Adversarial Evaluation of Machine Learning-Based Python Source Code Vulnerability Detectors

Talaya Farasat*, Ahmed Bouzid and Joachim Posegga*

University of Passau, Passau, Germany

Abstract

Machine learning models for Python source code vulnerability detection have demonstrated impressive accuracy in identifying security vulnerabilities. However, their robustness against adversarial manipulation remains largely unexplored. In this work, we evaluate the adversarial resilience of two strong Python vulnerability detection models. We apply simple semantics-preserving perturbations that maintain the original Python code functionality while misleading the detection models. Our evaluation shows that such adversarial examples can cause substantial performance degradation. We observe that even simple, semantics-preserving transformations are sufficient to mislead the models, without requiring complex attack strategies. These findings expose critical weaknesses in current machine learning-based detectors and underscore the urgent need for more robust, semantics-aware approaches to secure code analysis.

Keywords

Machine Learning, Vulnerability Detection, Adversarial, Python

1. Introduction

Code flaws or vulnerabilities are prevalent in software systems and can potentially lead to system compromise, information leaks, or denial of service [8]. Recognizing the constraints of traditional code analysis methods (static and dynamic code analysis [8, 12]), and with the growing accessibility of open-source software repositories, it has been recommended to adopt a data-driven approach for software vulnerability detection. Therefore, various machine learning techniques have been applied to learn vulnerable features of source code, and to automate the process of software vulnerability identification, with varying success[1, 2, 3, 4, 5, 6, 7, 13, 15, 16, 19, 21].

Many researchers have dedicated considerable attention to source code vulnerability detection written in different programming languages like Java, C, and C++ [1, 2, 3, 5, 8, 9, 10, 16, 20, 21]. In 2025, Python continues to maintain its prominent position as one of the top programming languages [30]. Therefore, many studies [6, 13, 17, 18, 19, 23, 25, 34, 35] focus specifically on vulnerability detection in Python. However, there is a major hazard lying with these software vulnerability detection models – they lack adversarial robustness.

Adversarial examples are inputs intentionally crafted by an adversary to mislead a trained machine learning model into making incorrect predictions. These examples are generated by applying carefully designed, minor perturbations to the original inputs. Although these perturbations are often imperceptible to human observers, they can significantly degrade the performance of machine learning models.

Unlike natural languages, programming languages are governed by strict lexical, grammatical, and syntactic rules. Consequently, adversarial examples for source code must be both syntactically valid and semantically correct; otherwise, they may fail to compile [26]. This makes adversarial manipulation more challenging than in continuous domains like images, where small pixel-level changes suffice. In source code, valid perturbations include (i) renaming variables, (ii) inserting dead code, and (iii)

GeCoIn 2025: Generative Code Intelligence Workshop, co-located with the 28th European Conference on Artificial Intelligence (ECAI-2025), October 26, 2025 — Bologna, Italy

*Corresponding author.

tf@sec.uni-passau.de (T. Farasat); bouzid01@ads.uni-passau.de (A. Bouzid); jp@sec.uni-passau.de (J. Posegga)

0000-0002-0560-0334 (T. Farasat)

reordering independent statements. These transformations require a deep understanding of program semantics to avoid altering functionality [27].

Different studies explore adversarial attacks on source code. Zhang et al. [26] introduce the Metropolis-Hastings Modifier for generating adversarial examples (applied on C/C++). Yefet et al. [27] propose Discrete Adversarial Manipulation of Programs (DAMP) to attack code models (applied on Java and C). Liu et al. [29] develop a ChatGPT-based evasion attack against detectors (applied on C/C++). Henkel et al. [28] evaluate the robustness of current machine learning architectures for vulnerability detection. Despite these advances, research on adversarial attacks in this domain remains limited [29].

