CodeBERT Detection Pipeline for SQL Injection Detection: A Comparison with other ML Models

Leonardo Antonetti da Motta[1], **Alan Demétrius Baria Valejo**[2], **Lina Garcés**[1]

[1] Laboratório de Engenharia de Software (LabES)
Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP)
CEP: 13566-590 - São Carlos - SP, Brazil

l.a.motta@usp.br, linagarces@usp.br

[2]Departamento de Computação (DC)
Universidade Federal de São Carlos (UFSCar).
CEP: 13565-905 - São Carlos - SP, Brazil

alanvalejo@ufscar.br

Abstract. *SQL injection (SQLi) remains a prevalent web application vulnerability, while existing detection approaches often struggle to generalise across obfuscated and long SQL payloads. This paper presents* ***CodeBERT-SQLi****, a CodeBERT-based detection pipeline that adapts transformer models to arbitrary-length queries through sliding-window tokenisation and majority-vote aggregation. We compile and standardise a large-scale SQLi corpus from ten public sources to produce a unified dataset of 12.7 million labelled queries. We then conduct a comparative evaluation involving CodeBERT-SQLi, untuned BERT and CodeBERT, and traditional ML baselines (Logistic Regression, Random Forest, Linear SVM). Experiments on the complete 1.9-million-query test partition show that CodeBERT-SQLi achieves MCC = 0.947 and AUC = 0.971, outperforming all compared methods. The results further reveal that code-domain pre-training (Δ MCC = 0.059) contributes more to performance gains than task-specific fine-tuning (Δ MCC = 0.026), highlighting representation quality as the key factor in SQLi detection.*

Keywords: SQL Injection, CodeBERT, Web Application Security, Vulnerability Detection, Security Testing, Machine Learning

1. Introduction

SQL injection (SQLi) consistently ranks among the most exploited web application vulnerabilities, appearing in every edition of the OWASP Top Ten Most Critical Risks [OWASP Foundation 2021]. By embedding malicious SQL syntax into user-supplied input, an attacker can read, modify, or delete database contents without authorisation. Despite decades of research, SQLi remains prevalent because the attack surface is large and detection methods struggle to generalise across the variety of obfuscation strategies available to attackers [Halfond et al. 2006, Ross et al. 2018].

Existing detection approaches generally fall into two families: rule- and signature-based systems (WAFs) [OWASP Foundation 2024], which match input against curated

rule sets, and feature-engineering classifiers that extract character-level statistics such as query length, keyword counts, and Shannon entropy to train models such as Logistic Regression or Random Forest [Joshi and Geetha 2014, Hasan et al. 2019]. Although these methods can capture superficial lexical patterns, they often struggle to generalise across obfuscated payloads and structurally complex queries, as their performance depends on the quality of hand-designed features.

Large language models pre-trained on source code, particularly CodeBERT [Feng et al. 2020], have demonstrated strong capability in learning syntactic and semantic code representations for software engineering tasks [Hanif and Maffeis 2022, Fu and Tantithamthavorn 2022]. However, it remains unclear whether such representations transfer effectively to SQL injection detection under large-scale and heterogeneous evaluation settings. Furthermore, prior studies typically rely on small or private datasets and employ inconsistent evaluation protocols [Joshi and Geetha 2014, Hasan et al. 2019, Alghawazi et al. 2023, Farea et al. 2021, Ben Ammar and Alharbi 2025], making it difficult to isolate the effects of code-domain pre-training, task-specific fine-tuning, and traditional feature engineering.

Contributions

This work addresses the lack of large-scale and controlled evaluations of transformer-based approaches for SQL injection detection. Our main contributions are:

- We propose **CodeBERT-SQLi**, a CodeBERT-based SQLi detection pipeline capable of handling arbitrarily long SQL queries through sliding-window tokenisation and majority-vote aggregation. Unlike conventional transformer classifiers limited by fixed input length, this approach preserves contextual information from long and obfuscated payloads commonly found in real SQLi attacks.
- We compile, standardise, and deduplicate a large-scale SQLi corpus from ten public sources, yielding approximately 12.7 million labelled queries. Beyond aggregating existing data, the compilation required schema normalisation, cross-source deduplication, and consistent label harmonisation.
- We conduct a controlled comparative evaluation across six models from three methodological families: fine-tuned CodeBERT-SQLi, frozen-encoder linear probes (BERT-base, CodeBERT-base), and traditional ML classifiers (Logistic Regression, Random Forest, SVM).

