Limits of Machine Learning for Automatic Vulnerability Detection

Niklas Risse
Max-Planck-Institute for Security and Privacy
Bochum, Germany

Marcel Böhme
Max-Planck-Institute for Security and Privacy
Bochum, Germany

arXiv:2306.17193v1 [cs.CR] 28 Jun 2023

ABSTRACT

Recent results of machine learning for automatic vulnerability detection have been very promising indeed: Given only the source code of a function f, models trained by machine learning techniques can decide if f contains a security flaw with up to 70% accuracy.

But how do we know that these results are general and not specific to the datasets? To study this question, researchers proposed to *amplify* the testing set by injecting semantic preserving changes and found that the model's accuracy significantly drops. In other words, the model uses *some* unrelated features during classification. In order to increase the robustness of the model, researchers proposed to train on amplified training data, and indeed model accuracy increased to previous levels.

In this paper, we replicate and continue this investigation, and provide an actionable model benchmarking methodology to help researchers better evaluate advances in machine learning for vulnerability detection. Specifically, we propose (i) a cross validation algorithm, where a semantic preserving transformation is applied during the amplification of either the training set or the testing set, and (ii) the amplification of the testing set with code snippets where the vulnerabilities are fixed. Using 11 transformations, 3 ML techniques, and 2 datasets, we find that the improved robustness only applies to the specific transformations used during training data amplification. In other words, the robustified models still rely on unrelated features for predicting the vulnerabilities in the testing data. Additionally, we find that the trained models are unable to generalize to the modified setting which requires to distinguish vulnerable functions from their patches.

1 INTRODUCTION

Recently a number of different publications have reported high scores on vulnerability detection benchmarks using machine learning (ML) techniques [1, 9, 11, 13, 14, 30]. The resulting models seem to outperform methods from other research directions (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?

In order to be safely applied to real-world software systems, ML techniques for vulnerability detection need to be reliable and trustworthy. This is why researchers have tried to explore the capabilities and limits of machine learning techniques in ways that go beyond simple evaluations on benchmark testing sets.

One approach to finding these limits is to create situations in which ML techniques fail to meet expectations. For example, it is possible to apply small semantic preserving changes to the input programs of a state-of-the-art model and then measure, whether the model changes its predictions and whether it still performs well. Examples for such transformations are identifier renaming [17, 38, 39, 41, 42], insertion of unexecuted statements [17, 36, 39, 41] or replacement of code elements with equivalent elements [3, 20]. The impact of applying these transformations to testing data has been explored for many different tasks in software engineering, and the results seems to be clear: ML techniques lack robustness against semantic preserving transformations [3, 5, 17, 21, 33, 36, 38, 39, 41, 42].

A common strategy to address the robustness problem is training data amplification; applying the same or similar transformations to the training dataset. Many of the works that reported the lack of robustness of ML models when trained on untransformed data also investigated training data amplification using their respective methods [5, 17, 21, 36, 38, 39, 41, 42]. They found a restoration or at least improvement towards the initial high performance. But does training data amplification actually improve the ability of these models to detect vulnerabilities, or are they just over-fitting to a different set of data?

We contribute to answering this question by proposing a general benchmarking methodology that can be used to evaluate the capabilities of machine learning models for vulnerability detection by using data amplification. 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 measure of over-fitting to the specific semantic preserving transformations that were used during training data amplification. Second, we propose Algorithm 2, in which a target model that was trained on an amplified training dataset is evaluated on a testing dataset, that contains both vulnerable programs and their respective fixes. The results provide a measure of the model's ability to generalize to a slightly modified vulnerability detection setting and the effects of training data amplification on this ability.

In addition to the general methodology, we present the results of an empirical study, in which we implement and validate the proposed algorithms on three state-of-the-art ML techniques for vulnerability detection. For Algorithm 1, we implemented 11 different semantic preserving transformations (see Table 1) for data amplification and evaluated the trained models using two different vulnerability detection datasets. As expected, we find a strong benefit of training data amplification (59.8% and 54.1% average restoration of performance for the two datasets) when the transformations applied to training and testing dataset are the same. However, we find no improvement in performance when the transformations applied to training and testing dataset are different. In fact, we even find an additional 35.7% and 97.4% average decrease

https://t.me/learningnets

in performance for the two datasets. In other words, state-of-the-art ML techniques severely over-fit to the specific label-unrelated features introduced by training data amplification. The improved robustness only applies to the specific type of transformations used during training.

