Master Internship

Cyber Security & AI

Radboud University

Enhancing Vulnerability Detection in Source Code using Adversarial Training

Author:
Pien Rooijendijk
s1054190

Supervisor:
Dr Stjepan Picek
`associate professor in the Digital Security (DiS) group`
`stjepan.picek@ru.nl`

September 3, 2025

Contents

Abstract

With the increasing use of deep learning (DL) models for vulnerability detection in source code, ensuring their robustness against adversarial attacks has become a critical challenge. Recent advances, such as the EaTVul attack framework, have demonstrated the ability to generate adversarial examples that can successfully evade vulnerability detection models. This research aims to implement adversarial training as a defense against such attacks. Using CodeBERT the training dataset is expanded by augmenting it with adversarial example generated by the EaTVul attack framework, examining the model's ability to correctly classify vulnerabilities when tested on adversarial examples. The findings suggest that adversarial improves the model's robustness to adversarial attacks, as the Attack Success Rate (ASR) dropped from 63% to 36% after adversarial training. This paper provides valuable insights for developing more robust AI-based vulnerability detection solutions, making them more suitable for real-world applications.

1 Introduction

Detection of security vulnerabilities in software is a recurring problem when it comes to automating the process (Chakraborty et al., 2021). Static analysis tools result in high false positive, detecting non-vulnerable code as vulnerable (Bardas et al., 2010). In the ideal world these tools should be complete and sound, meaning the vulnerability detector would find all errors in the software, while a sound detector would only report the real vulnerabilities and have no false positives (Bardas et al., 2010). Lint is one of the first analyzers which could find flaws, which were undetected, in code. These analyzers have the potential to find rare occurrences or hidden backdoors in the code. It does not replace the manual review of the source code completely, analyzers only facilitate the process of detecting possible defects and inaccuracies (Stefanović et al., 2020). One of the limitations of using static analyzers is that it is a challenge to develop tests when a new attack or failure mode is discovered (Bardas et al., 2010).
To stay acquainted of emerging vulnerabilities in source code, a more flexible and responsive strategy is required. Deep Neural Networks (DNN) are an upcoming framework for detecting these vulnerabilities. However, the DNNs are not performing as is expected. The research performed by Chakraborty et al., 2021 already showed that there was a reduction in the precision of the model to 11.12% when used on real world datasets, compared to the model which was trained on a synthetic dataset. The models not only exhibit weak performance on the datasets but also lack any substantial robustness to attacks, particularly those involving adversarial examples.
Adversarial examples can exploit vulnerabilities within deep neural networks, posing a significant threat to system security. The study presented by Liu et al., 2024 highlights the vulnerability of deep learning models to adversarial attacks, demonstrating a potential 100% success rate for such exploits. The evasion attack EaTVul highlights the need for more robust defenses within the system to ensure resilience. This study focuses on the nature and effectiveness of evasion attacks, it is essential to consider how defense mechanisms could be integrated into the DL based models to improve its resistance to adversarial examples.

1.1 Background

1.1.1 Deep Learning based Vulnerability Detection

The traditional vulnerability detection methods as static analysis, rely on predefined rules and a human in the loop, which makes it limited in the scope and prone to missing complex or subtle vulnerabilities (Bardas et al., 2010; Novak et al., 2010). DL has the ability to learn patterns and representations from large datasets and offers a more dynamic approach to detecting vulnerabilities in code. The state of the art DL vulnerability detection mechanisms are mainly divided in two categories: graph-based and sequence-based. The sequential models make use of architecture such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, which are effective for processing source code as a sequence of tokens (Lampe and Meng, 2023). The state-of-the art detection models are transformer-based pre-trained DL models, which are pre-trained and fine-tuned on vulnerability detection but also on authorship attribution and code completion (Chen et al., 2021).
Feng et al., 2020 introduced CodeBERT, the first bimodel pre-trained model designed to process both natural language and programming languages across multiple languages. Additionally, the model utilizes encoder-only architectures. In addition to this model, there are several other pre-trained DL models such as PLBART, CodeT5, and CodeGPT (Wang et al., 2024). However, this report primarily focuses on CodeBERT due to its demonstrated strong performance (Feng et al.,

2020). Its robustness is investigated in vulnerability detection in line with the results from Liu et al., 2024.
Automating vulnerability detection using DL models has shown significant promises in recent years. Various studies report impressive performance metrics, with accuracies reaching up to 95% and F1-scores of 90% for detecting vulnerabilities in source code (Ni et al., 2024). However, these models face limitations when applied to real-world scenarios, where they often fail to give the correct classification when tested on real-world data (Chakraborty et al., 2021). Additionally, critical challenges remain, such as understanding the complexity and computational cost of deploying these models on large-scale codebases, and resolving inconsistencies in predictions among other DL models (Steenhoek et al., 2023). Adversarial attacks such as EaTVul (Liu et al., 2024) amplify these challenges by targeting DL models with high success rates. EaTVul not only exposes vulnerabilities in these models but also raises concerns about their robustness and their readiness for real-world applications. This highlights the pressing need to develop and implement stronger defense mechanisms to ensure the reliability and security of vulnerability detection models in real-world deployments.