2. Related Work

Web application firewalls (WAFs) detect SQLi by matching input against curated regular-expression patterns; the OWASP Core Rule Set [OWASP Foundation 2024] deployed via ModSecurity is the canonical example. These systems offer near-zero latency and require no labelled data but do not generalise to payloads absent from their rule sets. As SQLi payloads became increasingly obfuscated and structurally complex, research gradually shifted from static signatures toward representation-learning approaches capable of modelling contextual token dependencies.

A large body of work applies supervised classifiers to character-level statistics such as query length, SQL keyword frequency, Shannon entropy, and punctuation ratios.

Joshi and Geetha [Joshi and Geetha 2014] and Hasan et al. [Hasan et al. 2019] demonstrate SVM and tree-based classifiers achieving high accuracy on public SQLi benchmarks. Shar and Tan [Shar and Tan 2013] use Naïve Bayes and Multilayer Perceptron to predict both SQLi and XSS vulnerabilities by mining input-sanitisation patterns from source code. Tripathy et al. [Tripathy et al. 2020] apply Random Forest to SQLi detection in cloud SaaS environments, reporting near-98% accuracy. Ross et al. [Ross et al. 2018] analyse multi-source datasets and find that query length, punctuation count, and distinct-byte counts are the most informative features. These models are interpretable and fast, but their generalisation depends critically on the feature set: novel obfuscation shifts the feature distribution without retraining.

Neural architectures have also been applied. Alghawazi et al. [Alghawazi et al. 2023] report an RNN autoencoder for binary SQLi detection (94% accuracy, F1 = 0.92); Farea et al. [Farea et al. 2021] apply a bidirectional LSTM to multiclass SQLi and XSS detection (99.26% accuracy). Liu et al. [Liu et al. 2020] use a Transformer to *generate* test inputs rather than classify payloads, outperforming SQLmap on real applications.

Although recurrent architectures improved sequential modelling compared with hand-crafted approaches, they still struggle with long-range dependencies and highly obfuscated payload structures commonly found in SQLi attacks.

Pre-trained language models have been applied to source-code vulnerability detection more broadly. VulBERTa [Hanif and Maffeis 2022] adapts RoBERTa by pre-training it on a large C/C++ function corpus for vulnerability classification, showing that code-domain pre-training outperforms general-English baselines on NIST SARD and NVD benchmarks. LineVul [Fu and Tantithamthavorn 2022] fine-tunes CodeBERT for line-level vulnerability prediction in C/C++ projects. Both studies evaluate on compiled source code; neither targets SQLi at the input-string level.

The closest prior work to ours is Ben Ammar and Alharbi [Ben Ammar and Alharbi 2025], who also fine-tune CodeBERT for binary SQLi detection. However, their study is limited in several ways that prevent a direct comparison. Training uses a single 30,919-row dataset — approximately 400 times smaller than the 12.7-million-query corpus assembled here — raising questions about generalisation to diverse payloads and benign traffic patterns. Evaluation is similarly small-scale and reports only accuracy and F1, omitting MCC and AUC, which are necessary for assessing performance under class imbalance. Queries exceeding the 512-token transformer limit are handled by simple truncation, discarding potentially malicious tail regions; no sliding-window strategy is applied. Finally, the study does not include any baseline comparison that isolates the contribution of code-domain pre-training from task-specific fine-tuning, leaving unanswered whether CodeBERT's advantage over general BERT stems from its pre-training objective or from the fine-tuning data alone. Our work addresses each of these gaps.

Unlike natural-language text, SQL queries follow rigid syntactic and structural rules that resemble programming languages, suggesting that representations learned from source code may transfer effectively to SQLi detection. However, it remains unclear whether code-oriented transformer representations generalise effectively to SQLi detec-

tion under controlled large-scale evaluation settings. In addition, prior studies often rely on heterogeneous datasets and inconsistent protocols, limiting fair comparisons across model families.

