Compressing Large Language Models for SQL Injection Detection: A Case Study on Deep Seek-Coder and Meta-llama-3-70b-instruct

Borhanullah Hairan, Mehmet Akif Şahman
Selcuk University
Konya, Turkey
hairanborhan@gmail.com, asahman@selcuk.edu.tr

***Abstract*—With the ongoing incorporation of large language models (LLMs) into cybersecurity operations, it is important to examine how they perform in real-life attacks. This paper presents a comparative evaluation of two advanced large language models, Deep Seek Coder and meta-llama-3-70b-instruct, in detecting SQL injection vulnerabilities in source code. This article places greater emphasis on Boolean-based attacks. To test these models, we use a labelled dataset, which contains malicious and legitimate SQL queries. The most important parameters of classification, such as precision, recall, F1-score, and accuracy, are used in the measurement of performance. The results of the evaluation indicate that the meta-llama-3-70b-instruct model performed better in important parameters of accuracy with scores of 68.75% precision, 88.00% recall, 72.42% F1 score, and 74.00% overall accuracy. Comparatively, Deep Seek Coder attained a precision of 71.50%, a recall rate of 44.00%, an F1 score of 57.17%, and an accuracy rate of 60.00%. This comparison shows that Deep Seek Coder is particularly strong in precision, which hints at the ability to find common vulnerabilities, but also very weak in recall and accuracy. Meta-LLaMA-3-70B-Instruct has a better recall and overall accuracy and is best able to detect more varieties of malignant queries. So, it should have significant issues with detecting more context-sensitive or obfuscated security concerns. The results make it clear that these forms of models can be an efficient helper in a secure coding process, but they cannot be used as stand-alone security analysis tools at the current stage. Suggestions are being made to develop a combination of LLMs and classic static analysis and fine-tuning with domain-specific security datasets.**

I. INTRODUCTION

As many users and developers know, web applications form the backbone of digital interactions, but they are also prime targets for cyber threats. SQL injections have been considered one of the greatest and most common security vulnerabilities in web applications as it is today [1].It arises when input that the user controls is not correctly placed into database queries without sanitization, hence the attackers are able to perform different manipulations like manipulating the logic of queries, extraction of sensitive data, or even gaining full access to the backend databases [2]. Even without exciting mitigation methods, SQL injection has been observed many times in popular software systems such as content management systems, e-commerce web applications, and enterprise APIs, even after decades of awareness and mitigation techniques.

The core of modern digital infrastructures is web applications through which a large amount of sensitive data are handled in a single day. According to the OWASP Top 10 report, SQLi attacks have continued to happen and develop in recent years, and have been continuously ranked as the top security risk, allowing attackers to manipulate database queries and extract confidential information [3].

The Boolean-based SQL injection is a modification whereby respective database queries are employed to covertly manipulate true/false responses to the user's benefit and in favour of data theft. The nature of its functionality makes it difficult to detect since it uses conditional logic instead of syntax mistakes, which makes it highly obfuscating [4], [5].

Modern intrusion detection systems (IDSs) typically use signature-based or anomaly-based processes, which, to a large part, fail when it comes to dealing with changing adversarial strategies. As exemplified in the empirical work by Qbea'h, M., these mechanisms, by their very nature as being static, break quickly under the pressure of mutating query types since they produce very high rates of false-negative results as well as the inability to identify obfuscated or contextually subtle types of queries with reasonable accuracy [3]. Study of the empirical evidence shows that use of Boolean-based attacks may often take advantage of the vulnerabilities through the use of logical operators (e.g., AND/OR) to construct queries that seemingly pose no threats at first. The given findings demonstrate the high urgency of more sophisticated, responsive countermeasures in the modern web-security system [4], [6].

Recently, there has been a rapid development of large language models (LLMs), due to which their creativity and operational uses have gained additional attention.

