future internet

Article

GenSQLi: A Generative Artificial Intelligence Framework for Automatically Securing Web Application Firewalls Against Structured Query Language Injection Attacks

Vahid Babaey and Arun Ravindran *

Department of Electrical and Computer Engineering, University of North Carolina at Charlotte, Charlotte, NC 28223, USA; vbabaey@charlotte.edu
* Correspondence: arun.ravindran@charlotte.edu

Abstract: The widespread adoption of web services has heightened exposure to cybersecurity threats, particularly SQL injection (SQLi) attacks that target the database layers of web applications. Traditional Web Application Firewalls (WAFs) often fail to keep pace with evolving attack techniques, necessitating adaptive defense mechanisms. This paper introduces a novel generative AI framework designed to enhance SQLi mitigation by leveraging Large Language Models (LLMs). The framework achieves two primary objectives: (1) generating diverse and validated SQLi payloads using in-context learning, thereby minimizing hallucinations, and (2) automating defense mechanisms by testing these payloads against a vulnerable web application secured by a WAF, classifying bypassing attacks, and constructing effective WAF security rules through generative AI techniques. Experimental results using the GPT-4o LLM demonstrate the framework's efficacy: 514 new SQLi payloads were generated, 92.5% of which were validated against a MySQL database and 89% of which successfully bypassed the ModSecurity WAF equipped with the latest OWASP Core Rule Set. By applying our automated rule-generation methodology, 99% of previously successful attacks were effectively blocked with only 23 new security rules. In contrast, Google Gemini-Pro achieved a lower bypass rate of 56.6%, underscoring performance variability across LLMs. Future work could extend the proposed framework to autonomously defend against other web attacks, including Cross-Site Scripting (XSS), session hijacking, and specific Distributed Denial-of-Service (DDoS) attacks.

Keywords: generative AI; adaptive defense mechanism; vulnerability; LLM; SQL injection; WAF; ModSecurity WAF; AWS WAF; database; cybersecurity

check for updates

Academic Editors: Raha Moraffah and Li Yang

Received: 5 December 2024
Revised: 20 December 2024
Accepted: 30 December 2024
Published: 31 December 2024

Citation: Babaey, V.; Ravindran, A. GenSQLi: A Generative Artificial Intelligence Framework for Automatically Securing Web Application Firewalls Against Structured Query Language Injection Attacks. *Future Internet* **2025**, *17*, 8. https://doi.org/10.3390/fi17010008

1. Introduction

The rapid proliferation of web services has revolutionized the way organizations deliver functionality and data to users. From e-commerce platforms to online banking systems, web services form the backbone of modern digital infrastructure. However, the growing reliance on web applications has also led to an increase in their exploitation as attack vectors, exposing them to various cybersecurity threats.

Among the most prevalent and damaging forms of web attacks are SQL injection (SQLi) attacks [1,2], which target the database layer of web applications. By exploiting vulnerabilities in query handling, attackers can manipulate backend databases, retrieve sensitive information, or even disrupt services. The significance of SQLi attacks cannot be overstated; they remain a top concern for organizations due to their frequency and the severe impact they can have on data confidentiality, integrity, and availability [3].

To defend against SQLi attacks, Web Application Firewalls (WAFs) are widely deployed as a first line of defense [4]. WAFs detect and block malicious payloads by analyzing incoming HTTP traffic for known attack patterns. However, traditional WAFs often struggle to keep pace with the constantly evolving landscape of attack techniques. Sophisticated attackers can craft novel SQLi payloads that bypass even state-of-the-art WAF rules, highlighting the need for adaptive and proactive defense mechanisms.

Generative AI offers a promising avenue for addressing this challenge [5]. Generative AI Large Language Models (LLMs), such as GPT-4o from OpenAI [6], Gemini from Google [7], Llama 3.2 from Meta [8], and Claude Sonnet 3.5 from Anthropic [9], represent the cutting edge of artificial intelligence, capable of producing human-like text, generating code, and even simulating complex attack vectors. By leveraging large-scale neural networks and in-context learning, generative AI systems can synthesize novel outputs based on carefully curated input examples, making them highly adaptable across a range of domains.

The use cases for generative AI are diverse, spanning creative content generation, natural language processing, and advanced problem-solving tasks. In the realm of cybersecurity, generative AI has shown potential for both offensive and defensive applications [10]. It can simulate attack scenarios to stress-test defenses, assist in vulnerability detection, and automate the creation of countermeasures. These capabilities position generative AI as a transformative tool for addressing emerging security threats.

In this paper, we propose a novel generative AI framework for proactively and automatically securing WAFs against SQL injection attacks. Our framework leverages the power of generative AI to achieve two primary objectives:

1. Generation of Novel SQL Injection Attacks: Using in-context learning, our approach uses LLMs to generate diverse SQLi payloads based on curated examples. These attacks are validated against actual database implementations to ensure authenticity and guard against hallucinated payloads.
2. Automated Defense Mechanisms: By testing the generated attacks against a vulnerable web application secured by a Web Application Firewall (WAF), we identify payloads that successfully bypass state-of-the-art WAF rules. These bypassing attacks are classified using machine-learning algorithms, and new WAF rules are automatically generated using LLMs. The updated rules are then incorporated into the WAF, and the attack scenarios are re-executed to validate their effectiveness.

Through this iterative process, the framework demonstrates a unique capability to improve WAF security dynamically. By using generative AI for both attack generation and defense optimization, the framework provides a **proactive and automated approach** to addressing SQL injection vulnerabilities. This contrasts with the existing approach, where cybersecurity personnel manually update rules reactively based on observed threats. Additionally, we observe that existing SQLi attack generation approaches in the literature (see Section 2) address attack generation and mitigation techniques separately. In contrast, GenSQLi leverages generative AI to automatically generate SQLi attacks and the corresponding WAF rules for their mitigation.

Our experimental results demonstrate the efficacy of this approach. Using GPT-4o LLM, we successfully generated 514 new SQLi payloads, 92.5% of which were validated against a MySQL database. Of these attacks, 89% bypassed state-of-the-art WAFs such as ModSecurity [11] equipped with the latest version of the OWASP CoreRuleSet [12]. By applying our machine-learning classification and GPT-4o LLM rule-generation methodology, we were able to develop and validate security rules that effectively blocked 99% of all previously successful attacks with only 23 new rules. To facilitate the replication of our work, we have provided detailed appendices that include the prompts used. We also present results with the Google Gemini-Pro LLM, where a lower percentage of attacks (56.5%) were successful. Our experiments were carried out with the MySQL database. We have also validated GenSQLi with the PostgreSQL database and AWS WAF. We plan to open-source the GenSQLi framework so that the capabilities of generative AI can be harnessed by cybersecurity practitioners to safeguard against evolving SQLi attacks.

The paper makes the following contributions—

1. Novel Framework for SQLi Mitigation: Development of a generative AI-based framework that generates novel SQL injection attacks and automatically evolves WAF defenses.
2. In-Context Learning for Attack Generation: Utilization of generative AI models with curated examples to produce diverse and validated SQLi payloads, minimizing hallucinations.
3. Automated WAF Rule Optimization: Integration of machine-learning and generative AI techniques to classify bypassing attacks and generate effective WAF rules.
4. Demonstrated Efficacy: Experimental validation showing improved WAF security by blocking 99% of previously bypassing SQLi attacks with only a limited number of new rules.

The remainder of the paper is structured as follows: Section 2 reviews the existing literature on the application of generative AI in cybersecurity. Section 3 provides a concise background on SQL injection (SQLi) attacks and generative AI. Section 4 outlines the proposed framework in detail. Section 5 presents the evaluation results. Section 6 discusses the broader implications of the proposed approach. Finally, Section 7 concludes the paper by summarizing the findings and suggesting directions for future research.

2. Related Work

Web Application Firewalls (WAFs) are one of the most widely used defense systems against SQL injection (SQLi) attacks (See Section 3 for a brief background on WAFs). WAFs utilize two common strategies to defend against SQLi attacks—

1. The rule-based approach detects attacks based on known attack signatures. Rules are typically written as a sequence of regular expressions designed to detect many forms of attack. Rule-based methods enjoy the advantages of being deterministic and transparent, having low false positive rates, are quickly deployable, being compliance friendly, and are resource efficient. However, they are limited to known patterns, require regular updates, are unable to adapt to slight variations in attack patterns, and have scalability issues.
2. The machine-learning-based approach detects attacks by learning from data to identify anomalous or malicious patterns in SQL queries. These methods typically involve training models on historical data, including both benign and malicious queries, to classify or detect suspicious behavior. Machine-learning-based methods enjoy the advantages of being adaptable to novel or obfuscated attack patterns, detecting deviations from normal behavior rather than relying on static signatures, improving continuously with more data, and reducing the need for explicit rule maintenance. However, they have drawbacks such as high false positive rates in poorly trained models, being resource-intensive, having complex and non-transparent decision-making processes, depending heavily on the availability and quality of labeled data, and requiring significant time and effort for initial deployment and fine-tuning.

El Ghawazi et al. [13] and Applebaum et al. [14] present a review of literature that employs machine-learning and deep-learning techniques to detect SQLi attacks.

Since our work focuses on enhancing WAF robustness through a generative AI-driven framework capable of generating novel SQLi attacks and updating existing WAF rules, we review related work in the areas of SQLi attack generation, rule adaptation, and WAF robustness improvement.

