Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection

Niklas Risse and Marcel Böhme, *MPI-SP, Germany*

https://www.usenix.org/conference/usenixsecurity24/presentation/risse

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection

Niklas Risse
MPI-SP, Germany

Marcel Böhme
MPI-SP, Germany

Abstract

Recent results of machine learning for automatic vulnerability detection (ML4VD) have been very promising. Given only the source code of a function f, ML4VD techniques can decide if f contains a security flaw with up to 70% accuracy. However, as evident in our own experiments, the same top-performing models are unable to distinguish between functions that contain a vulnerability and functions where the vulnerability is patched. So, how can we explain this contradiction and how can we improve the way we evaluate ML4VD techniques to get a better picture of their actual capabilities?

In this paper, we identify overfitting to unrelated features and out-of-distribution generalization as two problems, which are not captured by the traditional approach of evaluating ML4VD techniques. As a remedy, we propose a novel benchmarking methodology to help researchers better evaluate the true capabilities and limits of ML4VD techniques. Specifically, we propose (i) to augment the training and validation dataset according to our cross-validation algorithm, where a semantic preserving transformation is applied during the augmentation of either the training set or the testing set, and (ii) to augment the testing set with code snippets where the vulnerabilities are patched.

Using six ML4VD techniques and two datasets, we find (a) that state-of-the-art models severely overfit to unrelated features for predicting the vulnerabilities in the testing data, (b) that the performance gained by data augmentation does not generalize beyond the specific augmentations applied during training, and (c) that state-of-the-art ML4VD techniques are unable to distinguish vulnerable functions from their patches.

1 Introduction

Recently several different publications have reported high scores on vulnerability detection benchmarks using machine learning (ML) techniques [1,12–15,28]. The resulting models seem to outperform traditional program analysis methods, e.g. static analysis, even without requiring any hard-coded knowledge of program semantics or computational models. So, does this mean that the problem of detecting security vulnerabilities in software is solved? Are these models actually able to detect security vulnerabilities, or do the reported scores provide a false sense of security?

Even though ML4VD techniques achieve high scores on vulnerability detection benchmark datasets, there are still situations in which they fail to meet expectations when presented with new data. For example, it is possible to apply small semantic preserving changes to augment the testing dataset of a state-of-the-art model and then measure whether the model changes its predictions. If it does, it would indicate a dependence of the prediction on unrelated features. Examples of such transformations are identifier renaming [18,38,39,41,42], insertion of unexecuted statements [18, 35, 39, 41] or replacement of code elements with equivalent elements [2, 21]. The impact of augmenting testing data using these transformations has been explored for many different software-related tasks and the results seem to be clear: Learning-based models fail to perform well when testing data gets augmented using semantic preserving transformations of code [2, 5, 18, 30, 35, 38, 39, 41, 42].

In our own experiments, we were able to reproduce the findings of the literature and made additional observations: ML4VD techniques that were trained on typical training data for vulnerability detection are also unable to distinguish between vulnerable functions and their patched counterparts. If a patched function is also predicted as vulnerable, this indicates that the prediction critically depends on features unrelated to the presence of a security vulnerability.

It has previously been proposed to reduce the dependence on unrelated features by augmenting not just the testing data but also the training data [5, 18, 35, 38, 39, 41, 42]. Indeed, this seems to restore the lost performance back to previous levels, but does it really reduce the dependence on unrelated features, or are the models just overfitting to different unrelated features of the data?

In this paper, we propose a novel benchmarking methodology that can be used to evaluate the capabilities of ML4VD techniques by using data augmentation. First, we propose

Algorithm 1, in which a selected semantic preserving transformation is applied to the training dataset of a model, and a *different* transformation is applied to the testing dataset. When repeated for all possible pairs out of a set of transformations, the resulting scores provide a better measure of overfitting to the unrelated features that are introduced by the semantic preserving transformations during training data augmentation. Second, we propose Algorithm 2, in which a trained model is evaluated on a testing dataset that contains both vulnerable programs and their respective patches. The results provide a measure of the model's ability to generalize to a modified vulnerability detection setting.

In order to validate Algorithm 1 and Algorithm 2 empirically, we selected six state-of-the-art ML4VD techniques. All evaluated ML4VD techniques happen to be token-based large language models (LLMs). As our selection criterion, we defined the top-performing ML4VD techniques on the most widely known ML vulnerability detection benchmark CodeXGLUE [24, 25] that are available as open source. This gave us ranks 1, 2, 6, 10, and 12 of the leaderboard, all of which are token-based LLMs. In fact, 9 of the Top-10 solutions on the leaderboard are token-based LLMs. By applying Algorithm 1 and Algorithm 2 in our empirical study of six state-of-the-art ML4VD techniques and three datasets, we confirmed that ML4VD techniques continue to leverage unrelated features when deciding whether a function contains a vulnerability.

For Algorithm 1, we implemented 11 different semantic preserving transformations for data augmentation and evaluated the trained models using two popular vulnerability detection datasets. As expected, we find a strong benefit of training data augmentation (69.0% and 66.2% average restoration of accuracy/f1-score for the two datasets) when the transformations applied to training and testing datasets are the same. However, we find no improvement in performance when the transformations applied to training and testing datasets are different. In fact, we even find an additional 30.2% and 77.5% average *decrease* in accuracy/f1-score for the two datasets. In other words, ML4VD techniques still severely overfit to the specific label-unrelated features introduced by training data augmentation. The improvement in performance gained by data augmentation only applies to the specific type of transformations used during training.

For Algorithm 2, we introduce a new dataset, VulnPatchPairs, which contains 26.2k C functions and is derived from the CodeXGLUE/Devign vulnerability detection dataset [43]. Exactly half of the functions in VulnPatchPairs contain security vulnerabilities. The other half are patched versions of the first half.[1] We investigated six ML4VD techniques using VulnPatchPairs and evaluated their ability to generalize from their typical training data to VulnPatchPairs, and vice versa. To our surprise, all six ML4VD techniques that were trained on a typical training dataset were unable to distinguish between the vulnerable functions and their patched counterparts in VulnPatchPairs. On average, the accuracy turned out to be worse than random guessing. The trained models are unable to generalize from a standard vulnerability detection dataset to the modified setting.

In summary, this paper contributes two novel algorithms that can be used to uncover major problems of ML4VD techniques that are not detected using the standard evaluation setup: Overfitting to semantic preserving code changes and the inability to generalize between related vulnerability detection settings. Additionally, we provide an empirical evaluation of six state-of-the-art ML4VD techniques using the proposed methodology.

- ⋆ We present a general methodology consisting of two algorithms, that can be used to evaluate ML4VD techniques.
- ⋆ We show empirically, that state-of-the-art ML4VD techniques overfit to the unrelated features introduced by semantic preserving transformations during data augmentation.
- ⋆ We introduce *VulnPatchPairs*, a new dataset that contains vulnerable C function and the corresponding patched versions of the same functions. It is available at `https://github.com/niklasrisse/VPP`.
- ⋆ We demonstrate, that six state-of-the-art ML4VD techniques are not able to distinguish between the vulnerable and patched functions in VulnPatchPairs.
- ⋆ We publish all of our code and results for reproducibility. They are available at `https://github.com/niklasrisse/USENIX_2024`.

2 Related Work

One of the main tools to study the limits of ML4VD techniques are semantic preserving transformations of code. Previous work [2, 5, 18, 21, 22, 30, 35, 38, 39, 41, 42] proposed methods to generate semantic preserving transformations for source code datasets and investigated their impact when used to augment testing data of learned models.

Many of the works that reported the failures of learned models when testing data was augmented also investigated training data augmentation using their respective methods [5,18,35,38,39,41,42]. A common finding in all of these publications is that training data augmentation using a specific type of semantic preserving transformation leads to improved performance on testing sets that have been augmented the same way. But does the performance gained by data augmentation generalize beyond the specific augmentations applied during training?

Some of the publications that propose methods for data augmentation [5, 18, 35, 38, 41] take it a step further; they

[1] See Section 4.4 for details.

```
1 static inline int coeff_unpack_golomb(GetBitContext *gb, int qfactor, int
      qoffset)
2 {
3     int coeff = dirac_get_se_golomb(gb);
4     const int sign = FFSIGN(coeff);
5     const unsigned sign = FFSIGN(coeff);
6     if (coeff)
7         coeff = sign*((sign * coeff * qfactor + qoffset) >> 2);
8     return coeff;
9 }
```

(a) Code Snippet

```
1 static inline int coeff_unpack_golomb(GetBitContext *gb, int qfactor, int
      qoffset)
2 {
3     int coeff = dirac_get_se_golomb(gb);
4     const int sign = FFSIGN(coeff);
5     const unsigned sign = FFSIGN(coeff);
6     if (coeff)
7         coeff = sign*((sign * coeff * qfactor + qoffset) >> 2);
8     if (0)
9         coeff = 666;
10    return coeff;
11 }
```

(b) Transformed Code Snippet

Figure 1: Example of a simple semantic preserving transformation. The change (orange) has no effect on the vulnerability label. Both code snippets contain a security vulnerability (integer overflow in line 4). The code was taken from the Ffmpeg GitHub repository (URL: `https://github.com/FFmpeg/FFmpeg/commit/92da2309`) and is part of the CodeXGLUE/Devign dataset.

augment the training data using a slightly different but related type of transformation than for the testing data. For example, Henkel et al. [18] apply their gradient-based approach for identifier renaming to the training data and a random renaming strategy to the testing data. Similarly, Yang et al. [38] apply their method for variable renaming to a training dataset and the method proposed by Zhang et al. [42] to a testing dataset. All of these works find an improved performance when the training dataset is augmented in a similar way than the testing dataset. However, the transformations used for augmenting the training and testing data in these publications are all similar in type, e.g. they both rename identifiers. But does the performance also improve when training data is augmented in a different way than the testing data? Our work aims to fill this gap in the literature by carrying out a thorough empirical study that considers a diverse set of 11 different transformations, six state-of-the-art ML4VD techniques, and two high-quality datasets.

Similar to other related publications listed above [18, 35, 38], Rahman et al. [31] investigate overfitting of ML4VD techniques to variable and API names by transforming them in the testing data. Additionally, they propose a new method to address the overfitting based on causal learning, which aims to disable models from using superficial features (e.g. variable names) entirely. While their approach seems to be effective to avoid overfitting for concrete and simple transformations (e.g. changing variable names), the authors do not investigate how their method performs when faced with unseen semantic preserving transformations, that were not specifically trained for. Our proposed methodology can be used as a tool to do this, which allows to draw conclusions about overfitting to unrelated features fundamentally, irrespective of the type of transformation applied to training- and testing data.