The recent development of large language models (LLMs) trained on code understanding, including Deep Seek Coder [7] and Meta-LLaMA-3-70B-Instruct [8] has created new opportunities to conduct vulnerability detection in an automatic way. These models, pre-trained over giant repositories of open-source code and natural language instructions, are able to derive the semantics of code, reason about the behaviour of programs and answer targeted questions. LLMs come with a deep understanding of natural language as well as contextual analysis, placing them in the best position to deal with the parsing of sophisticated SQL commands. Recent studies on LLMs for code-centric tasks demonstrated that they can also be

applied on semantic analysis for detecting vulnerabilities which may change the threat detection forever [9]. Consequently, they are capable of being triggered to perform as smart code checkers who can detect security weaknesses.

Nonetheless, the assessment of the models dealing with certain types of vulnerabilities, such as SQL injection, remains insufficient. Fine-grained analysis of security-critical patterns does not exist in most studies, which are mostly on general bug detection or code repair [10].

This paper fills this gap by testing the potential of the Deep Seek Coder and Meta-LLaMA-3-70B-Instruct models in finding SQL injection vulnerabilities in code snippets that simulate scenarios realistically. We have made a contribution comprising.

- Comparative performance examination based on precision, recall, F1 score, and accuracy
- Qualitative insights into failure modes and contextual limitations of current LLMs

We demonstrate that while both models achieve high precision, their recall is significantly lower, highlighting a critical challenge for deployment in production security workflows.

A. How boolean-based sql injection works?

In Boolean-Based SQL Injection, the attacker enters specially programmed SQL queries on an input box (e.g. a login form or search box) that contains a condition statement. Whether a condition is true or false can be inferred by what the application does with the response to the request: to show success message, error page or time out. Incrementally testing each of the conditions, the attacker is able to determine which bits make up the contents of the database [5].

Example Scenario:

Assume an insecure web application contains a login form that makes a query of something similar to the following (in pseudocode):

- SELECT * FROM user WHERE username = 'input_username' AND password = 'input_password';

An attacker might manipulate the input to test for Boolean conditions. For instance:

- Original query: SELECT * FROM user WHERE id = 1 AND 1=1;
- Attacker's injection: SELECT * FROM user WHERE id = 1 AND 1=1; ' OR 1=1 –

If the response changes (e.g., the page loads fully), it indicates the condition is true.

- Further injection: SELECT * FROM user WHERE id = 1 AND 1=2;

If the page fails to load or shows an error, it indicates the condition is false.

Using this method, an attacker could enumerate database values. For example, to extract a username:

- Inject: username' AND SUBSTRING(username, 1, 1) = 'A' –

If true, the first character is 'A'; if false, test the next letter. In order to effectively and efficiently combat and prevent these threats we are able to use modern approaches and advanced technologies.

B. Models evaluated

Two code-specialized LLMs were selected for evaluation:

1) Deep Seek Coder (33B) : Deep Seek Coder is composed of a series of code language models, each trained from scratch on 2T tokens, with a composition of 87% code and 13% natural language in both English and Chinese. We provide various sizes of the code model, ranging from 1B to 33B versions. Each model is pre-trained on project-level code corpus by employing a window size of 16K and an extra fill-in-the-blank task to support project-level code completion and infilling. For coding capabilities, Deep Seek Coder achieves state-of-the-art performance among open-source code models on multiple programming languages and various benchmarks. The result shows that DeepSeek-Coder-Base-33B significantly outperforms existing open-source code LLMs in Fig. 1[7].

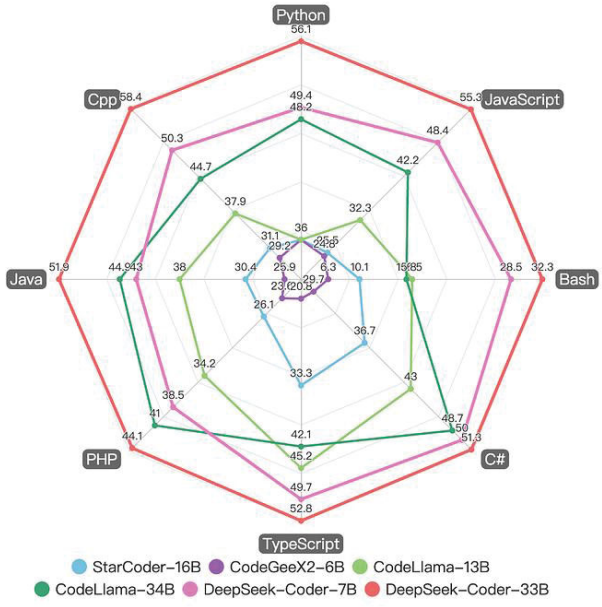

Fig. 1. Deep seek coder performance [7]