1.1.2 EaTVul

In the context of vulnerability detection algorithms, adversarial attacks involve modifying code samples such that it disrupts the model's ability to analyze them accurately (Feng et al., 2020). These modifications can include renaming variables, changing code formatting or inserting dead code. Other methods which resemble these attacks are obfuscating techniques to make the code harder for AI models to analyze. The EaTVul attack leverages ChatGPT to generate adversarial code snippets, which are then injected into vulnerable source code. This process aims to evade detection by altering the classification outcomes of DL based vulnerability detection models, exposing critical weaknesses in their robustness.
EaTVul consists of two main phases:

1. **Adversarial data generation**: A surrogate model based on BiLSTM with an attention mechanism is trained to generate adversarial data using ChatGPT with the attention words retrieved from the surrogate model.
2. **Adversarial learning**: The generated adversarial data is used to execute evasion attacks by inserting the adversarial code into vulnerable test cases, showcasing the susceptibility of deep learning models to adversarial attacks. See subsection 8.1 for an example of such adversarial example also in comparison with a benign example.

The following subsections describe the different mechanisms EaTVul uses when performing the evasion attack.

SVM

The SVM is used to identify the important code segments in important non-vulnerable samples, which are the data points that lie closest to the decision boundary of the ML model. The objective is to select the most important non-vulnerable samples and then identify the features that contribute the most to the prediction.

BiLSTM

The most important features are chosen based on the indices which where produced from the SVM. The goal of the adversary is to influence the decision boundary of the vulnerability deep learning algorithm to shift the prediction to non-vulnerable. These important features will be used to generate adversarial data. The stealthiness of the attack is maintained by excluding low-frequency terms. The outcome of the BiLSTM will be features which are deemed to be significant for the EaTVul attack.

Generating Adversarial Examples

ChatGPT generates adversarial examples by leveraging its ability to understand and manipulate text. Important features in the code snippets are identified using attention mechanisms. These features are crucial for the model's decision-making process. ChatGPT uses these identified features to create slightly altered versions of the original code snippets. These alterations are designed to be subtle enough to evade detection by the vulnerability detection model but significant enough to change the model's output. The generated adversarial examples are tested against the vulnerability detection model.
The goal of generating adversarial data is to identify weaknesses in the ML models used for detecting vulnerabilities in software. Adversarial data should include all the important features identified by the attention mechanisms and the data must maintain the code's functionality and should not introduce any syntactic errors or alter its operation. Also, the size of the data should be limited to less than 8 lines to enhance its concealment and make it challenging to detect, according to Liu et al., 2024.

Fuzzy Genetic Algorithm (FGA)

FGA method employs a fuzzy clustering approach to ensure that all reserved members have an opportunity to pass on tho the next generation. It is a genetic algorithm which breeds fuzzy logic controllers as agent programs (Geyer-Schulz, 1998). Fuzzy clustering is a type of clustering algorithm which assigns each data point to multiple clusters with corresponding probabilities instead of a single cluster. They aim to avoid sub-optimal combinations. The two best clusters are chosen based on the magnitude of the centroid, which is also known as the fitness score. In the context of the EaTVul attack, FGA select the best two clusters to get the optimal combination of templates to be fed into the target model, the enhance the success rate of the evasion attack. Enhancing this success rate entails choosing a random seed sample from which FGA produces an optimized adversarial data snippet. The code snippet is added into the vulnerable code and these will be fed into the prediction model.

1.2 Attack model

The attack model is the viewpoint and capabilities of the attacker (Steffan and Schumacher, 2002). These models are useful for the discovery of vulnerabilities in systems. The following is the attack model based on the goal of EaTVul and the capabilities of the attacker in the case of deep learning models used for vulnerability detection.

- Attacker's capability: attacker is allowed to insert a certain number of non-functional statements in arbitrary locations.
- Attacker's knowledge: black-box framework for the deep-learning model. Attacker has no access to the architecture and parameters of the target model(s).
- Attacker's goal: the objective is to deceive the targeted vulnerability detection tools through imperceptible modifications to the inputs. They aim to manipulate a system's decision-making

process by introducing carefully crafted adversarial examples that resemble normal samples but are misclassified by the system.

For a more comprehensive explanation of the EaTVul attack, refer to the paper by Liu et al., 2024.

1.3 Motivation

This research proposes adversarial training as a defense strategy for Deep Learning (DL) models and evaluates its effectiveness in enhancing robustness against the EaTVul attack. EaTVul represents a state-of-the-art adversarial attack specifically designed to compromise DL models, achieving a high success rate as demonstrated by Liu et al., 2024.
The continuous development of such sophisticated attacks enhances the necessity of integrating robust defense mechanisms when designing vulnerability detection algorithms based on DL. Several researchers have identified weaknesses in DL based vulnerability detection models and proposed diverse adversarial attack techniques, with the goal leading to erroneous outputs through strategies like identifier replacement, dead code injection, and structure transformation (Wang et al., 2024). With the source code of EaTVul publicly available, the attack becomes accessible to a wide audience, increasing the risk of it being used to bypass vulnerability detection systems. This research emphasizes the critical need for defensive measures to mitigate these risks and improve the reliability of DL-based algorithms in real-world applications.
By applying adversarial training as a defense strategy to CodeBERT, the model is forced to learn stronger decision boundaries to fit the data. While adding the adversarial examples into the training set, the model should also be able to correctly process benign data. The goal is to achieve a robust model which can resist adversarial attacks and correctly fit benign code samples.

2 Related work

Prior work has been done on generating adversarial examples. This work has mainly been done in the field of image classification, automatic speech recognition and natural language processing. However, these fields are completely different from the field of source code processing (Yu et al., 2023). This process is more challenging since the perturbations that can be applied to get the adversarial example is limited, the source code still needs to be syntactically correct.