In order to evaluate the general capabilities of ML4VD techniques, we collected a new dataset (VulnPatchPairs), which contains both vulnerable functions and their respective patches. The collection of a pairwise vulnerability-patch dataset has been proposed by previous work [4, 8, 29], e.g. for the research field of automated fixing. However, to the best of our knowledge, we are the first to utilize such a dataset to evaluate the general capabilities of ML4VD techniques.

Two recently published papers [10, 36] report poor generalization capabilities of different ML-based techniques (e.g. LLMs and GNNs) when evaluated on functions from unseen git projects. Our Algorithm 2 also investigates the generalization capabilities of ML4VD techniques, but using a different setup, in which functions in the evaluation data belong to a modified vulnerability detection setting (e.g. vulnerable functions and their patches), but can be from the same projects.

3 Methodology

We propose a novel benchmarking methodology to help researchers better evaluate advances in ML4VD techniques. The methodology consists of two parts, Algorithm 1 (A1) and Algorithm 2 (A2).

3.1 Data Augmentation

A central component of our methodology is data augmentation, and the expectations for vulnerability detection models that emerge from using code transformations for data augmentation.

We define data augmentation as the application of one or multiple code transformations onto all code snippets of a given code snippet dataset $CD \subset \mathcal{C}$, where $\mathcal{C}$ is a space that represents all possible code snippets $c \in \mathcal{C}$ in a given programming language.

A code transformation $t : \mathcal{C} \to \mathcal{C}$ is a function that maps from and to $\mathcal{C}$. Let's assume we have an oracle function $g : \mathcal{C} \to \{0, 1\}$, which maps from the space of code snippets $\mathcal{C}$ to either 0 or 1. The oracle function g represents the ground truth, i.e. it shows whether a code snippet c does (1) or does not (0) contain a security vulnerability. For a given code snippet dataset $CD \subset \mathcal{C}$, a code transformation t can be characterized by its effect on $g(t(c)) \; \forall \; c \in CD$:

Semantic Preserving Transformation. We call a transformation t_p semantic preserving w.r.t. CD, if the changes

Algorithm 1 Detecting Overfitting to Code Changes

Input: Semantic Preserving Transformations $T := \{t_1, ..., t_N\}$
Training Dataset Tr
Testing Dataset Te
ML Training Method *train_model*
ML Evaluation Method *evaluate_model*
Performance Metric M

$MLM[Tr] = train\_model(Tr)$
$score[MLM[Tr], Te] = evaluate\_model(MLM[Tr], Te, M)$
for each $t_k \in T$ **do**
  $Te_k = t_k(Te)$ *// testing data augmentation*
  $score[MLM[Tr], Te_k] = evaluate\_model(MLM[Tr], Te_k, M)$
  $effect[Tr, Te_k] = score[MLM[Tr], Te_k] - score[MLM[Tr], Te]$
  $Tr_k = t_k(Tr)$ *// training data augmentation*
  $MLM[Tr_k] = train\_model(Tr_k)$
  $score[MLM[Tr_k], Te_k] = evaluate\_model(MLM[Tr_k], Te_k, M)$
  $MLM[Tr_k, Te_k] = score[MLM[Tr_k], Te_k] - score[MLM[Tr], Te]$
  for each $t_{j \neq k} \in T$ **do**
    $Te_j = t_j(Te)$ *// testing data augmentation*
    $score[MLM[Tr_k], Te_j] = evaluate\_model(MLM[Tr_k], Te_j, M)$
    $effect[Tr_k, Te_j] = score[MLM[Tr_k], Te_j] - score[MLM[Tr], Te]$
  end for
end for

Output: $output_{A1.1} = (\sum_k effect[Tr, Te_k])/N$
$output_{A1.2} = (\sum_k effect[Tr_k, Te_k])/N$
$output_{A1.3} = (\sum_k \sum_{j \neq k} effect[Tr_k, Te_j])/(N(N-1))$

introduced by applying it do not affect the ground truth vulnerability label, $g(c) = g(t_p(c)) \; \forall \; c \in CD$. Figure 1 shows an example of a simple semantic preserving transformation applied to a real-world code snippet.

Label Inverting Transformation. We call a transformation t_d label inverting w.r.t. CD, if the changes introduced by applying it change the ground truth vulnerability label, $g(c) \neq g(t_d(c)) \; \forall \; c \in CD$. In other words, a label inverting transformation either adds or removes a vulnerability from a code snippet.

In general, we expect a vulnerability detection model to correctly predict, whether a given code snippet contains a security vulnerability, independent of any semantic preserving or label inverting transformations that have been previously applied to the code snippet. Specifically, we can formulate the following expectations:

1. If we change a code snippet without affecting the vulnerability label (semantic preserving transformation), we expect a vulnerability detection tool to compute the same correct prediction as before applying the change.

2. If we add or remove a vulnerability from a code snippet (label inverting transformation), we expect a vulnerability detection tool to still deliver a correct prediction, or i.e. we expect it to change its prediction with the ground truth label of the code snippet.

In the following sections, we present two algorithms, which allow to evaluate ML4VD techniques using the two formulated expectations.

3.2 A1: Detecting Overfitting to Code Changes

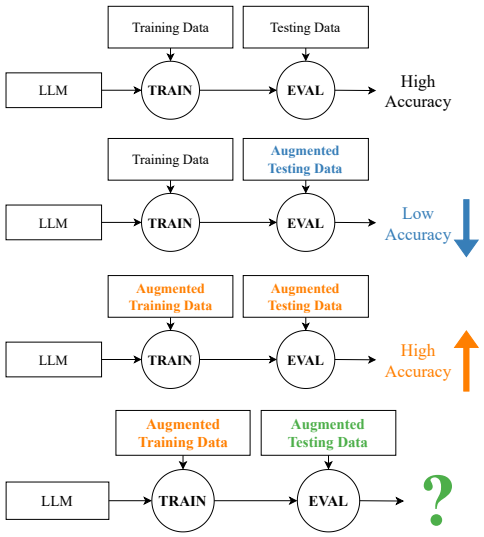


Figure 2: Visualization of Algorithm 1, which we created to detect overfitting of ML4VD techniques to unrelated features introduced by data augmentation. Colors represent that either only testing data is augmented (blue), training- and testing data are augmented using the same (orange), or different augmentation methods (green).

The goal of Algorithm 1 is to measure, whether ML4VD techniques overfit to augmentations of their training data that are unrelated to the respective vulnerability labels and whether the performance gained by data augmentation generalizes beyond the specific augmentations applied during training. We provide a simple visualization of the idea behind the algorithm in Figure 2, the algorithm itself in Algorithm 1, and a description of the most important parts in the following paragraphs. We use the colors blue, orange, and green, to connect the basic ideas of the algorithm with the experimental results across the paper[2].

What are the inputs? The inputs of Algorithm 1 are a set of different semantic preserving transformations $T := \{t_1, ..., t_N\}$, a training dataset Tr, a testing dataset Te, a ML training method *train_model*, a ML evaluation method *evaluate_model*, and a performance metric M. The training and testing datasets Tr and Te consist of code-label pairs (c_i, v_i), with $c_i \in \mathcal{C}$ representing code snippets and $v_i \in \{0, 1\}$ representing labels that indicate the absence (0) or presence (1) of security vulnerabilities in the respective code snippets. The method *train_model* can utilize the training dataset Tr to train a machine learning model $MLM : \mathcal{C} \rightarrow \{0, 1\}$, which maps from the space of code snippets $\mathcal{C}$ to either 1 (vulnerability) or 0 (no vulnerability). The method *evaluate_model* can use the performance metric M to quantify and aggregate the performance of a trained model MLM on a testing dataset Te into a single number between 0 (bad) and 1 (perfect).

[2] See Figure 2, Algorithm 1, Figure 5, Figure 7, Figure 7c and Table 2.

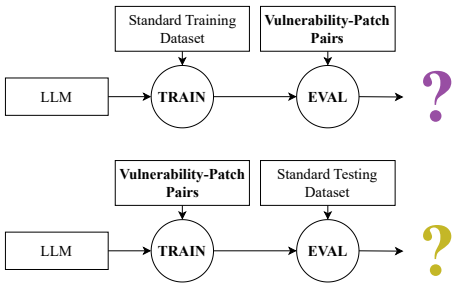


Figure 3: Visualization of Algorithm 2, which we created to test whether ML4VD techniques are able to generalize to a modified setting, which requires to distinguish between vulnerabilities and patches.

What is computed? Algorithm 1 computes the average effects of (a) augmenting the testing data of the selected ML technique using transformations $t_k \in T$ ($out\,put_{A1.1}$), (b) using the same transformations to also augment the training data ($out\,put_{A1.2}$), and (c) using different transformations to also augment the training data ($out\,put_{A1.3}$).

In lines 4-6, Algorithm 1 computes the effect of augmenting the testing dataset Te with the transformation t_k on the performance of the trained model $MLM[Tr]$. The result is $effect[Tr, Te_k]$, the absolute difference between the scores of $MLM[Tr]$ on the clean testing dataset Te and the augmented testing dataset Te_k. In other words, $effect[Tr, Te_k]$ quantifies how many points in score are lost if we augment the testing dataset with transformation t_k. $out\,put_{A1.1}$ aggregates this intermediate result over all transformations $t_k \in T$.

In lines 7-10, Algorithm 1 goes a step further and computes the effect of both augmenting the training dataset Tr and the testing dataset Te using the same transformation t_k. The result is $effect[Tr_k, Te_k]$, the absolute difference between scores of $MLM[Tr_k]$ on the augmented testing dataset Te_k and $MLM[Tr]$ on the testing dataset Te. In other words, $effect[Tr_k, Te_k]$ quantifies how many points in score are lost if we augment both the training and the testing dataset with transformation t_k. $out\,put_{A1.2}$ aggregates this intermediate result over all transformations $t_k \in T$.

In lines 12-14, the algorithm computes the effect of augmenting the testing dataset using a *different* transformation t_j than for the training dataset. The result is $effect[Tr_k, Te_j]$, the absolute difference between scores of $MLM[Tr]$ on the testing dataset Te and $MLM[Tr_k]$ on the augmented testing Te_j. In other words, $effect[Tr_k, Te_j]$ quantifies how many points in score are lost if we augment the training and the testing dataset with different transformations t_k and t_j. $out\,put_{A1.3}$ aggregates this intermediate result over all transformations $t_k \in T, t_{j \neq k} \in T$.