2) Meta-LLaMA-3-70B-Instruct: Meta-LLaMA-3-70B-Instruct is a state-of-the-art large language model introduced by Meta AI as one of the members of the LLaMA 3 family and is used in more advanced natural language processing tasks, such as instruction-following, code generation, and contextual reasoning. It is an instruct-tuned kind, which implies that it has been trimmed to one that can very well react to user requests and provide useful, coherent results to a broad variety of applications [8].

The LLaMA 3 models and 70B-Instruct version in particular are pre-trained on a large dataset of a wide variety of web-scale texts, code, and multilingual texts, amounting to more than 15 trillion tokens [8]. This also covers about half of

the natural language information (mainly in English, but other languages are supported), 30 percent code-related information (such as GitHub), and 20 percent other structured and unstructured information to add flexibility [11].

The LLaMA 3 series has different size models that can be used to suit different computational requirements. In particular, the 70B-Instruct model has an advantage in the parameter size and can thus perform better in tasks of complicated applications like filling in code, SQL query parsing, and real-time decisions [8].

II. Related works

In this section, we survey existing literature on SQL injection detection and the application of large language models (LLMs) in cybersecurity.

A. SQL injection detection

The classic methods of SQL injection detection are usually classified as signature-based methods, anomaly-based methods, or static analysis. The signature-based technique will entail matching predetermined patterns in the queries. Initial contributions in this field may be seen in the contributions made by Halfond and Orso [12], where tools centered around tracking and countering SQL injection attacks through a signature check were developed. Deep learning has proven highly useful at identifying vulnerabilities to SQL injection, achieving accuracies of up to 98.4 % on generic datasets [13]; however, these models are easily subverted by SQL injection vulnerabilities based on Boolean values, so they appear to follow logical operations to avoid naive string comparisons. (Pan) [14] introduce a deep learning framework to identify SQL injection vulnerability and claim to achieve higher accuracy in detecting vulnerability in general datasets, but their method does not identify vulnerable queries that are obfuscated using Boolean logic [14]. Similarly, Demilie and Deriba researched machine-learning classifiers, including random forest and support vector machines, to detect SQL injection. Their output proved its effectiveness in the recall and constituted high false positives in a dynamic web environment [15].

Although great strides may have been made in detecting malware, the current techniques lack contextual awareness and are therefore ineffective in real-time detection of subtle Boolean-based exploits, a conclusion that replicates other surveys on web vulnerability [4, 5].

B. LLM applications in security

The use of LLM in modern cybersecurity practices has proved to be a topic of special interest to researchers due to their ability to understand and evaluate code semantics [16]. Gui, Z., Wang, provides a survey of LLMs on code-related datasets, specifically the tasks in vulnerability detection; they state that models like the GPT variants make impressive results in detecting vulnerabilities such as buffer overflows and injection flaws [16]. Nafis Tanveer Islam introduces a pre-trained language modelling task defined over program source code that achieves outstanding results in semantic analysis of code and has already been tuned to an alternative task, API vulnerability detection [17]. A number of studies have considered how LLMs can be used in general SQL injection attacks. According to conventional thinking, even though LLMs have been effectively used to address a broad range of tasks, they are still hindered by high computing demands and the need to conduct domain-specific fine-tuning when applied in fields such as web security.

Overall, Studies in the relevant field of endeavour show that there were evident gaps in the mitigation of Boolean-based SQL injections. the literature reveals a gap in handling Boolean-based SQL injection, where traditional methods lack the semantic depth of LLMs, and current LLM applications often overlook fine-grained attack types or real-world scalability [10].