Qu et al. [15] propose a technique that transforms an original SQLi payload into generated SQLi payloads that can bypass WAFs through the use of Context-Free Grammar (CFG) for generation and a Monte Carlo tree search for selection.

Zhang et al. [16] address the challenge of unevenly distributed effective attack payloads by leveraging Context-Free Grammar (CFG) to describe attack patterns and decomposing payloads into tokens. Feature vectors are extracted using label cleaning, and cosine distance metrics are employed to measure similarity between payloads. By adaptively selecting and executing promising payloads based on these metrics, they reduce unsuccessful attack trials.

Demetrio et al. [17] use a series of mutation operators, such as case swapping, whitespace substitution, and comment injection, to iteratively generate new payloads, starting with a malicious payload that is detected by the WAF. These mutation operators preserve the semantic integrity of the payloads, ensuring they remain functionally identical from the adversary's perspective. However, these transformations gradually reduce the confidence of the WAF's classification algorithm. This process continues until the payload's classification falls below the rejection threshold, resulting in successful SQLi attacks.

Appelt et al. [18] propose an automated approach to generate SQL injection attacks capable of bypassing WAFs. They utilize a genetic algorithm to evolve effective attack patterns. By training a decision tree model on a dataset of labeled attacks, the system identifies successful patterns and generates new attacks based on them. This iterative process allows for the discovery of increasingly sophisticated attacks that can evade WAF defenses.

Hemmati et al. [19] utilize deep reinforcement learning algorithms, specifically DQN and PPO, to iteratively apply a set of modification operators to an initial malicious SQLi attack. The goal is to generate new SQLi attacks that successfully evade the WAF while preserving the malicious intent of the original payload.

In contrast to the aforementioned works, our approach leverages generative AI trained on SQLi attack data to generate novel SQLi attacks capable of bypassing WAFs. Additionally, we close the loop by using generative AI to generate WAF security rules that can effectively detect these attacks. A key advantage of generative AI over techniques such as context-free grammars is the reduced technical expertise required to enhance WAF robustness. Furthermore, methods like context-free grammars, genetic algorithms, and reinforcement learning could be employed to generate in-context data for training generative AI models. Exploring this integration is left for future work.

3. Background

In this section, we provide a brief background on SQL injection attacks, web application firewalls, and generative AI.

3.1. SQL Injection Attacks

SQL injection (SQLi) attacks are a type of security vulnerability that occurs when an attacker manipulates SQL queries executed by a web application [2]. These attacks exploit improper handling of user inputs in SQL statements, allowing attackers to access, modify, or delete sensitive database information. Common attack vectors include login forms, search fields, and URL parameters. Figure 1 illustrates the operational mechanism of a SQL injection attack.

As a simple illustrative example, consider a web application with a login form where users enter their username and password. The backend code might construct an SQL query like this:

```
SELECT * FROM users WHERE username = 'user_input' AND password = '
    password_input';
```

If the application does not properly sanitize user inputs, an attacker could input the following in the username field:

```
' OR '1' = '1
```

This modifies the query to:

```
SELECT * FROM users WHERE username = '' OR '1' = '1' -- ' AND password = '';
```

The - - marks the rest of the query as a comment, so everything after it is ignored. Since '1' = '1' is always true, the query retrieves all rows from the users table.

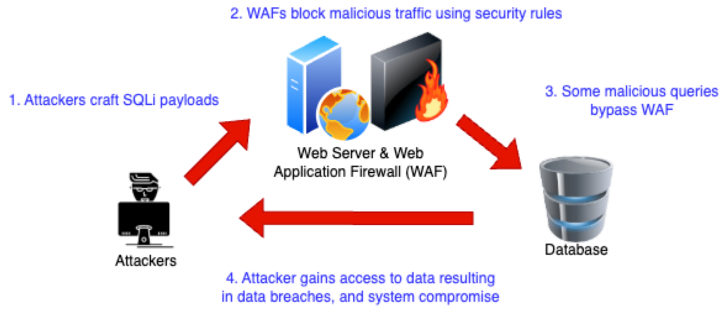

Figure 1. Anatomy of an SQL injection attack. Existing methods update WAF security rules manually in response to attacks. GenSQLi automates the generation of SQLi attacks and corresponding WAF security rules, enhancing security systems against SQLi threats.

SQLi attacks today are far more complex. SQLi attacks can be broadly categorized based on the techniques used [2]. In this paper, we focus on the following types of SQLi attacks: **Conditional attacks** use payloads that manipulate query conditions to infer database information by observing changes in the application's behavior. **Error-based attacks** exploit error messages returned by the database to extract details about its structure or configuration. **Tautology attacks** involve injecting always-true conditions (e.g., 1 = 1) into queries to bypass authentication or retrieve unintended data. **Tautology and conditional attacks** combine these techniques, using tautologies to identify vulnerable points and conditions to refine the extraction of specific information. **Exist-based attacks** rely on verifying the existence of certain data or tables in the database through crafted payloads, often by leveraging subtle response differences.

In addition to the above attacks, other attack types include the following—**Union-based attacks** use the UNION operator to merge legitimate query results with attacker-crafted data, allowing data retrieval from other tables. **Inferential attacks** (Blind SQLi) rely on deducing database information indirectly through application responses or execution delays, without receiving direct query results. **Time-based attacks**, a subset of Blind SQLi, inject queries that trigger delays to infer true/false conditions. **Out-of-band attacks** exploit alternative communication channels, such as DNS or HTTP, for data exfiltration, often useful when traditional channels are unavailable. **Stored attacks** involve inserting malicious payloads into a database, which executes later when accessed, making it particularly dangerous in shared or dynamic environments. **Second-order attacks** embed payloads that are stored and later executed in a different context within the application, often bypassing traditional input sanitization.

Obfuscation in SQL Injection Attacks

Obfuscation refers to techniques used by attackers to disguise or modify malicious SQLi payloads in order to evade detection by security mechanisms such as Web Application Firewalls (WAFs), Intrusion Detection Systems (IDSs), or input validation filters [20].

By altering the appearance of SQLi payloads without changing their functionality, attackers aim to bypass pattern-based defenses or heuristic checks.

Some common obfuscation techniques include the following—**Encoding and escaping** involves using character encoding schemes like URL encoding, hexadecimal, or Unicode to mask malicious inputs. **Case alteration** changes the capitalization of SQL keywords to bypass case-sensitive filters. **Comment injection** adds inline or end-of-line comments to split payloads and confuse signature-based detection. **White space manipulation** replaces spaces with tabs, newlines, or comment breaks to modify the query structure. **String concatenation** breaks strings into smaller parts and recombines them during query execution. **Keyword substitution** replaces standard SQL keywords with database-specific alternatives or synonyms. **Hexadecimal or binary representations** encode strings or commands in hexadecimal or binary formats. **Using false logic or complex queries** introduces redundant logic or unnecessary operations to make the query appear legitimate. **Exploiting database-specific functions** leverages database-specific syntax, features, or quirks to evade detection.**Dynamic query construction** employs nested queries or dynamic expressions to obscure the true intent of the malicious payload.

A comprehensive review of SQL injection attacks can be found in [21].

3.2. *Web Application Firewalls (WAFs)*

A Web Application Firewall (WAF) is a security mechanism designed to protect web applications by monitoring, filtering, and blocking malicious HTTP/S traffic. Unlike traditional firewalls that safeguard networks, WAFs focus on protecting the application layer (Layer 7 of the OSI model) [22]. They are commonly used to defend against web-based attacks such as SQL injection (SQLi), Cross-Site Scripting (XSS), and Distributed Denial-of-Service (DDoS) attacks. WAFs match incoming traffic against a database of known attack patterns. Some notable WAFs include AWS WAF, Microsoft Azure WAF, CloudFlare WAF, Imperva, Fortinet , and ModSecurity. Table 1 compares the open source ModSecurity WAF and the commercial AWS and Cloudflare WAF along multiple features such as types of attack detection mechanisms supported, ease of customization, and the manner in which threat intelligence is incorporated.

Table 1. Comparison of SQLi detection mechanisms in WAFs.

Feature	ModSecurity	AWS WAF	Cloudflare WAF
Rule-based	OWASP CRS + Custom Rules	Managed Rules + Custom Rules	Managed Rules
Machine Learning	Optional (via integrations)	Integrated with AWS ecosystem	Integrated, advanced capabilities
Behavioral Analysis	Limited (custom scripts required)	Available (via AWS services)	Advanced
Ease of Customization	High (fully open-source)	Medium	Medium
Threat Intelligence	None (manual updates needed)	AWS threat feeds	Cloudflare threat feeds

In this research, we use ModSecurity [11], since it is an open-source WAF and used in conjunction with Apache and Nginx servers. It uses rules to identify and mitigate malicious HTTP/S requests. These rules form the backbone of ModSecurity's ability to protect web applications by defining patterns, behaviors, and conditions that determine what traffic should be allowed, denied, or flagged for review. Each ModSecurity rule is written in its SecRule directive and typically includes the following:

1. Input to Monitor—specifies which part of the request or response to inspect (e.g., headers, parameters, or body).
2. Operator—defines the condition to match (e.g., checking for specific patterns or keywords).
3. Action—Specifies what to do if the rule is triggered (e.g., deny, log, or redirect).