In contrast to prior work, our study focuses specifically on Python-based vulnerability detection models. We demonstrate that even simple, Python-specific perturbations can significantly degrade the performance of strong models—the accuracy of VUDENC [6] drops from 97.8% to 59.6%, while the model by Farasat and Posegga [24] falls from 98.8% to 40.4%. These results suggest that such models remain highly susceptible to straightforward adversarial manipulations, indicating that complex attacks are not necessary to reveal their weaknesses. This highlights the urgent need for more robust, semantics-aware detection methods tailored to the analysis of Python source code.

2. Experimental Design

2.1. Selection of Vulnerability Detection Models

We select two strong models for Python source code vulnerability detection: VUDENC [6] and the model by Farasat and Posegga [24]. In this study, we focus exclusively on their Cross-Site Scripting (XSS) detection models.

2.2. Adversarial Perturbations

We perform an adversarial evaluation using the designated final test set from the `plain_xss` dataset, available at [31, 33]. The test set is defined by a list of indices provided in the `xss_dataset_keysfinaltest` file. For each indexed code block, we extract the relevant context using precomputed "bad parts" annotations and retrieve code segments of up to 200 tokens via utility functions.

To ensure the selected blocks are relevant to XSS, we apply a regular-expression-based function, `is_xss_payload()`, which identifies blocks containing known XSS markers such as `<script>` tags, `alert()` calls, and event handler keywords like `onerror` and `onload`. We specifically filter for blocks labeled as vulnerable (label 0) that contain syntactic evidence of XSS attacks.

These filtered vulnerable blocks are perturbed using a custom function that applies a series of semantics-preserving transformations:

- **Identifier Renaming:** Replaces known XSS-related variable names using a predefined mapping.
- **String Obfuscation:** Splits signature patterns like `<script>` into concatenated substrings (e.g., `"<scr" + "ipt>"`).
- **ASCII Encoding:** Rewrites payloads such as `alert()` using `chr()` representations to evade pattern matching.
- **Logic Refactoring:** Alters detection-relevant conditions, such as replacing substring search logic with membership tests.
- **No-op Insertion:** Randomly adds semantically neutral variable assignments (e.g., `dummy = None`) to increase structural noise.
- **Dummy Computations:** Appends irrelevant arithmetic operations to inflate the complexity of the code.

Each perturbed sample is tokenized using a custom tokenizer, embedded with a pretrained Word2Vec model available at [32], and padded to a fixed maximum sequence length of 200 tokens. These vectorized samples are then passed through pretrained vulnerability detection models for evaluation.

3. Evaluation

The adversarial evaluation of the VUDENC model [6] on 42 test samples yields an accuracy of 59.5%, a recall of 59.6%, and an F1 score of 74.6%. As shown in Table 1, this represents a significant performance degradation compared to the model's original metrics.

Similarly, the model proposed by Farasat and Posegga [24], when subjected to the same adversarial samples, achieves an accuracy of 40.4%, a recall of 40.3%, and an F1 score of 57.6%. The performance decline for this model is also illustrated in Table 1.

These results demonstrate substantial performance degradation, confirming that the applied perturbations effectively evade detection.

Figure 1 provides a visual example of this failure. A code used in visual demonstration available at [31] in examples folder. Table 2 describes the specific perturbations used in the demonstration script shown in Figure 1. As illustrated, the original Farasat and Posegga model (left side) detects vulnerabilities that the perturbed version (right side) misses, showing the significant impact of our adversarial modifications. We follow the same color scheme and confidence levels used in [6] for the visualizations in Figure 1.

Note: On 42 test samples, we apply all the perturbations described above in Section 2.2 Adversarial Perturbations. However, for the demonstration script shown in Figure 1, we apply only the relevant perturbations listed in Table 2.

