Research Article

Hacking Back: Using Genetic Algorithms to Outsmart Hackers

[1]Ghosoun Al Hindi, [2,3]Mohammad Alshraideh, [4]Abdelrahman H. Hussein, [3]Lubna Fayez Eliyan and [5]Arafat Al-Dhaqm

[1]Computer Science Department, The University of Jordan, Amman, Jordan
[2]Artificial Intelligence Department, The University of Jordan, Amman, Jordan
[3]Information Technology College, Lusail University, Lusail, Qatar
[4]Networks and Cybersecurity Department, Hourani Center for Applied Scientific Research, Al-Ahliyya Amman University, Amman, Jordan
[5]School of Computer Science, Taylor's University, Subang Jaya, Malaysia

Article history
Received: 07-12-2024
Revised: 01-01-1970
Accepted: 07-05-2025

Corresponding Author:
Mohammad Alshraideh
Artificial Intelligence Department, The University of Jordan, Amman, Jordan;
Information Technology College, Lusail University, Lusail, Qatar
Email: mshridah@ju.edu.jo

Abstract: Web applications are widely used in today's digital landscape, necessitating robust security measures to protect against unauthorized access by malicious users. Ensuring the security of these applications requires effectively identifying and addressing vulnerabilities. This paper proposes an automated methodology for vulnerability detection, utilizing a genetic algorithm to generate test cases, which offers greater efficiency and performance compared to resource-intensive and time-consuming manual approaches. Our research highlights the effectiveness of genetic algorithms as test data generators, leveraging insights from previous studies. Given the varying severity of Structured Query Language (SQL) injection attacks, those capable of executing destructive commands, such as the "drop" command, pose a more significant threat than those that merely disclose information. We employ both white-box and black-box testing methodologies to detect SQL injection vulnerabilities. Black box testing is utilized when the source code is unavailable, while white box testing is applied when the source code is accessible. Our findings suggest that white-box testing, particularly static analysis, is more effective in identifying vulnerabilities. This study aims to enhance web application security by utilizing genetic algorithms to generate optimal test cases for vulnerability detection, providing a comprehensive approach that integrates white-box and black-box testing.

Keywords: Web Application, Vulnerabilities, Test Cases, Detection, SQLI Vulnerability, Attacker, White-box Testing, Black-box Testing, Genetic Algorithm

Introduction

The proliferation of web applications over the past decade has transformed how businesses and individuals interact with digital services. These applications often store and process sensitive user data, including financial information, authentication credentials, and transaction records. However, the widespread adoption of web applications has also exposed them to various security threats, making robust vulnerability detection mechanisms an essential aspect of modern software security.

Among the many vulnerabilities affecting web applications, SQL Injection (SQLI) remains one of the most prevalent and damaging. SQLI attacks involve injecting malicious SQL code into input fields to manipulate the backend database, potentially leading to unauthorized access to data, corruption, or system compromise. Despite the availability of secure coding practices and mitigation techniques, SQLI vulnerabilities persist due to insecure programming practices and inadequate security testing. Therefore, an efficient and automated mechanism for detecting SQL Injection (SQLI) vulnerabilities is crucial for reducing security risks and enhancing the resilience of web applications.

Traditional vulnerability detection approaches, such as manual code reviews and penetration testing, are often time-consuming, costly, and limited in scope. To address these challenges, this paper proposes an automated test case generation method utilizing Genetic Algorithms

(GA) to identify SQL Injection (SQLI) vulnerabilities in web applications. GA is an evolutionary algorithm inspired by natural selection that is particularly effective for optimizing test case generation. Unlike traditional testing techniques, GA can efficiently explore complex input spaces, thereby minimizing false positives and enhancing the accuracy of vulnerability detection.

The application of GA for SQLI detection offers several advantages over traditional approaches:

- Efficiency in Test Case Generation: GA automates the creation of test inputs that maximize vulnerability coverage while minimizing redundant test cases.
- Optimization of Test Execution: GA prioritizes high-risk input patterns, improving the detection of SQLI vulnerabilities with fewer test cases.
- Reduction of False Positives: GA can dynamically refine test cases compared to static analysis tools, leading to more accurate vulnerability detection results.
- Adaptability to Different Testing Scenarios: GA can be applied in white-box (source code available) and black-box (source code unavailable) testing environments, making it versatile for real-world applications.

This study makes several significant contributions to the field of web application security and automated vulnerability detection. It introduces a Genetic Algorithm (GA)-based framework for detecting SQL Injection (SQLI) vulnerabilities in web applications through automated test case generation. The proposed approach offers a scalable and efficient alternative to traditional manual testing methods, improving detection accuracy and reducing both false positives and false negatives.

The framework further integrates GA with white-box and black-box testing paradigms, enabling comprehensive vulnerability detection under varying conditions of source code availability. This hybrid integration enhances the robustness, adaptability, and real-world applicability of the method.

To validate its effectiveness, the proposed system is empirically evaluated on real-world web applications, demonstrating optimized test case generation, reduced execution time, and improved vulnerability coverage. In addition, a comparative analysis with other detection techniques, such as static analysis, machine learning, fuzzing, and Reinforcement Learning (RL), highlights GA's superior adaptability and efficiency in identifying SQLI vulnerabilities.

Finally, the study employs ANOVA-based statistical analysis to assess the significance of detection outcomes across different SQLI variants and test configurations. The results confirm the robustness and generalizability of the proposed framework, positioning it as a promising tool for enhancing web application security.

Materials and Methods

This study employed a Genetic Algorithm (GA) to automatically generate test cases aimed at detecting SQL Injection (SQLI) vulnerabilities in web applications. The method integrates both white-box and black-box testing paradigms to accommodate scenarios with and without access to source code.

Genetic Algorithm Configuration

A steady-state genetic algorithm was implemented, where chromosomes are selected to survive based on their fitness across generations. The GA employed tournament selection, single-point crossover, and creep/random mutation. The Character Distance (CD) fitness function was used to evaluate candidate strings by comparing their ASCII character values with target SQL injection patterns. Key parameters included:

- Population sizes: 30, 50, 70, 90, 110
- Mutation probability: empirically tuned
- Crossover method: single-point
- Fitness function: Character Distance (CD)

Experimental Environment

The system was implemented in C# using Visual Studio 2015 on a Windows 8.1 (64-bit) operating system with 4 GB of RAM and an Intel Core i3 processor (2.4 GHz). Table 6 in the manuscript details the hardware specifications.

Test Subjects

Five web applications were selected, including open-source platforms such as WebGoat.NET and Bryian Tan's Login Page, as well as custom applications with SQLI-prone input fields (login, email, credit card, and search forms). The web applications ranged in size from 173 to 1473 lines of code.

Testing Methodology

- White-box testing was applied to applications with accessible source code to perform static analysis
- Black-box testing targeted applications without source code, using the GA to probe input fields for SQLI vulnerabilities

Evaluation Metrics

Performance was assessed based on:

- Average number of generations needed to find a successful injection
- Execution time (in seconds)
- Detection success rate

Each test scenario was repeated three times per population size, and ANOVA was applied to compare mean detection performance across different SQLI input groups.

Related Work and Background

SQL Injection Attacks (SQLIA) continue to be one of the most critical cybersecurity threats, allowing attackers to manipulate databases by injecting malicious SQL queries. The growing reliance on web applications and databases has made developing robust SQL Injection detection and prevention techniques essential. Various strategies have been proposed, including static analysis, Machine Learning (ML), Reinforcement Learning (RL), and fuzzing techniques. However, Genetic Algorithm (GA)-based methods have emerged as promising solutions due to their adaptability and capacity to optimize detection parameters dynamically.

Genetic Algorithm

GA performs well in handling splitting predictors, for which the exhaustive enumeration is accepted (Alshraideh *et al.*, 2011; 2010; Alshraideh, 2008).

Alshraideh *et al.* (2011) merged two selection methods in a genetic algorithm: roulette selection and rank selection. However, MATLAB was implemented to solve a problem related to the traveling salesman. Selection, crossover, and mutation are among the most critical operators in genetic algorithms. Figure 1 illustrates the basic flowchart of the genetic algorithm.