III. Methodology

This part explains the procedure used to evaluate deep seek-coder and Meta-LLaMA-3-70B-Instruct in identifying Boolean based SQL injection attacks. The aim is to build a reproducible, extensible, and closely aligned to the recent research practices concerning machine learning and cybersecurity assessment framework. The methodology will include preparation of the datasets, choice of the model, design of the detection mechanism, metrics of evaluation, and experimental setup.

The experimental testbed is a web-based application that provides a simulation of a real-world scenario in which SQL query execution and SQL injection attacks can be detected (Fig. 2). The application is developed with Python 3.13 version, and Flask framework is used to develop a web interface and application logic. The main database is the Jobs Database, which holds job announcements and associated records.

The LLMs are guided using a zero-shot, instruction-following prompt. This means that the models are not fine-tuned with labeled training examples for SQL injection detection, where each SQL query is embedded in a controlled instruction template. The goal is to evaluate whether the model could distinguish between benign and malicious (Boolean-based SQLi) queries. Specifically, each SQL query is inserted into a standardized instruction template that requires the model to analyze the query and make a binary decision. If the model determines that the query is legitimate, the system executes the query on the backend database and returns the appropriate results. Conversely, if the model classifies the query as an SQL injection attempt, the system blocks execution and returns only an explanatory message indicating why the query is identified as malicious. This prompt-driven workflow ensures consistency across models and allows for an evaluation of their ability to reliably distinguish between benign and malicious inputs in a realistic web application context. This prompt design created a decision-driven workflow in which the LLMs effectively acted as gatekeepers between the user input and the database execution layer.

Both LLMs are accessed and used via the OpenRouter API. The top-k value is set to 50 for meta-llama/llama-3-70b-instruct and 40 for deepseek-ai/deepseek-coder. The top-p value is set to 1.0 for both models. Additionally, to minimize the randomness of the models and ensure consistent and deterministic output, a common Temperature value of 0.1 is

selected. These settings are specifically focused on detecting SQL injection attempts in a realistic and reliable manner.

In SQL request, if the input is validated as safe, the query would be allowed to pass through to the database, and legitimate results (e.g., user records in the example) would be returned to the user. This Fig. 2 explains how legitimate queries are processed successfully through the same system.

The first process flow diagram depicts how a valid SQL request is executed when no malicious pattern is detected by the LLM.

Step 1: User Input Submission

The process begins with the user submitting a valid query through the input and response area.

Step 2: API Request Handling

The application backend forwards the request to the API, which communicates with the LLM. At this stage, the LLM is tasked with analysing the query.

Step 3: LLM Verification and Database Execution

Upon inspection, the LLM does not detect any SQL injection signatures in the query. Since the input is validated as safe, the request is executed against the database.

Step 4: Returning Results to the User

The database processes the query and returns the requested records to the LLM. The LLM then relays these results back to the application interface, where they are displayed in the response area for the user.

The second process flow diagram in Fig. 2 reflects a process workflow to detect and mitigate SQL injection (SQLi) attacks using the combination of LLMs. This workflow shows how the user input is intercepted, analyzed and filtered before any query is left to the database, thus the chances of exploitation are minimized.

Step 1: User Input Submission

The process begins when a user interacts with the application interface and submits an input query through a code input box. In the example provided, the malicious query is:

SELECT username FROM user WHERE id=3 OR '1'='1' --

This input has the conditional expression 1=1 and this is always true. Such expressions are widely recognized as a standard SQL injection technique used to bypass authentication and extract unauthorized data. The query is first received by the application backend.

Step 2: Request Processing via API and LLM

The request is passed through the application backend to the API, a middleware layer. The API does not transfer the query to the database directly but instead sends it to a LLM. In this stage the query is examined by the LLM in order to identify potentially malicious patterns. It has a duty to categorize the input as an SQL injection attempt or as a valid database query. With the help of natural language processing and contextual processing, the LLM detects suspicious conditions 1=1 in this example.

