Mitigating Keyword Bias in Java Vulnerability Detection through Dual-Stream CodeBERT with Security Feature Engineering

Arjun Khurana, Talaya Farasat, Joachim Posegga and Florian Lemmerich

University of Passau, Passau, Germany

Abstract

Deep learning models for vulnerability detection often learn to associate the presence of security-related keywords with vulnerability labels, rather than understanding the actual code semantics that distinguish vulnerable from secure code. As a result, models that exceed 97% validation accuracy can fail entirely on real-world secure code that uses security APIs such as `PreparedStatement` or `HtmlUtils.htmlEscape`. This paper presents a dual-stream CodeBERT architecture that addresses this problem which is referred to as *keyword bias*—by combining pre-trained Transformer representations with a 50-dimensional hand-engineered security feature vector, supported by targeted data augmentation and two-stage adversarial fine-tuning. The model is trained on 16,652 real-world Java samples from three established vulnerability databases across five classes (SQL injection, command injection, path traversal, cross-site scripting, and secure code), the dual-stream CodeBERT achieves 94.17% accuracy on a curated 120-sample evaluation set, 97% on a 100-case adversarial robustness suite, and 73% on a dedicated 100-sample keyword-heavy evaluation. A memorisation gap of only 2.75 percentage points between validation and adversarial accuracy confirms that the model learns genuine code semantics rather than surface-level patterns.

Keywords

vulnerability detection, keyword bias, CodeBERT, transformer models, Java security, security feature engineering

1. Introduction

Software vulnerabilities remain a persistent and costly threat. The average cost of a data breach now exceeds four million dollars [1], and from 2020 to 2024, the National Vulnerability Database records over fifteen thousand CVEs in Java-based systems alone [2]. Traditional static analysis tools [3] suffer from false-positive rates of 50–80% [4, 5], which causes alert fatigue and allows genuine vulnerabilities to go undetected. Machine learning offers a scalable alternative, as models can process code at scale and identify complex patterns from historical vulnerability data [6, 7, 8, 9, 10].

However, deep learning-based detectors suffer from a critical failure mode. When models learn to associate the mere presence of security-related tokens with vulnerability labels—rather than understanding the semantic conditions that make code actually vulnerable—they rely on superficial shortcuts instead of genuine code understanding. This failure mode is known as *keyword bias*. For example, a Java `PreparedStatement` with parameterised bindings contains the same SQL keywords (`SELECT`, `WHERE`, `setString`) as a vulnerable string-concatenation query, but a keyword-biased model treats both as vulnerable because it responds to the keywords rather than to how they are used.

Several independent studies confirm how widespread this problem is. Chakraborty et al. [4] evaluate the ReVeal vulnerability detector and show that it learns variable and function names rather than the underlying vulnerability causes, which leads to a 50% accuracy drop on production code despite 95% benchmark accuracy. Arp et al. [11] identify ten fundamental pitfalls in machine learning for security and report a 48% accuracy reduction when unused template code is removed from vulnerability samples, which exposes reliance on file-structure conventions rather than semantics. Ding et al. [12] report

GeCoIn'26: Generative Code Intelligence Workshop, IJCAI-ECAI 2026, August 15–17, 2026, Bremen, Germany

khuran01@ads.uni-passau.de (A. Khurana); talaya.farasat@uni-passau.de (T. Farasat); joachim.posegga@uni-passau.de (J. Posegga); florian.lemmerich@uni-passau.de (F. Lemmerich)

0009-0002-2816-3573 (A. Khurana)

the most striking evidence: a state-of-the-art 7B-parameter model achieves 68.26% F1 on BigVul but only 3.09% F1 on PrimeVul—a rigorously curated dataset that removes the label noise, duplication, and data leakage issues on which models silently rely. Together, these findings establish keyword bias as a widespread and unresolved problem in current vulnerability detection.