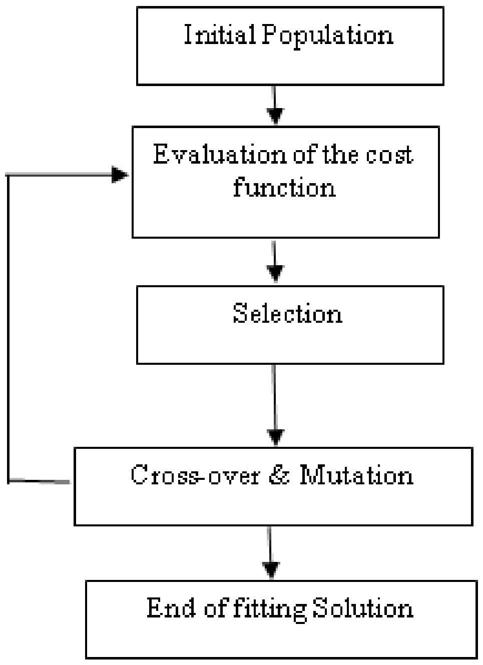


Fig. 1: Genetic algorithm approach

The merging of the previously mentioned methods aims to generate a perfect new selection that combines exploration and exploitation, influenced by optimization techniques (Alshraideh, 2008). The researchers compared the blended method (RS & RWS) with each selection method used individually. They found that the performance of the proposed method depends on the current number of generations.

The two main steps of GA are selection and manipulation. However, the probability of parent selection increases when an individual shows the best fitness value. There are three selection methods: tournament selection, roulette wheel selection, and rank-based selection. The conclusion (Razali & Geraghty, 2011) was that tournament selection is more suitable for minor problems, while the rank-based roulette wheel is better suited for larger ones. Furthermore, the first operation in rank-based selection involves sorting the individuals in the population according to their fitness values. This allows the second operation to calculate the probability of each being selected based on rank rather than fitness values. The first step could be manipulated using crossover or mutation (Razali & Geraghty, 2011).

Selecting suitable parameter values (i.e., population size, mutation probability, and crossover probability) establishes a framework for developing an effective GA (Razali & Geraghty, 2011).

Genetic Algorithms (GAs) have emerged as a practical and robust optimization and search method over the last three decades (Thakore & Torana, 2012).

The approach mentioned in Thakore and Torana (2012) combined model inference and evolutionary fuzzing. However, the fuzzy method focuses on data rather than state transition, which is considered one of its problems.

Some attacks against web applications occur due to the misuse of access specifiers. However, assigning access specifiers requires humans to thoroughly understand the software. To enforce security in software, security must be ensured during the implementation stage (Thakore & Torana, 2012).

The genetic algorithm generates test cases for the GUI flow graph (Alsorory and Alshraideh, 2023). The main advantage derived from the Genetic algorithm, as noted in Alsorory and Alshraideh (2023), is the automation of test case generation, which increases the test coverage of these cases.

In conclusion, the genetic algorithm in Avancini and Ceccato (2011) generates inputs to detect vulnerabilities. To help GA avoid local optima, symbolic constraints are collected at run time and passed to the solver. This search enables developers to identify and resolve security issues more effectively. This approach is applied to real-world PHP Code (Avancini and Ceccato, 2011; 2010). The kind adopted by Medeiros *et al.* (2016) is the reflected one, like Avancini and Ceccato (2010)

Genetic Algorithm Operator

Selection

The first GA operator is a selection used to reproduce the population. Primarily, this operator selects two individuals from a generation as parents for reproduction, creating a child or offspring for the next generation. The selection of individuals depends on their fitness; the fitness function calculates the suitability of chromosomes to survive in an environment. Before it appears in the subsequent generation, the chromosome must undergo other processes, such as crossover and mutation, which will be briefly explained. Selection is a critical factor influencing the performance of evolutionary algorithms (Allawi *et al.*, 2020; Al-Shaikh *et al.*, 2019).

A chromosome can undergo various types of selection; however, the most common types are tournament selection, roulette wheel selection, and random selection. Tournament selection is the most common Genetic Algorithm (GA) selection method because it is simple to implement and efficient (Noraini and Geraghty, 2011; Bataineh *et al.*, 2022).

Tournament selection involves conducting a tournament among S competitors, where S represents the number of competitors in the tournament. The selected individual is the one with the highest fitness among the competitors. This individual is then added to the mating pool. Increasing the tournament size raises the selection pressure, which typically means that the winner of a larger tournament will, on average, have greater fitness than the winner of a smaller tournament (Miller and Goldberg, 1995).

The roulette wheel method depends on the probability of selecting individuals, which is related to their fitness values (Noraini and Geraghty, 2011). The better individuals are, the more likely they are to be selected. Each population is defined as a section of an imaginary roulette wheel.

On the other hand, the random selection method involves randomly selecting a parent from the population. Therefore, each member of the population has an equal chance of being selected for reproduction.

Crossover

The Crossover operator concatenates two chromosomes to produce a new offspring. This concept brings new individuals closer to a global optimum, where the best genes from the parent chromosomes are combined to yield offspring that are superior to their parents (Spears and Anand, 1991).

Crossover occurs through evolution, determined by a specified probability. There are various types of crossover, including one-point crossover, multi-point crossover, and uniform crossover.

One crossover point is chosen randomly during the one-point crossover. A binary string from the start point of a chromosome to the crossover point is copied from one parent, while the rest is copied from the second parent (Figure 2). In a multi-point crossover, multiple crossover points are selected. A binary string from the start of a chromosome to the first crossover point is copied from one parent, the section from the first to the second crossover point is taken from the second parent, and the remainder is copied from the first parent (Figure 3).

Uniform crossover is a combined method of 1-point and N-point crossover, where N is the number of crossover points (Figure 4).

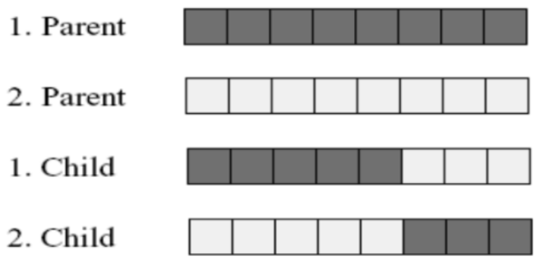


Fig. 2: Single Crossover at k = 5

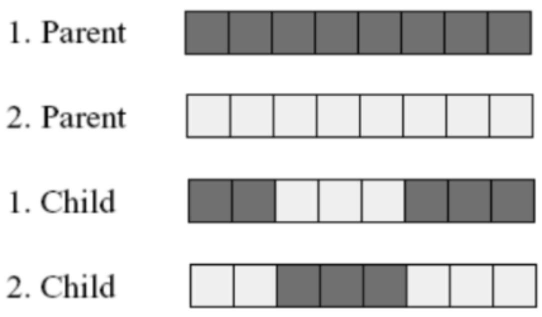


Fig. 3: Multi-point Crossover at k = 2 and k = 5

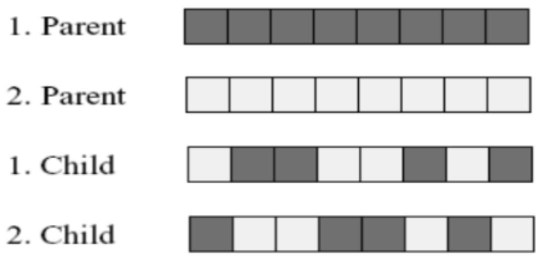


Fig. 4: Uniform Crossover at k = 1, 4, 5, and 7

Mutation

The mutation target is maintaining diversity between generations of a population's chromosomes. In mutation, one or more gene values are altered based on a predefined mutation probability from their initial values, resulting in new genes that will be added to the gene pool. Consequently, mutation helps prevent the solution from converging on local optima in the search space.

SQL Injection Detection Methods

Several methodologies have been explored for SQLI detection, categorized as follows.