How can the results be used? Using this algorithm, researchers can effectively evaluate new ML4VD techniques. Specifically, for a selected technique researchers can answer

Algorithm 2 Distinguish between Vulnerability and Patch

Input: Standard Training Dataset Tr
Standard Testing Dataset Te
Vulnerability-Patch Training Dataset VPTr
Vulnerability-Patch Testing Dataset VPTe
ML Training Method *train_model*
ML Evaluation Method *evaluate_model*
Performance Metric M

1: $MLM[Tr] = train\_model(Tr)$
2: $MLM[VPTr] = train\_model(VPTr)$
3: $score[MLM[Tr], Te] = evaluate\_model(MLM[Tr], Te, M)$
4: $score[MLM[Tr], VPTe] = evaluate\_model(MLM[Tr], VPTe, M)$
5: $score[MLM[VPTr], VPTe] = evaluate\_model(MLM[VPTr], VPTe, M)$
6: $score[MLM[VPTr], Te] = evaluate\_model(MLM[VPTr], Te, M)$

Output: $out\,put_{A2.1} = score[MLM[Tr], Te]$
$out\,put_{A2.2} = score[MLM[Tr], VPTe]$
$out\,put_{A2.3} = score[MLM[VPTr], VPTe]$
$out\,put_{A2.4} = score[MLM[VPTr], Te]$

the following questions:

1. How much does the performance of the selected ML technique decrease if we augment the code snippets for testing without affecting the vulnerability labels? Answer: On average, the performance does change by $out\,put_{A1.1}$ points.

2. How much performance of the selected ML technique can be restored, if we augment the training code snippets in a similar way than the testing code snippets? Answer: On average, $out\,put_{A1.2} - out\,put_{A1.1}$ of the initial decrease can be restored.

3. How much performance of the selected ML technique can be restored, if we augment the training code snippets in a different way than the testing code snippets? Answer: On average, $out\,put_{A1.3} - out\,put_{A1.1}$ of the initial decrease can be restored.

4. Does the selected ML technique overfit to specific augmentations of the training data that are unrelated to the respective vulnerability labels? Answer: If $out\,put_{A1.2} >> out\,put_{A1.3}$: Yes, otherwise No.

3.3 A2: Distinguish between Vulnerability and Patch

The main goal of Algorithm 2 is to evaluate, whether ML4VD techniques are able to generalize from their typical training data to a modified setting, which requires to distinguish security vulnerabilities from their patches. Additionally, the algorithm also aims to evaluate the reverse, or i.e. whether ML4VD techniques that were trained to distinguish between vulnerabilities and their patches are able to perform well on standard testing data. We provide a simple visualization of the idea behind the algorithm in Figure 3, the algorithm itself in Algorithm 2, and a description of the most important parts in the following paragraphs. We use the colors purple and

yellow to connect the basic ideas of the algorithm with the experimental results across the paper [3].

What are the inputs? In addition to the inputs of Algorithm 1, Algorithm 2 requires a special vulnerability-patch testing dataset $VPTe$ and a vulnerability-patch training dataset $VPTr$. $VPTe$ and $VPTr$ also consist of code snippets $c_i \in \mathcal{C}$ and vulnerability labels $v_i \in \{0,1\}$, but for every code snippet c_j with label $v_j = 0$, they also contain a snippet $c_{l \neq j}$ with $v_l = 1$ which represents the patched version of c_j. The relationship between a code snippet and its patched version can be characterized as a label inverting transformation $t_{PATCH} : \mathcal{C} \rightarrow \mathcal{C}$, which maps code snippets c_j to their patched versions c_l.

What is computed? The purpose of Algorithm 2 is to quantify the ability of the selected ML technique to generalize between two related vulnerability detection settings. The first setting, represented by the standard training and testing datasets Tr and Te, consists of code snippets, which either contain or do not contain a vulnerability. This setting is most frequently used in the related literature [1, 12–15, 28]. The second setting, represented by the vulnerability-patch training and testing datasets $VPTr$ and $VPTe$, consists of vulnerable code snippets and their respective patches. As formulated in Section 3.1, a perfect vulnerability detection model should be able to perform well in both settings, irrespective of the setting on which it was trained. In other words, a vulnerability detection model should be able to generalize between the settings.

In total, Algorithm 2 computes four scores. In line 3, Algorithm 2 computes the score of a model $MLM[Tr]$, which was trained on the standard training dataset Tr, on the standard testing dataset Te. This score represents the standard evaluation and serves as a baseline for the other scores.

In line 4, Algorithm 2 computes the score of a model $MLM[Tr]$, which was trained on the standard training dataset Tr, on the vulnerability-patch testing dataset $VPTe$. When compared to the first score, this result serves as a measure of $MLM[Tr]$s ability to generalize to the modified setting, which requires to distinguish vulnerabilities from their patches.

In line 5, Algorithm 2 computes the score of a model $MLM[VPTr]$, which was trained on the vulnerability-patch training dataset $VPTr$, on the vulnerability-patch testing dataset $VPTe$. Again, this score serves as a baseline for the other scores.

In line 6, Algorithm 2 computes the score of a model $MLM[VPTr]$, which was trained on the vulnerability-patch training dataset $VPTr$, on the standard testing dataset Te. When compared to the third score, this result serves as a measure of $MLM[VPTr]$s ability to generalize back to the standard vulnerability detection setting.

The four computed scores are returned as the outputs of the algorithm ($output_{A2.1}$, $output_{A2.2}$, $output_{A2.3}$ and $output_{A2.4}$).

[3] See Figure 3, Algorithm 2, and Table 3.

How can the results be used? Using Algorithm 2, researchers can effectively evaluate whether the high scores of ML4VD techniques are specific to the testing datasets on which they were computed. Specifically, for a selected technique researchers can answer the following questions:

1. Does the performance of the selected ML technique generalize from a standard vulnerability detection dataset to a modified setting, which requires to distinguish vulnerabilities from their patches? Answer: The selected ML technique can distinguish between vulnerabilities and their patches with performance $output_{A2.2}$, compared to a score of $output_{A2.1}$ on the standard testing dataset.

2. Does the performance of the selected ML technique generalize back to the standard vulnerability detection setting when it is explicitly trained to distinguish vulnerabilities from their patches? Answer: The selected ML technique achieves a score of $output_{A2.4}$ on the standard testing dataset, compared to a score of $output_{A2.3}$ in the modified setting.

4 Experimental Setup

4.1 Research Questions

Our objective is to validate empirically, whether the two proposed algorithms can be used to evaluate state-of-the-art ML4VD techniques. Specifically, we aim to answer the following research questions.

RQ.1 How is the performance of ML4VD techniques affected, if we augment the input code snippets without affecting the vulnerability labels? (a) Can we measure a decrease in performance, if we augment the testing data of ML4VD techniques using semantic preserving transformations? (b) Does training data augmentation using the same transformations restore the initial performance? (c) Are there differences between the individual transformations?

RQ.2 Do ML4VD techniques overfit to specific augmentations of their training data that do not affect the respective vulnerability labels? Can we still restore the performance, if we augment the training dataset with a different semantic preserving transformation than the testing dataset?

RQ.3 Are the high scores of ML4VD techniques specific to benchmark datasets or do they generalize to a modified vulnerability detection setting? (a) Are state-of-the-art ML4VD techniques able to distinguish between vulnerable functions and their patches? (b) Does training to distinguish between vulnerable functions and their patches improve the performance on standard testing data?

Table 1: The semantic preserving transformations that we used in our experiments.

Identifier	Type	Description
t_1	Identifier Renaming	Rename all function parameters to a random token.
t_2	Statement Reordering	Reorder all function parameters.
t_3	Identifier Renaming	Rename the function.
t_4	Statement Insertion	Insert unexecuted code.
t_5	Statement Insertion	Insert comment.
t_6	Statement Reordering	Move the code of the function into a separate function.
t_7	Statement Insertion	Insert white space.
t_8	Statement Insertion	Define additional void function and call it from the function.
t_9	Statement Removal	Remove all comments.
t_{10}	Statement Insertion	Add code from training set as comment.
t_{11}	Random Transformation	One transformation sampled from $\{t_1,\ldots,t_{10}\}$ is applied to each function.

4.2 Semantic Preserving Transformations

One of the central components of algorithms 1 and 2 are semantic preserving transformations of code. The most common semantic preserving transformations that are used in the literature to investigate the limits of learned models for source code related tasks are identifier renaming [18, 35, 38, 39, 41, 42], insertion of unexecuted statements [18, 35, 39, 41], replacement of statements with equivalent statements [21], reordering of unrelated statements [27], deletion of unexecuted statements (e.g. comments) [21], or combinations of the before mentioned [18, 35, 41].

Table 1 shows the 11 semantic preserving transformations that we implemented for the experiments presented in this paper. We tried to cover all types of transformations commonly used in the literature. The table lists all transformations, categorizes them by type, and provides short descriptions for each of them.

Since ML4VD techniques are ultimately aimed at detecting security vulnerabilities in real-world code, augmenting code using our semantic preserving transformations should also result in code that looks natural, or i.e. looks like it could occur in real-world software. To address this, we experimentally confirmed that our semantic preserving transformations do not decrease the naturalness (measured by cross entropy) of the code. We provide more information on this as supplementary material in Appendix A.

4.3 Vulnerability Detection Datasets

We use two publicly available vulnerability detection datasets for our experiments.

CodeXGLUE/Devign. CodeXGLUE is a machine learning benchmark for code understanding and generation [24]. It consists of several datasets for different source code related tasks. In our experiments, we only use the dataset for vulnerability detection, which is based on the Devign dataset [43]. Throughout this paper, we refer to this dataset as the *CodeXGLUE/Devign dataset* or just as the CodeXGLUE dataset. The CodeXGLUE dataset contains 26.4k C functions, from which 45.6% contain vulnerabilities, i.e. the dataset is fairly balanced. The types of vulnerabilities were not formally classified, but based on the collection process the authors found most vulnerabilities in the dataset to be memory-related, e.g. memory leaks, buffer overflows, memory corruption, or crashes. The authors of the CodeXGLUE benchmark also maintain a leaderboard [25], which tracks the performance of popular learning-based techniques on the different datasets of the benchmark.

VulDeePecker. The other vulnerability detection dataset that we use in this paper is the Code Gadget Database, which was introduced with the VulDeePecker bug detector [23]. We refer to this dataset as the *VulDeePecker dataset*. The original dataset contains 61.6k C/C++ code samples derived from the NVD [7] and the SARD [6], from which 17.7k contain vulnerabilities, mainly buffer (CWE-119) and resource management errors (CWE-399).