The OWASP CRS [12] is a set of generic attack detection rules for use with ModSecurity. Commercial WAFs employ similar proprietary rule sets. The security rules are manually updated by cybersecurity personnel in response to observed threats. In this work, we demonstrate how generative AI can be used to **proactively and automatically** enhance the robustness of ModSecurity. It should be noted that the proposed GenSQLi framework is generic and can be applied to strengthen any WAF against SQLi attacks.

3.3. Generative AI

Generative AI refers to a branch of artificial intelligence that creates new data or content, such as text, images, audio, or code, by learning patterns from existing datasets. Central to this field are Large Language Models (LLMs), which are deep-learning architectures trained on massive amounts of text data to generate coherent and contextually appropriate language [23,24]. These models, such as GPT (Generative Pre-trained Transformer), leverage advancements in machine learning and deep learning, particularly transformer architectures, to capture complex relationships within data. Deep learning enables these models to process and understand vast datasets through multi-layered neural networks, where each layer extracts increasingly abstract features. The complexity of LLMs arises from their scale, with billions (or even trillions) of parameters, requiring significant computational resources for training.

The applications of generative AI are diverse, including natural language processing tasks like chatbots, translation, and summarization, as well as creative fields such as content generation, code completion, and synthetic data creation for research. As of the time of writing of this paper, GPT-4o from OpenAI [6], Gemini-Pro from Google [7], Llama 3.2 from Meta [8], and Claude Sonnet 3.5 from Anthropic [9] represent the cutting edge of LLMs. For a modest price, these LLMs can be accessed programmatically via REST APIs without requiring direct deployment or infrastructure management. Additionally, Llama models from Meta are open source and can be hosted locally.

Customizing Large Language Models (LLMs) involves adapting pre-trained models to specific tasks, domains, or user requirements. Below are common techniques used for this purpose [25]:

1. Fine-Tuning: The model is retrained on a smaller, task-specific dataset while retaining knowledge from its pre-training phase.
2. Prompt Engineering: Carefully crafting input prompts to guide the model's behavior without altering its parameters.
3. In-Context Learning: Providing examples of the desired task within the input prompt so the model can infer patterns and mimic behavior.
4. Reinforcement Learning with Human Feedback (RLHF): Uses reinforcement learning to align the model's outputs with human preferences

Additionally, LLMs can be augmented with additional data sources (Retrieval Augmented Generation) [26], call external APIs or invoke specific tools, and be augmented with persistent memory to store and recall context beyond a single prompt-response cycle [27]. Furthermore, with agentic LLMs, multiple LLM-based agents can act collaboratively to reason, plan, and execute actions in a dynamic or goal-oriented manner [28].

4. GenSQLi Framework

In this section, we present the details of GenSQLi, a general and extendable framework for robustifying WAFs against SQLi attacks. To the best of our knowledge, GenSQLi represents the first reported application of generative AI to both generate and defend against SQLi attacks. The proposed framework is generally independent of any particular LLM, database, application, or WAF; however, it must be suitably adapted to align with the specific characteristics and requirements of each particular instance in which it is implemented.

4.1. Architecture

Figure 2 illustrates the architecture of GenSQLi. The process begins with generative Large Language Models (LLMs) being prompted using carefully curated in-context examples of SQL injection (SQLi) attacks to generate potential SQLi payloads. Since LLMs may produce inaccurate or irrelevant outputs (referred to as "hallucinations"), generated SQLi samples are validated against a target database to confirm their effectiveness as legitimate SQLi attacks.

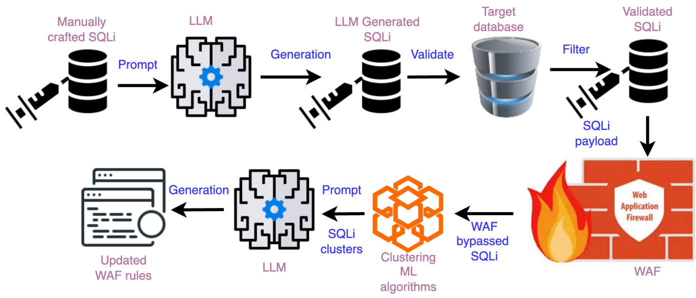

Figure 2. GenSQLi framework architecture.

Validated SQLi samples are then evaluated within a test setup comprising a vulnerable application and a Web Application Firewall (WAF). SQLi attacks that successfully bypass the WAF are recorded and analyzed using machine-learning clustering algorithms to group similar attack types. LLMs are subsequently tasked with generating WAF rules tailored to mitigate these identified attack patterns.

To optimize the effectiveness of the WAF rules, a Reinforcement Learning with Human Feedback (RLHF) methodology is applied. This iterative refinement process continues until all SQLi attacks are successfully mitigated. The entire workflow is repeatable to iteratively develop a robust set of security rules.

In the remainder of this section, we describe each of these steps in detail.

4.2. SQLi Generation

The in-context learning examples used to prompt the Large Language Models (LLMs) were manually crafted and validated against a vulnerable application connected to the target database. These examples served as a foundation for the LLMs to generate obfuscated SQL injection (SQLi) samples. The examples include a diverse range of SQLi types and consist only of validated payloads that successfully bypass the target WAF in the specified database environment. They prioritize modern attack patterns for real-world relevance while balancing complexity and clarity to effectively guide generative models. The prompts provided clear instructions, guiding the LLMs to systematically apply various obfuscation techniques to create complex and diverse SQLi payloads while ensuring compatibility with the target database syntax.

The prompt used in SQLi generation is presented in detail in Appendix A. The prompt construction includes four key elements: (1) Instruction for in-context learning, where the

LLM is guided to generate creative yet logically consistent variations using systematic obfuscation techniques; (2) Examples of obfuscation methods, providing clear explanations and examples to inform the generation process; (3) Database context, ensuring the generated SQLi samples are compatible with the target database schema for practical applicability; and (4) Generation task, which mandates the systematic application of relevant obfuscation techniques, producing functional, syntactically valid SQL samples. The prompt emphasizes relevance, quality, and diversity, ensuring the output is both comprehensive and suitable for real-world testing.

4.3. SQLi Validation

The validation process involved a predefined test database containing a users table with three columns: id, username, and password. The SQLi attacks used as in-context learning examples were first tested in a vulnerable application configured with the target database to ensure their effectiveness. These examples were then evaluated against the target WAF to assess their ability to bypass defenses. Similarly, the SQLi attacks generated by LLMs were validated within the same vulnerable application setup to confirm their functionality. This step ensured that all generated SQLi queries adhered to target database syntax and were successfully executed as intended within the database context. The validated SQLi attacks were subsequently used in further evaluations of WAF effectiveness and rule refinement.

4.4. SQLi Clustering

To cluster and analyze SQLi attacks, two machine-learning algorithms were used: TF-IDF with Hierarchical Agglomerative Clustering (HAC) and SequenceMatcher with DBSCAN. Each method captures unique aspects of SQLi attack patterns. **TF-IDF + HAC:** TF-IDF evaluates term importance in SQLi queries by emphasizing unique terms and de-emphasizing common ones [29]. Queries are represented as TF-IDF vectors, and HAC clusters them based on vector similarity [30]. For example, SELECT 1 = 1 and SELECT 1 = 1 OR 0 $\times$ 61 = 0 $\times$ 61 are grouped due to shared terms. **SequenceMatcher + DBSCAN:** SequenceMatcher calculates character-level similarity scores between queries, while DBSCAN clusters queries with high similarity density [31]. For instance, SELECT 1 = 1 and SELECT 0 $\times$ 61 = 0 $\times$ 61 are grouped due to their structural similarity.

Advantages of the Selected Clustering Approaches

The combination of TF-IDF + HAC and SequenceMatcher + DBSCAN are highly effective for clustering SQLi payloads due to their complementary strengths in handling the complexities of SQLi patterns. TF-IDF emphasizes rare, attack-specific terms while reducing the weight of common ones, ensuring meaningful feature extraction, and HAC effectively handles irregular, hierarchical relationships without assuming cluster shapes or predefined counts. Similarly, SequenceMatcher captures string-based similarities, and DBSCAN excels in managing overlapping clusters, tolerating noise, and dynamically identifying the number of clusters. Together, these approaches outperform alternatives like k-means, regex-based clustering, or topic modeling by offering superior accuracy, robustness against noisy data, and adaptability to diverse and irregular SQLi datasets. From a computational perspective, TF-IDF + HAC is suitable for smaller, structured datasets with clear hierarchical relationships, while SequenceMatcher + DBSCAN is more scalable and robust for larger, unstructured, and noisy datasets [32]. Experimental evaluation of the clustering algorithms is presented in Section 5.

4.5. WAF Security Rule Generation

To generate Web Application Firewall (WAF) security rules based on clusters of SQL injection (SQLi) attacks, the process involved crafting specialized prompts for LLMs. The prompts are presented in detail in Appendix B. This prompt provides structured guidance for an LLM to generate ModSecurity SecRules that detect obfuscated SQL injection (SQLi) patterns effectively. Key elements include the following: (1) System Role: The LLM acts as a security expert specializing in creating efficient ModSecurity rules for SQLi protection. (2) Clustered Input Organization: SQLi patterns are categorized into six clusters with unique traits and examples, aiding the LLM in focusing on specific obfuscation types for targeted rule creation. (3) Clear Task Definition: The LLM generates rules with broad pattern coverage and minimal false positives. The rules are presented as a continuous block for easy integration, with comments explaining each rule's purpose and effectiveness.