Static Analysis-Based Approaches

Static analysis techniques inspect source code to detect potential SQL injection vulnerabilities without executing it. These methods typically employ lexical analysis, syntactic analysis, and taint tracking to identify suspicious query structures.

- Strengths: Early detection during development, no runtime overhead.
- Limitations: High false positives, difficulty with dynamically generated queries.
- Example: Pixy, a PHP vulnerability scanner, effectively detects SQL injection vulnerabilities but lacks scalability for large applications (Su *et al.*, 2021).

Machine Learning-Based Approaches

Machine Learning (ML) has been increasingly adopted for SQL Injection (SQLI) detection due to its ability to learn attack patterns and dynamically detect anomalies. Various Machine Learning (ML) models, such as Decision Trees, Support Vector Machines (SVMs), and Neural Networks, have been explored.

- Strengths: High adaptability and reduced false positives.
- Limitations: Requires large, labeled datasets that are vulnerable to adversarial attacks.
- Example: A hybrid model combining Random Forest and LSTM achieved 89.7% accuracy in detecting SQLI attacks while maintaining a low false positive rate (Hasan *et al.*, 2019).

Reinforcement Learning-Based Approaches

Reinforcement Learning (RL) techniques leverage an agent-based learning model that dynamically detects SQL injection vulnerabilities based on rewards and penalties. These methods excel in adaptive learning and in identifying evolving attack strategies.

- Strengths: Continual learning and efficiency against novel attacks.
- Limitations: High computational cost requires extensive training.
- Example: An RL-based SQLI detector outperformed traditional classifiers by reducing false negatives by 23% in dynamic web applications (Wang *et al.*, 2019).

Fuzzing Techniques

Fuzzing generates and inputs random SQL queries into an application to test for vulnerabilities. Advanced fuzzing tools incorporate AI-driven testing to enhance accuracy and reliability.

- Strengths: Uncovers vulnerabilities not identified by static analysis.
- Limitations: Time-consuming; may miss sophisticated SQLI payloads.
- Example: An AI-enhanced fuzzing tool discovered 47% more SQLI vulnerabilities than traditional fuzzing techniques (Bisht *et al.*, 2018).

Genetic Algorithm-Based Approaches for SQLI Detection

Genetic Algorithm (GA) is an evolutionary optimization technique inspired by natural selection. GA-based SQLI detection models generate multiple candidate solutions (queries) and iteratively refine them using selection, crossover, and mutation operators.

SQL Injection (SQLI) is recognized as one of the critical security vulnerabilities that occur when malicious code is executed within a web application. It is among the most severe and detrimental security threats (Halfond and Orso, 2006). As highlighted by Thomé *et al.* (2014), SQL injection remains a prevalent vulnerability in web applications. The primary objective of Thomé *et al.* (2014) is to detect this vulnerability using a search-based technique that identifies inputs capable of exposing such security flaws. The proposed prototype, "BIOFUZZ," demonstrates its capability to uncover vulnerabilities in real-world web applications within minutes. BIOFUZZ is a security tester that leverages evolutionary black-box testing to identify vulnerabilities, particularly SQL injection (SQLi), without requiring direct access to the source code.

SQL injection vulnerabilities can be exploited by inserting user-provided input into a web application's text box. This input is concatenated with an SQL command and executed within the database (Dukes *et al.*, 2013). Such vulnerabilities enable attackers to read, modify, or corrupt sensitive and critical data within the database (Dukes *et al.*, 2013; Halfond and Orso, 2006). Furthermore, attackers can leverage this flaw to gain unrestricted access and complete control over the application's database (Halfond and Orso, 2006).

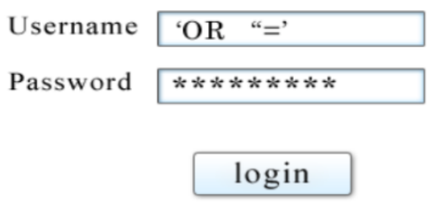


Fig. 5: SQLI example

The root cause of SQLI vulnerabilities in web applications is often the lack of proper validation of user input fields (Halfond and Orso, 2006). In an SQLI security breach, an attacker can inject SQL commands or statements into an input field, which are subsequently executed by the web application, thereby compromising the integrity of the database (Figure 5). For instance, SQLI enables attackers to extract sensitive user data, thereby facilitating unauthorized access to its primary objective (Kieyzun *et al.*, 2009; Thomé *et al.*, 2014). Unlike cross-site scripting (XSS), SQLI does not directly exploit the system's resources (Zhang, 2022).

Several types of SQL Injection (SQLI) attacks exist, each exploiting vulnerabilities within a web application's database.

Tautology-Based SQL Injection (SQLI)

This attack typically occurs during the user authentication phase (Zhang, 2022). Attackers bypass authentication mechanisms by constantly manipulating SQL statements to return true conditions.

Illegal or Logically Incorrect Queries

In this attack, the attacker deliberately introduces a type mismatch error by injecting malformed input to generate an error message. This error message can expose vulnerable parameters within the database system (Zhang, 2022; Baria & Gandhi, 2013; Kieyzun *et al.*, 2009).

Example: When a user inputs the URL http://www.example.com/?id=1234, the system may return an error message revealing table structures and field names, such as:

SELECT Username FROM users WHERE id = 1234\';

This information can aid attackers in crafting further SQLI attacks.

Union-Based SQLI

This attack exploits the SQL UNION operator to merge the results of two or more SQL statements, allowing attackers to extract unauthorized data from different database tables (Baria & Gandhi, 2013; Zhang, 2022).

Piggy-Backed Queries

In this method, attackers exploit the SQL statement delimiter (;) to append and execute additional malicious queries (Baria & Gandhi, 2013; Zhang, 2022).

Example: The following query manipulates a login authentication system by appending a malicious query:

SELECT * FROM users WHERE id = 'user' AND password = '123'; DELETE FROM users;

The DELETE FROM users command executes immediately after authentication, potentially erasing all records in the database.

Table 1: Examples of SQLI statements and inputs by the User

Input	Statement
': UPDATE [User] SET Department='it' WHERE username='service'--	select a username, Age, Department from [User] where Username like '; **UPDATE [User] SET Department='it' WHERE username='sarvi'__'**
x'; INSERT INTO members ('email','passed','login_id','full_name') VALUES ('steve@unixwiz.net','hello','steve','Steve Friedl');--	SELECT email, passwd, login_id, full_name FROM members WHERE email='x'; **INSERT INTO members('email','passwd','login_id','full_name') VALUES ('steve@unixwiz.net','hello','steve','Steve Friedl');--**
3; insert values into some_other_table	Select value1, value2, num_value3 from database where num_value3=3; **Insert values into some_other_table**
1' or '1' = '1'))/*	SELECT * FROM Users WHERE ((Username='**1' or '1' = '1'))/***>') AND (Password=MD5('$password'))
101 and ascii(substr((select+table_name+from+user_tables where rownum=1),1,1))>100	http://192.168.2.199/ora.php?id=101 **and ascii(substr((select+table_name+from+user_tables where rownum=1),1,1))>100**
105; DROP TABLE Suppliers	SELECT * FROM Users WHERE UserId = 105; **DROP TABLE Suppliers**
' union all select User from dual --	http://192.168.2.10/ora1.php?name='**union all select User from dual --**
' OR '='	SELECT name from users WHERE name='' **OR '='** AND password='' **OR '='**
John' --	SELECT name from users WHERE name='**John' --**' AND password=''
x'; DROP TABLE members; --	SELECT email, passwd, login_id, full_name FROM members WHERE email='x';**DROP TABLE members;--;**
a';DROP TABLE users; SELECT * FROM userinfo WHERE 't'='t	SELECT * FROM userinfo WHERE username = '**a';DROP TABLE users; SELECT * FROM userinfo WHERE 't'='t**';
' --	SELECT * FROM users WHERE username = 'admin**' --**' AND password = 'some_password';
' union select username, password,'1' from [User]--	SELECT name, description, price FROM products WHERE name = '**' union select username, password,'1' from [User]--**';
or 1=1 --	SELECT * FROM Users WHERE username='' **or 1=1 --** AND password='';

Stored Procedure Exploitation

Attackers can manipulate stored procedures by exploiting their input parameters and injecting malicious SQL statements into predefined database methods (Baria & Gandhi, 2013; Halfond and Orso, 2006; Zhang, 2022). As outlined in Table 1, stored procedures enable programmers to encapsulate multiple SQL statements within a named block that remains stored in the database. If inadequately secured, attackers can leverage these procedures to execute arbitrary commands.