To our knowledge, no prior study (i) fine-tunes a code-oriented LLM for binary SQLi detection on a large-scale, deduplicated multi-source corpus (12.7 million queries from ten sources), (ii) isolates the effects of code-domain pre-training versus task-specific fine-tuning via controlled linear probing, and (iii) benchmarks transformer-based and traditional ML approaches under the same unified evaluation protocol. This paper addresses all three gaps.

3. Methodology

3.1. Dataset Compilation

We assembled a corpus from ten publicly available SQLi datasets [Mullick et al. 2026, Quetel et al. 2025, Kaldi 2024, Issakhani et al. 2023a, Sajid 2021, Mathew 2021, Yu 2023, Trinity 2022, Rayten 2024, Hussain 2021], summarised in Table 1.

Table 1. Dataset sources and their contribution to the unified SQLi corpus. Raw size is the original row count before any deduplication. *Fully subsumed by higher-priority sources after cross-source deduplication; contributed zero rows.

Source	Type	Raw Size	Final Contrib.	Mal.%	Deduped
RbSQLi [Mullick et al. 2026]	Synthetic (rule-based)	10,190,450	10,190,450	26.5	No
Superviz25-SQL [Quetel et al. 2025]	Semi-synthetic	3,688,977	2,292,919	13.6	No
SQL Inj. Dataset [Kaldi 2024]	Mixed (incl. NL text)	163,476	163,424	47.6	No
BCCC-SFU-SQLInj [Issakhani et al. 2023a]	Adversarial	11,011	11,011	100.0	No
SQL Inj. Dataset [Rayten 2024]	Synthetic	122,982	74,198	≈50	No
Modified SQL [Sajid 2021]*	Synthetic	30,919	0	—	Yes
SQL-data [Mathew 2021]*	Mixed	37,961	0	—	Yes
Biggest SQLi [Yu 2023]*	Synthetic	148,326	0	—	Yes
SQLi Extended [Trinity 2022]*	Synthetic	109,518	0	—	Yes
SQLiV3 [Hussain 2021]*	Synthetic	30,609	0	—	Yes
Total		**14,534,229**	**12,732,002**	**24.5**	

The five contributing sources span complementary collection strategies:

RbSQLi [Mullick et al. 2026] 10.19 million synthetic queries covering six attack categories (Union-based, Boolean-based, Time-based, Error-based, Stackqueries-based, and Meta-based), generated from public payload repositories and AI-assisted mutation; additionally provides per-query annotations for attack category, SQL command, and target table.

Superviz25-SQL [Quetel et al. 2025] 3.69 million semi-synthetic queries (2.29 million after within-source deduplication) from Télécom Paris, LAAS CNRS, and Inria; benign queries represent realistic workloads and malicious samples carry detailed attack-stage and tamper-method metadata.

SQL Inj. Dataset (Kaldi) [Kaldi 2024] 163K queries whose benign class notably includes natural-language text (e.g., news snippets) rather than SQL, serving as a proxy for realistic production traffic.

BCCC-SFU-SQLInj-2023 [Issakhani et al. 2023a] 11K adversarially crafted malicious queries generated by a genetic algorithm optimised to evade detection [Issakhani et al. 2023b]; all rows are malicious by design, making this the hardest subset of the malicious class.

SQL Inj. Dataset (Rayten) [Rayten 2024] 122K synthetic queries distributed as a pre-split dataset; the original split assignment is discarded and rows are folded into the global stratified split.

Compilation proceeds in two passes. First, each source is normalised to a common schema with three fields: `query`, `label`, and `source`. Second, a cross-source deduplication pass retains only the first occurrence of each query by exact text match, using a fixed source priority order; rows with null or sub-two-character queries are dropped. Cross-source deduplication is particularly important in SQLi research because many public datasets partially overlap or reuse canonical attack payloads; without it, train-test leakage may artificially inflate performance estimates. Five of the ten sources were fully subsumed and contributed zero rows; the remaining five yield approximately **12.7 million labelled queries** (75.5% benign, 24.5% malicious).