This paper addresses keyword bias through a dual-stream CodeBERT architecture that combines pre-trained code representations with explicit security feature engineering. The contributions are threefold:

1. A **dual-stream architecture** that concatenates CodeBERT's 768-dimensional [CLS] embedding with a 50-dimensional hand-engineered security feature vector, producing an 818-dimensional joint representation. The use of direct concatenation (rather than a learned gate) prevents the model from suppressing explicit security signals when they conflict with keyword-based patterns.
2. A **50-dimensional security feature extractor combined with two-stage adversarial fine-tuning**. The feature extractor encodes 35 syntactic features, 12 semantic taint-flow features, and 3 novel validation-quality metrics that capture the distinction between vulnerable and secure API usage. The two-stage training protocol progressively exposes the model to 100 adversarial edge cases while a safety threshold prevents regression on previously learned patterns.
3. A **100-sample keyword-heavy evaluation suite** with Wilson score confidence intervals, which provides a rigorous diagnostic for measuring keyword bias.

The approach is evaluated on three complementary test sets: a *standard evaluation set* of 120 curated real-world samples that measures overall detection accuracy, an *adversarial robustness suite* of 100 edge cases that tests resilience to challenging patterns, and a *keyword-heavy suite* of 100 secure samples saturated with security keywords that directly measures keyword bias. On these three sets, the dual-stream CodeBERT achieves 94.17%, 97%, and 73.0% respectively.

2. Related Work

2.1. The Java Vulnerability Detection Gap

Most vulnerability detection research targets C/C++ and Python, leaving Java underrepresented despite its widespread use in enterprise systems [4, 11]. Large-scale datasets such as Big-Vul provide 188,000 C/C++ samples but offer limited Java coverage. Java presents distinct security challenges—serialisation gadget chains, reflection-based exploits, and framework-specific injection patterns in Spring, Hibernate, and Apache Struts—that require specialised modelling approaches. Prior work on Python vulnerability detection [13] reveals similar robustness concerns under adversarial perturbations, motivating the need for architectures that resist superficial pattern exploitation. Recent dataset releases, including JavaVFC [14], CVEFixes [15], and MegaVul [16, 17], begin to close this data gap, but none of them address the keyword bias problem that undermines all models regardless of dataset quality.

2.2. Transformer Models for Code Understanding

CodeBERT [18] is a pre-trained model that learns code representations from 2.1 million code–natural-language pairs across six programming languages using masked language modelling. GraphCode-BERT [19] extends this by incorporating data-flow information through graph-based positional encoding. LineVul [20] demonstrates the effectiveness of Transformer architectures for vulnerability detection through attention-based line localisation on C/C++ code, but does not evaluate on secure code containing security keywords. PrimeVul's evaluation of fifteen code-language models reveals that architectures achieving 92–97% benchmark accuracy drop to 47–63% under minor code changes [12], which indicates reliance on superficial patterns rather than genuine semantic understanding.

2.3. Keyword Bias in Security Machine Learning

As introduced in Section 1, keyword bias arises because security-related keywords correlate with vulnerability labels in CVE-derived training data, but that correlation breaks down in production, where secure code necessarily uses the same APIs and keywords as vulnerable code [4, 11, 12]. No existing method systematically measures or mitigates this bias for Java vulnerability detection.

3. Methodology

3.1. Dataset Construction

Three established vulnerability databases are selected for complementary coverage: CVEFixes [15], JavaVFC [14], and MegaVul [16, 17]. Together, these sources provide CVE-verified vulnerability-fixing commits, manually verified Java-specific fixes, and function-level vulnerable/fixed code pairs.

Three quality issues require systematic resolution. First, 39% of initial samples are exact or near-exact duplicates; code-similarity hashing reduces them to unique samples. Second, commit-based splitting is implemented to prevent data leakage [21]: vulnerable and fixed versions from the same CVE commit are placed exclusively in either the training or test set, never both. Third, manual inspection of 500 randomly sampled labels reveals a 17% misclassification rate ($\kappa = 0.81$ inter-rater agreement).

The final dataset comprises 16,652 samples classified into five groups following the OWASP Top Ten and CWE taxonomy: SQL injection (1,213 samples, 7.3%), command injection (4,003, 24.0%), path traversal (1,603, 9.6%), cross-site scripting (1,211, 7.3%), and secure code (8,622, 51.8%). The pronounced class imbalance—with Secure accounting for over half of all samples and XSS severely underrepresented—motivates the use of class-weighted loss (Section 3.5). A 90/10 train-validation split with commit-based stratification yields 14,987 training and 1,665 validation samples. All samples preserve full function context, with an average length of 247 tokens, which fits within CodeBERT's 512-token window.