Example: having the following procedure (Halfond and Orso, 2006), CREATE PROCEDURE Authenticated @userName varchar2, @password varchar2, @pin int AS EXEC("SELECT account no FROM users WHERE username='" +@userName+ "' and password='" +@password+ '"and pin=" +@pin;);

Hence, the attacker injection will utilize the stored procedure through a piggybacked query attack. Therefore, the query after injection will be "SELECT account no FROM users WHERE username='foo' AND pass=' '; pass; -- AND pin=

Advantages of GA for SQLI Detection

Genetic Algorithms (GA) present significant advantages for detecting SQL Injection (SQLI) attacks, primarily due to their dynamic and intelligent nature. A key strength lies in their adaptive learning capability, which allows the detection model to continuously evolve and recognize new or mutated attack variants that might otherwise go unnoticed. Furthermore, GA's inherent optimization capabilities are instrumental in refining the detection process by improving feature selection and enhancing the classification of SQL queries, thereby increasing overall accuracy. This learning-based approach also leads to a significant reduction in false positives; unlike static rule-based systems that can be overly rigid, GA learns from past attack patterns to more effectively distinguish between genuine threats and benign anomalies in user input.

Comparative Analysis of GA and Other Techniques

To highlight the novelty of GA-based approaches, Table 2 compares GA with other detection techniques. The table shows that GA-based detection strikes a balance between high accuracy, adaptability, and low false positives, making it a strong contender against traditional SQLI detection techniques.

Alternative Techniques for SQLI Detection

Besides the commonly used methods, researchers have explored alternative approaches a few of which are mentioned below.

Hybrid Approaches

Hybrid models integrate multiple detection techniques to enhance accuracy and reduce false positives. Example: Combining Machine Learning (ML) and Static Analysis increased detection accuracy by 14% (Alazab *et al.*, 2021).

Blockchain for SQLI Prevention

Blockchain technology has been proposed to secure database transactions against SQL Injection (SQLI) attacks by creating immutable logs. For example, a blockchain-secured database prevented SQL Injection (SQLI) exploitation in 94% of test cases (Zhang, 2022).

As a summary, Table 3 presents a comparative overview of traditional and modern SQL injection detection methods. It highlights each approach's detection mode, strengths, and limitations, demonstrating how modern machine learning, reinforcement learning, and genetic algorithms offer enhanced adaptability and accuracy compared to conventional methods.

Table 2: Comparison of selection strategies used in genetic algorithms based on execution time and success rate

Technique	Accuracy	False Positive Rate	Adaptability	Computational Cost
Static Analysis	Moderate	High	Low	Low
Machine Learning	High	Moderate	Moderate	High
Reinforcement Learning	High	Low	High	Very High
Fuzzing	Low	High	Moderate	Moderate
Genetic Algorithm (GA)	High	Low	High	Moderate

Table 3: Summary of traditional and modern SQL injection detection methods, highlighting their strengths and weaknesses

Approach	Examples	Detection Mode	Advantages	Limitations
Static Analysis	Pixy (Su *et al.*, 2021)	White-box	Early detection during development; no runtime overhead	High false positives; limited in handling dynamic queries
Manual Testing	Code Review, Penetration Testing	White-/Black-box	Human expertise identifies nuanced vulnerabilities	Time-consuming; expensive; not scalable
Fuzzing	AI-enhanced fuzzing (Bisht *et al.*, 2018)	Black-box	Effective in uncovering unexpected vulnerabilities	May miss sophisticated or logic-based SQLI payloads
Machine Learning (ML)	SVM, Random Forest (Hasan *et al.*, 2019)	Supervised Learning	Learns attack patterns; reduced false positives	Requires large, labeled datasets; sensitive to adversarial examples
Reinforcement Learning (RL)	Wang *et al.* (2019)	Agent-based	Adaptive to evolving attacks; reward-based learning	Computationally expensive; training required
Genetic Algorithm (GA)	Alshraideh *et al.* (2011), Current Study	Evolutionary Algorithm	Optimized test case generation, low false positives, adaptable to input types	It may require tuning, as slower convergence can occur if it is not well-configured.
Hybrid Approaches	ML + Static (Alazab *et al.*, 2021)	Combined Techniques	Balanced performance; improved accuracy	Complexity in integration and tuning
Blockchain-based	Zhang (2022)	Immutable Ledger System	Prevents tampering; ensures data traceability	Overhead integration challenges with legacy systems

Vulnerability Detection in Web Applications Using Genetic Algorithm

This section outlines the detailed steps for applying GA to automatically generate test data for web applications, enabling the detection of SQL injection vulnerabilities. Web applications are used in vital areas of our lives, making it essential to observe the exact behavior of user input in text boxes. This process is crucial because many web applications do not validate user input. Input validation ensures that the value entered by a user into a text box does not contain unique characters that could lead to an SQL injection attack. Such values must be tested to guarantee that their behavior will not result in an attack. The user input is concatenated with the SQL command in the backend code of the web application. Therefore, the entire SQL command is typically executed without syntax errors, which can hinder normal code execution.

Consider, for example, a commercial web application with a text box for the credit card number, which takes a value from the user. If the value entered is valid, one of the SQL injection cases can lead to the revelation or destruction of sensitive data stored in the web application's database. The high cost of revealing or destroying sensitive data affects critical and confidential information.

Unfortunately, the source code of all web applications requiring testing is not always available. In conclusion, black-box testing is essential in addition to white-box testing.

Consequently, using a tool to generate test cases and evaluate these web applications becomes essential to ensure they are not vulnerable to SQL injections. GA is one of the most successfully applied tools in software testing and vulnerability detection.

Motivations for Using Genetic Algorithms

Previously, we mentioned that software testing aims to design test cases that reveal as many faults as possible to improve the quality of the software and increase the reliability of the software product. We also noted that testing a software system requires considerable effort, consumes time, and incurs costs. One solution to these challenges is to automate the testing process.

Automatic software testing significantly reduces costs and time while increasing confidence in the results. Researchers have proposed various methods for generating test data cases to utilize software testing automation, including using GA automatically. The motivations for using GA are listed below:

- GA evolves faster around possible solutions.
- GA outperforms the exhaustive search and local search techniques.
- GA can be easily implemented and does not require complex programming.
- GA is a practical, robust optimization technique and search method.
- GA is an influential and innovative search method; it can efficiently solve large-scale and complex problems.

Genetic Algorithm Operators, Configurations, and Specifications

This section provides a detailed explanation of the specifications for each operator we use. A selection operator is implemented to determine how individuals are chosen to become parents for mating based on their fitness. We mentioned earlier in Section 2 the selection types for reproduction operators. A steady-state-style genetic algorithm is employed in this work. This selection method is favored because of its high performance and straightforward implementation.

The main idea in steady-state selection is that many chromosomes will survive to the next generation. This selection type works in the following way: Every generation selects chromosomes with higher fitness values to create new offspring. Next, chromosomes with lower fitness values are omitted, and the new offspring are placed in their place. The remainder of the population survives to a new generation.

In other words, fitness value is used to select better chromosomes from the population for the next generation. This value is derived from applying a fitness function that depends on the specific problem being studied. Significantly, each individual's solution has a calculated fitness value; an individual near an optimum solution receives a higher fitness level than one farther away. The fitness value is the only feedback from the search space for the Genetic Algorithm (GA).

In this paper, calculating the fitness value for an individual solution depends on the Character Distance (CD) fitness function.