4.4 New Dataset: VulnPatchPairs

In order to investigate the ability of ML4VD techniques to distinguish between vulnerabilities and their patches (RQ3), we collected a new dataset, which we call *VulnPatchPairs*. We provide a simple visualization of the collection process for VulnPatchPairs in Figure 4.

VulnPatchPairs is an extension of the CodeXGLUE/Devign vulnerability detection dataset [43], which consists of C functions from two popular open source repositories, FFmpeg[4] and Qemu[5]. The creators of the dataset describe the collection process in their original publication [43]. As a first step, they filtered the selected repositories for security-related commits using a list of keywords. Then, they invested 600 work hours of a four-person team of security researchers to classify the security-related commits into vulnerability-fix commits (VFCs) and non vulnerability-fix commits (non-VFCs) and extracted the respective functions before the commits were applied as vulnerable (before VFCs) and non-vulnerable (before non VFCs) functions. The actual patched versions of the functions (after the VFCs were applied) are not part of their original dataset. However, for each function of their dataset, the authors released the respective commit ID from the two open-source repositories as additional information. We used this information to extract the actual patched versions of the vulnerable functions in the CodeXGLUE dataset from the FFmpeg and Qemu repositories and created a new dataset: VulnPatchPairs.

We manually verified 100 randomly chosen functions from the CodeXGLUE dataset that were labelled as vulnerable. We found 68 out of these 100 functions to actually contain a security vulnerability, 23 to contain no security vulnerability, and 9

[4] FFmpeg repository: `https://github.com/FFmpeg/FFmpeg`
[5] Qemu repository: `https://github.com/qemu/qemu`

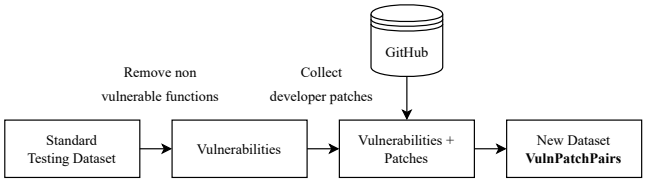


Figure 4: Visualization of the collection process for our new dataset VulnPatchPairs.

with no decision after 10 minutes of manual effort. The results are available at `https://github.com/niklasrisse/VPP_label_accuracy`. Even though these results show that there are inaccurate labels in the CodeXGLUE dataset, we do not expect any changes to our main findings. While the absolute performance might change if we used perfectly labeled data, we expect the relative performance (e.g., augmented vs non-augmented) to remain comparable for all of our individual evaluations.

In total, VulnPatchPairs consists of 26.2k C functions from the two open source repositories FFmpeg and Qemu. Exactly half (13.1k) of the 26.2k functions contain security vulnerabilities and were copied from the CodeXGLUE/Devign vulnerability detection dataset. The other 13.1k are the respective patches of the vulnerable functions, which we extracted from the open-source repositories. We published VulnPatchPairs as supplementary material in an open GitHub repository[6].

4.5 Machine Learning Techniques

We selected six state-of-the-art ML4VD techniques for our experiments.

Selection Criteria. In order to select techniques that represent the state-of-the-art of ML4VD, we chose the top techniques from the CodeXGLUE leaderboard [25] for which the authors provide open-source implementations. Measured by citations[7] (496) and GitHub Stars[8] (1.2k), CodeXGLUE is the most well-known benchmark for source code related machine learning techniques. The vulnerability detection dataset of the benchmark [43] is also highly cited (407 citations[9]) and widely used to evaluate ML4VD techniques for automatic vulnerability detection.

Selected Techniques. Based on the described criteria, we selected UniXcoder [13], CoTexT [28], VulBERTa [15], PLBart [1], and CodeBERT [12] for our experiments. At submission time of this paper, the six techniques hold rank 1 (UniXcoder), rank 2 (CoTexT), rank 6 (VulBERTa), rank 10 (PLBart), and rank 12 (CodeBERT) on the CodeXGLUE leaderboard for vulnerability detection [25]. In addition to the five techniques from the CodeXGLUE leaderboard, we selected GraphCodeBERT (abbreviated as *GraphCB* in Table 3) [14], a technique related to CodeBERT, which utilizes graph representations of source code during the training process.

Model Details. Since the selected techniques belong to the same family of techniques, they also share the same basic architecture:

1. Tokenization: A given code function is split into tokens (small sequence of characters that forms a semantic entity), based on a given strategy.

2. Embedding: Tokens are transformed into numbers, usually by indexing via a learned vocabulary and the addition of positional information.

3. Transformer Network: Several steps of parametrized computation are applied, resulting in a final embedding.

4. Prediction Layer: The final layer of the model is a parametrized prediction layer, which computes an output number between 0 and 1 based on the final embedding of the previous step. The output number represents the predicted probability that the input function contains a security vulnerability.

During model training, the parametrized computational layers of steps 3. and 4. are optimized to compute the correct predictions for given training data. The six selected techniques mainly differ in tokenization strategy, training data, optimization objective, and the specific transformer network architecture used. We provide additional information on the specific models in Appendix C.

The authors of all six techniques provide publicly available implementations of their techniques [1, 12–15, 28].

4.6 Model Training Pipeline

We used a similar training setup for all model instances that we trained for our experiments.

Pre-training. All models that we train in our experiments have been pre-trained by the authors of the respective publications using various source code datasets. The size of the pre-training datasets spans from 2.3 million (VulBERTa) to 680 million code snippets (PLBart). The original publications provide more information on the pre-training datasets [1, 13, 15, 28]. We use the pre-trained models released by the authors of the respective techniques as starting points for our pipeline and finetune the models on our selected datasets.

Data split. For the CodeXGLUE/Devign dataset, we used the train-/validation-/testing dataset split provided by the authors of the benchmark [24], which resulted in a training dataset with 21k functions, a validation dataset with 2.7k functions, and a testing dataset with 2.7k functions. For the VulDeePecker dataset, we used the split provided by Hanif et

[6] VulnPatchPairs: `https://github.com/niklasrisse/VPP`
[7] `https://api.semanticscholar.org/CorpusID:231855531`
[8] `https://github.com/microsoft/CodeXGLUE`
[9] `https://api.semanticscholar.org/CorpusID:202539112`

al. [15], which resulted in a training dataset with 128.1k functions, a validation dataset with 16k functions, and a testing dataset with 16k functions. The split for VulnPatchPairs is derived from the split of CodeXGLUE, such that *all and only* vulnerable functions in training, validation, and testing sets, respectively, of CodeXGLUE were taken as training, validation, and testing sets of VulnPatchPairs, augmented by their corresponding patches.

Pre-processing. For the VulDeePecker dataset, we removed all duplicate functions and also replaced all label-revealing tokens (e.g. comment with token "bad" above a vulnerable function) that we found by manual inspection of the dataset with randomly selected tokens. For the CodeXGLUE and VulnPatchPairs datasets, we applied no additional preprocessing steps.

Hyperparameters. For all six ML4VD techniques, we used the pre-trained models and tokenizers provided by the respective authors as starting points for our experiments. Similar to Hanif et al. [15], we noticed a relatively quick convergence of our performance metrics in our initial experiments on the validation dataset (after 2-7 epochs), which is why we trained each model instance for 10 epochs. For all model-specific hyperparameters, we used the values that were reported in the original papers [1, 12–15, 28]. Consult our published training scripts[10] for the complete list of hyperparameter values that we used.

Performance metrics. We tracked and quantified the performance of our trained models on the selected testing datasets using several commonly used performance metrics. We report six metrics in this paper: Accuracy, f1-score, precision, recall, false positive rate (FPR), and false negative rate (FNR).

For CodeXGLUE/Devign as a balanced dataset (45.6% vulnerable functions), we use accuracy as the main performance metric, since it is also used exclusively in the CodeXGLUE benchmark [24] and on the leaderboard [25].

For VulDeePecker as a relatively imbalanced dataset (28.7% vulnerable functions) we use the f1-score as the main performance metric. The f1-score is defined as the harmonic mean of precision and recall and is most suitable when the positive class (in our case vulnerable functions) is the minority class in an imbalanced dataset.

Hardware. We used a setup of five Nvidia A100 GPUs, each equipped with 40 GB RAM. One run of all our experiments takes approximately 60 days on a single A100 GPU.

5 Experimental Results

RQ.1 Testing- and Training Data Augmentation

We investigate, whether (a) testing data augmentation using semantic preserving transformation decreases the performance of state-of-the-art ML4VD techniques, whether (b) training data augmentation using the same transformations restores the performance towards previous levels, and whether (c) there are differences between the individual transformations.

Methodology. We used Algorithm 1 to investigate all three questions. We ran the algorithm for each ML technique and dataset separately and measured the outcomes using the respective preferred performance metrics (see Section 4.6). We did not only record the outcomes after completing the full training of the respective models but also after each training epoch in order to observe the progression of the learning process.

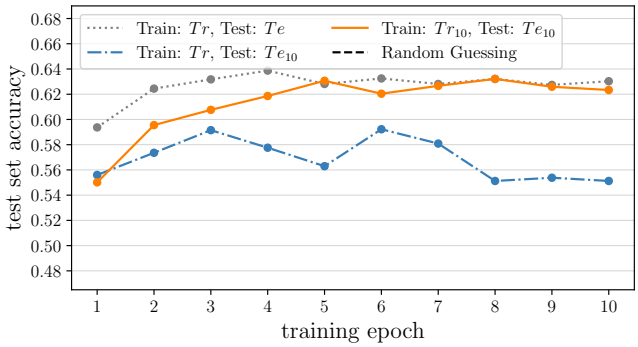
Results. Figure 5a shows the test set accuracy of different VulBERTa models measured after each of the ten training epochs. We can observe, that augmenting the testing dataset Te with transformation t_{10} leads to a substantial drop in accuracy, represented by the gap between the dotted gray and blue graphs in the figure. We can also observe, that augmenting the training dataset Tr with the same transformation t_{10} as the testing dataset, restores the accuracy back to previous levels (orange graph).

Figure 5b extends the results of Figure 5a to all semantic preserving transformations $t_k \in T$, and to all six ML4VD techniques. Instead of showing the accuracy for each training epoch, we only use the maximum accuracy across the full training. Across all six ML4VD techniques, we can observe, that augmenting the testing dataset Te with transformations $t_k \in T$ (represented by the blue boxplots), on average, leads to a drop in accuracy compared to evaluations on the standard testing dataset Te (represented by the horizontal lines with stars). In parallel to Figure 5a we can also observe that, on average, training data augmentation using the same transformation as for testing data augmentation leads to a restoration of the observed drops in accuracy (represented by the orange boxplots).