3.2. Architecture Selection: CodeBERT

CodeBERT is selected as the base architecture for three reasons. First, its pre-training on 2.1 million code–natural-language pairs across six programming languages—with a significant portion comprising Java code—provides general programming knowledge that is not achievable by training from scratch on the relatively small vulnerability datasets available for Java [18]. Second, the self-attention mechanism in the Transformer architecture captures long-range dependencies that are critical for vulnerability detection, such as when a variable is designated as user-controlled input in one location and subsequently used unsafely elsewhere in the function [22]. Third, most existing Transformer-based vulnerability detectors target C/C++ (e.g., LineVul [20]), leaving Java insufficiently addressed despite its distinct attack surface.

3.3. Dual-Stream Architecture

The standard CodeBERT baseline—`microsoft/codebert-base` with a linear classification head over the `[CLS]` token—consists of twelve Transformer encoder layers, 768-dimensional embeddings, twelve attention heads, and approximately 125 million parameters. This baseline achieves 97.2% validation accuracy but fails to correctly classify any keyword-heavy secure sample. Attention-weight analysis reveals that the model consistently assigns high importance to tokens like `SELECT` and `PreparedStatement`, regardless of whether the code uses unsafe concatenation or secure parameterisation.

The dual-stream architecture (Figure 1) addresses this problem by processing each Java function through two parallel streams. The **CodeBERT stream** produces a 768-dimensional `[CLS]` representation through the pre-trained encoder. The **feature stream** produces a 50-dimensional security feature vector (described in Section 3.4). These two streams are concatenated into an 818-dimensional joint representation, which passes through dropout ($p = 0.1$) and a single dense layer that maps to five output classes.

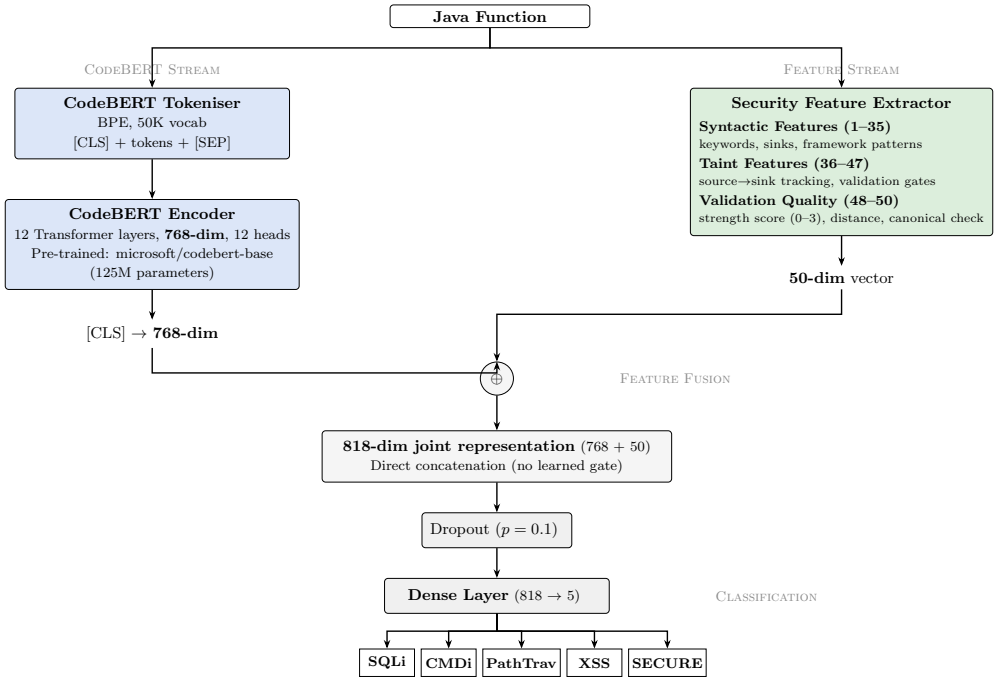

Figure 1: Dual-stream CodeBERT architecture. The CodeBERT stream produces a 768-dimensional `[CLS]` embedding while the feature stream produces a 50-dimensional security vector.