Models	Accuracy	Recall	F1 Score
XSS (Original VUDENC Model [6])	97.8%	80.8%	86.0%
XSS (With Perturbations (VUDENC Model))	59.5%	59.6%	74.6%
XSS (Original Farasat and Posegga Model [24])	98.8%	91.3%	93.0%
XSS (With Perturbations (Farasat and Posegga Model))	40.4%	40.3%	57.6%

Table 1
Performance Metrics Before and After Adversarial Perturbations on 42 test samples

Perturbation	Description	Applied in Script
Identifier Renaming	Rename sensitive identifiers based on mapping dictionary (e.g., `rules` → `result_record`, `mapping_delete` → `net_client.mapping_delete`)	Yes
String Literal Breaking	Break strings like `mark_safe` into concatenated parts (e.g., `"mark_" + "safe"`) to evade exact string matching	Yes
No-op Line Insertion	Randomly insert no-op lines with dummy variables (e.g., `abcxyz = None # no-op filler`) to confuse static analysis	Yes
Dummy Math Insertion	Insert dummy math calculations (e.g., `_ghost = sum([42, 1337, 7]) # dummy calc`) inside specific functions to add noise	Yes
Literal Obfuscation	Replace fixed strings like `"XSS"` or `"Cross Site Scripting"` with concatenations (e.g., `"X" + "SS"`) to avoid detection	No (not applicable)
ASCII Encoding of Payloads	Encode suspicious strings (e.g., `<script>`, `alert()`) as `chr()` expressions to evade pattern matching	No (not applicable)
Logic Refactoring	Change logic expressions like `.find(...) != -1` to alternate forms (e.g., `in` operator)	No (not applicable)

Table 2
Perturbations applied in the Python script (see Figure 1 for results)

(a) (b)

Figure 1: Visualization of model predictions before (a) (the original Farasat and Posegga Model [24]) and after (b) adversarial perturbations (perturbations applied as described in Table 2) over XSS vulnerability script available at [31]

4. Defense Suggestions

To defend against the proposed adversarial perturbations, a combination of static analysis and model-level enhancements is necessary. First, integrating semantic-aware models—such as those based on abstract syntax trees (ASTs) or control/data flow graphs—can improve resistance to superficial modifications like identifier renaming and no-op insertions, as these models capture deeper structural and behavioral properties of the code. Second, applying adversarial training with examples that include obfuscation techniques (e.g., string splitting, ASCII encoding, and logic refactoring) can help the model learn to generalize over such evasive patterns. To mitigate the impact of dummy computations and structural noise, feature extraction methods that emphasize semantically meaningful elements—such as

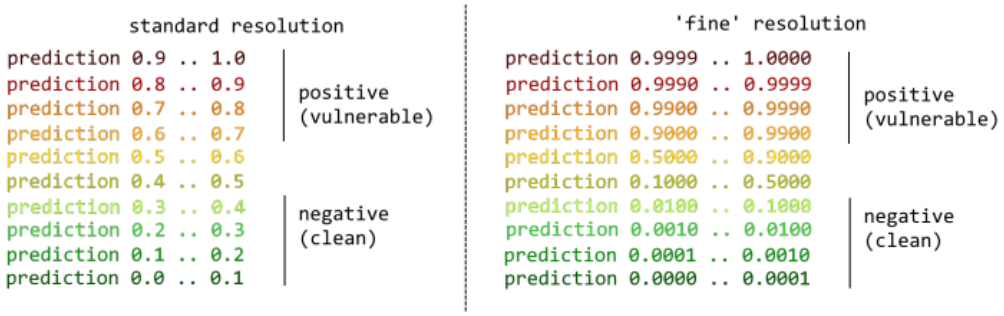

Figure 2: Use same Color and Confidence level as in [6] for Figure 1 (demonstration)

data flow dependencies or function call patterns—should be prioritized over raw token sequences. Lastly, deploying runtime monitoring or post-prediction semantic validation mechanisms can help detect anomalous inputs that deviate from typical coding patterns, signaling possible adversarial manipulation even when syntactic correctness is maintained.