This approach relies on the pairwise comparison of character values. The comparison involves summing the absolute differences among the character values of each pair. If the pairs differ in length, any missing character will increase the cost by 128, corresponding to the character search space size (Alshraideh & Bottaci, 2006).

The following example illustrates the function:

$$Sting\,(st) = st_0 st_1 st_2 \ldots, st_{l-1}, \ldots \tag{1}$$

$$Sting\,(st) = ri_0 ri_1 ri_2 \ldots, ri_{l-1}, \ldots \tag{2}$$

Then:

$$CD\,(st, ri) = \sum_{i=0}^{i=l-1} |st_i - ri_i| + 128\,(e-1) \ldots \tag{3}$$

Where, Character Distance (CD) generally refers to the total of the absolute ASCII code differences between corresponding characters in the two strings.

st and *sr* are two strings, and *e* represents the absolute difference in length between the two strings.

For example:

CD(“SET”, “CASE”) = |83-67| + |69-65| +|84-83| +|0 -69| =95.

CD(“BBB”, “AAA”) = 3;

CD(“ZZZ”, “AAA”) =75;

Subsequently, the two individuals selected from the previous step are combined to generate new offspring. This combination operation involves swapping genes or sequences of bits between the two selected individuals (Sharma *et al.*, 2014). This process is repeated to obtain a subsequent generation with a sufficient number of individuals.

The combination previously mentioned is known as a crossover operation. In the previous section, we illustrated the three types of crossovers: single crossover, multi-point crossover, and uniform crossover. In this paper, the single-point crossover is implemented because adding additional crossover points may lead to low performance and significant changes in the solution structure.

After the crossover operation, the mutation is performed by another GA operator. A mutation operator means that the elements of individuals are slightly altered according to a specific probability of producing variants in the solutions within a population. The probability of mutation indicates how often the bits of an individual will be mutated (Avdeenko & Serdyukov, 2023). This probability significantly impacts GA performance and helps prevent falling into local extremes (Liu & Fan, 2014; Korejo *et al.*, 2009). If the likelihood equals 100%, all the individual bits will be mutated, while 0% probability indicates that none will be altered.

In this paper, the Creep and random mutation methods were employed. Randomly, one of the characters in an individual is selected to be used in the mutation of one bit in the second individual. Creep mutation depends on the position of the selected character in a specific individual. Random mutation is applied by choosing a random position for both individuals and mutating the first individual's bit with the second individual's. One of the two previously mentioned mutation methods will be selected based on the Creep Mutation flag.

SQL Injection Vulnerable Web Applications

This research utilized a genetic algorithm in a web application to detect SQL Injection (SQLI) vulnerabilities. The detection is performed by generating test cases using a genetic algorithm, as the web application features user-input text boxes that an attacker may exploit to insert a malicious part into a text box concatenated with a full SQL command. Figures 6, 7, and 8 show examples of web applications that concern us.

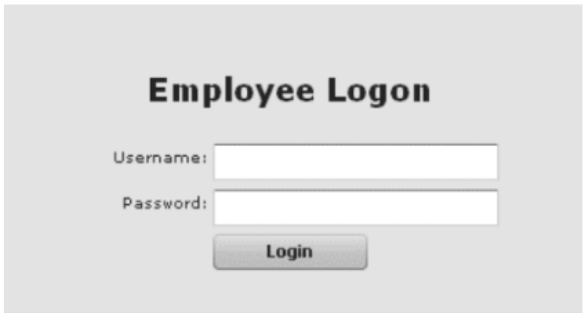


Fig. 6: "Login" web application example

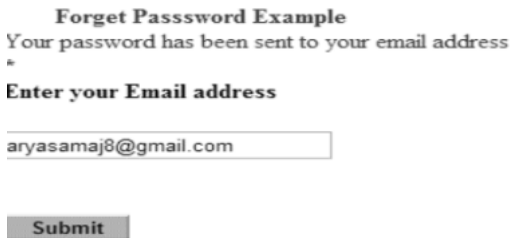


Fig. 7: "Email me" part in a web application example

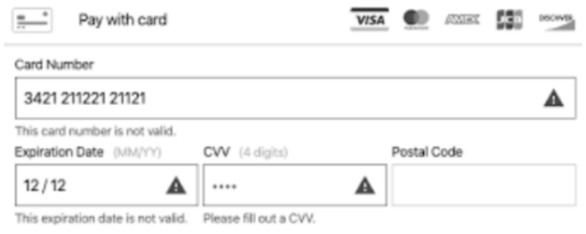


Fig. 8: "Credit card" part in a web application example

In Figure 6, the User will input their Username and password to log in to the web application and use it as a legitimate user. Meanwhile, malicious input could be inserted by an illegal user to cause a violation at the database layer of the application. In some cases, this violation results in an SQL Injection (SQLI). As a result, sensitive and critical data will be revealed or destroyed by an illegal user or attacker. In this case, the attacker could read and obtain the passwords of each User to use them to log in legally to the web application afterward.

Moreover, he will know the table names, which he can drop by inserting a drop command into the web application's username or password text box. Figure 7, with my web application's email, will work in some ways, but it will have only one text box. Although Figure 8 shows more than one text box, one could be exploited to compromise the web application's credit card number text box under SQL injection.

SQL Injection Statements

In Table 1, variant examples of SQLI statements’ clauses are listed. Instead of using all the statements’ clauses to detect, we emphasized some general cases and others that could have resulted from them. The selected clauses are illustrated in Table 4. The first column of Table 4 defines an input in the text box of a web application. As column two shows, this input will be concatenated with a full SQL command or statement. The bold part of the statement indicates the input from the User in the form of the full SQL command.

Table 4: SQLI statements' clauses and user inputs

Input	Statement
'1' or '1'='1')/*	SELECT * FROM Users WHERE (Username='1' or '1'='1')/*) AND (Password=MD5('password'))
union all select User from dual --	http://192.168.2.10/oral.php?name=union all select User from dual --
' OR '='	SELECT name from users WHERE name='' OR '=' AND password=''
x'; DROP TABLE members; --	SELECT email, passwd, login_id, full_name FROM members WHERE email='x'; DROP TABLE members; --
or 1=1 --	SELECT * FROM Users WHERE username = '' or 1=1 --' AND password = ''

White Box and Black Box Testing of Web Applications

Testing operations should be conducted to ensure the security of web applications. As mentioned, the two primary testing approaches are white-box and black-box testing. The black-box testing method is employed because the source code of some web applications is not available for consideration in this study. Conversely, white-box testing is applied to web applications that have accessible, open-source code. The choice between these two methods depends on whether the web application's source code is available.

Tested Web Applications

In this study, we investigated the application of genetic algorithms for detecting SQL Injection (SQLI) vulnerabilities, focusing on sample web applications. The testing process included both white-box and black-box testing methodologies applied to these web applications.

Figures 9 and 10 show examples of a genetic algorithm used to generate test cases, as detailed in Section 3.4. These test cases were designed to identify SQL Injection (SQLI) vulnerabilities within the web application at two distinct levels: the username and password levels.

Furthermore, Figure 11 presents another example of a web application featuring a single text box as an input component. This web application is designed to extract specific columns from a particular database table. The data displayed upon clicking the search button depends on the user's input into the search text box. Typically, the columns selected are considered more critical for users. However, SQLI attacks can be executed to unveil columns beyond the initially specified ones, often containing sensitive data. In this scenario, an attacker may attempt to reveal the password column from the user table or even try to drop the user table entirely.