For Algorithm 2, we introduce a new dataset, which we call *VulnPatchPairs*. VulnPatchPairs contains 26.2k C functions and is derived from the CodeXGLUE/Devign vulnerability detection dataset [43]. Exactly half (13.1k) of the 26.2k functions in VulnPatchPairs contain vulnerabilities. The other 13.1k functions are patched versions of the first 13.1k functions (see Section 4.3 for details). Again, we use the 11 transformations that we implemented and three ML techniques to train $11 \cdot 3 = 33$ models, and evaluate each of them on the testing subset of VulnPatchPairs. To our surprise, no trained model is able to distinguish between the vulnerable functions and their fixed counterparts. On average, the performance is only 1.8% better than random guessing. The trained models are unable to generalize from a standard vulnerability detection dataset to the slightly modified setting.

In summary, this paper makes the following contributions:

- ⋆ We present a general methodology that can be used to evaluate ML models for vulnerability detection using data amplification.
- ⋆ We show empirically, that the robustness gained by data amplification only applies to the specific transformations used during training, and that robustified models over-fit to the unrelated features introduced by semantic preserving transformations.
- ⋆ We introduce *VulnPatchPairs*, a new dataset that contains vulnerable C function and the corresponding patched versions of the same functions.
- ⋆ We demonstrate, that three state-of-the-art ML techniques for vulnerability detection are not able to distinguish between the vulnerable and patched functions in VulnPatchPairs and that training data amplification leads to no significant improvement.
- ⋆ We publish all of our datasets, code, and results for reproducibility. They are available at `https://github.com/niklasrisse/LimitsOfML4Vuln`.

2 RELATED WORK

One of the main tools to study the robustness of deep learning based vulnerability detection systems are semantic preserving transformations of code. Previous work [3, 5, 17, 20–22, 33, 36, 38, 39, 41, 42] proposed methods to generate semantic preserving transformations for source code datasets. The effects of training data amplification using semantic preserving transformations of code have been studied for many software engineering tasks, such as code summarization [17, 36, 39], type prediction [5], code functionality classification [42] and vulnerability detection [21, 22, 38, 41]. A common finding in all of these publications is that training data amplification using a specific type of semantic preserving transformation leads to improved performance on testing sets that have been amplified the same way.

Some of the works [5, 17, 36, 38] go a step further; they evaluate the effects of amplifying the training dataset using their proposed method on a testing dataset that has been amplified using a slightly different but related type of transformation. For example, Henkel et al. [17] apply their gradient-based resolution of transformation sketches to training data, e.g. they select a local variable for renaming and determine the name in such a way that the loss of the target model is maximized, and evaluate their method on a testing dataset where variables are randomly renamed. Similarly, Yang et al. [38] apply their method for variable renaming to a training dataset and evaluate using the method proposed by Zhang et al. [42], which also renames variables but determines the new names in a different way. All of these works find an improved robustness when the training dataset is amplified in a similar way than the testing dataset.

A recent publication [41] investigates the benefits of data amplification when the transformations applied to training and testing dataset are of completely different type. The authors propose two methods, one for identifier renaming and one for statement insertion, and explore robustness when using one for training and the other for testing data amplification. They find lower performance compared to when training and testing dataset were amplified using the same type of transformation. However, two questions remain unanswered in this work: Is this preliminary result specific to the two transformations proposed by the authors or does it hold up in general, i.e. across different transformations, models and datasets? What are the effects of training data amplification on the general capabilities of the model to detect vulnerabilities? Our work aims to answer both open questions by proposing a methodology that generalizes the initial result proposed by Zhang et al. [41], and by carrying out a thorough empirical study that considers a diverse set of 11 different transformations, three state-of-the-art ML techniques and two high-quality datasets.