Direct concatenation is chosen over a learned attention gate specifically to prevent the model from suppressing the feature signal. Given the 768:50 dimensional asymmetry, a learned gate would likely discount the feature stream whenever it contradicts dominant keyword patterns—precisely the cases where it is most needed.

3.4. 50-Dimensional Security Feature Extractor

The feature extractor produces a 50-dimensional vector organised into three functional groups.

Features 1–35 (syntactic patterns) count security-domain keywords by vulnerability class and encode signals for sink patterns, validation-API usage, and framework-specific constructs (e.g., `CriteriaBuilder`, Thymeleaf escaping, `ProcessBuilder` array vs. string construction).

Features 36–47 (semantic taint analysis) implement lightweight intra-function taint tracking, capturing whether user-supplied inputs reach unsafe sinks (`ProcessBuilder`, `Runtime.exec()`, `File`) and whether a validation gate intervenes.

Features 48–50 (validation quality—novel contribution). Feature 48 provides an ordinal validation-quality score (0: none; 1: null/empty check; 2: regex pattern; 3: canonical path via `getCanonicalPath`/`toRealPath`). Feature 49 measures the line distance between the validation gate and the security sink. Feature 50 flags the use of `File.getCanonicalPath()` or `Path.toRealPath()`.

3.5. Training Protocol

Targeted data augmentation. As discussed in Section 1, keyword bias arises partly because CVE-derived datasets underrepresent secure code that uses security-domain vocabulary. To address this imbalance, 1,500 targeted augmentation samples are generated across six categories that correspond to observed failure modes: command injection array-vs-string construction patterns (secure `ProcessBuilder` with argument arrays vs. vulnerable single-string commands), SQL ORM and corrective query patterns (secure `CriteriaBuilder` and `JdbcTemplate` usage), path traversal canonical-path fix samples, and XSS framework-escape pairs. Each augmentation exposes the model to additional

examples of secure code that is saturated with vulnerability-associated keywords.

Adversarial edge cases. A 100-case adversarial suite is constructed in two rounds of 50. The first round covers framework confusion, weak validation, injection variants (JNDI, LDAP, XPath, SpEL), encoding bypasses, and structural false positives. The second round targets modern Java idioms, reflection, enterprise patterns (JMS, JMX, EJB), file-upload attacks, and TOCTOU conditions.

Stage 1 trains the full dual-stream model on 18,202 samples (16,652 base + 1,500 augmentations + 50 original adversarial cases) using AdamW [23] with a learning rate of 2×10^{-5}, weight decay of 0.01, batch size of 16, and up to 15 epochs with early stopping (patience 3). Class-weighted cross-entropy with inverse-frequency weights addresses class imbalance:

$$\mathcal{L}_{\text{CE}} = -\sum_{i=1}^{N} w_{y_i} \log P(y_i \mid c_i; \theta) \tag{1}$$

where w_{y_i} is the inverse class-frequency weight and c_i is the input representation.

Stage 2 fine-tunes the Stage 1 checkpoint on the full 18,252-sample corpus (adding 50 expansion adversarial cases) at a reduced learning rate of 5×10^{-6} for at most three epochs. Fine-tuning on the full corpus, rather than only on the 50 new adversarial cases, is used to prevent catastrophic forgetting of previously learned patterns while integrating the new cases. A safety threshold reverts to the Stage 1 checkpoint if validation accuracy drops by more than one percentage point.

3.6. Evaluation Protocol

Three complementary test sets are used, each targeting a different aspect of model performance. The **standard evaluation set** contains 120 curated real-world Java samples: 25 per vulnerability class (13 vulnerable, 12 secure) plus 20 additional secure samples; it measures overall detection accuracy across realistic code patterns. The **adversarial robustness suite** consists of all 100 adversarial cases, reported separately for the original and expansion halves; it tests whether the model handles challenging edge cases that go beyond typical benchmark patterns. The **keyword-heavy evaluation** consists of 100 manually constructed secure samples balanced across four vulnerability categories (25 each), where each sample uses correct defensive programming but is saturated with security-domain keywords. Approximately 85% of the samples employ defensive patterns present in the training distribution (e.g., `PreparedStatement` with parameterised bindings, `getCanonicalPath` with prefix validation), while the remaining 15% test generalisation to framework-specific conventions not directly represented in training data (e.g., jOOQ type-safe queries, Thymeleaf auto-escaping). Wilson score 95% confidence intervals [24] are reported for the keyword-heavy accuracy estimates to provide statistical rigour.