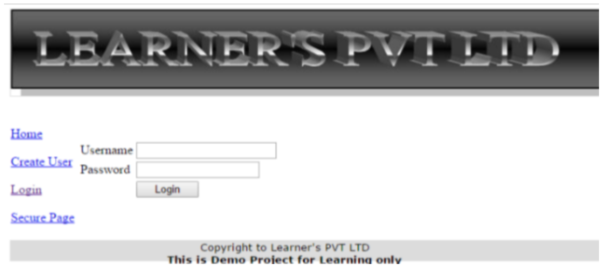


Fig. 9: Login Web application sample 1

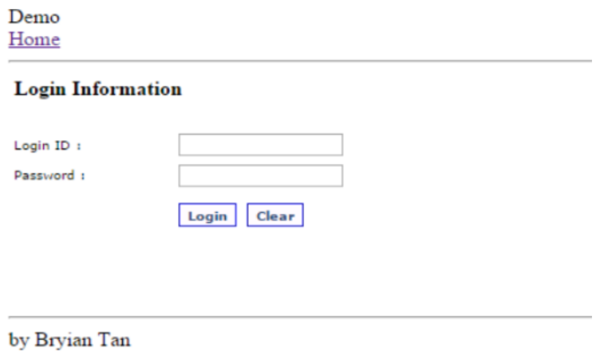


Fig. 10: Login Web application sample 2

Fig. 11: Search Web application sample

Web application Prevention from SQLI

This paper uses one of three methods to prevent SQL Injection (SQLI) in a web application. First, a regular expression identifies and replaces specific user input characters in the text box with null or space characters. After the replacement, the SQL command is executed, establishing a trusted database connection. Next, the input text is concatenated with the trusted SQL command.

The second method utilizes a parameterized query to achieve the required prevention. The input text is treated as a parameter in the SQL command. Similarly, the database connection opens and concatenates the SQL command, like in the first method.

The final method involves a specific stored procedure based on a designated value parameter from the input text box.

SQLI Sample Dataset Analysis Using ANOVA

This paper utilizes a sample from the dataset available on the Kaggle website and evaluates it using One-Way Statistical ANOVA (Analysis of Variance). The ANOVA test assesses whether the difference between the averages of two or more groups is significant based on sample data. ANOVA is generally applied when at least three groups are involved, since the two-tailed pooled variance t-test and the right-tailed ANOVA test yield the same result for two groups.

In this evaluation, we utilize three groups of 500 records, which consist of various SQL statement clauses, as shown in the table. Table 5 displays the mean and standard deviation of these data samples. Group 1 contains 1512 values, Group 2 includes 1706 values, and Group 3 comprises 2924 values.

Table 5: Anova Test - Mean & S 1

	Group 1	Group 2	Group 3
Mean	7.22	3.47	3.47
S	2.80	1.43	5.40

Results

This Section shows GA's performance as a test case generator for web applications. It also discusses experiments and evaluation parameters, presenting experimental results and analysis.

Experiment Environment

The experiments were implemented in Visual Studio 2015, making coding easier to maintain and manipulate. Table 6 presents the machine's implementation specifications.

Table 6: Hardware Specifications

Specifications Type	Value
Computer Type	HP
RAM	4 GB
CPU Type	Core i3
CPU Speed	2.40 GHz
System Type	64-bit Operating System
Operating System	Windows 8.1

Design and Implementation Issues

As mentioned, the algorithm was implemented in the C# programming language using Visual Studio, which provides various features.

The implementation output shows that our algorithm can be applied to test any web application. More significantly, this work highlighted the need to test web applications using GA to detect vulnerabilities as soon as possible, which may lead to potentially dangerous attacks. In other words, the algorithm automatically generates test data that traverses the web application's code commands under test.

Evaluation Parameters

This paper assessed GA's performance by automatically generating test cases for web applications analyzing the test case generation time and the average number of generations. The average values were calculated after running the algorithm five times, each with a different population size; the experiment was repeated three times. A population contains all generated solutions, with the population sizes being 30, 50, 70, 90, and 110. After each execution, we recorded the average number of generations and the execution time. The smallest average number indicates that GA generates the required test case with the least number of generations, while the shortest average execution time suggests that it has more advantages.

The Experiments

In this Section, we will demonstrate the GA execution for automatically generating test cases for several specific types of web applications to estimate the results of this technique.

Web Applications Under Test

We implemented the algorithm on five web applications and tested it using the Genetic Algorithm (GA). These web applications and their descriptions are provided in Table 7. The sizes of the web applications range from 173 to 1,473 lines of code. Among these, the login example is particularly crucial, as it represents a critical security risk—it can potentially cause significant damage if compromised and may also be an unsecured page. Detailed descriptions and testing results for each web application are presented in the following subsection.

Table 7: Programs under Test

Web application Name	Number of Code Lines
WebGoat.NET	1473
Login information by Bryian Tan	590
Learner's Pvtltd	217
SQL injection	173

Experimental Results

This section presents the web applications under Test and the outcomes of experiments implementing the GA to evaluate these web applications.

The WebGoat.NET web application is an open-source educational platform for users to learn about prevalent web security vulnerabilities. It encompasses common security flaws commonly found in web applications and offers lessons tailored to the .NET framework. These exercises are carefully designed to teach developers about web security attacks and how to address them effectively.

Each of the 60 pages is dedicated to specific functionalities, such as login, password modification, password retrieval, user addition, and other operations. The source code for this web application can be accessed at (Alshraideh & Bottaci, 2006; GitHub, 2025).

Of particular significance within this web application is the login page, which offers two distinct login options: customer login and employee login. Both login pages feature input fields for usernames and passwords. Notably, upon code analysis, it becomes evident that there are no built-in safeguards to protect these input textboxes from potential user-based attacks.

The data type assigned to these textboxes is a string, allowing for the acceptance of various character types, including letters, numbers, and special characters such as "% ", "$ ", "# ", and "!". Figure 12 illustrates a portion of the login code, highlighting the data types assigned to both the email and password fields, followed by a code snippet containing user and password information. Verifying the authenticity of the entered email and password is performed within the if statement. Code within this if statement is executed when the user provides invalid input for email, password, or both, and it is skipped when valid inputs are provided.

```
string email = txtUserName.Text;
string pwd = txtPassword.Text;

log.Info("User " + email + " attempted to log in
with password " + pwd);

if (!dlu.IsValidCustomerLogin(email, pwd))
{
   labelError.Text = "Incorrect
username/password";
   PanelError.Visible = true;
   return;
}
```

Fig. 12: Part of the Login Code

Table 8 shows the testing results of the WebGoat.NET web application for each testing metric. You can see from the table that when the population size was 30, the average number of generations required to find the test data was the highest; conversely, the execution time decreased. Note that this results from implementing the first case of SQLI, which is 1' or '1' = '1'))/*.

Table 8: WebGoat.NET Testing Results

Population Size	Average No. of Generations	Execution Time (Seconds)
30	186.8	2.9
50	4 73.	4.1
70	54.8	4.9
90	51.2	6.2
110	39	7.2

Figures 13 and 14 illustrate the behavior of GA as the population size increases. Figure 13 indicates that the average number of generations decreases with increased population. The probability of finding optimal solutions rises as the generation size of solutions from the search domain expands, resulting in a smaller number of generations. In contrast, Figure 13 shows that the execution time increases in tandem with population growth. The average number of generations steadily decreased in Figure 14, as the probability of finding optimal values increases gradually with the rise in population size.

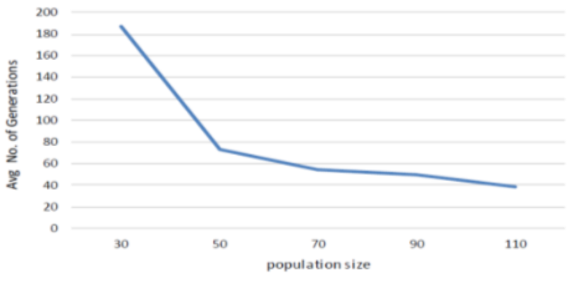


Fig. 13: Average Number of Generations for WebGoat.NET

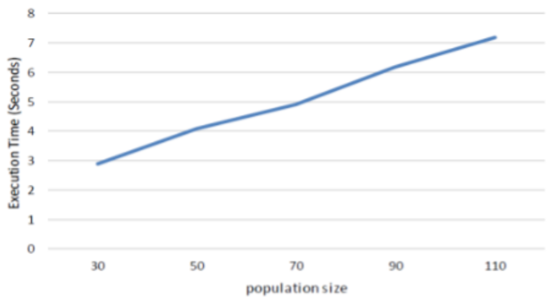


Fig. 14: Execution Time for WebGoat.NET

Login Information by Bryian Tan

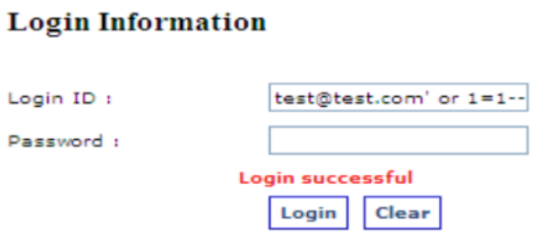
The Bryian Tan web application provides login information. This application represents a small sample code to test vulnerabilities, such as SQL Injection (SQLI) and cross-site scripting (XSS). The source code is available at CodeProject (2025).