4.6. Role of Reinforcement Learning with Human Feedback

Reinforcement Learning with Human Feedback (RLHF) significantly improved the effectiveness of ModSecurity WAF rules by addressing initial issues such as unstructured prompts, syntax errors, and incomplete coverage. Through iterative feedback, the process refined prompt design and rule generation to achieve higher accuracy and efficiency. The RLHF process involved testing and validating initial rule outputs, identifying issues like syntax errors, redundancy, and limited coverage. Feedback was then used to adjust the prompt, emphasizing generalization, minimizing redundancy, and enhancing rule efficiency. For example, syntax errors were corrected through ModSecurity testing, and redundant rules were consolidated without sacrificing detection accuracy. The refined prompts, informed by RLHF, produced rules with significantly fewer errors, broader coverage of SQL injection (SQLi) patterns, and reduced false positives. RLHF ensured that the rules effectively blocked obfuscated SQLi attacks while maintaining precision and scalability.

5. Evaluation and Results

This section presents an experimental evaluation of GenSQLi and reports its performance metrics. The evaluation addresses the following key research questions:

- RQ1: Generative Effectiveness—How effective are the SQL injection (SQLi) attacks generated by the Large Language Model (LLM)?
- RQ2: Attack Novelty—How many of these generated attacks successfully bypass Web Application Firewalls (WAFs) configured with state-of-the-art rules?
- RQ3: Rule Generation—How effective are the defense rules produced by our combined machine-learning and generative AI approach in mitigating these attacks?
- RQ4: LLM Portability—How well does the methodology generalize when applied to different LLMs?

5.1. Experimental Setup

The GenSQLi architecture of Figure 2 was implemented with the following open-source projects, as shown in Table 2.

A custom PHP application was developed to interact with a MySQL database by executing SQL queries based on user input received via HTTP requests. The application lacks proper input sanitization, making it vulnerable to SQL injection attacks. This design enables testing the effectiveness of the Web Application Firewall (WAF) independently of any application-level query sanitization mechanisms.

Table 2. System components used in the evaluation setup. All components used their default configurations. GPT-4o was configured with a temperature setting of 0.7, and Gemini Pro with a temperature setting of 1.

System	Details
LLM	OpenAI GPT-4o, Google Gemini Pro
Operating System	Ubuntu 22.04 LTS
Database	MySQL v8.0.40, PostgreSQL v14.15
Webserver	Apache v2.4.52
Web Application Firewall	ModSecurity v2.9.5 default configuration, AWS WAF
Attack Detection Rules	OWASP CRS v4.9.0
Vulnerable Application	Custom PHP app
Clustering algorithms	Scikit-learn v1.6.0

5.2. Sample SQLi Generation

To demonstrate the operation of GenSQLi, we present a specific example of a generated SQL injection (SQLi) attack that successfully bypasses ModSecurity.

5.2.1. Generated Payload

```
-7552 OR (SELECT CASE WHEN (6872=6872 AND ASCII('a')=97 COLLATE utf8mb4_bin)
    THEN 1 ELSE 0 END)=1
```

5.2.2. Analysis of the Generated Query

The payload is designed to exploit a numeric-based SQL injection vulnerability with the following key components:

- *-7552: A numeric value that integrates seamlessly into the backend SQL query syntax.*
- *OR*: A logical operator that overrides the original query condition if the injected condition evaluates to `TRUE`.
- *SELECT CASE WHEN…THEN 1 ELSE 0 END*: A subquery introducing conditional logic that evaluates specific conditions:
  - *6872 = 6872*: A tautology ensuring this condition always evaluates to `TRUE`.
  - *ASCII('a') = 97 COLLATE utf8mb4_bin*: Confirms the ASCII value of `'a'` matches 97, which is also `TRUE`.
- **=1**: Forces the overall query condition to evaluate as `TRUE` when the subquery result equals 1.

5.2.3. Payload Submission to the Vulnerable Application

The payload is typically injected through parameters such as URLs or form fields. For example, using a URL-based attack:

```
http://example.com/vulnerable_app?input=-7552 OR (SELECT CASE WHEN (6872 =
    6872 AND ASCII('a') = 97 COLLATE utf8mb4_bin) THEN 1 ELSE 0 END) = 1
```

The application appends the user-supplied input (`-7552 OR (SELECT CASE WHEN...)`) directly into its backend query due to inadequate input sanitization or the absence of parameterized queries.

5.2.4. Final Backend SQL Query

The vulnerable application generates the following backend query:

```
SELECT * FROM users WHERE id = -7552 OR (SELECT CASE WHEN (6872 = 6872 AND
    ASCII('a') = 97 COLLATE utf8mb4_bin) THEN 1 ELSE 0 END) = 1;
```

The query operates as follows:

- **id = -7552**: Checks for a user with an `id` of `-7552`. Typically, this returns no results unless such an ID exists.
- *OR (SELECT CASE WHEN…) = 1*:
  - The `OR` operator ensures that if the first condition fails, the injected condition is evaluated.
  - The `CASE WHEN` statement evaluates the following conditions:
    * *6872 = 6872*: A tautology, always `TRUE`.
    * *ASCII('a') = 97 COLLATE utf8mb4_bin*: Verifies that the ASCII value of `'a'` matches 97, also `TRUE`.
  - Since both conditions are `TRUE`, the `CASE WHEN` statement returns `1`, making the overall expression evaluate to `TRUE`.

5.2.5. Execution and Outcome

As the injected condition evaluates to `TRUE`, the query bypasses the original `id = -7552` condition and retrieves all rows from the `users` table.

5.2.6. Bypassing ModSecurity

The payload bypasses ModSecurity by leveraging the following techniques:

- *Numeric Input*: Begins with a numeric value (`-7552`), evading common WAF rules targeting string-based SQL injections.
- *Conditional Logic and Obfuscation*: Uses `CASE WHEN`, arithmetic (`6872 = 6872`), and character functions (`ASCII('a') = 97`) to introduce complexity and obscure intent.
- *Valid SQL Syntax*: Maintains proper syntax, allowing it to integrate seamlessly into the backend query without causing errors.

This example highlights the effectiveness of GenSQLi in generating novel and obfuscated SQLi attacks capable of bypassing modern WAFs.

5.3. Results

Table 3 summarizes the results of SQL injection (SQLi) generation using the GPT-4o and Gemini LLM models. Both models were provided with a prompt (detailed in Appendix A) along with 11 manually crafted obfuscated SQLi samples as in-context examples. The generated samples were validated for correctness using a MySQL database.

Table 3. SQLi attack generation results for GPT-4o and Gemini-Pro. The attacks are validated with MySQL to ensure that they are functional SQL queries.

LLM Model	Num SQLi Attacks	Num Valid	Num Invalid
GPT-4o	514	475	39
Gemini-Pro	393	209	184

- GPT-4o: Generated 514 samples, of which 475 were valid SQLi payloads, achieving a validity rate of 92.5%.
- Gemini: Generated 393 samples, of which 209 were valid, resulting in a validity rate of 53.1%.

These results indicate that GPT-4o is more consistent in generating valid SQLi attacks. Notably, Gemini Pro exhibits a higher tendency to generate hallucinated outputs compared to GPT-4o.

Tables 4 and 5 provide a breakdown of the validated SQLi samples generated by GPT-4o and Gemini-Pro, respectively. The SQLi attacks are listed by type of attack.

Table 4. Validated SQLi attacks by type generated by GPT-4o that bypassed ModSecurity.

Attack Types	Num SQLi Attacks	Num Blocked	Num Bypass WAF
Conditional	128	2	126
Error-Based	35	0	35
Tautology	220	51	158
Tautology and Conditional	47	0	47
Exist-Based	45	0	45
Total	475	53	422
Percentage	100%	11%	89%

Table 5. Validated SQLi attacks by type generated by Gemini-Pro that bypassed ModSecurity.

Attack Types	Num SQLi Attacks	Num Blocked	Num Bypass WAF
Conditional	61	25	36
Error-Based	13	0	13
Tautology	65	32	33
Tautology and Conditional	34	34	0
Exist-Based	36	0	36
Total	209	91	118
Percentage	100%	43.5%	56.5%

- GPT-4o: Of the 475 validated SQLi attacks, 422 (89%) bypassed ModSecurity.
- Gemini Pro: Of the 209 validated SQLi attacks, 118 (56.5%) bypassed ModSecurity.

Both models predominantly generated conditional and tautology-based SQLi attacks, accounting for over 60% of the total attacks. Overall, 89% of the SQLi attacks generated by GPT-4o and 56.5% for Gemini-Pro were able to bypass ModSecurity. These findings demonstrate the ability of GPT-4o to produce novel and effective SQLi payloads through proper prompting and in-context learning.

Table 6 presents the results of SQLi attacks generated using GPT-4o and tested against AWS WAF. The AWS WAF was configured with a Web ACL containing the SQL database rule set to detect and block SQL injection attacks. It was attached to an Application Load Balancer (ALB) directing traffic to an EC2 instance, ensuring malicious traffic is blocked before reaching the server. Results show that 41% of the attacks bypassed AWS WAF. While this performance is better than ModSecurity, the high success rate highlights generative AI's capability to effectively synthesize SQLi attacks.