Beyond the mandatory `query` and `label` fields, the compiled corpus preserves auxiliary metadata from sources that provide it. The dominant source, RbSQLi [Mullick et al. 2026], contributes per-query annotations including `injection_type` (attack category: Union-based, Boolean-based, Time-based, Error-based, Stackqueries-based, and Meta-based), `sql_command`, `target_table`, and `selected_columns`. These fields are not consumed by the BERT models evaluated in this work but are retained in the corpus to support future research, such as per-attack-category performance breakdowns.

The majority of the corpus (98.6%) originates from synthetic or semi-synthetic sources, which is consistent with the broader SQLi dataset landscape where real annotated production traffic is largely unavailable. Generalisation to live production traffic remains an open question and is discussed as a limitation in Section 4.

The corpus is split into training (70%), validation (15%), and test (15%) partitions using stratified sampling by *(source, label)* pair with fixed random seed 42, yielding approximately 8.86M training, 1.9M validation, and 1.9M test rows.

In summary, our compiled corpus combines queries originating from multiple public sources with different collection strategies, formatting conventions, and attack styles, resulting in substantial lexical and structural heterogeneity. This diversity is important for reducing source-specific bias and improving the robustness of the evaluation across distinct SQLi patterns and benign query distributions.

3.2. Proposed CodeBERT-SQLi Detection Pipeline

We fine-tune `microsoft/codebert-base` (125M parameters; [Feng et al. 2020]) as a binary sequence classifier using CodeBERT's built-in tokeniser.

SQLi payloads may contain nested queries, stacked statements, long obfuscation chains, and injected fragments distributed across distant parts of the query, frequently exceeding BERT's 512-token context limit. Direct truncation would discard potentially malicious regions and compromise detection reliability. We therefore apply a *sliding-window* strategy: each query is split into overlapping windows of 510 content tokens

(plus `[CLS]` and `[SEP]`). A stride of 256 tokens provides 50% overlap between adjacent windows, reducing the risk of splitting malicious patterns across window boundaries while preserving sufficient contextual continuity between consecutive segments.

During training, each window is treated as an independent labelled example. At inference, the query-level label is determined by *majority vote* over all windows, which improves robustness by reducing sensitivity to isolated window-level prediction errors and allowing the final decision to reflect the dominant behaviour across the entire query; window-level probabilities are averaged for AUC computation. Figure 1 illustrates the full pipeline.

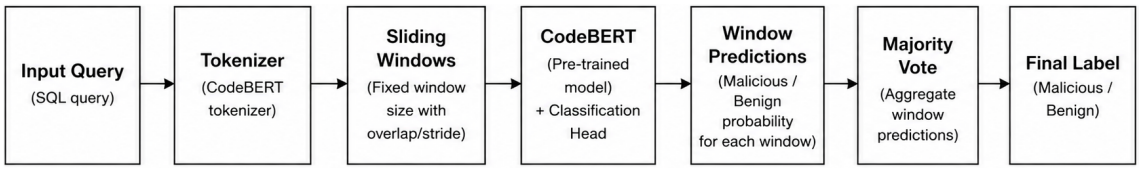

Figure 1. Proposed CodeBERT-SQLi detection pipeline. The input SQL query is tokenised and divided into overlapping sliding windows to handle arbitrarily long payloads. Each window is independently processed by the fine-tuned classifier; window-level predictions are aggregated through majority voting to produce the final query label.

Implementation. Listing 2 presents the core implementation of the two-phase pipeline. The upper block constructs a flat index of `(query_idx, window_start)` pairs. Queries that fit within a single 510-token window receive one entry; longer queries are iterated with a 256-token stride until the last window covers the end of the token sequence. Each window is assembled by slicing the pre-encoded token array, prepending `[CLS]`, appending `[SEP]`, and padding to 512 tokens. The lower block groups window-level model outputs by their source query and applies majority vote: a query is labelled malicious if more than half of its windows predict class 1. Window-level softmax probabilities for class 1 are averaged across windows to produce a continuous score used for AUC computation.