2.1 Adversarial Attacks on Code Models

There are many adversarial example generation techniques for DL vulnerability detection models, including methods to change variable names, identifier renaming techniques or variable replacement. CodeBERT-Attack (CBA) is such proposed attack programmed to perform a black-box adversarial attack on CodeBERT (H. Zhang et al., 2024). CBA locates the vulnerabilities and performs perturbations on these positions in the source code and the model generates adversarial examples with the lowest predictive probability. The results included a notable decrease in CodeBERT's performance by 17.5% when subjected to the CBA attack. The results included that when the CBA attack performed on CodeBERT there was a reduction of performance achieved of 17.5%. CodeBERT was used in the CBA attack for adversarial example generation and boosted the generation compared to the random Metropolis-Hastings algorithm modifier (MHM) attack. MHM generates adversarial examples of source code based on Metropolis-Hastings sampling (H. Zhang et al., 2020). The algorithm is a Markov chain Monte Carlo sampling approach.

Yu et al., 2023 already researched three main categories of adversarial example generation methods for source code tasks: changing identifier names, adding dead code, and changing code structure. However, these methods cannot be directly applied to vulnerability detection. This paper aims to generate adversarial samples for DL-based vulnerability detection models that differ in operator granularity, vector representation, and neural network structure. ALERT was the state-of-the-art adversarial example generation method for pre-trained models of code YANG et al., 2022. Recently, researchers have shown that models of code like code2vec and code2seq, can output different results for the two code snippets sharing the same operational semantics, one of which is generated by renaming some variables in the other.

2.2 Defenses on Code Models

The study by Liu et al., 2024 highlighted that the defense perspective of the attack remains an open area of research. The effectiveness of the attack on the defense side can be downgraded by several defense mechanisms. Liu et al., 2024 suggested input sanitization, anomaly detection and ensemble models as possible defenses, all of which could significantly decrease the success rate of the EaTVul attack. Additionally, a widely used method for enhancing the robustness of DL models is adversarial training, which will be discussed in the following section.

2.2.1 Recognizing Adversarial Examples

Recognizing adversarial examples involves using machine learning to analyze the patterns of adversarial inputs and detecting anomalous features indicative of these attacks. It detects individual adversarial examples. It exploits the statistical distinguishability of adversarial examples to design an outlier detection system, where it is integrated into the model. The model is trained with adversarial examples as their own class Grosse et al., 2017.

2.2.2 Input Sanitization

Input sanitization removes, replaces or escapes malicious inputs according to the context of sensitive program operation such that those inputs may not cause the model to perform unintended operations (Shar and Tan, 2013 & Malik et al., 2024). Static code attributes that reflect the characteristics of these methods could be used to predict vulnerability. It makes sure that the input data is safe and valid before it is processed. This is important in the context of for machine learning models that are susceptible to evasion attacks, since evasion attacks work at inference time (Malik et al., 2024). Sanitization techniques apply preprocessing steps to the input data to ensure that the quality, integrity and reliability of the data is maintained.
Key components:

- Validation: checks the input for expected format, value ranges, or patterns.
- Transformation: apply transformations to inputs in a way that neutralizes adversarial manipulations.
- Filtering: remove or modify parts of the input that are unnecessary, potentially harmful, or do not contribute to the task at hand.
- Encoding: converting the input into a standard, safe representation to avoid potentially harmful data from being executed or processed incorrectly.

Since input sanitization is used as a defense mechanism before training consisting of sanitizing the data, the technique requires access to the training data (Cinà et al., 2024). The underlying idea is that adversarial samples can be removed by outlier detection techniques, as they have to be very different from samples within the same class label to induce the model to learn a significantly-different decision function.

2.2.3 Anomaly Detection

Anomaly detection used as a defense for machine learning models used for vulnerability detection identifies deviations from normal behavior within a dataset (Chandola et al., 2009). By monitoring the inputs and outputs of the detection models, anomaly detection can identify attempts to manipulate the model through adversarial attacks. The detection can flag unusual code patterns that may signify novel or rare vulnerabilities which were adversarially created (Lazarevic et al., 2003).

2.2.4 Adversarial training

Adversarial training is proved to be most effective among these methods, which simply requires to train models on adversarial examples progressively (Bai et al., 2020). It solves a min-max game with a simple process of training on the adversarial examples until the model learns to classify them correctly as adversarial examples (Athalye et al., 2018). The main challenge is to solve the maximization problem. Adversarial training is used to make the model more robust to attack or to reduce its test error on benign inputs (Kurakin et al., 2016). The model behaves more normally when facing adversarial examples than standard trained models (Bai et al., 2021).
The idea of adversarial training is to inject adversarial examples into the training set, continually generating new adversarial examples at every step of training. Batch normalization should be used since it was originally developed for small models, so the normalization is used for scaling. Datasets with few labeled examples where overfitting is the concern, adversarial training reduces the test error. Kurakin et al., 2016 state that when security against adversarial example is a concern, adversarial training is the method which provides the most security of any known defense, while losing only a small amount of accuracy.

Debicha et al., 2021 showed already that training the model with adversarial training can improve the robustness of intrusion detection systems. Relating this to vulnerability detection, the model can also be adversarial trained to be more robust. They found that adversarial training can improve to some extent the robustness of deep learning-based intrusion detection systems. However, it comes with a trade-off of slightly decreasing detector accuracy on unattacked network traffic.
Tian et al., 2024 introduced HardVD which enhanced semantic richness by expanding token capacity limits and leverage cross-modal adversarial reprogramming techniques to reduce the number of training parameters, thereby minimizing training efforts in terms of time and data. It segments source code into lines that adhere to human coding logic. The scheme naturally deals with variable token lengths and maximizes the capacity to learn comprehensive semantics from source code. It makes use of adversarial reprogramming, which allows repurposing a model pretrained in a source domain to perform a target-domain task, where the domains and tasks can be entirely different.