Table 7 presents the evaluation results for ModSecurity rules generated using the two clustering techniques described in Section 4. For TF-IDF + HAC, ward linkage was used with a distance_threshold of 2.8 to identify a minimal number of meaningful clusters without predefining the cluster count. For DBSCAN, the metric parameter was set to precomputed using a custom distance matrix based on $1 -$ similarity ratio from Sequence-Matcher. The parameters eps = 0.4 and min_samples = 2 were chosen after testing, as they yielded a minimal yet meaningful set of clusters. We used silhouette scores to evaluate cluster cohesion and separation, which range from −1 to +1, with +1 indicating strong clustering and −1 indicating poor clustering. The silhouette scores for TF-IDF + HAC and SequenceMatcher + DBSCAN were 0.37 and 0.48, respectively, reflecting moderate clustering performance.

Table 6. Validated SQLi attacks by type generated by GPT-4o that bypassed AWS WAF.

Attack Types	Num SQLi Attacks	Num Blocked	Num Bypass WAF
Conditional	128	56	72
Error-Based	35	35	0
Tautology	220	99	121
Tautology and Conditional	47	47	0
Exist-Based	45	45	0
Total	475	282	193
Percentage	100%	59.3%	40.7%

Table 7. ModSecurity rules generated by GPT-4o based on different algorithms used to cluster attacks that initially bypassed ModSecurity. A small set of 23 rules is able to block over 99% of the attacks.

Attack Types	Num Samples	Clustering Algorithms	Num Rules	Num Blocked
Conditional	128	TF-IDF + HAC	6	127
		SeqMatcher+DBSCAN	6	125
Tautology	220	TF-IDF + HAC	8	189
		SeqMatcher+DBSCAN	3	217
Tautology & Conditional	47	TF-IDF + HAC	5	47
		SeqMatcher+DBSCAN	5	47
Error-Based	35	TF-IDF + HAC	4	35
		SeqMatcher+DBSCAN	5	35
Exist-Based	45	TF-IDF + HAC	5	45
		SeqMatcher+DBSCAN	4	45

The clusters generated were input to the LLM for rule generation. We note that SeqMatcher+DBSCAN clustering-driven rule generation by GPT-4o shows the best performance. A total of 23 SecRules, generated using the SeqMatcher algorithm combined with the DBSCAN classifier, successfully blocked 99% of the 475 SQLi attacks that had previously bypassed ModSecurity. Samples of these rules are provided in Appendix B.

In Table 8, we present False Positive, Precision, Recall, and F1 Score metrics to provide a comprehensive evaluation of the performance of the security rules, balancing its accuracy in detecting true positives while minimizing errors and assessing its overall effectiveness. Real-world web traffic often exhibits a higher proportion of normal samples compared to attacks. A commonly used ratio is 4:1 (80% normal, 20% attack). For this study, we created 1596 normal samples spanning four operations (Create, Read, Update, Delete) and 475 attack samples. The definitions of the metrics and their calculated values are provided in Table 8.

Table 8. Performance metrics for generated ModSecurity security rules.

Metric	Definition/Calculation
True Positives (TP)	Number of attacks correctly blocked by the WAF: $TP = 469$
False Negatives (FN)	Number of attacks not blocked by the WAF: $FN = 6$
True Negatives (TN)	Number of normal samples correctly not blocked by the WAF: $TN = 1596$
False Positives (FP)	Number of normal samples incorrectly blocked by the WAF: $FP = 0$
Accuracy	$\frac{TN+TP}{TN+TP+FN+FP} = 0.9956$
Precision	$\frac{TP}{TP+FP} = 1.0$
Recall	$\frac{TP}{TP+FN} = 0.9874$
F1-Score	$2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = 0.9936$

In the context of SQLi detection, the high precision means that nearly every detected attack is a true attack, with very few false positives (normal traffic wrongly classified as an attack). The high recall indicates that almost all SQL injection attacks are correctly identified and blocked, with very few false negatives (missed attacks).

We provide a brief assessment of the performance of the clustering strategies based on the results of Table 7. DBSCAN excels at grouping SQLi queries with similar structures, even when obfuscated, by dynamically adapting to the dataset and identifying natural clusters without needing predefined cluster counts. It focuses on meaningful patterns while excluding outliers, though it may cluster noisy data if their structure appears similar. In contrast, TF-IDF identifies important terms across the dataset, emphasizing the unique features of SQLi attacks and reducing noise by de-emphasizing irrelevant terms. This approach enables precise clustering and rule generation, making it effective for targeting specific attack types with high precision and minimal false positives. Overall, TF-IDF provides a global perspective on query patterns, while DBSCAN offers localized similarity clustering. Our results suggest that both strategies could be used to identify the most effective SecRules that can be generated. Future investigations could explore integrating regex for initial pattern matching with machine-learning models to develop more computationally efficient classification systems.

5.4. SQLi Verification on Other Databases

We tested the 475 generated SQLi samples on a PostgreSQL database, where 270 samples (56.8%) failed due to syntax errors and the absence of MySQL-specific functions such as HEX() and ORD(), which are not supported in PostgreSQL. These failures do not imply that the samples are flawed; all of them were validated successfully on MySQL, the target database for which the prompt was initially designed. The failures instead highlight compatibility issues between database systems, highlighting the critical role of prompt engineering in tailoring outputs to specific environments.

To address this, we developed a highly structured and detailed prompt optimized for MySQL-compatible syntax and obfuscation techniques to ensure sample validity. By including the full prompt in this paper (Appendices A and B), we aim to provide a blueprint for researchers and practitioners to adapt prompts for other database systems (e.g., PostgreSQL, Microsoft SQL Server). This flexibility to generate tailored data using intuitive natural language instructions highlights a key strength of generative AI.

6. Discussion

The increasing capabilities of generative AI are significantly influencing the field of cybersecurity. Threat actors are leveraging generative AI to scale the scope and sophistication of their attacks. In this paper, we introduced GenSQLi, a generative AI-based framework designed to help cybersecurity defenders strengthen their systems against SQL injection (SQLi) attacks. GenSQLi employs generative AI to synthesize novel SQLi attack patterns capable of bypassing existing Web Application Firewall (WAF) security rules. The framework then combines machine learning with generative AI to generate defense rules that address entire classes of these attacks. The approach employed by GenSQLi is in contrast to the traditional approach of the manual crafting of WAF security rules. Table 9 compares the generative AI approach with the manual approach across multiple features.

Table 9. Comparison of hand-crafted and AI-generated firewall rules.

Feature	Hand-Crafted Firewall Rules	AI-Generated Firewall Rules
Attack Generation Capabilities	Limited to known signatures; updates are manual and reactive.	Can adapt to novel and evolving threats
Defense Adaptability	Slower to adapt; changes require human intervention to add or refine rules.	Faster adaptability; models can incorporate new threat intelligence and retrain on evolving attack patterns with only a few examples.
Technical Expertise Required	High; crafting and maintaining rules demand deep understanding of protocols, vulnerabilities, and patterns.	Moderately lower; leveraging pre-trained models reduces the need for low-level rule-writing, though ML expertise may be needed for fine-tuning.
Computational Efficiency	Generally efficient; simple pattern-matching is computationally inexpensive.	Variable; depending on model complexity and inference cost, may require more computational resources.
Scalability	Scales by adding more rules, but management complexity can increase substantially over time.	Potentially more scalable; a well-trained model can generalize broadly, though high throughput might require more computing resources.
Limitations	Prone to false positives if rules are too broad; limited coverage of novel threats due to slow update cycles.	LLM-based approaches may face token constraints and may be prone to hallucinations.

This approach exemplifies the potential of using generative AI to proactively enhance cybersecurity defenses by synthesizing potential attack scenarios before they are encountered in real-world systems. The process can be iterated with multiple generation strategies until a satisfactory level of robustness is achieved. Existing SQLi generation techniques, such as context-free grammars and evolutionary algorithms (see Section 2), can serve as in-context learning examples to create families of similar attacks. Additionally, newly discovered attacks can immediately be incorporated as in-context examples to generate similar attack vectors and corresponding defense rules. Although GenSQLi primarily targets rule-based firewalls, the generated SQLi attacks can also enhance machine-learning-based attack detection models by supplying additional training data. Overall, the proposed generative AI framework provides a foundation for automated defense against a wide range of cybersecurity threats.

An advantage of using generative AI in this capacity is the relative ease of generating both attacks and defenses, particularly when compared to traditional methods like context-free grammars.This lowers the technical barrier, enabling a broader range of cybersecurity practitioners to enhance system defenses against cyberattacks. Unlike traditional manual drafting of firewall rules, a generative AI-driven approach offers faster adaptation to newly observed and even previously unseen attacks. However, validation mechanisms are essential to address the risk of incorrect outputs caused by LLM hallucinations.

6.1. Observations on Generative AI Model Restrictions

Our research highlights an important observation: some publicly available models, such as Anthropic Claude and Meta's LLaMA, refused to generate SQLi attack vectors. While such restrictions aim to limit misuse by attackers, they also inhibit defenders from leveraging these models for creating proactive defenses. This creates a double-edged sword; while the restrictions may curb certain malicious activities, they can also disadvantage smaller defenders who lack the resources to develop proprietary generative AI models.

We argue that determined adversaries, such as nation-states and large criminal organizations, are likely to possess the resources to build their own generative AI systems for crafting sophisticated attacks. Consequently, enabling smaller defenders to access similar capabilities becomes crucial. However, we recognize this as a significant public policy question requiring further investigation.

Our work aims to catalyze discussions on the role of publicly available generative AI models in cybersecurity. The balance between restricting generative AI for ethical concerns and empowering defenders to address emerging threats is a nuanced issue deserving deeper exploration.