Class imbalance is handled by 2:1 undersampling of the majority class (benign) in the training split only, combined with a weighted loss that penalises missed malicious queries more heavily. Training uses AdamW [Loshchilov and Hutter 2019] (lr $= 2\times10^{-5}$, batch size 16, 3 epochs) on the full 8.86-million-row training partition. Fine-tuning ran on a single NVIDIA A100 40 GB GPU for approximately 108 hours. All experiments use a single fixed random seed (42).

3.3. Comparison Baselines

To isolate the contribution of code-domain pre-training from task-specific fine-tuning, we evaluate `bert-base-uncased` [Devlin et al. 2019] and `microsoft/codebert-base` using a *linear probing* protocol [Alain and Bengio 2017]: each model's encoder is frozen as a fixed feature extractor, producing a single vector per query (via the `[CLS]` token) without updating any weights. A logistic regression classifier is then trained on these vectors using a 50,000-row stratified subsample (seed 42).

```
CONTENT = MAX_TOKEN_LENGTH - 2  # 510 content tokens

# Build flat index: one (query_idx, window_start) per window
index = []
for q_idx, ids in enumerate(token_id_arrays):
    if len(ids) <= CONTENT:
        index.append((q_idx, 0))
    else:
        start = 0
        while True:
            index.append((q_idx, start))
            if start + CONTENT >= len(ids): break
            start += WINDOW_STRIDE  # 256 tokens; 50% overlap

# Retrieve window i: slice tokens, add specials, pad to 512
chunk = token_ids[q_idx][start : start + CONTENT]
ids   = [CLS] + chunk + [SEP] + [PAD] * (512 - len(chunk) - 2)
```

```
# Collect per-window predictions grouped by source query
for label, pred, prob, q_idx in zip(labels, preds, probs, q_idxs):
    preds_by_q[q_idx].append(pred)
    probs_by_q[q_idx].append(prob)

# Majority vote for classification; mean probability for AUC
for q_idx in sorted(label_by_q):
    votes      = preds_by_q[q_idx]
    final_pred = 1 if sum(votes) > len(votes) / 2 else 0
    final_prob = sum(probs_by_q[q_idx]) / len(probs_by_q[q_idx])
```

Figure 2. Core sliding-window implementation. *Top:* window index construction and per-window token assembly (`MAX_TOKEN_LENGTH` = 512, `WINDOW_STRIDE` = 256). *Bottom:* majority-vote query-level aggregation; window probabilities are averaged for AUC computation.

Three traditional ML classifiers are included as baselines: Logistic Regression, Random Forest, and Linear SVM with probability calibration. All three are trained on 18 hand-crafted features from a pre-engineered dataset [Alyasiri and Al-Araji 2025]. This feature schema follows prior work on character-level SQLi classifiers [Joshi and Geetha 2014, Hasan et al. 2019, Alyasiri and Al-Araji 2025]. The selected hand-crafted features were chosen based on prior SQLi detection literature and include lexical, syntactic, and statistical characteristics commonly associated with malicious payloads such as length, operator usage, punctuation patterns, and entropy-based descriptors. These features provide strong traditional baselines while remaining computationally inexpensive and interpretable. Baselines are trained on a 100,000-row stratified sample (2:1 benign-to-malicious ratio, seed 42), yielding approximately 73,000 effective training examples.

3.4. Evaluation Protocol

All models are evaluated on the complete 1.9-million-row test partition (approx. 75.5% benign / 24.5% malicious, seed 42). The primary metric is the Matthews Correlation Coefficient (MCC; 1 = perfect, 0 = random), which accounts for all four cells of the confusion

matrix and is robust to class ratio [Chicco and Jurman 2020]. Secondary metrics are AUC (discrimination across decision thresholds), F1, Accuracy, Precision, and Recall. Results are reported for a single run with seed 42; confidence intervals from repeated runs are not available.

4. Results Analysis and Discussion

Table 2. Model comparison on the complete 1.9-million-row test partition (75.5% benign / 24.5% malicious, natural class distribution, seed 42). †Used without fine-tuning: frozen encoder, logistic classifier trained on 50K-row subsample. MCC is the primary metric.