However, this application demonstrates that vulnerabilities can occur through the Query string or Form input box.

As a query string, the web application shows UNION SQL Injection like the command:

SELECT * FROM dbo.MyComments WHERE ID = 1 --ORDER BY [Name]

After executing this command, the results show that the database holds various tables: My Comments, tbl_SQLInjection, tbl_users, and TestTable. After revealing the names of tables in the database, the attacker will retrieve data from the sysprocesses table, for example. Then, updating the table, deleting table data, truncating the table, or dropping the table may occur.


Login Information

Login ID : test@test.com' or 1=1--

Password :

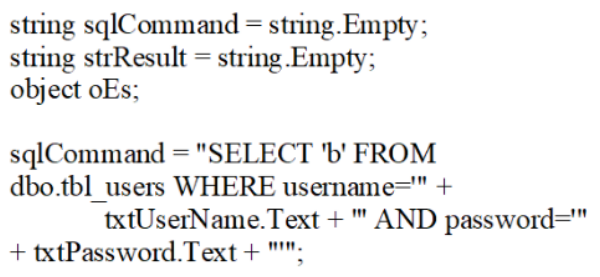
Login successful

Fig. 15: Login screen example of SQLI

On the other hand, as Forms input, illegal users can bypass the login page by adding ' or 1=1 -- or ') or 1=1--

to the login ID and putting any value in the password field. Figure 15, which illustrates an example that was previously explained. This figure shows that the input text: ‘ or 1= 1 – makes a successful login to the site.

Figure 16 shows part of the code that illustrates the types of text boxes and the complete SQL command line to be executed after the User provides input.



```
string sqlCommand = string.Empty;
string strResult = string.Empty;
object oEs;

sqlCommand = "SELECT 'b' FROM
dbo.tbl_users WHERE username='" +
            txtUserName.Text + "' AND password='"
+ txtPassword.Text + "'";
```

Fig. 16: Code snippet showing the login authentication logic for the Bryian Tan web application

Table 9: WebGoat.NET Testing Results

Population Size	Average No. of Generations	Execution Time (Seconds)
30	527.6	2
50	321.6	2.9
70	184.8	4.2
90	155.8	6.3
110	111.6	9 6.

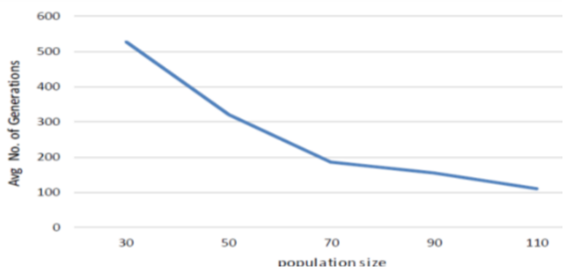


Fig. 17: Average Number of Generations for Login Information by Bryian Tan web Application

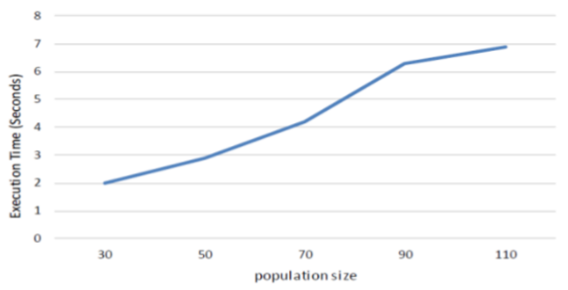


Fig. 18: Execution Time for Login information by Bryian Tan web Application

The results of running the algorithm on this program are shown in Table 9 and Figures 17 and 18. Figure 17 shows the inverse relationship between the average number of generations and population size. As the population size increases, the average generation decreases. Unfortunately, this is reflected negatively in the time the algorithm requires to find what it looks for in many generations, illustrated in Figure 18.

This results from implementing the case: 1' or '1' = '1'))/*.

However, Learner’s Pvtltd and SQLInjection web applications show the same behavior as the previous two examples of web applications.

Discussion

The results from our GA-based SQL Injection detection experiments demonstrate a significant improvement in efficiency, adaptability, and detection accuracy over traditional testing methods. The GA achieved a consistent reduction in the average number of generations required as population size increased, albeit with a corresponding rise in execution time.

The use of the Character Distance fitness function proved effective in guiding the search process, enabling the GA to evolve payloads that closely mimic actual attack vectors. Compared to manual penetration testing and static analysis tools, our GA-based approach significantly reduces false positives and adapts well to diverse input patterns.

The results also confirmed that white-box testing, enhanced with static analysis, was more effective in detecting embedded vulnerabilities. In contrast, black-box testing demonstrated practical utility when source code was unavailable. The integration of both paradigms provides a comprehensive strategy applicable to various real-world scenarios.

Additionally, the statistical analysis using ANOVA confirmed the significant differences in the detection performance of various SQLI clauses. The evaluation across multiple applications revealed consistent patterns that support the generalizability of the approach.

Nevertheless, the method's effectiveness is influenced by the proper tuning of genetic parameters (e.g., mutation rate, selection pressure). Moreover, execution time may increase with higher population sizes, warranting a balance between detection depth and resource efficiency.

Future work will investigate the integration of machine learning classifiers, such as Decision Trees or Deep Neural Networks, to automate the classification of SQLI commands and further reduce detection time and error rates. We also plan to explore advanced evolutionary strategies and multi-objective optimization to enhance both performance and interpretability.

Conclusion

This study addressed the critical challenge of detecting SQL Injection (SQLI) vulnerabilities in web applications by employing a Genetic Algorithm (GA)-

based test case generation approach. Given the increasing reliance on web applications for handling sensitive user data, ensuring their security is paramount. SQL injections remain among the most prevalent and damaging security threats, allowing attackers to manipulate database queries and gain unauthorized access to confidential data.

Our proposed approach leverages GA to generate optimized test cases that effectively identify SQLI vulnerabilities while minimizing the required test cases. This optimization reduces computational overhead and enhances the efficiency of the testing process, contributing to the development of more secure web applications. Automated test case generation also lowers maintenance and development costs, making it a practical and scalable solution for addressing vulnerability detection.

The findings of this study highlight the potential of evolutionary algorithms in enhancing the security of web applications. However, further improvements can be achieved by integrating additional machine-learning techniques. Future research will investigate the application of decision trees as a classifier to categorize SQL commands, aiming to enhance detection accuracy and expedite the identification of SQL Injection (SQLI) vulnerabilities. By refining the vulnerability detection process, this research contributes to the ongoing efforts to fortify web applications against cyber threats.

While this study focuses on SQLI detection using Genetic Algorithms (GA) with ANOVA-based evaluation, future research will expand the assessment by incorporating additional performance metrics. Specifically, we plan to analyze the false positive rate (FPR) and false negative rate (FNR) to quantify the accuracy of our detection approach. Additionally, we will compare execution times between GA-based detection and conventional SQLI detection methods to assess computational efficiency. Moreover, we aim to explore the integration of machine learning classifiers, such as decision trees or deep learning models, to improve SQL command classification and enhance detection precision. These extensions will further validate our approach and contribute to the development of more effective web application security mechanisms.

Acknowledgment

The Authors sincerely thank the Editor-in-Chief, Associate Editor, and anonymous reviewers for their invaluable feedback and constructive criticism. Their meticulous reviews and insightful comments have significantly enhanced the clarity, rigor, and quality of this research. The expertise and dedication demonstrated throughout the review process were instrumental in refining the manuscript and strengthening its scientific contributions.