4. Results

4.1. Standard Test Set Performance

Table 1 reports overall metrics comparing the CodeBERT baseline (standard `[CLS]` head, no feature stream) against the full dual-stream architecture. The dual-stream CodeBERT achieves 94.17% accuracy (113/120 correct), maintaining competitive standard performance while substantially improving keyword-heavy accuracy to 73%.

Table 2 provides a per-class breakdown. The dual-stream CodeBERT achieves perfect accuracy on SQL injection, path traversal, and XSS, with an overall secure-class accuracy of 88.2% (60/68). The remaining errors concentrate on command injection (11/13), where `ProcessBuilder` with string arrays and `find -exec` patterns remain ambiguous, and on eight secure samples misclassified as the vulnerability class whose keywords they contain.

Table 1
Performance comparison: Baseline CodeBERT baseline vs. dual-stream CodeBERT.

Metric	Baseline	Dual-Stream	Change
Validation accuracy	97.2%	99.75%	+2.55pp
Standard accuracy (120)	93.50%	94.17%	+0.67pp
Keyword-heavy (100)	0%	73.0%	+73.0pp
Adversarial (100)	—	97%	—

Table 2
Per-class standard evaluation.

Class	Correct	Acc
SQL Injection	13/13	100%
Command Injection	11/13	84.6%
Path Traversal	13/13	100%
Cross-Site Scripting	13/13	100%
Secure Code	60/68	88.2%
Overall	**113/120**	**94.17%**

Table 3
Adversarial robustness by category.

Category	Cases	Acc
9 at 100%[†]	69	100%
Injection variants	8	87.5%
File upload attacks	6	83%
False positives	5	80%
TOCTOU patterns	4	75%
Total	**100**	**97%**

[†] Framework confusion (8), weak validation (10), modern Java (10), reflection (8), enterprise Java (8), adv. encoding (6), complex patterns (7), context confusion (6), encoding bypasses (6).

Table 4
Per-category keyword-heavy accuracy (dual-stream CodeBERT).

Category	Accuracy	n
SQL Injection	68.0%	25
Command Injection	88.0%	25
Path Traversal	80.0%	25
Cross-Site Scripting	56.0%	25
Overall	**73.0%**	**100**

4.2. Adversarial Robustness

The dual-stream CodeBERT achieves 97% accuracy on the 100-case adversarial suite (98% on the original 50 cases, 96% on the expansion 50). This high accuracy on out-of-distribution edge cases—compared to 99.75% on the training-distribution validation set—yields a *memorisation gap* of only 2.75 percentage points. The memorisation gap measures the difference between a model's performance on data that resembles its training distribution and its performance on deliberately challenging cases. A small gap indicates that the model has learned generalisable patterns rather than memorising surface-level cues.

Robustness by category. Table 3 summarises performance across adversarial categories. The model achieves perfect accuracy across nine categories spanning framework confusion, modern Java idioms, enterprise patterns, and encoding bypasses. The three failures involve inter-procedural patterns (a JNDI injection variant, an SVG-based XSS in file uploads, and a TOCTOU race condition) that require cross-function analysis beyond the scope of a single function.

4.3. Keyword Bias Evaluation

Table 4 breaks down keyword-heavy accuracy by vulnerability category. The highest accuracy appears on command injection (88%) and path traversal (80%), where the validation-quality features (48–50) and taint-flow features (36–47) most directly encode the semantic distinction between vulnerable and secure usage. The lowest accuracy is on cross-site scripting (56%), where residual failures concentrate on framework-specific auto-escaping conventions (Thymeleaf `th:text`, Freemarker, JSF `h:outputText`) that are not represented in the training data.

Table 5
Ablation study: component contributions.