6.2. Ethical Considerations

The use of generative AI in cybersecurity raises important ethical considerations, particularly when it is used to automatically generate both attacks and corresponding defenses, as demonstrated by GenSQLi. A key concern is the potential for attackers to exploit the attack-generation capabilities of such tools for malicious purposes. However, we contend that with the widespread availability of generative AI tools and the significant motivation of attackers, withholding this research would ultimately do more harm than good. Instead, there is a pressing need to ensure that generative AI tools are equally accessible to defenders, enabling them to stay ahead of evolving threats. By publicizing and integrating these tools into open-source projects, such as OWASP ModSecurity, we can ensure that widely used defense systems remain up to date and capable of addressing modern attack techniques. The paradigm we aim to establish is that generative AI, when leveraged effectively, can tilt the balance in favor of defenders, empowering them to proactively strengthen systems against emerging threats.

6.3. Limitations

While the generated SQLi samples offer significant insights, this research has certain limitations. Although GPT-4o successfully generated a diverse range of SQLi samples, it has a high token limit of 128k tokens. However, the outputs from other LLMs may be limited by token constraints, impacting both the quantity and complexity of the samples produced. Additionally, the validation was conducted in a controlled environment with ModSecurity as the sole defense layer. This approach may not fully capture the challenges faced in more complex, real-world scenarios where multiple security mechanisms (for example, input sanitization) are involved.

6.4. Generalizability

LLMs: The research highlights that while GPT-4o and Gemini Pro are state-of-the-art models from leading commercial entities, their performance on the SQLi generation task varies significantly. Despite extensive tuning of the Gemini Pro prompts, we observed that both GPT-4o and Gemini Pro performed best with the same prompts provided in Appendices A and B. Both LLMs have sufficiently large input token capacities (128,000 tokens for GPT-4o and 2 million for Gemini Pro), ensuring token length is not a limiting factor for these prompts. However, as these models are commercial black-box implementations, differences in performance cannot be readily attributed to their internal architectures. As discussed in Section 6.1, Claude and Llama declined to generate attack vectors.

Databases: In terms of database compatibility, only 56.8% of the SQLi attacks validated on MySQL were successful on PostgreSQL. However, the prompts provided in Appendices A and B allow users to adapt the generation process to PostgreSQL or other databases. This adaptability to create customized outputs using natural language instructions is a core strength of generative AI.

Web Application Firewalls: Our experimental results show that the generated SQLi attacks were successful against both ModSecurity and AWS WAF, though to a lesser extent against AWS WAF (89% vs. 40.7%). We also tested Cloudflare and Imperva WAFs; however, their setups required a valid registered domain name, making them more complex to configure. While our current work focuses on synthesizing security rules for ModSecurity, we believe similar rules can be generated for other WAFs. The use of natural language to synthesize these rules, combined with in-context examples to guide the LLM, significantly reduces the technical expertise required for cybersecurity engineers to secure systems against SQLi attacks.

7. Conclusions and Future Work

In this paper, we presented a novel generative AI framework designed to enhance the security posture against SQLi attacks by leveraging the advanced capabilities of Large Language Models (LLMs). Our framework achieves two pivotal objectives:

1. Generation of Novel SQLi Attacks: Utilizing in-context learning with carefully curated examples, we harnessed generative AI to produce a diverse array of SQLi payloads. This approach significantly reduces the occurrence of hallucinations—a common issue in AI-generated content—by ensuring that generated attacks are both syntactically and semantically valid. The high validation rate of 92.5% against a MySQL database attests to the effectiveness of this method.
2. Automated Defense Mechanisms: By systematically testing the generated payloads against a vulnerable web application fortified with a WAF, we identified attacks that successfully bypassed existing security measures. These bypassing attacks were then subjected to classification using machine-learning algorithms. Leveraging natural language processing techniques, we automatically generated new WAF rules tailored to counter these specific attack patterns. The iterative refinement of WAF defenses culminated in the successful blocking of all previously bypassing attacks.

Our experimental evaluation demonstrated the framework's efficacy and generality. With the GPT-4o LLM, we generated 514 new SQLi payloads, of which 89% bypassed the ModSecurity WAF equipped with the latest OWASP Core Rule Set. Remarkably, the introduction of just 23 new WAF rules was sufficient to block 475 unique SQLi attacks, highlighting the broad applicability and effectiveness of the generated defenses.

By planning to open-source our framework, we aim to democratize access to advanced cybersecurity tools, enabling a wider range of practitioners to proactively defend against evolving threats. This approach lowers the technical barriers associated with traditional methods, such as manual rule crafting, and fosters a collaborative environment where defense strategies can be rapidly shared and improved upon.

Our work also brings to light important considerations regarding the dual-use nature of generative AI in cybersecurity. While these technologies offer powerful tools for defenders, they can equally be exploited by malicious actors. The reluctance of some publicly available models to generate attack vectors reflects a broader debate on responsible AI usage. We acknowledge the complexity of this issue and advocate for ongoing dialogue among policymakers, technologists, and the cybersecurity community to establish guidelines that balance innovation with security.

7.1. Future Work

The promising results of our framework open several avenues for future research and development:

- Extension to Other Attack Types: The methodology could be adapted to generate and defend against other forms of web application attacks, such as Cross-Site Scripting (XSS) or Remote Code Execution (RCE) exploits, enhancing the overall security landscape.
- Real-Time Adaptation: Integrating real-time monitoring and adaptive learning capabilities could enable the WAF to respond dynamically to emerging threats without manual intervention.
- Ethical Frameworks and Policy Development: Collaborating with stakeholders to develop ethical guidelines and policies governing the use of generative AI in cybersecurity will be crucial for ensuring these tools are used responsibly.
- Enhanced Validation Mechanisms: Improving the validation process for generated payloads to include a wider range of database systems and configurations could increase the robustness of the framework.
- Machine-learning-based SQLi detection: Generative AI could be used to generate datasets to train machine-learning-based SQLi detection models, so as to improve the robustness of the models.

7.2. Integration of GenSQLi in Enterprise Systems

We have developed a prototype that leverages generative AI to automatically generate novel SQLi attacks and corresponding security rules to defend against them, demonstrated using the ModSecurity WAF. As web threats continue to evolve rapidly, enterprises face increasing challenges in keeping their WAF configurations updated to defend against sophisticated and emerging attack patterns. To address this need, as a part of future efforts, we propose offering this capability as a SaaS (Software-as-a-Service) solution. This service would enable enterprises to dynamically generate tailored WAF rules based on the latest AI-generated attack simulations, ensuring proactive and automated defense. By integrating this solution into their security workflows, organizations can enhance their web application security posture, reduce manual effort, and stay ahead of adversaries. Such a service would complement existing WAF solutions, providing enterprises with continuous updates to their defense mechanisms, informed by generative AI-driven threat modeling. The SaaS service could also be extended to address other types of web attack, such as XSS, RCE, and similar threats.

In summary, our generative AI framework represents a significant advancement in the proactive defense against SQL injection attacks. By uniting the offensive and defensive potentials of generative AI, we have demonstrated a holistic approach to cybersecurity that not only anticipates and neutralizes current threats but also adapts to future challenges. We believe that this work contributes valuable insights to the cybersecurity field and encourages further exploration into the integration of artificial intelligence for safeguarding critical digital infrastructures.

Author Contributions: Conceptualization, A.R.; Software, V.B.; Validation, V.B.; Writing—original draft, V.B.; Writing—review & editing, A.R.; Supervision, A.R. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Appendix A.1. Instruction for In-Context Learning

The model is explicitly informed that the input serves as in-context learning examples for guiding the LLM to generate creative yet logically consistent variations. Then, LLM transforms the base input into complex, obfuscated forms using a systematic application of multiple bypass techniques.

```
sqli_attack = "-9111 OR (3038+0) = 3038"
You will receive a SQLi attack input, indicated as {sqli_attack}. This input is an example that you can use as in-context learning. Apply all relevant bypass methods to the {sqli_attack} to create obfuscated and complex variations.
```

Appendix A.2. Examples of Obfuscation Methods

A comprehensive list of obfuscation strategies is provided. Each method includes a brief explanation and an example to guide the LLM in crafting variations:

```
The new samples can use obfuscation techniques that are compatible with MySQL.
Use methods such as:
Hexadecimal Encoding: Convert characters or strings into hexadecimal format using MySQL functions like UNHEX() and HEX(). Ensure that the logical conditions remain equivalent.
Example: 0x61 instead of 'a'.

String Manipulation: Use functions like CONCAT(), ASCII(), ORD(), or CHAR() to manipulate strings and characters in a way that maintains the original logic.
Example: CONCAT('a', '') = 'a'.

Inline Comments: Add comments using /* comment */ to break up the syntax and make detection more difficult while ensuring the query still works.
Example: 1 = 1 /*hidden*/ AND 'a' /*comment*/ = 'a'.

Mathematical Logic: Use mathematical expressions that evaluate to the same result, such as ORD('a')-96 = 1 to represent 'a' = 'a'.
Example: (ORD('a')-96) = 1.

Whitespace Variations: Insert or remove spaces strategically and use tab characters or newline characters to make the query more obfuscated.
Example: 1 = 1 AND 'a'
nLIKE 'a%'.

Nested Subqueries: Use subqueries to add complexity while preserving the overall logic of the statement.
Example: (SELECT 'a') = 'a'.
```

NULLIF and Conditional Functions: Use functions like NULLIF() to introduce additional complexity.
Example: NULLIF('a','b') = 'a'.