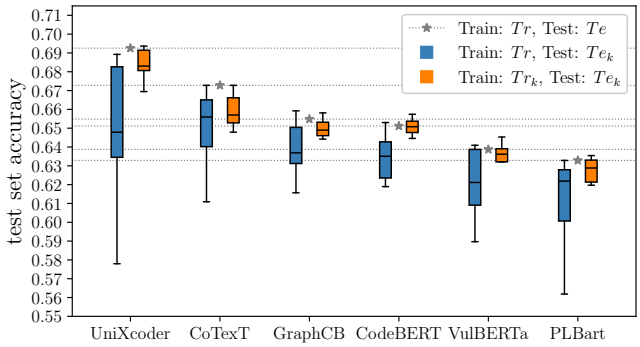
Table 2 summarizes the outputs of Algorithm 1 for all six ML4VD techniques and both datasets. Specifically, it shows the average recorded changes in the respective metrics, when only the testing dataset was augmented (blue columns, $output_{A1.1}$), when the training and testing datasets were augmented using the same transformation (orange columns, $output_{A1.2}$), and when the training and testing datasets were augmented using a different transformation (green columns, $output_{A1.3}$). We observe that, on average, augmenting the testing dataset leads to a drop in accuracy/f1-score (-0.025 for CodeXGLUE, -0.043 for VulDeePecker), and augmenting the training dataset using the same transformation restores that decrease towards previous levels. On average, approximately 69.0% (CodeXGLUE) and 66.2% (VulDeePecker) of the lost accuracy/f1-score is restored.

Figure 6 shows the impact on accuracy caused by augmenting only the testing data using the individual transformations $t_k \in T$ ($impact(t_k) := accuracy[MLM[Tr], Te_k] - accuracy[MLM[Tr], Te]$) for all six ML4VD techniques. In other words, Figure 6 displays the severity of the performance

[10] GitHub: https://github.com/niklasrisse/USENIX_2024



(a) Test set accuracy over ten training epochs of different models trained with VulBERTa on the CodeXGLUE/Devign dataset. Augmenting the testing set Te with transformation t_{10} (blue) decreases the accuracy, but applying the same transformation also to the training dataset Tr (orange) restores the accuracy back to previous levels.



(b) Extension of the results in Figure 5a, for all transformations $t_k \in T$, and for all six ML4VD techniques. The boxplots represent distributions of the resulting accuracies. Augmenting the testing set Te with transformations $t_k \in T$ (blue boxplots) decreases the accuracy, but applying the same transformation also to the training dataset Tr (orange boxplots) partially restores the accuracy, although not always to its previous levels.

Figure 5: Effects of augmenting the testing data and the training data using the same semantic preserving transformations.

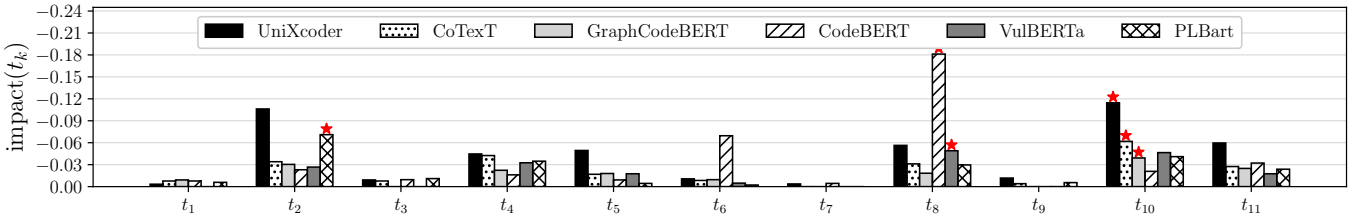


Figure 6: Impact on accuracy caused by augmenting the testing data using the individual transformations $t_k \in T$ ($impact(t_k) := accuracy[MLM[Tr], Te_k] - accuracy[MLM[Tr], Te]$). The most impactful transformations for each ML technique are marked by red stars.

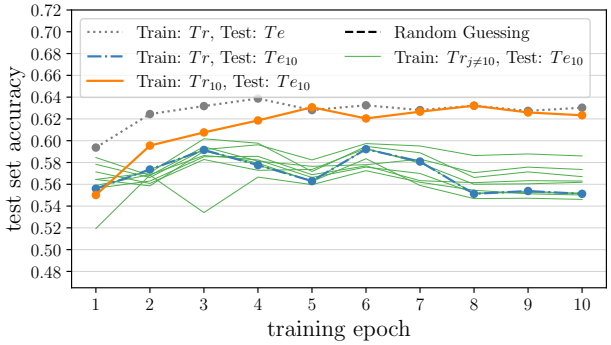
decline of the techniques when only applying semantic preserving transformations on the testing set. The most impactful transformations for each ML technique are marked by red stars.

If a specific semantic preserving transformation has a high negative impact on the accuracy of an ML technique, we can assume that either (a) the ML technique partly relied on unrelated features that were removed or modified by the transformation (e.g. removal of comments) to achieve its original high accuracy, or (b) that the ML technique relied on unrelated features introduced by the transformation (e.g. addition of comments) to achieve the decreased accuracy after applying the transformation.

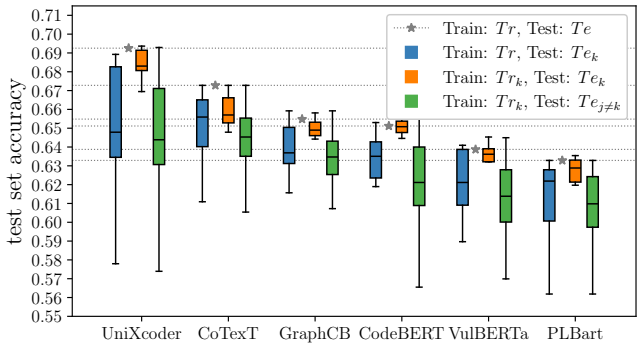
From the results presented in Figure 6 we can observe, that there are clear differences both between transformations and ML4VD techniques. As one might expect, a trivial transformation such as adding whitespace (t_7) has little to no impact on the accuracy of all six ML4VD techniques. The severity of this impact is, on average, below 1% accuracy, which means that below 1% of predictions are changed from correct to incorrect by applying this transformation. The ML4VD techniques also seem to be robust against identifier renaming (t_1 and t_3) and removal of comments (t_9), for which the severity of impact is also below or close to 1%. The most impactful transformations are changing the order of the function parameters (t_2), defining an additional void function (t_8), and adding code snippets from the training set as comments (t_{10}). For these transformations, a substantial part of the predictions is changed from correct to incorrect: Between 2.3% and 10.6% for t_2, between 1.8% and 18.1% for t_8, and between 2.1% and 11.5% for t_{10}. Overall, transformations that insert statements (e.g. t_4, t_5, t_8, and t_{10}) or reorder statements (e.g. t_2 and t_6) seem to have a higher impact than the other types.

Additionally, there are also differences between the six ML4VD techniques. For example, moving the code into a separate function (t_6) only seems to have a high impact on CodeBERT, and inserting a simple comment (t_5) seems to have a much higher impact on UniXcoder than on the other ML4VD techniques. Future work is required to determine why the ML4VD techniques are more or less robust against specific transformations.

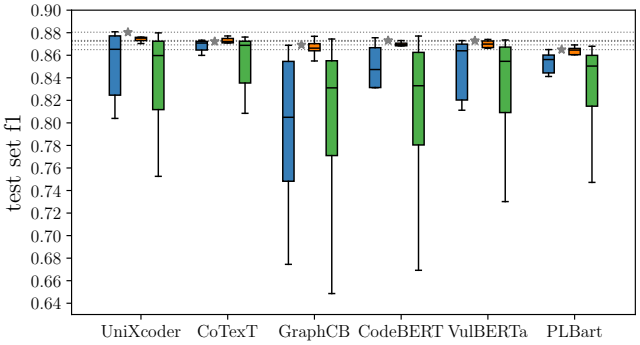
We also investigated the impact of each individual transformation when not only the testing data but also the training data is augmented using a different transformation than for



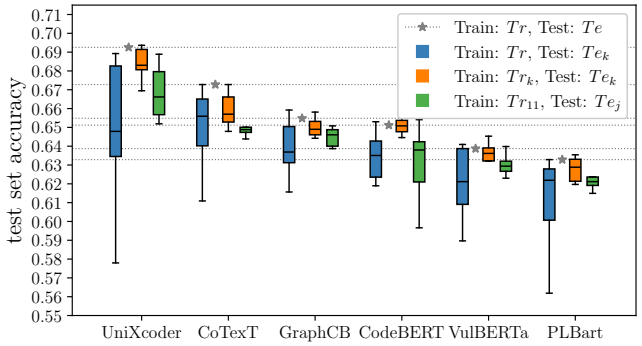
(a) Test set accuracy over ten training epochs of different models trained with VulBERTa on the CodeXGLUE/Devign dataset. Augmenting the training set Tr with different transformations $t_{j\neq 10}$ than the testing dataset (green lines) does not restore the accuracy back to previous levels.



(b) Extension of the results in Figure 7a, for all transformations $t_k \in T$ and for all six ML4VD techniques. The boxplots represent distributions of the resulting accuracies. Augmenting the training set Tr with different transformations $t_{k\neq j}$ than the testing dataset (green boxplots) does not restore the accuracy back to previous levels. Instead of restoring, the accuracy sometimes even drops further compared to using standard training data (green below blue).



(c) Same setup as for Figure 7b, but using the VulDeePecker dataset. The boxplots represent distributions of the resulting f1-scores. Augmenting the training set Tr with different transformations $t_{k\neq j}$ than the testing dataset (green boxplots) does not restore the f1-score back to previous levels.



(d) Same setup as for Figure 7b, but the green boxplots represent the accuracies achieved by augmenting the training data using the meta transformation t_{11}, in this case sampled from $\{t_1,\ldots,t_{10}\}\setminus\{t_j\}$, and the testing data using a single left-out transformation t_j. Augmenting the training dataset Tr with t_{11} partially restores the accuracy, although not to its previous levels.

Figure 7: Effects of augmenting the training data with different semantic preserving transformations than the testing data.

the testing data. However, due to the results being very similar to Figure 6, we decided to omit this from the paper and provide it as supplementary material in Appendix B.