3 Experimental Setup

3.1 Dataset

The primary objective of this research is to implement a defense against evasion attacks to improve the robustness of deep learning models used for vulnerability detection. The methodology includes training a deep learning model to detect code vulnerabilities while ensuring robustness against adversarial attacks. This is achieved by integrating adversarial training into the model-training process.
Vulnerability detection is a binary classification task to identify whether a source code sample is vulnerable or not. The dataset as described in Table 1 contains the descriptions of the C projects. This dataset was also used by Liu et al., 2024 for generating adversarial examples. The original data generated by EaTVul consists of abstract syntax trees (AST), a tree-shaped representation of the source code (J. Zhang et al., 2019). The tree represents the syntactic structure of the source code. Multiple datasets were used, including:

Dataset	Description
Asterisk Project	open-source project containing vulnerabilities.
OpenSSL Project	security-focused open-source library.
CWE119 Dataset	focuses on buffer errors.
CWE399 Dataset	contains resource management errors.

Table 1: The figure provides an overview of the dataset descriptions containing C/C++ code samples in AST representation

The data includes examples of vulnerable, non-vulnerable, and adversarial instances generated by EaTVul. The adversarial examples are crafted from vulnerable test cases. These were generated by inserting lines of code to produce examples that could potentially evade detection. The dataset details are summarized in Table 2.

Data	# Vulnerable	# Non-vulnerable	# Total	# Adversarial examples
Asterisk	70	1177	1247	50
OpenSSL	333	400	733	50
CWE119	2170	5951	8121	200
CWE399	215	585	800	200

Table 2: The figure provides an overview of the dataset distribution and the adversarial examples used.

3.2 Target Model

The training process is configured with specific parameters to optimize performance as specified in Table 3. The architecture includes a custom classification head, designed specifically for vulnerability detection and inspired by previous research, such as VulnCodeBERT. This classification head is composed of multiple layers: a dropout layer for regularization, two fully connected layers with 768 to 3072 and 3072 to 3072 dimensions, a tanh activation function, and a final classification layer that outputs logits for the two classes (vulnerable and non-vulnerable). The pooled representation from CodeBERT's hidden states is used as input to the classification head. The final output logits are passed to a binary cross-entropy loss function with logits, weighted to handle class imbalance.

Parameter	**Description**
Batch Size	8
Number of epochs	15
Learning Rate	1×10^{-5}
Weight Decay	0.05
Layers in Classification Head	• Dropout layer for regularization. • Fully connected layers: 768 $\rightarrow$ 3072 and 3072 $\rightarrow$ 3072 dimensions. • Tanh activation function. • Final classification layer outputs logits for the two classes (vulnerable and non-vulnerable).
Input Representation	Pooled representation from CodeBERT's hidden states, used as input to the classification head.
Loss Function	Binary cross-entropy with logits.

Table 3: Training parameters and Model architecture.

This setup aligns with Thapa et al., 2022 and is designed to evaluate the robustness of a pre-trained CodeBERT when fine-tuned for the task of vulnerability detection. It also provides a foundation for testing the proposed defense mechanism, adversarial training.

3.3 Adversarial Training

The process of adversarial training begins by generating adversarial examples from the existing dataset. This process was already done by EaTVul. These modified inputs are then labeled as vulnerable, ensuring that the adversarial dataset is consistent and well-annotated. Once the adversarial examples are generated, they are combined with the benign training data to form the dataset to perform adversarial training. The dataset for standard training and the adversarial dataset have the same size to ensure that the results are not dependent on the size of the dataset. This combined dataset exposes the model to a diverse range of inputs, including both benign and adversarially examples.
During adversarial training (see Figure 1), the model trains on the adversarial dataset. The existing training setup, with binary cross-entropy loss weighted to account for class imbalance, is used without requiring major changes. Adversarial examples are presented to the model during each epoch, ensuring that it learns to identify features distinguishing benign inputs from adversarial ones. The dropout layers and weight decay parameters help to prevent overfitting and maintain a robust learning process.
The model is evaluated during each epoch to indicate the effectiveness of adversarial training. After the training process, the model is tested on a separate dataset that includes both original and adversarial examples and also on a benign test dataset. The metrics accuracy, F1-score, loss and Attack Succes Rate (ASR) with respect to adversarial inputs are measured and compared against a baseline model that was trained without adversarial examples (the benign model).