Step 3: Response Generation and Prevention of Database Compromise

When the LLM determines that the input is malicious, it blocks execution and prevents the query from reaching the database. Instead, the model generates a response message that is sent back to the user through the response area. This ensures that the database remains protected from unauthorized access or data leakage.

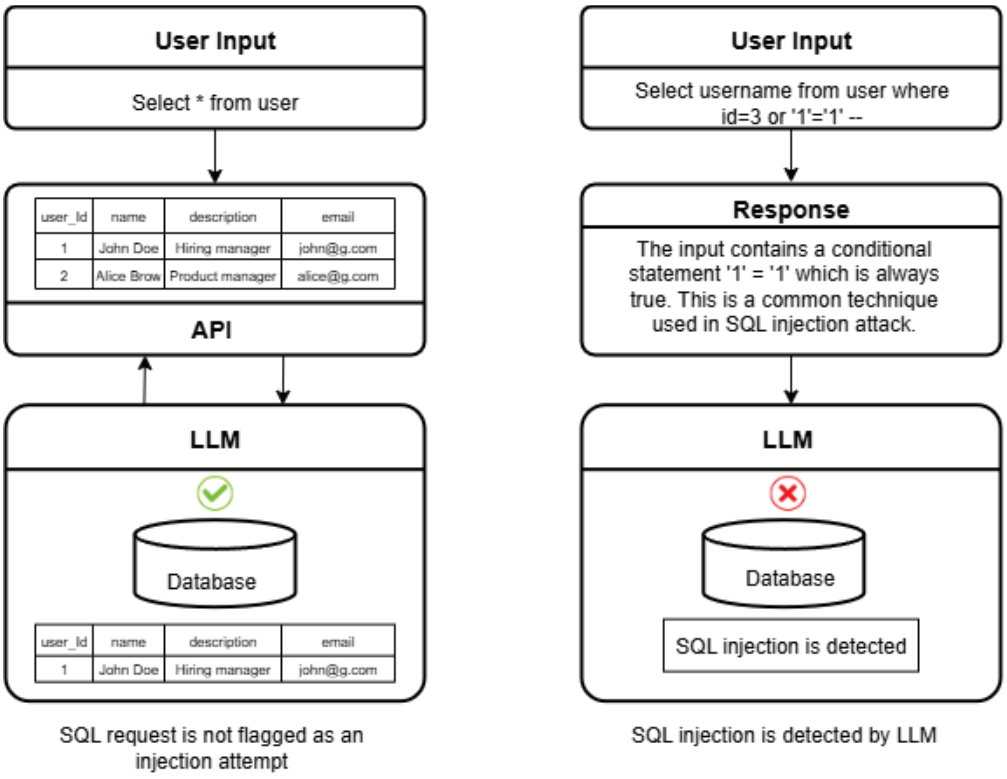

Fig. 2. LLM supported SQL query processing

A. Dataset preparation

Effective training and evaluation of systems aimed at detecting SQL injections requires the availability of a well-labelled, comprehensive dataset. We constructed a custom dataset comprising a set of labelled SQL queries, with a focus on Boolean-based attacks that manipulate true/false responses. This experimental structure allowed assessment of the detection potential of the pre-trained LLMs in a systematic way with the normal and injection-based queries. The dataset is fairly balanced, more than 350 queries were performed, including about 70 percent of the elements had malicious actions (e.g., using AND/OR and equal operators in queries) and 30 percent of the elements in the dataset had benign activities (e.g., typical database actions). SQLite was used as backend storage and as a support to the evaluation process through the Flask framework.

To generate samples, we used the SQLMap tool to create synthetic malicious queries, simulating real-world attack scenarios.

B. Model selection and adaptation

We selected the deep seek-coder and Meta-LLaMA-3-70B-Instruct models for their advanced capabilities in natural language processing and efficient handling of contextual tasks. Both of these models, like Deep Seek-Coder (large-scale transformers optimized for coding, evolution) and Meta-LLaMA-3-70B-Instruct with the instruction-tuned design, exhibit a reasonable trade-off between the rate of performance and resources. This especially brings them to web-security applications, e.g., real-time SQL injection detection in the query processing of an actual database.