Configuration	Std.	Adv.
CodeBERT baseline	93.50%	—
+ Features (20-dim)	91.20%	—
+ Augmentations	92.50%	88
+ Features (50-dim)	93.30%	89
+ Stage 1 adversarial	93.50%	90
+ Stage 2 (full system)	94.17%	97
— Valid. quality (48–50)	92.50%	88
— Taint feat. (36–47)	93.10%	91
— Stage 2 only	93.50%	88

Table 6
Comparison with published methods.

Method	Metric	Kw.	Lang.
FindSecBugs [25]	82.8% F1[†]	—	Java
VUDENC [21]	80–90% F1	—	Py
QCNN [26]	99.2% Acc[‡]	—	Java
LineVul [20]	0.91 F1[§]	—	C++
Ours	94.17%	73%	Java

[†]OWASP synthetic; real-world 12.7%.
[‡]Synthetic SARD benchmark.
[§]C/C++ Big-Vul; not comparable.

4.4. Ablation Analysis

Table 5 quantifies each component's contribution by progressively adding components to the baseline and by removing individual components from the full system. The validation-quality triplet (features 48–50) is the most impactful single component: removing it drops standard accuracy from 94.17% to 92.50% and adversarial accuracy from 97 to 88 correct cases. Taint-flow features (36–47) contribute the next largest adversarial improvement, with their removal reducing adversarial accuracy from 97 to 91. Stage 2 adds nine correct adversarial cases and 0.67 points of standard accuracy, which confirms that the safety threshold effectively prevents regression. Each augmentation exposes the model to additional examples of secure code that is saturated with vulnerability-associated keywords, directly counteracting the distribution in the base dataset. The ablation study (Table 5) confirms that augmentations improve standard accuracy from 91.20% to 92.50% and demonstrate adversarial robustness evaluation.

4.5. Comparison with Existing Methods

Table 6 places the dual-stream CodeBERT alongside established vulnerability detection methods. Metrics are drawn from the original publications rather than re-run under a unified protocol; since datasets, languages, and metrics differ across studies, the comparison is indicative rather than conclusive. Within these limits, the dual-stream CodeBERT is the only evaluated approach that reports both competitive standard accuracy and an explicit keyword-bias measurement on Java.

5. Discussion

The dual-stream architecture's effectiveness stems from three factors: self-attention establishes direct connections across all sequence positions, enabling the model to relate distant security sinks and validation gates [22]; pre-training on 2.1 million code–natural-language pairs provides general programming knowledge that transfers to vulnerability detection [18]; and the 50-dimensional feature stream provides an explicit channel for semantic signals that attention alone cannot reliably derive from token sequences.

Figure 2 illustrates this using line-occlusion attribution, a post-hoc technique that measures each line's contribution to the prediction. In a `JdbcTemplate` query with parameterised placeholders, the dual-stream CodeBERT assigns full attribution to the SQL statement containing `?` bindings, correctly classifying the sample as Secure. This shows that the model distinguishes parameterised queries from vulnerable concatenation—the distinction that keyword-biased models fail to capture.