Across two datasets, six ML4VD techniques, and 11 transformations, on average, (a) testing data augmentation using semantic preserving transformations leads to a drop in accuracy/f1-score (CodeXGLUE: −0.025, VulDeePecker: −0.043), (b) training data augmentation using the same transformations restores 69.0% (CodeXGLUE) and 66.2% (VulDeePecker) of the lost accuracy/f1-score, and (c) transformations that insert or reorder statements seem to be more impactful than other types of transformations.

RQ.1 has already been studied in the literature, for many different techniques, datasets, and tasks [5, 18, 22, 35, 38, 39, 41, 42]. Based on our evidence, we can approve the findings of the literature.

RQ.2 Overfitting to Specific Transformations

We investigate, whether the performance of ML4VD techniques can still be restored if we augment the training dataset with a different semantic preserving transformation than the testing dataset. We use Algorithm 1 to investigate RQ.2, with the same setup as for RQ.1.

Results. Figure 7a is similar to Figure 5a, it also shows the test set accuracies of different VulBERTa models measured after each of the ten training epochs. In addition to the results displayed in Figure 5a, Figure 7a shows the accuracies (green lines) of VulBERTa models trained on data that was augmented using all transformations except t_{10}, which was used to augment the testing dataset. We observe, that augmenting the training dataset Tr with different transformations $t_{j\neq 10}$ as the testing dataset does not restore the accuracy back to previous levels.

Figure 7b visualizes the same extended results for all semantic preserving transformations $t_k \in T$ and for all six

Table 2: Algorithm 1: Average changes when augmenting only the testing data ($out\,put_{A1.1}$), training and testing data using the same ($out\,put_{A1.2}$), or a different transformation ($out\,put_{A1.3}$).

Metric	Technique	CodeXGLUE Tr / Te	CodeXGLUE $out_{A1.1}$ Tr / Te_k	CodeXGLUE $out_{A1.2}$ Tr_k / Te_k	CodeXGLUE $out_{A1.3}$ Tr_k / $Te_{j\neq k}$	VulDeePecker Tr / Te	VulDeePecker $out_{A1.1}$ Tr / Te_k	VulDeePecker $out_{A1.2}$ Tr_k / Te_k	VulDeePecker $out_{A1.3}$ Tr_k / $Te_{j\neq k}$
accuracy	UniXcoder	0.693	-0.043	-0.011	-0.050	0.975	-0.005	-0.002	-0.010
	CoTexT	0.673	-0.022	-0.014	-0.030	0.973	-0.001	-0.001	-0.003
	GraphCB	0.655	-0.015	-0.006	-0.021	0.973	-0.014	-0.002	-0.015
	CodeBERT	0.651	-0.034	-0.000	-0.040	0.974	-0.007	-0.002	-0.012
	VulBERTa	0.639	-0.017	-0.004	-0.025	0.973	-0.011	-0.002	-0.012
	PLBart	0.633	-0.021	-0.007	-0.026	0.972	-0.003	-0.002	-0.008
			-0.025	**-0.007**	**-0.032**		**-0.007**	**-0.002**	**-0.010**
f1-score	UniXcoder	0.680	-0.037	-0.007	-0.041	0.880	-0.028	-0.012	-0.054
	CoTexT	0.635	0.001	0.006	0.006	0.872	-0.006	-0.006	-0.020
	GraphCB	0.629	-0.024	-0.013	-0.033	0.869	-0.093	-0.006	-0.091
	CodeBERT	0.596	-0.005	0.012	0.001	0.873	-0.045	-0.007	-0.082
	VulBERTa	0.652	-0.050	-0.014	-0.048	0.873	-0.082	-0.009	-0.073
	PLBart	0.618	-0.009	-0.006	-0.016	0.865	-0.014	-0.007	-0.035
			-0.021	**-0.004**	**-0.022**		**-0.045**	**-0.008**	**-0.059**
recall	UniXcoder	0.787	-0.025	-0.016	-0.009	0.893	-0.034	-0.011	-0.041
	CoTexT	0.851	0.066	0.074	0.105	0.900	-0.004	-0.015	-0.029
	GraphCB	0.661	-0.023	-0.029	-0.031	0.895	-0.153	-0.005	-0.101
	CodeBERT	0.572	0.079	0.031	0.106	0.890	-0.090	-0.004	-0.103
	VulBERTa	0.759	-0.097	-0.029	-0.056	0.898	-0.094	-0.000	-0.073
	PLBart	0.658	0.025	0.003	0.024	0.884	-0.008	-0.003	-0.015
			0.004	**0.006**	**0.023**		**-0.064**	**-0.006**	**-0.060**
precision	UniXcoder	0.717	-0.033	0.020	-0.033	0.939	-0.008	0.030	-0.045
	CoTexT	0.684	-0.029	-0.027	-0.044	0.955	-0.015	0.013	-0.012
	GraphCB	0.734	-0.026	-0.019	-0.035	0.995	-0.016	-0.005	-0.040
	CodeBERT	0.847	-0.115	-0.009	-0.107	1.000	-0.000	-0.007	-0.031
	VulBERTa	0.643	-0.004	-0.019	-0.034	0.850	0.023	-0.012	-0.010
	PLBart	0.675	-0.031	-0.015	-0.049	0.971	-0.048	0.008	-0.068
			-0.040	**-0.012**	**-0.051**		**-0.011**	**0.005**	**-0.034**
FPR	UniXcoder	0.142	0.033	-0.018	0.049	0.006	0.001	-0.003	0.006
	CoTexT	0.211	0.018	0.028	0.034	0.004	0.001	-0.001	0.001
	GraphCB	0.079	0.040	-0.003	0.027	0.000	0.002	0.000	0.003
	CodeBERT	0.026	0.140	0.005	0.145	0.000	0.000	0.001	0.003
	VulBERTa	0.194	-0.036	0.028	0.040	0.018	-0.004	0.002	-0.001
	PLBart	0.137	0.016	0.020	0.026	0.003	0.004	-0.001	0.007
			0.035	**0.010**	**0.053**		**0.001**	**-0.000**	**0.003**
FNR	UniXcoder	0.213	0.025	0.016	0.009	0.107	0.034	0.011	0.041
	CoTexT	0.149	-0.066	-0.074	-0.105	0.100	0.004	0.015	0.029
	GraphCB	0.339	0.023	0.029	0.031	0.105	0.153	0.005	0.101
	CodeBERT	0.428	-0.079	-0.031	-0.106	0.110	0.090	0.004	0.103
	VulBERTa	0.241	0.097	0.029	0.056	0.102	0.094	0.000	0.073
	PLBart	0.342	-0.025	-0.003	-0.024	0.116	0.008	0.003	0.015
			-0.004	**-0.006**	**-0.023**		**0.064**	**0.006**	**0.060**

ML4VD techniques. Again, the blue and the orange boxplots represent the distributions of accuracies, when either only the testing dataset (blue) or training and testing datasets were augmented using the same transformation (orange). The green boxplots represent the distribution of accuracies achieved by models that were trained on data, which was augmented using a different transformation than for the testing data. Across all six ML4VD techniques, we observe that, on average, augmenting the training dataset Tr with a different transformation $t_{k\neq j}$ than the testing dataset does not restore the accuracy back to previous levels.

Figure 7c visualizes the same results as Figure 7b, but using the VulDeePecker dataset. In this figure, the y-axis measures the f1-score, since it is the preferred evaluation metric for the VulDeePecker dataset. Again, we observe that, on average, augmenting the training dataset Tr with a different transformation $t_{k\neq j}$ as the testing dataset does not restore the accuracy back to previous levels.

In Figure 7d the green boxplots represent the distribution of accuracies achieved by augmenting the training data using our *meta transformation* t_{11}. Slightly different to the definition of t_{11} in Table 1, each function in the training set is transformed using a random transformation t_k with $k \in [1,10] \setminus j$, with one left-out transformation t_j which is applied to the testing data. Since our set of implemented transformations contains groups of similar transformations (e.g. adding different types of comments), we would expect the accuracies to be higher compared to applying only a single different transformation to the training set (green boxplots of Figure 7d), but lower compared to applying exactly the same transformation to the training set (orange boxplots). Based on Figure 7d, we observe that is the case across all six ML4VD techniques. Augmenting the training dataset Tr with the meta transformation t_{11} does not fully restore the accuracy, but moves it closer towards the accuracy on unaugmented data compared to applying only a single different transformation to the training set.

In addition to the results for RQ.1, Table 2 also shows the average recorded changes in the respective metrics, when the training and testing datasets were augmented using different transformations (green columns, $out\,put_{A1.3}$). We observe that, on average, the score drops by 0.032 accuracy (CodeXGLUE) and 0.059 f1-score (VulDeePecker). Across the six techniques, the decrease is on average 30.2% (CodeXGLUE) and 77.5% (VulDeePecker) stronger than for training on unaugmented data. In other words, augmenting the training dataset using a different transformation than for the testing dataset did not restore the score towards previous levels, but instead decreased it even further.

For the other metrics (recall, precision, FPR and FNR), we generally observe similar patterns than for accuracy and f1-score. However, there are also slight deviations, e.g. for CodeXGLUE recall improves on average by 0.023 when training data is augmented using a different transformation than the testing data instead of decreasing as expected. These deviations can be explained by innate tradeoffs between recall/-precision and FPR/FNR, and can only be intepreted in context of the other metrics. Accuracy and f1-score provide a better summary of the performance, which is why they are used as the preferred metrics for the two datasets.

Across two datasets, six ML4VD techniques, and 11 transformations, augmenting the training dataset using a different transformation than for the testing dataset does not restore the performance back to previous levels. In other words, the ML4VD techniques overfit to the label-unrelated features introduced by semantic preserving transformations during training data augmentation.

In summary, we can observe that across the tested ML4VD

Table 3: Algorithm 2: Performance of six ML4VD techniques evaluated on the standard CodeXGLUE/Devign testing dataset Te or the vulnerability-patch testing dataset $VPTe$.