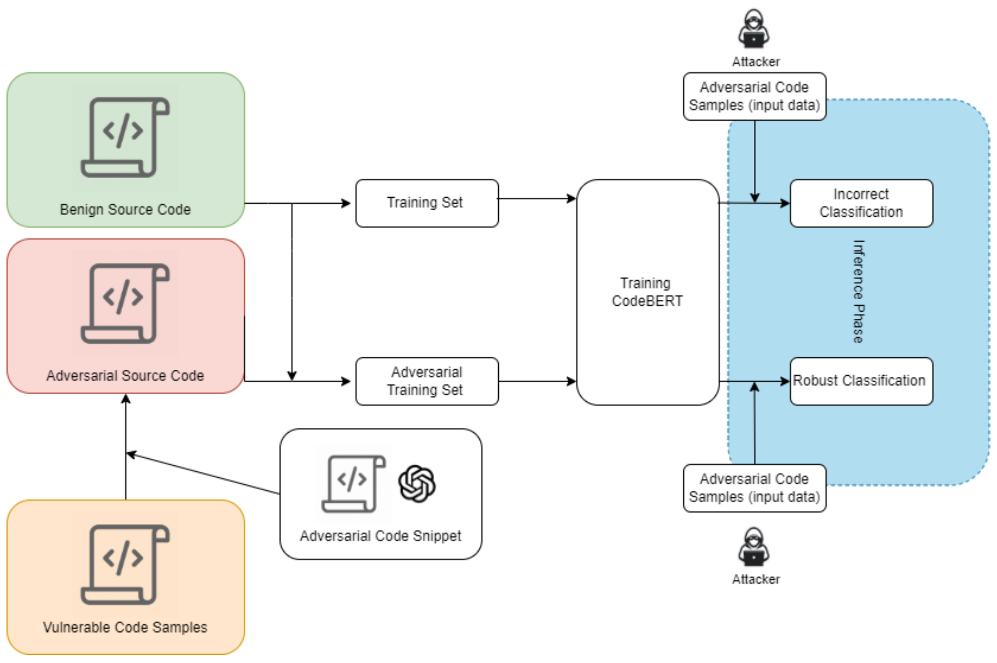


Figure 1: The figure shows the implementation of adversarial training on CodeBERT, showing the flow of data through various stages, including the generation of adversarial examples, model training with both benign and adversarial inputs, and the inference phase to test the model's performance to enhance its robustness against evasion attacks.

4 Results

The baseline model was trained on benign samples only including the Asterisk, OpenSSL, CWE119 and CWE399 datasets as described in Table 2 with the dataset split into training (80%), evaluation (10%) and test (10%) data. Then the baseline model achieved an accuracy of 90% when tested on benign test samples only. Also the adversarially trained model reached an accuracy of 90% when tested on the benign test data. Both the baseline model and the adversarially trained model were tested on benign and adversarial AST code samples, as can be seen in Figure 2. When the baseline CodeBERT model was tested on a subsection of the adversarial examples generated by EaTVul in AST representation, the accuracy dropped significantly to 26%. This highlights the effectiveness of the adversarial attack in degrading the model's performance. EaTVul achieved an ASR of 63% when attacking CodeBERT without adversarial training. However, after adversarial training, the ASR decreased to 36% when tested on the same subsection of adversarial examples indicating an improvement in the model's robustness against these code snippets.
The loss plot Figure 3 compares the training progress of the benign and adversarially trained models over 15 epochs. While the training of the benign model converged within the given epochs, the adversarially trained model had not yet converged, suggesting that adversarial training introduces additional challenges in optimization.

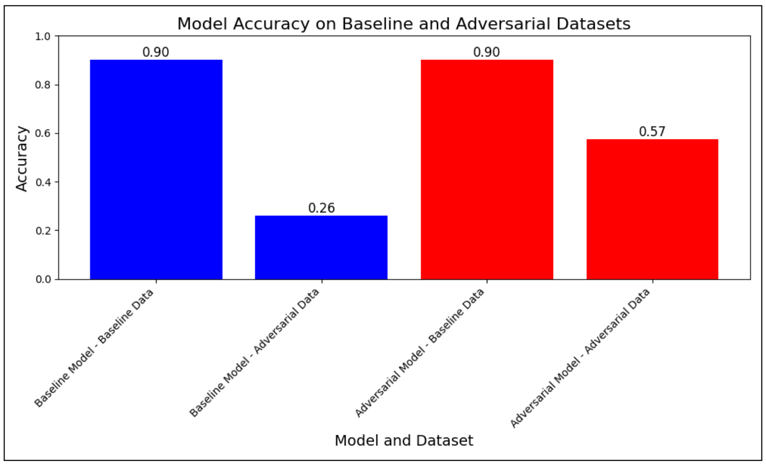
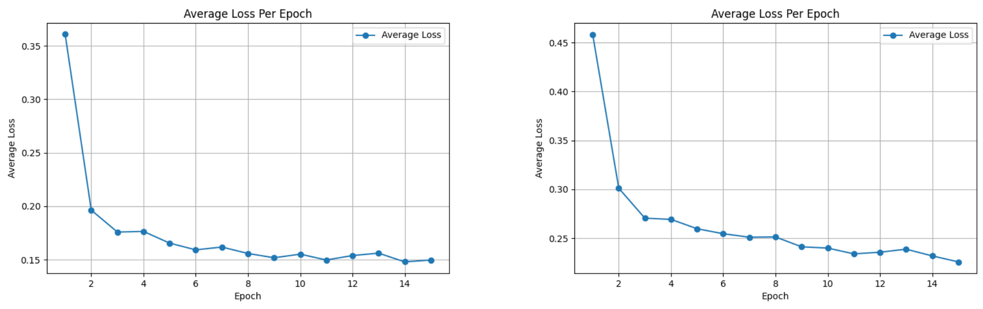


Figure 2: The accuracy of the CodeBERT model at inference time before and after adversarial training using AST representations as training and test data. The blue bars represent the baseline model (without adversarial training) and the red bars represent the model after adversarial training.



(a) Loss of the benign model using AST representations.

(b) Loss of the adversarially trained model using AST representations.

Figure 3: Loss plots of the trained models. The left plot shows the loss of the benign model, which was trained on benign samples only. The right plot shows the loss of the adversarially trained model, which was trained on both benign and adversarial samples.

5 Discussion

5.1 Interpretation of Results

This study investigated the effectiveness of the defense mechanism adversarial training in enhancing the robustness of deep learning-based vulnerability detection systems against the EaTVul adversarial attack. The results indicate that adversarial training significantly improves the model's resilience, reducing the attack success rate by 27%, while still maintaining the model to classify benign data samples. What stands out in the results is that the ASR is not as high as described by Liu et al., 2024. This difference is due to the use of a different model for vulnerability detection instead of the

original model used in the EaTVul study. The variation in model architectures and training specifics likely accounts for the observed difference in ASR, highlighting the importance of model choice in evaluating adversarial attack effectiveness.
The substantial improvement in robustness through adversarial training underscores its efficacy as a primary defense strategy against sophisticated attacks like EaTVul. This aligns with existing literature that highlights adversarial training as a cornerstone in enhancing model security (Kurakin et al., 2016).
The demonstrated effectiveness of adversarial training suggests that developers of vulnerability detection systems using DL should incorporate this strategy during the model training phase to enhance security against adversarial threats. Implementing input sanitization and anomaly detection can serve as additional layers of defense, thereby creating a more secure and trustworthy environment for using deep learning models in real-world applications.