5. Conclusion and Future Work

This study demonstrates that simple, semantics-preserving adversarial perturbations can significantly degrade the performance of strong Python vulnerability detection models. By applying transformations such as identifier renaming, string obfuscation, ASCII encoding, logic refactoring, and the insertion of no-op or dummy computations, we show that models like VUDENC [6] (accuracy drops from 97.8% to 59.6%) and the Farasat and Posegga model [24] (accuracy drops from 98.8% to 40.4%) are highly susceptible to misclassifying vulnerable code. Our experiments reveal a drastic drop in performance, confirming that even straightforward evasion strategies can bypass these models while preserving semantic integrity—eliminating the need for complex attacks. These findings expose a critical weakness in current machine learning-based Python vulnerability detection systems.

For future work, we plan to extend our evaluation to other vulnerability types beyond XSS, such as SQL injection and path traversal, and to other programming languages.

Declaration on Generative AI

During the preparation of this work, the author(s) utilized X-GPT-4 to assist with grammar and spelling corrections. Following the use of this tool, the author(s) thoroughly reviewed and revised the content as necessary and take(s) full responsibility for the accuracy and integrity of the published work.

References

[1] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. In *IEEE Transactions on Dependable and Secure Computing*, Volume: 19, 2022.

[2] G. Lin, S. Wen, Q. Han, J. Zhang, and Y. Xiang. Software Vulnerability Detection Using Deep Neural Networks: A Survey. In *IEEE Proceedings*, Volume: 108, 2020.

[3] Y. Zhou, S. Liu, , J. Siow, X. Du, and Y. Liu. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*, Vancouver Canada, 2019.

[4] S. Chakraborty, R. Krishna, Y. Din, and B. Ray. Deep Learning Based Vulnerability Detection: Are We There Yet?. In *IEEE Transactions on Software Engineering*, Volume: 48, 2021.

[5] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood and M. McConley. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *IEEE International Conference on Machine Learning and Applications (ICMLA)*, Orlando USA, 2018.

[6] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske. VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python. In *ELSEVIER Information and Software Technology*, Volume 144, 2022.

[7] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng and Y. Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *25th Annual Network and Distributed System Security Symposium (NDSS)*, California USA, 2018.

[8] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy and A. Ghose. Automatic feature learning for vulnerability prediction. In *IEEE Transactions on Software Engineering*, Volume: 47, 2018.

[9] K. Liu, D. Kim, T. F. Bissyand´e, S. Yoo and Y. Le Traon, Mining Fix Patterns for FindBugs Violations. In *, IEEE Transactions on Software Engineering*, Volume: 47, 2018.

[10] R. Rolim, G. Soares, R. Gheyi and T. Barik. Learning Quick Fixes from Code Repositories. In *ACM Brazilian Symposium on Software Engineering*, Brazil, 2021.

[11] T. Zimmermann, N. Nagappan and L. Williams. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. In *IEEE International Conference on Software Testing, Verification and Validation*, France, 2005.

[12] M. Ceccato and R. Scandariato. Static analysis and penetration testing from the perspective of maintenance teams. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, USA, 2016.

[13] R. Wang, S. Xu, X. Ji, Y. Tian, L. Gong and K. Wang. "An extensive study of the efects of diferent deep learning models on code vulnerability detection in Python code. In *Automated Software Engineering*, Volume 31, 2024.

[14] R. Scandariato, J. Walden, and W. Joosen. Static analysis versus penetration testing: A controlled experiment. In *IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, USA, 2013.

[15] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, I. Vats, H. Moazen and F. Sarro. A survey on machine learning techniques applied to source code. In *ELSEVIER Journal of Systems and Software*, Volume 209, 2024.

[16] A. O. A. Semasaba, W. Zheng, X. Wu, and S. Agyemang. Literature survey of deep learning-based vulnerability analysis on source code. In *WILEY IET Software*, Volume 14(6), 2020.