Model	MCC	AUC	F1	Acc.	Prec.	Rec.
CodeBERT-SQLi (fine-tuned)	**0.947**	**0.971**	**0.951**	**0.976**	**0.956**	**0.946**
CodeBERT-base†	0.921	0.956	0.929	0.966	0.938	0.921
BERT-base†	0.862	0.926	0.874	0.939	0.888	0.861
Logistic Regression	0.754	0.878	0.789	0.902	0.832	0.750
Random Forest	0.747	0.893	0.782	0.901	0.846	0.727
SVM (Linear)	0.743	0.868	0.779	0.896	0.814	0.746

This section evaluates whether the proposed CodeBERT-SQLi pipeline improves SQL injection detection compared with traditional feature-engineering approaches and general-language transformers. Table 2 summarises the performance of all evaluated models under the same large-scale test partition, using MCC as the primary metric.

The most informative comparison in Table 2 is between the two probed controls: BERT-base (MCC = 0.862) and CodeBERT-base (MCC = 0.921). Both are evaluated under identical condition so the 0.059-point gap is attributable entirely to the difference in pre-training objectives. The additional gain from full fine-tuning ($\Delta = 0.026$) is meaningful but smaller, suggesting that code-domain representations are more transferable to SQLi detection than general-English ones.

All three ML baselines cluster between MCC = 0.743 and MCC = 0.754, all outperformed by both pre-trained-only LLM models (MCC = 0.862–0.921). The 18 hand-crafted character-level features, while informative, do not fully capture the token-sequence patterns distinguishing SQLi from benign traffic. The fine-tuned model's 0.193-point MCC advantage over the best ML baseline requires no feature engineering but trades off inference speed: ML baselines complete 1.9M classifications in under 5 s on CPU, while fine-tuned model inference requires approximately 2.4 hours on a single A100 GPU. For offline security auditing or batch scanning in automated pipelines this latency is acceptable; real-time WAF integration would require model compression before deployment.

The large performance gap between transformer-based approaches and hand-crafted feature models suggests that SQL injection detection is fundamentally a structural sequence-modelling problem rather than a purely lexical classification task. While traditional ML baselines rely on shallow statistical descriptors, transformer representations are substantially more effective at capturing contextual relationships between SQL operators, clauses, and obfuscation patterns. This behaviour also helps explain the strong performance difference between BERT-base and CodeBERT-base: SQL queries are structurally closer to programming languages than to natural language.

The proposed sliding-window and majority-vote strategy also contributes to the reported performance. Instead of truncating long SQL payloads to the transformer context limit, CodeBERT-SQLi preserves broader contextual coverage across obfuscated and structurally complex queries, improving robustness in long-query classification scenarios.

4.1. Limitations

There are some limitations in the current work state: *(i) Dataset composition:* majority of the corpus originates from two synthetic-style sources, while organic queries were underrepresented. Although the synthetic sources are valid and were used for other research as well, significant updates in attack strategies might alter performance in a live environment. *(ii) Single random seed:* all training, sampling, and evaluation use seed 42; variance across seeds was not measured. Different seeds can give us variation in results. *(iii) Asymmetric training data:* the fine-tuned model was trained on 8.86M rows while ML baselines used a 100K-row sample, a factor of roughly 88.6, close to two orders of magnitude; ML baselines given the full 8.86M rows might reach higher performance than observed. *(iv) No external data validation:* the model was ultimately tested on the same corpus it was trained on, albeit with a held-out partition; generalisation to external datasets remains untested. This would be an important next step.

5. Conclusion

This paper presented a CodeBERT-based pipeline for SQL injection detection capable of handling arbitrarily long SQL queries through a sliding-window and majority-vote aggregation strategy. In addition to the proposed detection pipeline, we introduced a large-scale unified SQLi corpus compiled from ten public sources, including schema normalization and cross-source deduplication, and conducted a controlled large-scale benchmark involving fine-tuned transformers, frozen-encoder probes, and traditional machine learning baselines under a unified evaluation protocol. Together, these contributions provide a reproducible benchmark resource for future SQLi detection research.