5.2 Contributions

This research makes several key contributions to the field of DL security. Firstly, it provides a comprehensive evaluation of the defense mechanism adversarial training against a sophisticated adversarial attack, EaTVul, thereby offering actionable insights for enhancing model robustness. Secondly, by utilizing diverse datasets, including Asterisk, OpenSSL, CWE119, and CWE399, the study ensures that the findings are already generalizable across these different types of vulnerabilities and codebases. Lastly, the integration of adversarial training within the vulnerability detection pipeline sets a precedent for future research aimed at developing more secure and reliable DL models.

5.3 Limitations

Despite the promising results, this study has certain limitations. The evaluation was primarily conducted on specific datasets, which may not encompass the full spectrum of real-world vulnerabilities and coding practices. The code databases and also the attack were based on C/C++ code samples. Ideally there would be diversity in the programming languages such that this defense is not code specific. While adversarial training proved effective, it also increases the computational complexity and training time, potentially limiting its scalability for larger models and datasets. Furthermore, the study focused on the EaTVul attack; future research should explore the applicability of the proposed defenses against a broader range of adversarial strategies. The proposed defense involved augmenting the training set by incorporating the generated adversarial examples and retraining CodeBERT. However, this approach does not improve the robustness of the model at the model-level, which makes it difficult to resist multiple types of adversarial attacks simultaneously. Finally, while the defense was effective, it would have been more accurate and robust if new adversarial examples were generated in each iteration of the adversarial training process. Unfortunately, due to difficulties in reproducing the results of the original study by Liu et al., 2024, we were unable to follow this ideal methodology. As a result, the adversarial examples used in training may not have fully captured the diversity and complexity of attacks that could occur in a real-world setting.

5.4 Future Research

Future research should aim to evaluate the proposed defense mechanisms against a wider variety of adversarial attacks to assess their generalizability and robustness. Exploring the combination of multiple defensive strategies, such as integrating adversarial training with anomaly detection,

could yield a more robust model. Furthermore, investigating the trade-offs between model performance, computational cost, and security can inform the development of more efficient and scalable defense solutions. Lastly, extending the research to include more diverse programming languages and complex code databases will help towards making DL-based vulnerability detection effective in real-world applications.
The latter could involve converting the AST code samples to C/C++ code to test the model's performance on a programming language. While preliminary results indicate that the approach holds promise, the accuracy achieved on C code samples is lower compared to the results obtained with the original dataset (see Figure 6 and Figure 7). As can be seen in the figures in subsection 8.2 is that the adversarially trained model underfits the data and does not classify the benign nor the adversarially data successfully. Further research of the preprocessing steps and potentially fine-tuning the model specifically on C code may be required to bridge this gap and improve performance on using different languages for vulnerability detection.

6 Conclusion

In conclusion, this study shows the importance of implementing robust defense mechanisms, particularly adversarial training, to safeguard deep learning-based vulnerability detection models against adversarial attacks like EaTVul. By addressing the vulnerabilities exposed by such adversarial threats, this research contributes to the development of more secure and reliable deep learning systems, ultimately advancing the state of cybersecurity in software development. The results indicate that augmenting adversarial examples into the training process improves the model's performance in vulnerability detection. Moving forward, further research is needed to refine adversarial training techniques and better capture other existing adversarial attacks. Expanding the scope of the research to include a variety of programming languages and more complex codebases will be crucial in advancing AI-based vulnerability detection systems, making them ready for real-world applications.

7 Reference list

References

Athalye, A., Carlini, N., & Wagner, D. (2018). Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *International conference on machine learning*, 274–283.

Bai, T., Chen, J., Zhao, J., Wen, B., Jiang, X., & Kot, A. (2020). Feature distillation with guided adversarial contrastive learning. *arXiv preprint arXiv:2009.09922*.

Bai, T., Luo, J., Zhao, J., Wen, B., & Wang, Q. (2021). Recent advances in adversarial training for adversarial robustness. *arXiv preprint arXiv:2102.01356*.

Bardas, A. G., et al. (2010). Static code analysis. *Journal of Information Systems & Operations Management*, *4*(2), 99–107.

Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2021). Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, *48*(9), 3280–3296.

Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*, *41*(3), 1–58.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Cinà, A. E., Grosse, K., Demontis, A., Biggio, B., Roli, F., & Pelillo, M. (2024). Machine learning security against data poisoning: Are we there yet? *Computer*, *57*(3), 26–34.

Debicha, I., Debatty, T., Dricot, J.-M., & Mees, W. (2021). Adversarial training for deep learning-based intrusion detection systems. *arXiv preprint arXiv:2104.09852*.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. (2020). Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Geyer-Schulz, A. (1998). Fuzzy genetic algorithms. In H. T. Nguyen & M. Sugeno (Eds.), *Fuzzy systems: Modeling and control* (pp. 403–459). Springer US. https://doi.org/10.1007/978-1-4615-5505-6_12

Grosse, K., Manoharan, P., Papernot, N., Backes, M., & McDaniel, P. (2017). On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280*.

Kurakin, A., Goodfellow, I., & Bengio, S. (2016). Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*.