Metric	Technique	$out_{A2.1}$ Tr Te	$out_{A2.2}$ Tr $VPTe$	$out_{A2.3}$ $VPTr$ Test: $VPTe$	$out_{A2.4}$ $VPTr$ Te
accuracy	UniXcoder	0.693	0.414	0.616	0.546
	CoTexT	0.673	0.503	0.607	0.575
	GraphCB	0.655	0.342	0.596	0.546
	CodeBERT	0.651	0.294	0.571	0.548
	VulBERTa	0.639	0.527	0.602	0.564
	PLBart	0.633	0.524	0.598	0.572
		0.657	**0.434**	**0.598**	**0.559**
f1-score	UniXcoder	0.680	0.582	0.662	0.613
	CoTexT	0.635	0.667	0.665	0.616
	GraphCB	0.629	0.508	0.654	0.603
	CodeBERT	0.596	0.455	0.629	0.613
	VulBERTa	0.652	0.610	0.651	0.615
	PLBart	0.618	0.583	0.633	0.575
		0.635	**0.567**	**0.649**	**0.606**
recall	UniXcoder	0.787	0.819	0.870	0.896
	CoTexT	0.851	1.000	0.975	0.941
	GraphCB	0.661	0.680	0.835	0.873
	CodeBERT	0.572	0.589	0.770	0.883
	VulBERTa	0.759	0.758	0.909	0.928
	PLBart	0.658	0.680	0.741	0.738
		0.715	**0.754**	**0.850**	**0.876**
precision	UniXcoder	0.717	0.452	0.668	0.518
	CoTexT	0.684	0.502	0.724	0.702
	GraphCB	0.734	0.406	0.622	0.509
	CodeBERT	0.847	0.371	0.656	0.516
	VulBERTa	0.643	0.531	0.781	0.647
	PLBart	0.675	0.535	0.663	0.547
		0.717	**0.466**	**0.686**	**0.573**
FPR	UniXcoder	0.142	0.816	0.107	0.172
	CoTexT	0.211	0.823	0.060	0.041
	GraphCB	0.079	0.840	0.091	0.177
	CodeBERT	0.026	0.849	0.102	0.233
	VulBERTa	0.194	0.312	0.034	0.061
	PLBart	0.137	0.251	0.138	0.213
		0.131	**0.649**	**0.089**	**0.149**
FNR	UniXcoder	0.213	0.181	0.130	0.104
	CoTexT	0.149	0.000	0.025	0.059
	GraphCB	0.339	0.320	0.165	0.127
	CodeBERT	0.428	0.411	0.230	0.117
	VulBERTa	0.241	0.242	0.091	0.072
	PLBart	0.342	0.320	0.259	0.262
		0.285	**0.246**	**0.150**	**0.124**

techniques, transformations, and datasets, training data augmentation only restores the performance to previous levels when the testing dataset is augmented in a similar way than the training dataset.

The performance gained by data augmentation only applies to the specific transformations used during the training of the model. ML4VD techniques continue to leverage unrelated features when deciding whether a function contains a security vulnerability.

RQ.3 Generalization to VulnPatchPairs

We investigate, whether (a) ML4VD techniques are able to generalize from typical vulnerability detection training datasets to a modified setting, in which they are required to distinguish between vulnerabilities and their patches, and (b) whether training to distinguish between vulnerabilities and patches improves the performance on standard testing data.

Methodology. We used Algorithm 2 to investigate both questions. As inputs to the algorithm, we selected the training and testing subsets of the CodeXGLUE/Devign dataset as the standard training and testing datasets Tr and Te, and the training and testing subsets of VulnPatchPairs as the vulnerability-patch training and testing datasets $VPTr$ and $VPTe$. We ran the algorithm for all six ML4VD techniques separately.

Results. Table 3 shows the results of running Algorithm 2. Specifically, it shows the performance of different models evaluated on the standard CodeXGLUE/Devign testing dataset Te or the vulnerability-patch testing dataset $VPTe$. We focus our analysis on the results measured in accuracy since it is the preferred performance metric for balanced datasets such as CodeXGLUE and VulnPatchPairs.

We observe, that the accuracy of all six ML4VD techniques is highest (between 0.633 and 0.693) when trained and evaluated on standard training and testing data (second column, $out\,put_{A2.1}$). This is expected and consistent with the findings in the literature [1, 12–15, 28]. When trained and evaluated on VulnPatchPairs (fourth column, $out\,put_{A2.3}$) the accuracy is consistently lower than in the standard setting (between 0.558 and 0.617), but still significantly higher than the expected accuracy of a random guesser[11]. However, when trained on standard training data and evaluated on the VulnPatchPairs testing dataset (third column, $out\,put_{A2.2}$), the accuracy drops dramatically (between 0.294 and 0.527). Even the best model (VulBERTa) is only 0.027 points better than a random guesser. On average, the accuracy is worse than random guessing. In other words, all six ML4VD techniques that we evaluated are unable to distinguish between vulnerabilities and their patches when trained on a typical vulnerability detection dataset.

When trained on VulnPatchPairs and evaluated on standard testing data (fifth column, $out\,put_{A2.4}$), we get a similar picture. The performance is significantly worse (between 0.546 and 0.575) compared to models trained on standard training data (second column). However, the performance in this case is notably better than random guessing.

(a) All six ML4VD techniques are not able to distinguish between vulnerabilities and their patches when trained on standard training data. On average, the accuracy is lower than the expected accuracy of a random guesser. (b) When trained to distinguish between vulnerabilities and their patches, the ML4VD techniques are able to predict standard testing data better than a random guesser, but still significantly worse than when trained on standard training data. In other words, the ML4VD techniques are unable to generalize from their training data to a slightly modified vulnerability detection setting.

[11] Since $VPTe$ is perfectly class balanced (50% vulnerable, 50% clean), a random guesser (coin flip) would be expected to achieve an accuracy of 0.5.

6 Threats to Validity

As for any empirical study, there are various threats to the validity of our results and conclusions.

Internal validity. A common source of systematic error in empirical studies on ML4VD techniques is hyperparameter selection. Given a particular desired outcome, hyperparameters can be optimized to move the result in the desired direction. We tried to minimize this risk by taking the values for hyperparameters provided by the authors of the chosen ML4VD techniques.

Another potential source of systematic error is the training-/testing dataset split. Similar to hyperparameter selection, dataset split can also be varied to change a result in a desired direction. We tried to avoid this risk by taking the provided splits of the CodeXGLUE benchmark [24] and by Hanif et al. [26].

External validity. The degree to which our results generalize to other learning-based techniques, datasets, semantic preserving transformations, and performance metrics, are concerns of external validity. We tried to minimize the risk attached to these concerns by evaluating a wide set of six state-of-the-art techniques, two datasets, six performance metrics, and 11 semantic preserving transformations. For RQ.3, we only investigate the generalization between CodeXGLUE/Devign and VulnPatchPairs. To maximize generality, we tried to keep both Algorithm 1 and Algorithm 2 as general as possible, so that they can easily be adapted to other techniques, datasets, transformations, and metrics.

Simplicity of Transformations. Some of the semantic preserving transformations that we used (see Table 1) could be easily addressed by adding additional data pre-processing (e.g. mapping identifiers to standardized names). However, the specific transformations that we implemented are merely a tool to demonstrate, that the performance gained by training data augmentation only applies to the specific transformations used for training and that the techniques that we investigated overfit to the label-unrelated features introduced by these transformations. For a new technique, they could be replaced by a different set of transformations.

Class balance. Multiple works have shown that learning-based vulnerability detection techniques trained on fairly balanced datasets (such as CodeXGLUE) often fail to generalize to real-world code repositories [3,9,16], which usually contain a much smaller ratio of security vulnerabilities [20]. However, measured by citations, class-balanced datasets are still by far the most popular datasets to evaluate learning-based techniques for vulnerability detection. To our current knowledge, there is no vulnerability detection dataset with sufficient size (more than 10k code snippets), high-quality labels (manually provided by security experts), and a realistic distribution of vulnerable to non-vulnerable code snippets that is widely used in the research community (at least 50 citations). This is why we decided to focus our experiments on the CodeXGLUE and VulDeePecker datasets, even though they do not reflect a realistic class distribution.

7 Discussion and Future Work

Overfitting of learned models is a well-known problem in the machine learning research field [11, 40]. However, as shown in our experiments, the traditional approach to evaluating ML4VD techniques often fails to detect overfitting to label-unrelated features in the training data. Our proposed Algorithm 1 is a novel way to measure overfitting of ML4VD techniques, that goes beyond the traditional approach, and can even detect overfitting if there is no gap in the standard setup at all. There are several common strategies to reduce overfitting in the standard evaluation setup, e.g. early-stopping, dropout, or large pre-training datasets [40], which are already integrated in the ML4VD techniques that we used in our experiments. However, our experiments demonstrate that the techniques are still severely overfitting to label-unrelated features introduced by semantic preserving transformations during training data augmentation. Finding ways to robustify ML4VD techniques without or with minimal overfitting will be a central challenge of the ML4VD research area.

Generalization. The results for RQ.3 (see Section 5) reveal, that state-of-the-art ML4VD techniques lack the ability to generalize from their training data to a modified setting, which requires to distinguish between vulnerabilities and their patches. Since we can not assume that real-world software systems would be similar to the training data of these techniques, the ability to generalize to modified settings would be required for these techniques to be safely integrated into real software engineering environments.

The ability of a ML technique to generalize to testing data that is differently distributed than the training data is also called *out-of-distribution generalization*, and the lack of it for learning-based techniques has been recently identified (e.g. in the computer vision domain [17, 34]). Our proposed Algorithm 2 can be seen as a tool to measure out-of-distribution generalization for the domain of automatic vulnerability detection. It would be interesting to try approaches that have been used to address out-of-distribution generalization in other domains (e.g. causal representation learning [32]) on the task of automatic vulnerability detection and measure the success using our Algorithm 2.

Acknowledgements

Special thanks to Lukas Pirch, Konrad Rieck, Kevin Borgolte, and Seongmin Lee for their constructive feedback on earlier versions of this paper.

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics.

[2] Leonhard Applis, Annibale Panichella, and Arie van Deursen. Assessing robustness of ml-based program analysis tools using metamorphic program transformations. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1377–1381, 2021.

[3] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3971–3988, 2022.

[4] Guru Bhandari, Amara Naseer, and Leon 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, page 30–39, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Pavol Bielik and Martin Vechev. Adversarial robustness for code. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[6] Paul Black. A software assurance reference dataset: Thousands of programs with known bugs, 2018-04-16 2018.

[7] Harold Booth, Doug Rike, and Gregory Witte. The national vulnerability database (nvd): Overview, 2013-12-18 2013.

[8] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 464–468, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, PP:1–1, 06 2021.

[10] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '23, page 654–668, New York, NY, USA, 2023. Association for Computing Machinery.

[11] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM Comput. Surv.*, 27(3):326–327, sep 1995.

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[13] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[15] Hazim Hanif and Sergio Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2022.

[16] Jingxuan He, Luca Beurer-Kellner, and Martin Vechev. On distribution shift in learning-based bug detectors. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 8559–8580. PMLR, 17–23 Jul 2022.