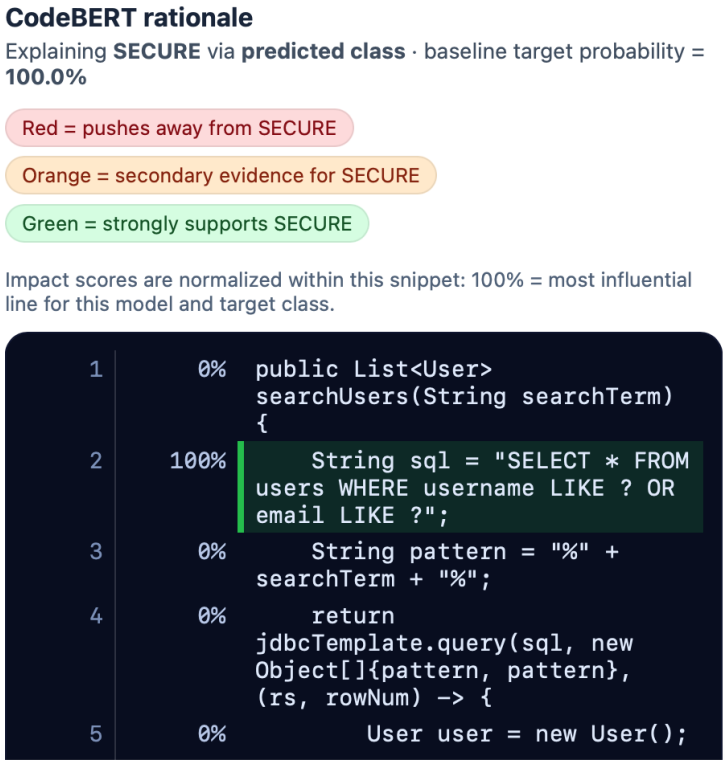

Figure 2: Line-occlusion attribution for a keyword-heavy secure sample (JdbcTemplate with parameterised query). The dual-stream CodeBERT assigns 100% attribution to the parameterised SQL statement (line 2), correctly recognising the ? placeholders as evidence of secure parameterisation despite the presence of SQL keywords.

6. Limitations

- Approximately 17% label noise remains in the training corpus despite the applied quality-assurance procedures.
- Classification is performed at the function level rather than at the line level, limiting localisation.
- Although the keyword-heavy evaluation suite is balanced across four categories, it does not encompass all Java defensive frameworks. Consequently, the model struggles with previously unseen auto-escaping conventions, which is reflected in the 56% accuracy achieved on XSS keyword-heavy samples.
- Comparison against more recent code language models such as GraphCodeBERT, UniXcoder, or larger CodeLlama variants is left for future work; part of the observed gains may therefore reflect limitations of the CodeBERT baseline rather than the dual-stream design alone.

7. Conclusion

This paper addresses keyword bias in Java vulnerability detection through a dual-stream CodeBERT architecture that combines pre-trained code representations with a 50-dimensional hand-engineered security feature extractor and two-stage adversarial fine-tuning. The approach achieves 73% keyword-heavy accuracy, 94.17% on the standard evaluation set, and 97% on the adversarial suite. The memorisation gap of only 2.75 percentage points between validation and adversarial accuracy confirms that the model learns genuine code semantics rather than surface-level keyword correlations.

The per-category analysis highlights cross-site scripting (56%) as lagging behind command injection (88%) and path traversal (80%), pointing to framework-specific auto-escaping as a concrete target for future work. Further directions include line-level localisation, cross-language transfer, and automatic feature learning through program synthesis or graph-based encoders.

The code and models are available at https://github.com/Automated-Vulnerability-Detection/Mitigating-Keyword-Bias-Java-Vulnerability-Detection-CodeBERT-Security-Feature-Engineering.

Declaration on Generative AI

The authors used Claude (Anthropic) for code debugging and synthetic sample generation. All content was reviewed by the authors, who take full responsibility for the publication.

References

[1] IBM Security, Ponemon Institute, Cost of a Data Breach Report 2024, Technical Report, IBM Corporation, 2024. URL: https://www.ibm.com/think/insights/cost-of-a-data-breach-2024-financial-industry.

[2] National Institute of Standards and Technology, National vulnerability database, 2024. URL: https://nvd.nist.gov/.

[3] T. Farasat, J. Posegga, A large-scale analysis of vulnerabilities in networking-oriented python github repositories, in: Springer Applied Cryptography and Network Security (ACNS), 2026.

[4] S. Chakraborty, R. Krishna, Y. Ding, B. Ray, Deep learning based vulnerability detection: Are we there yet?, IEEE Transactions on Software Engineering 48 (2020) 3280–3296. doi:https://doi.org/10.48550/arXiv.2009.07235.

[5] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, 2013, pp. 672–681. doi:10.1109/ICSE.2013.6606613.

[6] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, M. McConley, Automated vulnerability detection in source code using deep representation learning, 2018, pp. 757–762. doi:10.1109/ICMLA.2018.00120.

[7] T. Farasat, J. Posegga, Machine learning techniques for python source code vulnerability detection (2024). doi:10.48550/arXiv.2404.09537.

[8] T. Farasat, J. Posegga, Optimizing code embeddings and ml classifiers for python source code vulnerability detection, in: Proceedings of the IEEE/ACM 12th International Conference on Big Data Computing, Applications and Technologies, 2025.