The proposed model achieves $\mathrm{MCC} = 0.947$ and $\mathrm{AUC} = 0.971$, outperforming all three traditional ML baselines and both untuned LLM controls. The primary finding is that code-domain pre-training ($\Delta\,\mathrm{MCC} = 0.059$) contributes more to SQLi detection performance than task-specific fine-tuning ($\Delta\,\mathrm{MCC} = 0.026$), identifying representation quality as the key leverage point for future classifier design. However, fine-tuning on task specific data still provides a boost to the evaluated metrics, and the best performance for this SQLi task is achieved by combining both pre-training and fine-tuning.

Note, the proposed pipeline extends standard transformer-based classification by adapting CodeBERT to the characteristics of SQL injection payloads. The sliding-window strategy enables the processing of arbitrarily long queries beyond the native transformer context limit, while the majority-vote aggregation mechanism improves robustness across obfuscated and structurally heterogeneous payloads. Combined with task-specific fine-tuning, these adaptations allow the model to preserve contextual information that is often lost in traditional feature-engineering approaches.

Future work includes: **(i)** officially publicize model and dataset to be available as free software; **(ii)** multi-seed evaluation to quantify result variance and provide confidence intervals for reported metrics; **(iii)** evaluation on real production SQL traffic logs to

assess generalisation beyond the collected training distribution; **(iv)** comparison of energy and training time cost between simple models (millions of weights, BERT-like) to large models (billions of weights, GPT-like).

Artifact Availability

The dataset compilation scripts, fine-tuning code, evaluation pipeline, and comparison artefacts produced in this study are publicly available at `https://github.com/l-a-motta/sbseg2026-codebert-sqli`, including instructions to rebuild the full corpus from its ten public sources.

Use of Artificial Intelligence Tools

The authors acknowledge the use of Artificial Intelligence (AI)-based tools exclusively to support the research and manuscript preparation process. AI systems were employed for language revision, text refinement, organization of experimental materials, and software engineering support tasks.

Specifically, Claude Opus (Anthropic) was used as an auxiliary tool to assist in the creation of test and sanity-check scripts, generation of log files, preparation of commit messages, and note-taking. AI tools were also used to improve the readability and linguistic quality of the manuscript.

The use of these tools did not replace the authors' scientific judgment. All research questions, methodological decisions, experimental design, implementation, data collection, analysis, interpretation of results, and conclusions were conceived, validated, and approved by the authors. The authors assume full responsibility for the accuracy, originality, and integrity of the work presented in this manuscript.

The authors declare no conflicts of interest related to the use of AI tools in this study.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, through a CAPES/PROEX Master's scholarship (Grant number. 88887.924539/2023-00) granted to the first author. The authors also thank the PRPI-USP (Grant number: 22.1.09345.01.2), the São Paulo Research Foundation (FAPESP) (Grant number 22/03090-0), and the Brazilian National Council for Scientific and Technological Development (CNPq) (Grants 420025/2023-5 and 406941/2025-4).

References

Alain, G. and Bengio, Y. (2017). Understanding intermediate layers using linear classifier probes. In *ICLR Workshop*.

Alghawazi, M., Alghazzawi, D., and Alarifi, S. (2023). Deep learning architecture for detecting SQL injection attacks based on RNN autoencoder model. *Mathematics*, 11(15):3286.

Alyasiri, H. and Al-Araji, Z. H. (2025). SQL injection attack dataset. Mendeley Data, v1. License: CC BY 4.0.

Ben Ammar, B. and Alharbi, A. M. (2025). SQL injection detection using fine-tuned CodeBERT. *Engineering, Technology & Applied Science Research*, 15(5):27852–27857.

Chicco, D. and Jurman, G. (2020). The advantages of the matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genomics*, 21(1):6.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186. Association for Computational Linguistics.

Farea, A. A., Wang, C., Farea, E., and Alawi, A. B. (2021). Cross-site scripting (XSS) and SQL injection attacks multi-classification using bidirectional LSTM recurrent neural network. In *2021 IEEE International Conference on Progress in Informatics and Computing (PIC)*, pages 358–363. IEEE. IEEE Xplore doc. 9687064. `https://ieeexplore.ieee.org/document/9687064`.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547. Association for Computational Linguistics.