Some of the semantic preserving transformations proposed in the literature [17, 36, 39] are computed during the training of the respective models with the goal to optimize an additional training objective, which is designed to produce models that are robust against small transformations of their input (adversarial training). The difference of adversarial training to standard training data amplification is, that in adversarial training the transformations are computed during training (instead of before the training) and in such a way, that the loss of the model for each training sample is maximized. We use both simple semantic preserving transformations and adversarial training in our experiments.

In order to evaluate the general capabilities of ML techniques for vulnerability detection and the benefits of data amplification on them, 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, 31], e.g. for the research field of automated fixing. However, to the best of our knowledge, we are the first use such a dataset to evaluate the general capabilities of ML techniques for vulnerability detection and the effects of data amplification using semantic preserving transformations.

3 METHODOLOGY

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

https://t.me/learningnets

Algorithm 1 Detecting Over-fitting to Code Changes

Input: Semantic Preserving Transformations $T := \{t_1, ..., t_N\}$
Training Dataset Tr
Testing Dataset Te
Machine Learning Technique MLT
Performance Metric M

1: $f[Tr] = train\_model(Tr, MLT)$
2: $s[f[Tr], Te] = evaluate\_model(f[Tr], Te, M)$
3: **for each** $t_k \in T$ **do**
4: $Te_k = t_k(Te)$ *// testing data amplification*
5: $s[f[Tr], Te_k] = evaluate\_model(f[Tr], Te_k, M)$
6: $e[Tr, Te_k] = s[f[Tr], Te_k] - s[f[Tr], Te]$
7: $Tr_k = t_k(Tr)$ *// training data amplification*
8: $f[Tr_k] = train\_model(Tr_k, MLT)$
9: $s[f[Tr_k], Te_k] = evaluate\_model(f[Tr_k], Te_k, M)$
10: $e[Tr_k, Te_k] = s[f[Tr_k], Te_k] - s[f[Tr], Te]$
11: **for each** $t_{j \neq k} \in T$ **do**
12: $Te_j = t_j(Te)$ *// testing data amplification*
13: $s[f[Tr_k], Te_j] = evaluate\_model(f[Tr_k], Te_j, M)$
14: $e[Tr_k, Te_j] = s[f[Tr_k], Te_j] - s[f[Tr], Te]$
15: **end for**
16: **end for**

Output: $o_1 = (\sum_k e[Tr, Te_k])/N$
$o_2 = (\sum_k e[Tr_k, Te_k])/N$
$o_3 = (\sum_k \sum_{j \neq k} e[Tr_k, Te_j])/(N(N-1))$

```c
#include <stdio.h>
int main() {
   printf("Hello, World!");

   return 0;
}
```

(a) Code Snippet

```c
#include <stdio.h>
int main() {
   printf("Hello, World!");
   char x;
   return 0;
}
```

(b) Transformed Code Snippet

Figure 1: Example of a semantic preserving transformation. The change (orange) has no effect on the vulnerability label (no vulnerability).

3.1 Label-aware Transformations of Code

A central component of both algorithms are label-aware transformations of code, and the expectations for vulnerability detection models that emerge from using them for data amplification.

A code transformation $t : C \longrightarrow C$ is a function that maps from and to a space C, which represents all possible code snippets $c \in C$ in a given programming language. Lets assume we have an oracle function $g : C \longrightarrow \{0, 1\}$, which maps from the space of code snippets 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 $CD \subset C$, 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 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 a toy example of a semantic preserving transformation applied to a simple 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 machine learning techniques for vulnerability detection using these two expectations.

3.2 A1: Detecting Over-fitting to Code Changes

The goal of Algorithm 1 is to measure, whether the robustness gained by data amplification generalizes beyond the specific amplifications applied during training, and whether machine learning techniques for vulnerability detection still over-fit to amplifications of their training data that are unrelated to the respective vulnerability labels.

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 machine learning technique for vulnerability detection MLT, and a performance metric M. The training and testing datasets Tr and Te consist of code-label pairs (c_i, v_i), with $c_i \in C$ 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 machine learning technique MLT can utilize the training dataset Tr to train a model $f : C \longrightarrow \{0, 1\}$, which maps from the space of code snippets C to either 1 (vulnerability) or 0 (no vulnerability). The performance metric M quantifies and aggregates the performance of a trained machine learning model f on a testing dataset Te into a single number between 0 (bad) and 1 (perfect).