Lampe, B., & Meng, W. (2023). A survey of deep learning-based intrusion detection in automotive applications. *Expert Systems with Applications*, *221*, 119771.

Lazarevic, A., Ertoz, L., Kumar, V., Ozgur, A., & Srivastava, J. (2003). A comparative study of anomaly detection schemes in network intrusion detection. *Proceedings of the 2003 SIAM international conference on data mining*, 25–36.

Liu, S., Cao, D., Kim, J., Abraham, T., Montague, P., Camtepe, S., Zhang, J., & Xiang, Y. (2024). {Eatvul}:{chatgpt-based} evasion attack against software vulnerability detection. *33rd USENIX Security Symposium (USENIX Security 24)*, 7357–7374.

Malik, J., Muthalagu, R., & Pawar, P. M. (2024). A systematic review of adversarial machine learning attacks, defensive controls and technologies. *IEEE Access*.

Ni, C., Shen, L., Xu, X., Yin, X., & Wang, S. (2024). Learning-based models for vulnerability detection: An extensive study. *arXiv preprint arXiv:2408.07526*.

Novak, J., Krajnc, A., & Žontar, R. (2010). Taxonomy of static code analysis tools. *The 33rd International Convention MIPRO*, 418–422.

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

Steenhoek, B., Rahman, M. M., Jiles, R., & Le, W. (2023). An empirical study of deep learning models for vulnerability detection. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2237–2248.

Stefanović, D., Nikolić, D., Dakić, D., Spasojević, I., & Ristić, S. (2020). Static code analysis tools: A systematic literature review. *Ann. DAAAM Proc. Int. DAAAM Symp*, *31*(1), 565–573.

Steffan, J., & Schumacher, M. (2002). Collaborative attack modeling. *Proceedings of the 2002 ACM symposium on Applied computing*, 253–259.

Thapa, C., Jang, S. I., Ahmed, M. E., Camtepe, S., Pieprzyk, J., & Nepal, S. (2022). Transformer-based language models for software vulnerability detection. *Proceedings of the 38th Annual Computer Security Applications Conference*, 481–496.

Tian, Z., Li, H., Sun, H., Chen, Y., & Chen, L. (2024). Hardvd: High-capacity cross-modal adversarial reprogramming for data-efficient vulnerability detection. *Information Sciences*, 121370.

Wang, Y., Chen, Y., Zhao, Y., Gong, Z., Chen, J., & Hao, D. (2024). Mutual learning-based framework for enhancing robustness of code models via adversarial training. *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 1484–1496.

YANG, Z., SHI, J., HE, J., & LO, D. (2022). Natural attack for pre-trained models of code.(2022). *Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA*, 21–29.

Yu, X., Li, Z., Huang, X., & Zhao, S. (2023). Advulcode: Generating adversarial vulnerable code against deep learning-based vulnerability detectors. *Electronics*, *12*(4), 936.

Zhang, H., Li, Z., Li, G., Ma, L., Liu, Y., & Jin, Z. (2020). Generating adversarial examples for holding robustness of source code processing models. *Proceedings of the AAAI Conference on Artificial Intelligence*, *34*(01), 1169–1176.

Zhang, H., Lu, S., Li, Z., Jin, Z., Ma, L., Liu, Y., & Li, G. (2024). Codebert-attack: Adversarial attack against source code deep learning models via pre-trained model. *Journal of Software: Evolution and Process*, *36*(3), e2571.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., & Liu, X. (2019). A novel neural source code representation based on abstract syntax tree. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 783–794.

8 Appendix

8.1 Adversarial Code Examples

```
bool wmemset() {}
bool wmemset(bool a) {}
bool wmemset(bool a, bool b) {}
bool wmemset(bool a, bool b, bool c) {}
bool wmemset(bool a, bool b, bool c, bool d) {}
bool wmemset(bool a, bool b, bool c, bool d, bool e) {}
bool ALLOCA() {}
bool ALLOCA(bool a) {}
bool ALLOCA(bool a, bool b) {}
bool ALLOCA(bool a, bool b, bool c) {}
bool ALLOCA(bool a, bool b, bool c, bool d) {}
bool ALLOCA(bool a, bool b, bool c, bool d, bool e) {}

using namespace std;
namespace CWE121_Stack_Based_Buffer_Overflow__CWE806_wchar_t_alloca_ncpy_72 {
void badSink(vector<wchar_t*> dataVector) {};
void bad() {
  wchar_t* data;
  vector<wchar_t*> dataVector;
  wchar_t* dataBuffer;
  dataBuffer = (wchar_t*)ALLOCA(100 * sizeof(wchar_t));
  data = dataBuffer;
  wmemset(data, L 'A', 100 - 1);
  data[100 - 1] = L '\\0';
  dataVector.insert(dataVector.end(), 1, data);
  dataVector.insert(dataVector.end(), 1, data);
  dataVector.insert(dataVector.end(), 1, data);
  badSink(dataVector);
}
void goodG2BSink(vector<wchar_t*> dataVector) {};
static void goodG2B() {
  wchar_t* data;
  vector<wchar_t*> dataVector;
  wchar_t* dataBuffer;
  dataBuffer = (wchar_t*)ALLOCA(100 * sizeof(wchar_t));
  data = dataBuffer;
  wmemset(data, L 'A', 50 - 1);
  data[50 - 1] = L '\\0';
  dataVector.insert(dataVector.end(), 1, data);
  dataVector.insert(dataVector.end(), 1, data);
  dataVector.insert(dataVector.end(), 1, data);
  goodG2BSink(dataVector);
}
void good() { goodG2B(); }
}
using namespace CWE121_Stack_Based_Buffer_Overflow__CWE806_wchar_t_alloca_ncpy_72;
int main(int argc, char*[] argv) {
  srand((unsigned)time(NULL));
  printLine("Calling good()...");
  good();
  printLine("Finished good()");
  printLine("Calling bad()...");
  bad();
  printLine("Finished bad()");
  return 0;
}
int main() { return 0; }
```