C. Evaluation metrics

To rigorously assess the model's performance, we adopted standard metrics evaluations. These include:

- Accuracy: The proportion of correctly classified queries (both benign and malicious).
- Precision: The ratio of true positive detections to all positive predictions, critical for minimizing false alarms in web security.
- Recall: The ratio of true positive detections to all actual malicious queries, essential for capturing elusive Boolean-based attacks.
- F1-score: The harmonic means of precision and recall, providing a balanced measure.

These metrics were selected to address the high-stakes nature of SQL injection detection, where both false positives and negatives can have significant consequences.

IV. EXPERIMENTAL STUDIES

In accordance with the methodology of Section III, this part shows the practical results of the Meta-LLaMA-3-70B-Instruct and Deep Seek-Coder models when tested in the context of a practical study. In this practical setup, the system was integrated into a Python-based web project for job-related applications. Standard SELECT statements, which would have been associated with standard user requests, were run to make sure proper retrieval and display of records were done. In contrast, the SQL injection queries, especially the ones using Boolean logic (AND, OR) to process queries and retrieve confidential information, were blocked and detected by filtering. Both models were applied in their pre-trained model, without any extra fine-tuning or modification of the structure, as specified with the implementation of the user.

Both models categorized the incoming queries based on the patterns that they had learned, applying a validation prompt-based mechanism to classify between legitimate and malicious requests. These models recognized a query as valid, and the LLM performed it and provided the relevant information and presented the relevant data on a designated database.

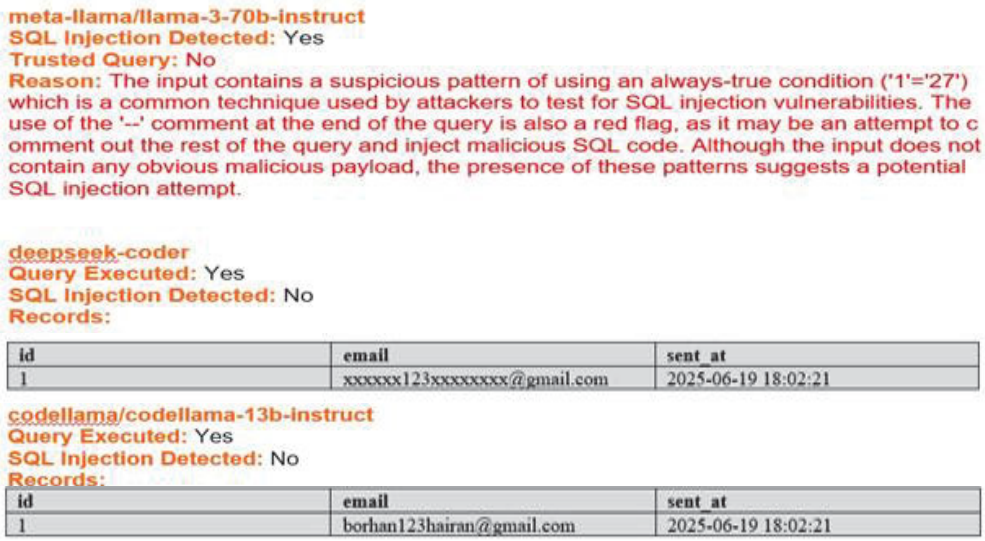

Fig. 3. Boolean base attack example

On the other hand, all the queries, which proved to be SQL injection attacks, would be prevented from executing. When this was the case, the model produced an explanatory message that pointed to the decision of a non-execution as well as the categorization of the query as a possible SQL injection attack.

In the example above (Fig. 3), an SQL injection query was used: select id, email, sent at from contact where id =1' or '1'='27' –

As it was shown in the example, both models demonstrate different behaviour to the same query, whereas in the real world, it is an SQL injection attempt. The Meta-LLaMA-3-70B-Instruct model was able to detect the query as being malicious, but the deep seek-coder was unable to detect the attack as an injection one. Rather, it categorized the request as correct and valid, and proceeded to execute the request, then showed the related information from the database.