[9] T. Farasat, J. Posegga, Enhancing python code security: A comparison of machine learning, chatgpt, and static analysis methods, in: IEEE International Conference on Electrical and Computer Engineering Researches (ICECER), 2025.

[10] T. Farasat, A. Ahmadzai, A. E. George, S. A. Qaderi, D. Dordevic, J. Posegga, Safepyscript: A web-based solution for machine learning-driven vulnerability detection in python, in: arXiv preprint arXiv:2411.00636, 2024.

[11] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, K. Rieck, Dos and don'ts of machine learning in computer security, 2021. doi:10.48550/arXiv.2010.09470.

[12] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, Y. Chen, Vulnerability detection with code language models: How far are we?, in: 47th IEEE/ACM International Conference on Software Engineering (ICSE), 2025. doi:10.1109/ICSE55347.2025.00038.

[13] T. Farasat, A. Bouzid, J. Posegga, Adversarial evaluation of machine learning-based Python source code vulnerability detectors, in: Proceedings of the Generative Code Intelligence Workshop (GeCoIn 2025), co-located with ECAI, CEUR Workshop Proceedings, 2025. URL: https://ceur-ws.org/Vol-4075/paper2.pdf.

[14] T. Bui, Y. N. Tun, Y. Cheng, I. Irsan, T. Zhang, H. Kang, Javavfc: Java vulnerability fixing commits from open-source software, 2024. doi:10.48550/arXiv.2409.05576.

[15] G. Bhandari, A. Naseer, L. Moonen, Cvefixes: automated collection of vulnerabilities and their fixes from open-source software, in: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2021, Association for Computing

Machinery, New York, NY, USA, 2021, pp. 30–39. URL: https://doi.org/10.1145/3475960.3475985. doi:10.1145/3475960.3475985.

[16] C. Ni, L. Shen, X. Yang, Y. Zhu, S. Wang, Megavul: A c/c++ vulnerability dataset with comprehensive code representation, 2024. doi:10.48550/arXiv.2406.12415.

[17] C. Ni, L. Shen, X. Yang, Y. Zhu, S. Wang, MegaVul: C/C++/Java vulnerability dataset — Java extension, GitHub repository, 2024. URL: https://github.com/icyrockton/megavul, java extension providing 2,433 vulnerable and 39,516 non-vulnerable Java functions via tree-sitter-java. Accessed: March 2026.

[18] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, T. Liu, D. Jiang, M. Zhou, Codebert: A pre-trained model for programming and natural languages, 2020, pp. 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139.

[19] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, M. Zhou, Graphcodebert: Pre-training code representations with data flow, 2021. URL: https://arxiv.org/abs/2009.08366. doi:10.48550/arXiv.2009.08366. arXiv:2009.08366.

[20] M. Fu, C. Tantithamthavorn, Linevul: a transformer-based line-level vulnerability prediction, in: Proceedings of the 19th International Conference on Mining Software Repositories, MSR '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 608–620. URL: https://doi.org/10.1145/3524842.3528452. doi:10.1145/3524842.3528452.

[21] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, L. Grunske, VUDENC: Vulnerability detection with deep learning on a natural codebase for python, Information and Software Technology 144 (2022) 106809. doi:10.1016/j.infsof.2021.106809.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need (2017). doi:10.48550/arXiv.1706.03762.

[23] I. Loshchilov, F. Hutter, Decoupled weight decay regularization, 2019. URL: https://arxiv.org/abs/1711.05101. doi:10.48550/arXiv.1711.05101. arXiv:1711.05101.

[24] E. B. Wilson, Probable inference, the law of succession, and statistical inference, Journal of the American Statistical Association 22 (1927) 209–212.

[25] K. Li, S. Chen, L. Fan, R. Feng, H. Liu, C. Liu, Y. Liu, Y. Chen, Comparison and evaluation on static application security testing (sast) tools for java, 2023, pp. 921–933. doi:10.1145/3611643.3616262.

[26] S. Hussain, M. Nadeem, J. Baber, M. Hamdi, A. Rajab, M. Al Reshan, A. Shaikh, Vulnerability detection in java source code using a quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction, Scientific Reports 14 (2024). doi:10.1038/s41598-024-56871-z.