Figure 4: Adversarial test example of the CWE119 dataset.

```
using namespace std;
namespace CWE121_Stack_Based_Buffer_Overflow__CWE806_wchar_t_alloca_ncpy_72
{
void badSink(vector<wchar_t *> dataVector);

void bad()
{
    wchar_t * data;
    vector<wchar_t *> dataVector;
    wchar_t * dataBuffer = (wchar_t *)ALLOCA(100*sizeof(wchar_t));
    data = dataBuffer;
    wmemset(data, L'A', 100-1);
    data[100-1] = L'\0';
    dataVector.insert(dataVector.end(), 1, data);
    dataVector.insert(dataVector.end(), 1, data);
    dataVector.insert(dataVector.end(), 1, data);
    badSink(dataVector);
}
static void goodG2B()
{
    wchar_t * data;
    vector<wchar_t *> dataVector;
    wchar_t * dataBuffer = (wchar_t *)ALLOCA(100*sizeof(wchar_t));
    data = dataBuffer;
    wmemset(data, L'A', 50-1);
    data[50-1] = L'\0';
    dataVector.insert(dataVector.end(), 1, data);
    dataVector.insert(dataVector.end(), 1, data);
    dataVector.insert(dataVector.end(), 1, data);
    goodG2BSink(dataVector);
}

void good()
{
    goodG2B();
}
using namespace CWE121_Stack_Based_Buffer_Overflow__CWE806_wchar_t_alloca_ncpy_72;

int main(int argc, char * argv[])
{
    srand( (unsigned)time(NULL) );
    printLine("Calling good()...");
    good();
    printLine("Finished good()");
    printLine("Calling bad()...");
    bad();
    printLine("Finished bad()");
    return 0;
}
```

Figure 5: Benign test example of the CWE119 dataset.

8.2 Preliminary Results using C/C++ code

When C/C++ representations were used, the accuracy was 95% when the benign model was tested on benign samples and 70% when the adversarially trained model was tested on benign samples, see Figure 6. When both models were tested on adversarial examples, the accuracy dropped to 30% for the benign model and to 14% for the adversarially trained model. The model trained on C/C++ code has an increase of 5% compared to the model trained on AST source code.

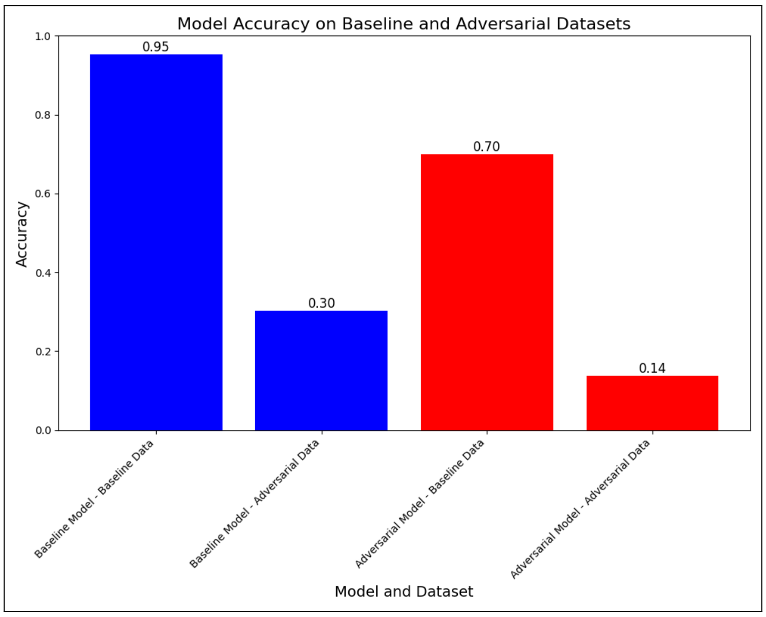
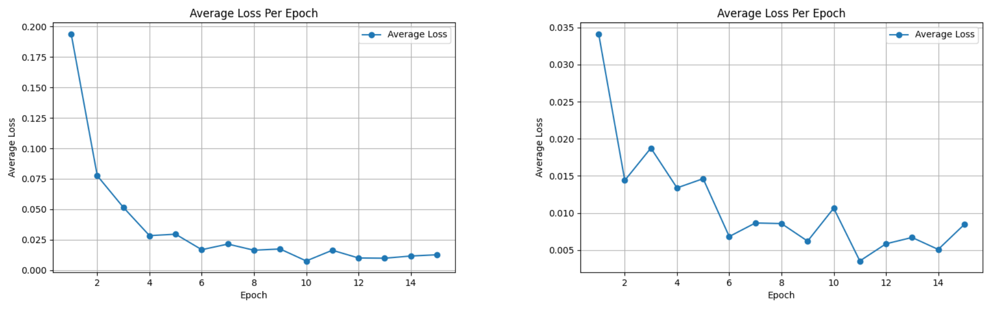


Figure 6: The accuracy of the CodeBERT model at inference time before and after adversarial training using C/C++ representations as training and test data. The blue bars represent the baseline model (without adversarial training) and the red bars represent the model after adversarial training.



(a) Loss of the benign model using C/C++ code representations.

(b) Loss of the adversarially trained model using C/C++ code representations.

Figure 7: Loss plots of the models trained on C/C++ code samples.