In Fig. 4, legitimate query was used: select id, salary, location from job where id =1.

The performance and reaction of both models to a valid query are shown in Fig. 4 as an example, demonstrating different responses to the same query. Although this query is a valid query, the first model did not identify it correctly and considered it as an injection query, whereas the second model recognized it as a valid and legitimate query and displayed the corresponding data from the database.

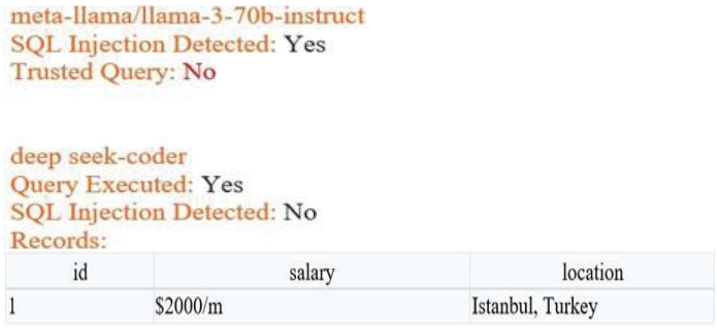

Fig. 4. Legitimate query

The most critical point is that, regardless of whether a query is valid or an SQL injection, if it is executed multiple times at different intervals, the models may sometimes produce inconsistent responses. For example, a query that was initially classified as an injection may, at another time, be classified as a valid query and consequently retrieve and display data from the database. This inconsistency constitutes a security vulnerability in itself.

V. RESULTS

The experimental analysis compared the detection of SQL injection attacks using Boolean methods of Meta-LLaMA-3-70B-Instruct and DeepSeek-Coder on a balanced set of legitimate and malicious queries. The evaluation of the performance was conducted in terms of precision, recall, F1-score, and accuracy (Table 1). These results indicate strong recall capabilities, meaning the model effectively captured a high proportion of actual SQL injection attempts.

TABLE I. PERFORMANCE OF MODELS

Model	Precision (%)	Recall (%)	F1 Score (%)	Accuracy (%)
Meta-LLaMA-3-70B-Instruct	68.75	**88.00**	**77.42**	**74.00**
Deep Seek-Coder	**71.50**	44.00	57.17	60.00

These findings bring to focus an important trade-off: Meta-LLaMA-3-70B-Instruct can be better at detecting more malicious queries, but DeepSeek-Coder is more reserved in the classification, and not as efficient in picking up all threats

VI. DISCUSSION

The results obtained highlight both promise and limitations in using LLMs for automated security analysis. Their relatively high levels of precision under both models make them useful as complementary tools in Integrated Development Environments (IDEs) or FRP-based continuous integration systems, where it is useful to produce as few false positives as possible in order to avoid developer burnout. Nevertheless, the lower recall, especially in DeepSeek-Coder, shows that one should not rely exclusively on these types of models to provide a high level of security in any production environment, as most of the attacks may go unnoticed.

It can be observed that there is an inconsistency in classification across multiple runs on the same query. In some other instances, a query that was considered to be malicious was changed to a genuine query by the same model, and this may result in security flaws. Such instability means susceptibility to immediate context or token-level differences that have to be fixed before safety in critical systems.

Moreover, Meta-LLaMA-3-70B-Instruct was found to be more flexible in detecting obfuscated forms of attack, maybe because it contains higher parameters and instruction-tuning data. Alternatively, the increased accuracy of DeepSeek-Coder implies that it might be more applicable to cases where it is crucial to reduce false positive detections, as opposed to ensuring detection against all attacks.

Implications:

- LLMs show promise as first-line security filters or developer assistants.
- For operational deployment, they should be paired with traditional static and dynamic analysis tools.
- Fine-tuning on domain-specific security datasets could substantially improve recall without degrading precision.