Logical Equivalence: Use logical operators or clauses that are equivalent but more complex, such as LIKE instead of =.
Example: 'a' LIKE 'a%'.

Appendix A.3. Database Context

Context about the database schema is provided, ensuring compatibility of the generated SQLi samples with the actual database environment. This step grounds the variations in practical scenarios.

I plan to test all generated SQLi samples directly in a MySQL database. The database contains a single database named test_db with a table called users. The users table has three columns: id, username, and password. If any SQLi sample contains references to table names or column names, replace those with the actual names from the MySQL database provided (test_db for the database name; users for the table; and id, username, and password for columns). Ensure these substitutions are applied consistently to maintain the sample's logical structure and functionality.

Appendix A.4. Generation Task

The prompt mandates the LLM to apply all relevant techniques systematically, ensuring comprehensive exploration of obfuscation strategies while skipping incompatible techniques. The output format is clearly specified, focusing on generating a seamless list of variations for ease of testing. The generated samples must remain functional and syntactically valid for MySQL, ensuring their utility for real-world testing. The prompt emphasizes relevance, instructing the LLM to avoid unnecessary or non-functional transformations, thereby enhancing the quality of the output. The required number of samples ensures a diverse set of obfuscated payloads for robust testing. This also ensures the LLM generates a comprehensive output that can be directly used for testing in the specified database environment.

Then, generate each obfuscated version of the SQLi attack based on the techniques listed above.

Avoid listing the attacks with numbers. Write each example on a new line without additional explanations, and samples must be back to back without any empty line between each sample.

Ensure each variation maintains the logical and functional integrity of the original attack while obfuscating it effectively. Also note that the generated queries will be tested in the MYSQL database, so the syntax should be correct according to the MYSQL database.
If you believe one or some of obfuscated techniques are not compatible with the input or the input does not contain critical words for that specific techniques, ignore them and do not apply them.

You Must generate 50 samples and provide the complete set of obfuscated SQLi attacks in response.

Appendix B

This prompt is a detailed instruction designed to guide LLM in generating ModSecurity SecRules tailored to detect obfuscated SQL injection (SQLi) patterns. Below is an explanation of each element, focusing on how it contributes to the creation of effective and efficient rules:

Appendix B.1. System Role

You are a security expert specialized in writing efficient ModSecurity rules to protect web applications against SQL injection (SQLi) attacks.

Appendix B.2. Cluster-Based Organization of Input

The prompt groups SQLi patterns into six clusters, each with unique characteristics and examples. This organization helps the LLM focus on distinct types of obfuscation and simplifies the process of creating tailored rules.

Cluster 0: Character Manipulation & String Obfuscation
Characteristics:
Heavy use of functions like CHAR, ORD, ASCII, and CONCAT. Obfuscated comparisons (e.g., 'a' = 'a' or NULLIF). Logical tautologies (1 = 1) with string manipulation. Inline obfuscated comments (e.g., /*hidden*/).
Example Queries:
or (SELECT CASE WHEN (1 = 1 AND CONCAT('a', '') = 'a') THEN 1 ELSE 0 END) = 1– or (SELECT CASE WHEN (1 = 1 AND CHAR(97) = CHAR(97)) THEN 1 ELSE 0 END) = 1– Generate SecRules to detect:
Character manipulation functions. Obfuscated comments. Tautologies using string equivalence or logical comparisons.

Cluster 1: Tautologies with Encoding and Collations
Characteristics:
Tautologies like 6872 = 6872 combined with obfuscated encoding (ASCII, ORD, CHAR). Use of collations (e.g., COLLATE utf8mb4_bin). Hexadecimal logic (0×61 = CHAR(97)).
Example Queries:
-7552 OR (SELECT CASE WHEN (6872 = 6872 AND ASCII('a') = 97) THEN 1 ELSE 0 END) = 1 -7552 OR (SELECT CASE WHEN (6872 = 6872 AND CHAR(97) = 'a' COLLATE utf8mb4_bin) THEN 1 ELSE 0 END) = 1 Generate SecRules to detect:
Encoding functions (ASCII, ORD, CHAR). Tautologies involving numeric or hexadecimal encoding. Collation usage in queries.

Cluster 2: Numeric Obfuscation & Logical Comparisons
Characteristics:
Numeric tautologies (3038 = 3038) and comparisons (HEX(3038) = HEX(3038)). Logical expressions with NULLIF and CONCAT. Obfuscation using inline comments (e.g., /*comment*/).
Example Queries:
-9111 OR (SELECT CASE WHEN (3038 = 3038) THEN 1 ELSE 0 END /*no = 0*/) = 1
-9111 OR (SELECT CASE WHEN (HEX(3038) = HEX(3038)) THEN 1 ELSE 0 END) = 1
Generate SecRules to detect:
Numeric tautologies and logical obfuscations. Inline comments masking intent.

Cluster 3: ASCII/ORD Obfuscation Characteristics:
Character comparisons using ASCII and ORD functions. Logical expressions with nested obfuscation. Heavy reliance on CHAR, ASCII, and numeric tautologies.
Example Queries:
or (SELECT CASE WHEN (1 = 1 AND ASCII('a') = 97) THEN 1 ELSE 0 END) = 1–
-9111 OR (SELECT CASE WHEN (ORD(CHAR(97)) = 97 AND 3038 = 3038) THEN 1 ELSE 0 END) = 1
Generate SecRules to detect:
ASCII and ORD logic. Character manipulation combined with numeric tautologies.

Cluster 4: Encoding Obfuscation with HEX/UNHEX
Characteristics:
Obfuscation with HEX and UNHEX. Nested encoding (e.g., UNHEX(HEX(...))). Comparisons of encoded strings.
Example Queries:
or (SELECT CASE WHEN (1 = 1 AND UNHEX(HEX('a')) = UNHEX(HEX('a'))) THEN 1 ELSE 0 END) = 1– -7552 OR (SELECT CASE WHEN (6872 = 6872 AND HEX('a') = HEX(CONCAT('a'))) THEN 1 ELSE 0 END) = 1
Generate SecRules to detect:
HEX and UNHEX obfuscations. Encoded string comparisons.

Cluster 5: Inline Comments & Simplified Tautologies
Characteristics:
Heavy use of inline comments (/*comment*/). Simplified tautologies combined with logical operators or string comparisons ('a' = 'a', LIKE).
Example Queries:
or (SELECT CASE WHEN (1 = 1 AND 'a' = 'a' /*comment*/) THEN 1 ELSE 0 END) = 1– or (SELECT CASE WHEN (1 = 1 AND 'a' LIKE 'a%') THEN 1 ELSE 0 END /*comment*/) = 1–
Generate SecRules to detect:
Inline comments. Logical operators and simple tautologies

Appendix B.3. Generation Task

The task is clearly defined: the LLM must generate rules that detect specific obfuscated SQLi patterns while minimizing false positives. This instruction emphasizes efficiency, precision, and the need for broad pattern coverage. The rules should be presented as a continuous block, which facilitates easy integration into ModSecurity configurations. Comments provide context for each rule, explaining what it detects and why it is effective.

Your task is to generate ModSecurity SecRules based on the provided cluster characteristics of SQLi queries. These rules must be minimal, efficient, and tailored to detect obfuscated patterns while avoiding false positives.

Write SecRules for ModSecurity that specifically address the characteristics of each cluster below.

Focus on creating as few rules as possible for each cluster while maintaining high coverage of the patterns. Avoid listing the rules with numbers. Write each rule on a new line without additional explanations and samples include comments explaining what the rule detects and why it is effective. Ensure rules are minimal but cover the cluster's characteristics comprehensively.

Appendix C

Table A1. Sample SQL injection generated by GPT-4o.

or (SELECT CASE WHEN (1 = 1 AND 'a' LIKE CONCAT('a', '%')) THEN 1 ELSE 0 END) = 1–
or (SELECT CASE WHEN (1 = 1 AND UNHEX(HEX('a')) = 0 × 61) THEN 1 ELSE 0 END) = 1–
−7552 OR (SELECT CASE WHEN (6872 = 6872 AND (SELECT 'a') = 'a') THEN 1 ELSE 0 END) = 1
−9111 OR (0 × 0bde + 0 × 0) = 0× 0bde
or (SELECT 1) = 1 AND (SELECT 1) = 1 AND (SELECT CHAR(52), CHAR(53), CHAR(57), CHAR(53)) = ('4', '5', '9', '5') –
−3520 OR 0 + 8571 = 8571 OR (SELECT CASE WHEN (8571 = 8571 OR 'b' = 'b') THEN 1 ELSE 0 END) = 1 –
or EXISTS(SELECT COUNT(*) FROM test_db.users AS sys1, test_db.users AS sys2, test_db.users AS sys3, test_db.users AS sys4, test_db.users AS sys5, test_db.users AS sys6, test_db.users AS sys7 WHERE 2462 = 2462 AND 2716 = (SELECT 2716))–

