08.775

Reeves, B. (2022). *Protego: a Python package for SQL injection detection*. ScholarWorks@CWU. https://digitalcommons.cwu.edu/source/2022/COTS/88/

*scrapy/protego: A pure-Python robots.txt parser with support for modern conventions.* (2019). GitHub. https://github.com/scrapy/protego?tab=readme-ov-file

# Appendices

## Link to the demo video (with max. length of 5 minutes)

Final Project Video Demo

## Link to the GIT website

https://github.com/Gitroars/CompilationTechniques_FinalProject

## RE Specification

Python regex: ^[a-zA-Z0-9][a-zA-Z0-9_.]*$'(( or '[0-9]?'='[0-9]?')|(--))
Tokens:

- Username: ^[a-zA-Z0-9][a-zA-Z0-9_.]*$
- Quote: ‘
- Equal: =
- Value: [0-9]
- OR: or
- Comment: --.*

# Contributions

Replit allowed the team to work effectively and efficiently due to it's real-time collaboration feature similar to Google Docs, but this caused difficulty in tracking individual contribution. Therefore, we listed our contributions in the table below for easy tracking of individual contributions.

| | Arvin Yuwono | Bernard Choa | Chellse Love | Ian Wirawan | Muhammad Alfin |
|---|---|---|---|---|---|
| Presentation Slides | ✔ | ✔ | ✔ | ✔ | ✔ |
| Introduction | | | | | ✔ |
| Related Works | ✔ | | | | ✔ |
| Implementation | ✔ | ✔ | | ✔ | ✔ |
| Evaluation | ✔ | ✔ | | | |
| Discussion | | | ✔ | | |
| Conclusion & Recommendation | | ✔ | | | |
| Program Manual | ✔ | | ✔ | | |
| Coding: Lexer | ✔ | ✔ | | ✔ | ✔ |
| Coding: Parser | ✔ | ✔ | | ✔ | |
| Demo video | | | ✔ | | |