VII. CONCLUSIONS

This paper performed a comparison test of Meta-LLaMA-3-70B-Instruct and DeepSeek-Coder on detecting SQL injection attacks that are based on Boolean. Findings indicate that the Meta-LLaMA-3-70B-Instruct has better recall and overall accuracy and is best able to detect more varieties of malignant queries. DeepSeek-Coder is a little more accurate, lowering the false positive rate, but failing to detect a large percentage of the attacks.

Although both models also hold promise as part of secure coding processes, neither model is currently sufficiently consistent to be used as an autonomous security mechanism. Their best-suited place is in multi-layered defence architectures, in which they serve together with traditional vulnerability scanners and in-person code reviews.

REFERENCES

[1] O. Foundation. "OWASP Top Ten 2025." https://owasp.org/www-project-top-ten/ (accessed 02.09.2025, 2025).

[2] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: techniques, datasets and challenges," *Cybersecurity,* vol. 2, no. 1, pp. 1-22, 2019.

[3] M. Qbea'h, S. Alrabaee, M. Alshraideh, and K. E. Sabri, "Diverse approaches have been presented to mitigate sql injection attack, but it is still alive: a review," in *2022 International Conference on Computer and Applications (ICCA)*, 2022: IEEE, pp. 1-5.

[4] S. Yaswanthraj, A. M, K. S, and J. R, "SQL Injection and Prevention," *International Journal of Research Publication and Reviews,* 2024.

[5] M. Meenakshi and D. Murugan, "Understanding the Threat: Exploring SQL Injection Attacks and Prevention Strategies," *International Research Journal of Modernization in Engineering Technology and Science,* 2024.

[6] J. Zhang, Y. Zhou, B. Hui, Y. Liu, Z. Li, and S. Hu, "TrojanSQL: SQL Injection against Natural Language Interface to Database," Singapore, December 2023: Association for Computational Linguistics, in Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pp. 4344-4359, doi: 10.18653/v1/2023.emnlp-main.264.

[7] D. Coder. "Deepseek-Ai. ." https://github.com/deepseek-ai/DeepSeek-Coder (accessed 02.09.2025, 2025).

[8] C. Cetin, *Authentication and SQL-Injection Prevention Techniques in Web Applications*. University of South Florida, 2019.

[9] H. Zhao *et al.*, "Explainability for large language models: A survey," *ACM Transactions on Intelligent Systems and Technology,* vol. 15, no. 2, pp. 1-38, 2024.

[10] V. Usha, N. C. Abhinash, S. N. Chowdary, V. Sathya, and E. R. Reddy, "Enhanced Database Interaction Using Large Language Models for Improved Data Retrieval and Analysis," in *2024 Second International Conference on Intelligent Cyber Physical Systems and Internet of Things (ICoICI)*, 2024: IEEE, pp. 1302-1306.

[11] S. Mishra, "SQL injection detection using machine learning," 2019.

[12] W. G. Halfond and A. Orso, "Combining static analysis and runtime monitoring to counter SQL-injection attacks," in *Proceedings of the third international workshop on Dynamic analysis*, 2005, pp. 1-7.

[13] M. M. Hassan, R. B. Ahmad, and T. Ghosh, "SQL injection vulnerability detection using deep learning: a feature-based approach," *Indonesian Journal of Electrical Engineering and Informatics (IJEEI),* vol. 9, no. 3, pp. 702-718, 2021.

[14] Y. Pan *et al.*, "Detecting web attacks with end-to-end deep learning," *Journal of Internet Services and Applications,* vol. 10, no. 1, pp. 1-22, 2019.

[15] W. B. Demilie and F. G. Deriba, "Detection and prevention of SQLI attacks and developing compressive framework using machine learning and hybrid techniques," *Journal of Big Data,* vol. 9, no. 1, p. 124, 2022.

[16] Z. Gui *et al.*, "SqliGPT: Evaluating and Utilizing Large Language Models for Automated SQL Injection Black-Box Detection," *Applied Sciences,* vol. 14, no. 16, p. 6929, 2024.

[17] N. T. Islam, M. B. Karkevandi, and P. Najafirad, "Code security vulnerability repair using reinforcement learning with large language models," *arXiv preprint arXiv:2401.07031,* 2024.