[17] Dan Hendrycks, Steven Basart, Norman Mu, Saurav Kadavath, Frank Wang, Evan Dorundo, Rahul Desai, Tyler Zhu, Samyak Parajuli, Mike Guo, Dawn Song, Jacob Steinhardt, and Justin Gilmer. The many faces of robustness: A critical analysis of out-of-distribution generalization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 8340–8349, October 2021.

[18] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. Semantic robustness of models of source code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, mar 2022.

[19] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847, 2012.

[20] Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. In *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*, pages 573–577, 2020.

[21] Yaoxian Li, Shiyi Qi, Cuiyun Gao, Yun Peng, David Lo, Zenglin Xu, and Michael R. Lyu. A closer look into transformer-based code intelligence through code transformation: Challenges and opportunities, 2022.

[22] Zhen Li, Jing Tang, Deqing Zou, Qian Chen, Shouhuai Xu, Chao Zhang, Yichen Li, and Hai Jin. Towards making deep learning-based vulnerability detectors robust, 2021.

[23] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, 2018.

[24] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. Codexglue: A machine learning benchmark dataset for code understanding and generation. In J. Vanschoren and S. Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021.

[25] Codexglue leaderboards, 2021. `https://microsoft.github.io/CodeXGLUE/#LB-DefectDetection`.

[26] Vuldeepecker function-level dataset, 2022. `https://github.com/ICL-ml4csec/VulBERTa/tree/main/data`.

[27] Pedro Orvalho, Mikoláš Janota, and Vasco Manquinho. Multipas: applying program transformations to introductory programming assignments for data augmentation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1657–1661, 2022.

[28] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Annibal, Alec Peltekian, and Yanfang Ye. CoTexT: Multi-task learning with code-text transformer. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pages 40–47, Online, August 2021. Association for Computational Linguistics.

[29] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, page 383–387. IEEE Press, 2019.

[30] Md Rafiqul Islam Rabin, Nghi D.Q. Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135:106552, jul 2021.

[31] Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. Towards causal deep learning for vulnerability detection. In *Proceedings of the 46th International Conference on Software Engineering*, ICSE '24. IEEE Press, 2024.

[32] B. Schölkopf, F. Locatello, S. Bauer, N. R. Ke, N. Kalchbrenner, A. Goyal, and Y. Bengio. Toward causal representation learning. *Proceedings of the IEEE*, 109(5):612–634, 2021.

[33] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Katrin Erk and Noah A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics.

[34] Zheyan Shen, Jiashuo Liu, Yue He, Xingxuan Zhang, Renzhe Xu, Han Yu, and Peng Cui. Towards out-of-distribution generalization: A survey, 2021.

[35] Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. Generating adversarial computer programs using optimized obfuscations. In *International Conference on Learning Representations*, 2021.

[36] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. An empirical study of deep learning models for vulnerability detection. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 2237–2248. IEEE Press, 2023.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser,

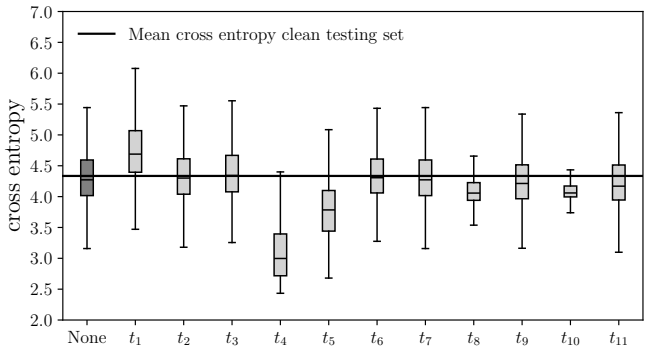


Figure 8: Naturalness of the semantic preserving transformations, that we used in our experiments. Lower cross entropy means higher naturalness. All transformations except t_1 (identifier renaming) lead to lower or equal cross entropy than no transformation (None).

and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[38] Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1482–1493, New York, NY, USA, 2022. Association for Computing Machinery.

[39] Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[40] Xue Ying. An overview of overfitting and its solutions. *Journal of Physics: Conference Series*, 1168(2):022022, feb 2019.

[41] Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. Towards robustness of deep program processing models—detection, estimation, and enhancement. *ACM Trans. Softw. Eng. Methodol.*, 31(3), apr 2022.

[42] Huangzhao Zhang, Zhuo Li, Ge Li, L. Ma, Yang Liu, and Zhi Jin. Generating adversarial examples for holding robustness of source code processing models. In *AAAI Conference on Artificial Intelligence*, 2020.

[43] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. Curran Associates Inc., Red Hook, NY, USA, 2019.

A Naturalness

Since vulnerability detection techniques are ultimately designed to be applied to real-world code, we also need to ensure that our transformations lead to code snippets that could occur in the real world, or i.e., lead to *natural* code. We measured the naturalness of our transformations using the method introduced by Hindle et al. [19] (implemented as a 2-gram markov model) and present the results in Figure 8. Using the method introduced by Hindle et al., we can compute the *cross entropy* of a given code snippet, which represents how surprising or *unnatural* the code snippet is relative to the code snippets observed in the training dataset (for a detailed explanation consult the work of Hindle et al. [19]). Using this approach, we computed the cross entropy for all code snippets in the CodeXGLUE testing dataset (dark gray boxplot), and for transformed versions of the CodeXGLUE testing dataset (light gray boxplots). The horizontal black line represents the average cross entropy for all code snippets in the untransformed CodeXGLUE testing dataset. We can observe, that for all transformations except t_1 (identifier renaming), the cross entropy is similar or lower than for the untransformed dataset. In other words, all transformations except t_1 (identifier renaming) lead to code snippets that are similar in naturalness compared to the real-world code of the CodeXGLUE testing dataset.

B Impact of Individual Transformations

Figure 9 shows the impact on accuracy caused by augmenting the testing data with a different transformation than the training data ($impact(t_k) := \frac{1}{(N-1)} \sum_{t_j \in T} accuracy[MLM[Tr_j], Te] - accuracy[LLM[Tr_j], Te_k]$). The most impactful transformations for each LLM are marked by red stars. The results are very similar to Figure 6, which is why we chose to omit Figure 9 from the main paper and provide it as supplementary material.

C Model Architecture Details

All selected ML4VD techniques happen to be token-based large language models (LLMs), specialized for the task of vulnerability detection. LLMs are based on the transformer architecture, which is a neural network model architecture for sequence-to-sequence tasks based on the attention mechanism [37]. The attention mechanism is essentially a weighted dot product that allows models to focus on specific parts of the input data that are most relevant to the task at hand, improving their ability to capture dependencies and context. In a transformer model, the attention mechanism is combined with parametrized feed-forward layers and repeated multiple times, resulting in a complex multi-layer network. A detailed

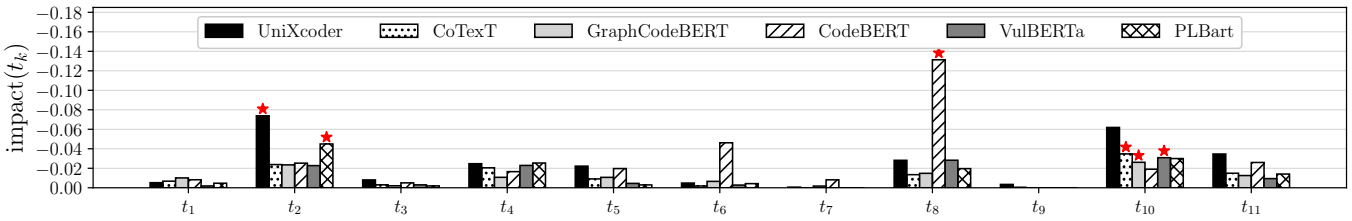


Figure 9: Impact on accuracy caused by augmenting the testing data with a different transformation than the training data ($impact(t_k) := \frac{1}{(N-1)} \sum_{t_j \in T} accuracy[MLM[Tr_j], Te] - accuracy[MLM[Tr_j], Te_k]$). The most impactful transformations for each ML technique are marked by red stars.

(a) Accuracy (b) F1-Score (c) Precision (d) Recall (e) FPR (f) FNR

Figure 10: Additional metrics for RQ.2 using the CodeXGLUE/Devign dataset. The results support the conclusions that we generated based on the main metric (accuracy).

(a) Accuracy (b) F1-Score (c) Precision (d) Recall (e) FPR (f) FNR

Figure 11: Additional metrics for RQ.2 using the VulDeePecker dataset. The results support the conclusions that we generated based on the main metric (f1-score).

description of the transformer architecture can be found in the original paper [37]. While the base architecture is the same for all of the six techniques, there are some notable differences in the pre-training setup, size, or specific parts of the model training:

VulBERTa. VulBERTa leverages a custom tokenization strategy, which is based on the *byte pair encoding* algorithm [33] combined with a set of pre-defined code tokens (standard C/C++ keywords, punctuation, and library API calls) to achieve better code encodings through maintaining the syntactical structure of source code.

CoTexT. CoTexT uses multi-task learning for pre-training, which means that the model is trained to perform multiple code-related tasks (e.g. vulnerability detection, code summarization, and code generation) in parallel. The model architecture also contains significantly more trainable parameters than the other techniques (222M parameters), because it contains more more multi-head attention layers. CoTexT also uses a different tokenizer than the other techniques.

UniXcoder. UniXcoder introduces a new pre-training setup, which includes tokenization of the abstract syntax trees (ASTs) of the code, and three different 'training modes' that leverage different self-attention masks.

PLBart. PLBart uses a special pre-training procedure called 'denoising autoencoding', which is a combination of token masking, token deletion, and token infilling, that has to be reversed by the model.

CodeBERT. CodeBERT uses a pre-training setup in which natural language (e.g. documentation) and code are combined to produce a more semantically stable representation of the input.

GraphCodeBERT. GraphCodeBERT uses a pre-training task, where in addition to natural language and the code, a graph-based representation of the data flow of the function is provided.

D Additional Metrics for RQ.2

Figure 10 and Figure 11 show additional results for RQ.2 using all of our available metrics (accuracy, f1-score, precision, recall, FPR, and FNR). Generally, the results support the conclusions that we generated based on the respective preferred metrics on the two datasets (accuracy for CodeXGLUE/Devign and f1-score for VulDeePecker). While the observed patterns deviate for some of the metrics (e.g. precision, recall, FPR, or FNR), these deviations can be explained by the relationships between them. For example, a model can have a really high precision, but low recall. Similarly, a model can have really low false-negative-rate, but the corresponding false-positive-rate is really high. Accuracy and f1-score provide a better summary of the performance, which is why they are used as the preferred metrics for the two datasets.