[17] A. Bagheri and P. Heged˝us. A Comparison of Different Source Code Representation Methods for Vulnerability Prediction in Python. In *Springer Quality of Information and Communications Technology*, 2021.

[18] M. Alfade, D. E. Costa and E. Shihab˝us. Empirical analysis of security vulnerabilities in Python packages. In *ACM Empirical Software Engineering*, Volume 28, 2023.

[19] N. S. Harzevili, J. Shin, J. Wang and S. Wang˝us. Characterizing and Understanding Software Security Vulnerabilities in Machine Learning Libraries. In *IEEE/ACM International Conference on Mining Software Repositories (MSR)*, Australia, 2023.

[20] J. Fan, Y. Li, S. Wang and T. N. Nguyen. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *ACM International Conference on Mining Software Repositories (MSR)*, South Korea, 2020.

[21] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin. µVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. In *IEEE Trans. Dependable Secure Computing*, Volume 18(5), 2019.

[22] M. Davari, M. Zulkernine, and F. Jaafar. An Automatic Software Vulnerability Classification Framework. In *IEEE International Conference on Software Security and Assurance (ICSSA)*, USA, 2017.

[23] M. Ehrenberg, S. Sarkani, and T. A, Mazzuchi. Python Source Code Vulnerability Detection with Named Entity Recognition. In *ElSEVIER Computers & Security*, 2024.

[24] T. Farasat and J. Posegga. Machine Learning Techniques for Python Source Code Vulnerability

Detection. In *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy (ACM CODASPY)*, Portugal, 2024.

[25] T. Farasat, A. Ahmadzai, A. Elsa George, S. Alisina Qaderi, D. Dordevic and J. Posegga. SafePyScript: A Web-Based Solution for Machine Learning-Driven Vulnerability Detection in Python. Available at: *https://arxiv.org/abs/2411.00636*, 2024.

[26] H. Zhang, Z. Li, G Li, L. Ma, Y. Liu, and Z. Jin. Generating adversarial examples for holding robustness of source code processing models. In Proceedings of the AAAI Conference on Artificial Intelligence, pages 1169–1176, 2020.

[27] N. Yefet, U. Alon, and E. Yahav. Adversarial examples for models of code. Proceedings of the ACM on Programming Languages, 4(OOPSLA):1–30, 2020.

[28] J. Henke, G. Ramakrishnan, Z. Wang, A. Albarghouth, S. Jha, and T. Reps. Semantic robustness of models of source code. In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 526–537, 2022.

[29] S. Liu, D. Cao, J. Kim, T. Abraham, P. Montague, S. Camtepe, J. Zhang, Y. Xiang. EaTVul: ChatGPT-based Evasion Attack Against Software Vulnerability Detection. In 33rd USENIX Security Symposium (USENIX Security 24), 2024

[30] Python. Available at https://www.tiobe.com/tiobe-index/, Accessed on: July, , 2025.

[31] VUDENC. Available at https://github.com/LauraWartschinski/VulnerabilityDetection, Accessed on: July. 01, 2025.

[32] Vulnerability Detection. Available at https://github.com/Tf-arch/Python-Source-Code-Vulnerability-Detection?tab=readme-ov-file, Accessed on: July, 2025.

[33] L. Wartschinski, Vudenc-datasets for vulnerabilities (2020). Available at https://zenodo.org/record/3559841#.XeVaZNVG2Hs, Accessed on: January, 2025

[34] T. Farasat and J. Posegga. Enhancing Python Code Security: A Comparison of Machine Learning, ChatGPT, and Static Analysis Methods. In *International Conference on Electrical and Computer Engineering Researches (ICECER 2025)*, Madagascar, 2025.

[35] T. Farasat and J. Posegga. Optimizing Code Embeddings and ML Classifiers for Python Source Code Vulnerability Detection. Available at: *https://arxiv.org/pdf/2509.13134*, 2025.