Funding Information

The author did not receive support from any organization for the submitted work.

Author's Contributions

Ghosoun Al Hindi: Led the conceptualization and design of the study, implemented the genetic algorithm for test case generation, and drafted the initial manuscript.

Mohammad Alshraideh: Supervised the research, refined the methodology, integrating white-box and black-box testing approaches, and critically reviewed the manuscript for scientific accuracy and clarity.

Abdelrahman H. Hussein: Contributed to the development of the static analysis component within the white-box testing framework, assisted with system implementation, and supported the visualization and interpretation of results.

Lubna Fayez Eliyan: Contributed to optimizing the genetic algorithm and significantly improved the experimental analysis and discussion sections.

Arafat Al-Dhaqm: Provided key theoretical insights into web application security, enhanced the interpretation of testing results, and contributed to refining the presentation of findings. All authors reviewed and approved the final version of the manuscript. Top of Form

Ethics

This study was conducted by all relevant ethical guidelines and standards governing research. No human subjects were directly involved in the study, and no personally identifiable information was collected. The authors affirm that all analyses, interpretations, and conclusions presented herein reflect impartial scientific inquiry and adhere strictly to principles of integrity, transparency, and academic honesty.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

Alazab, M. (2021). Hybrid AI Models for Cybersecurity. *Machine Learning and Data Analytics for Predictive Modeling*, *88*, 1–20. https://doi.org/10.1007/978-3-030-71965-8_1

Allawi, H. M., Al Manaseer, W., & Al Shraideh, M. (2020). A greedy particle swarm optimization (GPSO) algorithm for testing real-world smart card applications. *International Journal on Software Tools for Technology Transfer*, *22*(2), 183–194. https://doi.org/10.1007/s10009-018-00506-y

Al-Shaikh, A., Mahafzah, B. A., & Alshraideh, M. (2019). Metaheuristic approach using grey wolf optimizer for finding strongly connected components in digraphs. *J Theor Appl Inf Technol*, *97*(16), 4439–4452.

Alshraide, M. (2008). A Complete Automation of Unit Testing for JavaScript Programs. *Journal of Computer Science*, *4*(12), 1012–1019. https://doi.org/10.3844/jcssp.2008.1012.1019

Alshraideh, M., & Bottaci, L. (2006). Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, *16*(3), 175–203. https://doi.org/10.1002/stvr.354

Alshraideh, M., Bottaci, L., & Mahafzah, B. A. (2010). Using program data-state scarcity to guide automatic test data generation. *Software Quality Journal*, *18*(1), 109–144. https://doi.org/10.1007/s11219-009-9083-x

Alshraideh, M., Mahafzah, B. A., & Al-Sharaeh, S. (2011). A Multiple-Population Genetic Algorithm for Branch Coverage Test Data Generation. *Software Quality Journal*, *19*(3), 489–513. https://doi.org/10.1007/s11219-010-9117-4

Alsorory, H. S., & Alshraideh, M. (2024). Boosting Software Fault Prediction: Addressing Class Imbalance With Enhanced Ensemble Learning. *Applied Computational Intelligence and Soft Computing*, *2024*(1), 2959582. https://doi.org/10.1155/2024/2959582

Avancini, A., & Ceccato, M. (2010). Towards security testing with taint analysis and genetic algorithms. *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, 65–71. https://doi.org/10.1145/1809100.1809110

Avancini, A., & Ceccato, M. (2011). Security Testing of Web Applications: A Search-Based Approach for Cross-Site Scripting Vulnerabilities. *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, 85–94. https://doi.org/10.1109/scam.2011.7

Avdeenko, T., & Serdyukov, K. (2023). Modified Evolutionary Test Data Generation Algorithm Based on Dynamic Change in Fitness Function Weights. *15th International Conference "Intelligent Systems*, 23. https://doi.org/10.3390/engproc2023033023

Baria, J., & Gandhi, M. (2013). SQL Injection Attacks in Web Application. *International Journal of Soft Computing and Engineering*, *2*(6), 189–191.

Bataineh, A., Alshraideh, M., Hudaib, A., & Wedyan, F. (2022). The Automation of Java Smart Card Using Negative Testing. *International Journal of Computing and Digital Systems*, *12*(3), 707–714. https://doi.org/10.12785/ijcds/120157

Bisht, P. (2018). AI-Enhanced Fuzzing for Web Security. *Cybersecurity and Secure Information Systems*, *174*, 45–62. https://doi.org/10.1007/978-3-319-99073-6_3

Dukes, L., Yuan, X., & Akowuah, F. (2013). A case study on web application security testing with tools and manual testing. *2013 Proceedings of IEEE Southeastcon*, 1–6. https://doi.org/10.1109/secon.2013.6567420

Halfond, W. G. J., & Orso, A. (2006). Preventing SQL injection attacks using AMNESIA. *Proceedings of the 28th International Conference on Software Engineering*, 795–798. https://doi.org/10.1145/1134285.1134416

Hasan, M., Balbahaith, Z., & Tarique, M. (2019). Detection of SQL Injection Attacks: A Machine Learning Approach. *2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*. 2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA), Ras Al Khaimah, United Arab Emirates. https://doi.org/10.1109/icecta48151.2019.8959617

Kieyzun, A., Guo, P. J., Jayaraman, K., & Ernst, M. D. (2009). Automatic creation of SQL Injection and cross-site scripting attacks. *2009 IEEE 31st International Conference on Software Engineering*, 199–209. https://doi.org/10.1109/icse.2009.5070521

Korejo, I., Yang, S., & Li, C. (2009). A comparative study of adaptive mutation operators for metaheuristics. *At the VIII Metaheuristics International Conference*, 13–19.

Liu, D., & Fan, S. (2014). A Modified Decision Tree Algorithm Based on Genetic Algorithm for Mobile User Classification Problem. *The Scientific World Journal*, *2014*(1), 468324. https://doi.org/10.1155/2014/468324

Medeiros, I., Neves, N., & Correia, M. (2016). Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining. *IEEE Transactions on Reliability*, *65*(1), 54–69. https://doi.org/10.1109/tr.2015.2457411

Miller, B. L., & Goldberg, D. E. (1995). Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, *9*(3), 193–212.

Noraini, M. R., & Geraghty, J. (2011). Genetic algorithm performance with different selection strategies in solving TSP. *Proceedings of the World Congress on Engineering (WCE 2011*, 1–6.

Razali, N., & Geraghty, J. (2011). Genetic algorithm performance with different selection strategies in solving TSP. *World Congress on Engineering*, 1134–1.

Sharma, C., Sabharwal, S., & Sibal, R. (2014). A survey on software testing techniques using genetic algorithms. *Software Engineering*, *5*(1), 13. https://doi.org/10.48550/arXiv.1411.1154

Spears, W. M., & Anand, V. (1991). A study of crossover operators in genetic programming. *Methodologies for Intelligent Systems*, *542*, 409–418. https://doi.org/10.1007/3-540-54563-8_104

Su, T. (2021). Improving SQL Injection Detection via Code Analysis. *Machine Learning and Data Analytics for Predictive Modeling*, *88*, 21–40. https://doi.org/10.1007/978-3-030-71965-8_2

Thakore, D., & Kamble , T. (2012). Application of Genetic Algorithm in Software Security. *International Journal of Recent Technology and Engineering (IJRTE*, *1*(5), 28–24.

Thomé, J., Gorla, A., & Zeller, A. (2014). Search-based security testing of web applications. *Proceedings of the 7th International Workshop on Search-Based Software Testing*, 5–14. https://doi.org/10.1145/2593833.2593835

Wang, X. (2019). Reinforcement Learning-Based Web Security. *IEEE Access*, *7*, 160844–160853.

Zhang, Y. (2022). Blockchain for SQLI Prevention. *IEEE Internet of Things Journal*, *9*(18), 17852–17859. https://doi.org/10.1109/JIOT.2022.3098803