Fu, M. and Tantithamthavorn, C. (2022). LineVul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, pages 608–620. IEEE.

Halfond, W. G. J., Viegas, J., and Orso, A. (2006). A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE)*.

Hanif, H. and Maffeis, S. (2022). VulBERTa: Simplified source code pre-training for vulnerability detection. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Hasan, M., Balbahaith, Z., and Tarique, M. (2019). Detection of SQL injection attacks: A machine learning approach. In *Proceedings of the 2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*, pages 1–6. IEEE.

Hussain, S. S. (2021). SQL injection dataset (SQLiV3). Kaggle. `https://www.kaggle.com/datasets/syedsaqlainhussain/sql-injection-dataset`.

Issakhani, M., Huang, M., Tayebi, M. A., and Lashkari, A. H. (2023a). BCCC-SFU-SQLInj-2023: SQL injection attack dataset. Behaviour-Centric Cybersecurity Center (BCCC), York University / Simon Fraser University.

Issakhani, M., Huang, M., Tayebi, M. A., and Lashkari, A. H. (2023b). An evolutionary algorithm for adversarial SQL injection attack generation. In *2023 IEEE International Conference on Intelligence and Security Informatics (ISI)*, Charlotte, NC, USA. IEEE.

Joshi, A. and Geetha, V. (2014). SQL injection detection using machine learning. In *Proceedings of the 2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, pages 1111–1115. IEEE.

Kaldi, A. (2024). SQL injection dataset. Kaggle. https://www.kaggle.com/datasets/ayahkhaldi/sql-injection-dataset.

Liu, M., Li, G., and Chen, J. (2020). DeepSQLi: Deep semantic learning for testing SQL injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 286–297. ACM.

Loshchilov, I. and Hutter, F. (2019). Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*.

Mathew, A. (2021). SQL-data: Dataset for SQL injection detection. GitHub. https://github.com/ajinmathew/SQL-data.

Mullick, M. A. O., Ratul, R. R., Sharif, S. B., Anannaya, S. J., and Shibly, M. M. A. (2026). Rule-based SQL injection (RbSQLi) dataset. Mendeley Data, v5. License: CC BY 4.0.

OWASP Foundation (2021). OWASP top ten 2021. https://owasp.org/www-project-top-ten/. Accessed: 2026-05-13.

OWASP Foundation (2024). OWASP modsecurity core rule set. https://coreruleset.org/. Accessed: 2026-05-13.

Quetel, G., Pautet, L., Alata, É., Robert, T., and Gimenez, P.-F. (2025). Superviz25-SQL: SQL injection detection dataset. Zenodo, v2. License: MIT.

Rayten (2024). SQL injection dataset. Kaggle. https://www.kaggle.com/datasets/rayten/sql-injection-dataset. License: Apache 2.0.

Ross, K., Moh, M., Moh, T.-S., and Yao, J. (2018). Multi-source data analysis and evaluation of machine learning techniques for SQL injection detection. In *Proceedings of the ACMSE 2018 Conference*, pages 1–8.

Sajid (2021). SQL injection dataset. Kaggle. https://www.kaggle.com/datasets/sajid576/sql-injection-dataset.

Shar, L. K. and Tan, H. B. K. (2013). Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitisation patterns. *Information and Software Technology*, 55(10):1767–1780.

Trinity, A. (2022). SQL injection extended dataset. Kaggle. https://www.kaggle.com/datasets/alextrinity/sqlinjectionextend. License: CC0 Public Domain.

Tripathy, D., Gohil, R., and Halabi, T. (2020). Detecting SQL injection attacks in cloud SaaS using machine learning. In *2020 IEEE 6th International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC) and IEEE International Conference on Intelligent Data and Security (IDS)*, pages 145–150. IEEE.

Yu, G. (2023). Biggest SQL injection dataset. Kaggle. https://www.kaggle.com/datasets/gambleryu/biggest-sql-injection-dataset.