Appendix D

Table A2. Sample Modsecurity SecRules generated by GPT-4o.

```
SecRule REQUEST_URI|ARGS ‘‘@rx (?i)(char\(|ord\(|ascii\(|concat\()
    |/\*.*?\*/|(\b1\s*=\s*1\b|(\w\s*=\s*\w))’’ \
    ‘‘id:1001,phase:2,deny,status:403,msg:‘SQLi Pattern: Character
    manipulation, obfuscated comments, or~tautologies detected’’’
```

```
SecRule REQUEST_URI|ARGS ‘‘@rx (?i)(ascii\(|ord\(|char\()|(\b\d+\s*=\s*\d
    +\b)|0x[0-9a-f]+|collate\s+\w+’’ \
    ‘‘id:1002,phase:2,deny,status:403,msg:‘SQLi Pattern: Encoding
    functions, numeric tautologies, or~collation usage detected’’’
```

```
SecRule REQUEST_URI|ARGS ‘‘@rx (?i)(\b\d+\s*=\s*\d+\b)|hex\(\d+\)|nullif
    \(|concat\(|/\*.*?\*/’’ \
    ‘‘id:1003,phase:2,deny,status:403,msg:‘SQLi Pattern: Numeric
    tautologies, logical obfuscation, or~inline comments detected’’’
```

Table A2. *Cont.*

`SecRule REQUEST_URI|ARGS ''@rx (?i)(ascii\(|ord\(|char\(|\b\d+\s*=\s*\d+\b)'' \ ''id:1004,phase:2,deny,status:403,msg:'SQLi Pattern: ASCII/ORD obfuscation, character manipulation, or~numeric tautologies detected'''`
`SecRule REQUEST_URI|ARGS ''@rx (?i)(hex\(|unhex\(|\b\d+\s*=\s*\d+\b|concat\([^)]+\))'' \ ''id:1005,phase:2,deny,status:403,msg:'SQLi Pattern: HEX/UNHEX obfuscation or encoded string comparisons detected'''`
`SecRule REQUEST_URI|ARGS ''@rx (?i)(/\*.*?\*/|(\b\d+\s*=\s*\d+\b)|(\w\s*=\s*\w)|like\s+'[^']*')'' \ ''id:1006,phase:2,deny,status:403,msg:'SQLi Pattern: Inline comments, simplified tautologies, or~logical operators detected'''`
`SecRule REQUEST_URI|ARGS ''@rx (?i)(hex\(|unhex\(|\b\d+\s*=\s*\d+\b|concat\([^)]+\))'' \ ''id:1005,phase:2,deny,status:403,msg:'SQLi Pattern: HEX/UNHEX obfuscation or encoded string comparisons detected'''`
`SecRule REQUEST_URI|ARGS ''@rx (?i)(/\*.*?\*/|(\b\d+\s*=\s*\d+\b)|(\w\s*=\s*\w)|like\s+'[^']*')'' \ ''id:1006,phase:2,deny,status:403,msg:'SQLi Pattern: Inline comments, simplified tautologies, or~logical operators detected'''`

References

1. SQL Injection. 2024. Available online: https://owasp.org/www-community/attacks/SQL_Injection (accessed on 2 December 2024).
2. Clarke-Salt, J. *SQL Injection Attacks and Defense*; Elsevier: Amsterdam, The Netherlands, 2009.
3. OWASP Top Ten Project. 2024. Available online: https://owasp.org/www-project-top-ten/ (accessed on 2 December 2024).
4. Dermann, M.; Dziadzka, M.; Hemkemeier, B.; Hoffmann, A.; Meisel, A.; Rohr, M.; Schreiber, T. Best Practices: Use of Web Application Firewalls. The Open Web Security Application Project, OWASP Papers Program. 2008. Available online: https://wiki.owasp.org/images/b/b0/Best_Practices_WAF_v105.en.pdf (accessed on 2 December 2024).
5. Zewe, A. Explained: Generative AI. *MIT News*, 2023.
6. GPT-4o: OpenAI's Optimized GPT-4 Model. 2024. Available online: https://openai.com (accessed on 2 December 2024).
7. Gemini: Google's Multimodal AI Model. 2024. Available online: https://deepmind.com (accessed on 2 December 2024).
8. LLaMA: Large Language Model Meta AI. 2024. Available online: https://ai.facebook.com (accessed on 2 December 2024).
9. Claude: An AI Assistant by Anthropic. 2024. Available online: https://anthropic.com (accessed on 2 December 2024).
10. Gupta, M.; Akiri, C.; Aryal, K.; Parker, E.; Praharaj, L. From chatgpt to threatgpt: Impact of generative ai in cybersecurity and privacy. *IEEE Access* **2023**. [CrossRef]
11. Project, M. ModSecurity: Open Source Web Application Firewall. 2024. Available online: https://modsecurity.org/# (accessed on 3 December 2024).
12. Foundation, O. OWASP ModSecurity Core Rule Set Project. 2024. Available online: https://github.com/OWASP/www-project-modsecurity-core-rule-set (accessed on 3 December 2024).
13. Alghawazi, M.; Alghazzawi, D.; Alarifi, S. Detection of sql injection attack using machine learning techniques: A systematic literature review. *J. Cybersecur. Priv.* **2022**, *2*, 764–777. [CrossRef]
14. Applebaum, S.; Gaber, T.; Ahmed, A. Signature-based and machine-learning-based web application firewalls: A short survey. *Procedia Comput. Sci.* **2021**, *189*, 359–367. [CrossRef]

15. Qu, Z.; Ling, X.; Wang, T.; Chen, X.; Ji, S.; Wu, C. AdvSQLi: Generating Adversarial SQL Injections against Real-world WAF-as-a-service. *IEEE Trans. Inf. Forensics Secur.* **2024**, *19*, 2623–2638. [CrossRef]
16. Zhang, L.; Zhang, D.; Wang, C.; Zhao, J.; Zhang, Z. ART4SQLi: The ART of SQL injection vulnerability discovery. *IEEE Trans. Reliab.* **2019**, *68*, 1470–1489. [CrossRef]
17. Demetrio, L.; Valenza, A.; Costa, G.; Lagorio, G. Waf-a-mole: Evading web application firewalls through adversarial machine learning. In Proceedings of the 35th Annual ACM Symposium on Applied Computing, Brno, Czech Republic, 30 March–3 April 2020; pp. 1745–1752.
18. Appelt, D.; Panichella, A.; Briand, L. Automatically repairing web application firewalls based on successful SQL injection attacks. In Proceedings of the 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), Toulouse, France, 23–26 October 2017; pp. 339–350.
19. Hemmati, M.; Hadavi, M.A. Using deep reinforcement learning to evade web application firewalls. In Proceedings of the 2021 18th International ISC Conference on Information Security and Cryptology (ISCISC), Isfahan, Iran, 1–2 September 2021; pp. 35–41.
20. Halder, R.; Cortesi, A. Obfuscation-based analysis of SQL injection attacks. In Proceedings of the The IEEE Symposium on Computers and Communications, Riccione, Italy, 22–25 June 2010; pp. 931–938.
21. Nasereddin, M.; ALKhamaiseh, A.; Qasaimeh, M.; Al-Qassas, R. A systematic review of detection and prevention techniques of SQL injection attacks. *Inf. Secur. J. A Glob. Perspect.* **2023**, *32*, 252–265. [CrossRef]
22. Razzaq, A.; Hur, A.; Shahbaz, S.; Masood, M.; Ahmad, H.F. Critical analysis on web application firewall solutions. In Proceedings of the 2013 IEEE Eleventh International Symposium on Autonomous Decentralized Systems (ISADS), Mexico City, Mexico, 6–8 March 2013; pp. 1–6.
23. Bandi, A.; Adapa, P.V.S.R.; Kuchi, Y.E.V.P.K. The power of generative ai: A review of requirements, models, input–output formats, evaluation metrics, and challenges. *Future Internet* **2023**, *15*, 260. [CrossRef]
24. Feuerriegel, S.; Hartmann, J.; Janiesch, C.; Zschech, P. Generative ai. *Bus. Inf. Syst. Eng.* **2024**, *66*, 111–126. [CrossRef]
25. Sahoo, P.; Singh, A.K.; Saha, S.; Jain, V.; Mondal, S.; Chadha, A. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv* **2024**, arXiv:2402.07927.
26. Lewis, P.; Perez, E.; Piktus, A.; Petroni, F.; Karpukhin, V.; Goyal, N.; Küttler, H.; Lewis, M.; Yih, W.t.; Rocktäschel, T.; et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 9459–9474.
27. Li, X. A Review of Prominent Paradigms for LLM-Based Agents: Tool Use (Including RAG), Planning, and Feedback Learning. *arXiv* **2024**, arXiv:2406.05804.
28. Talebirad, Y.; Nadiri, A. Multi-agent collaboration: Harnessing the power of intelligent llm agents. *arXiv* **2023**, arXiv:2306.03314.
29. Ramos, J. Using TF-IDF to Determine Word Relevance in Document Queries. 2003. Available online: https://citeseerx.ist.psu.edu/document?repid=rep1;type=pdf;doi=b3bf6373ff41a115197cb5b30e57830c16130c2c (accessed on 2 December 2024).
30. Tokuda, E.K.; Comin, C.H.; Costa, L.d.F. Revisiting agglomerative clustering. *Phys. A Stat. Mech. Its Appl.* **2022**, *585*, 126433. [CrossRef]
31. Schubert, E.; Sander, J.; Ester, M.; Kriegel, H.P.; Xu, X. DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. *ACM Trans. Database Syst. (TODS)* **2017**, *42*, 1–21. [CrossRef]
32. Schütze, H.; Manning, C.D.; Raghavan, P. *Introduction to Information Retrieval*; Cambridge University Press: Cambridge, UK, 2008; Volume 39.