What is computed? Algorithm 1 computes the average effects of (a) amplifying the testing data of the selected ML technique using transformations $t_k \in T$ (Output o_1), (b) using the same transformations to amplify the training data (Output o_2), and (c) using different transformations to amplify the training data (Output o_3).

In lines 4-6, Algorithm 1 computes the effect of amplifying the testing dataset Te with the transformation t_k on the performance of the model $f[Tr]$. The result is $e[Tr, Te_k]$, the absolute difference between the scores of $f[Tr]$ on the clean testing dataset Te and the amplified testing dataset Te_k. In other words, $e[Tr, Te_k]$ quantifies how many points in score are lost if we amplify the testing dataset with transformation t_k. The output o_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 amplifying the training dataset Tr and the testing dataset Te using the same transformation t_k. The result is $e[Tr_k, Te_k]$, the absolute difference between scores of $f[Tr_k]$ on

https://t.me/learningnets

Algorithm 2 Distinguish between Vulnerability and Patch

Input: Semantic Preserving Transformations $T := \{t_1, ..., t_N\}$
Training Dataset Tr
Testing Dataset Te
Vulnerability-Patch Testing Dataset VPT
Machine Learning Technique MLT
Performance Metric M

1: $f[Tr] = train\_model(Tr, MLT)$
2: $s[f[Tr], VPT] = evaluate\_model(f[Tr], VPT, M)$
3: **for each** $t_k \in T$ **do**
4: $\quad Tr_k = t_k(Tr)$ // *training data amplification*
5: $\quad f[Tr_k] = train\_model(Tr_k, MLT)$
6: $\quad s[f[Tr_k], VPT] = evaluate\_model(f[Tr_k], VPT, M)$
7: $\quad e[Tr_k, VPT] = s[f[Tr_k], VPT] - s[f[Tr], VPT]$
8: **end for**

Output: $o_1 = s[f[Tr], VPT]$
$o_2 = (\sum_k s[f[Tr_k], VPT])/N$
$o_3 = (\sum_k e[Tr_k, VPT])/N$

the amplified testing Te_k and $f[Tr]$ on the testing dataset Te. In other words, $e[Tr_k, Te_k]$ quantifies how many points in score are lost if we amplify both the training and the testing dataset with transformation t_k. The output o_2 aggregates this intermediate result over all transformations $t_k \in T$.

In lines 12-14, the algorithm computes the effect of amplifying the testing dataset using a *different* transformation t_j than for the training dataset. The result is $e[Tr_k, Te_j]$, the absolute difference between scores of $f[Tr]$ on the testing dataset Te and $f[Tr_k]$ on the amplified testing Te_j. In other words, $e[Tr_k, Te_j]$ quantifies how many points in score are lost if we amplify the training and the testing dataset with different transformations t_k and t_j. The output o_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 machine learning techniques for vulnerability detection. Specifically, for a selected technique researchers can answer the following questions:

(1) How much does the performance of my technique decrease if we amplify the code snippets for testing without affecting the vulnerability labels? Answer: On average, the performance does change by o_1 points.
(2) How much performance of my technique can be restored, if we amplify the training code snippets in a similar way than the testing code snippets? Answer: On average, $o_2 - o_1$ of the initial decrease can be restored.
(3) How much performance of my technique can be restored, if we amplify the training code snippets in a different way than the testing code snippets? Answer: On average, $o_3 - o_1$ of the initial decrease can be restored.
(4) Does my technique over-fit to specific amplifications of the training data that are unrelated to the respective vulnerability labels? Answer: If $o_2 >> o_3$: Yes, otherwise No.

3.3 A2: Distinguish between Vulnerability and Patch

The goal of Algorithm 2 is to evaluate, whether machine learning techniques for vulnerability detection are able to generalize from their typical training data to a modified setting, which requires to distinguish security vulnerabilities from their patches.

What are the inputs? In addition to the inputs of Algorithm 1, Algorithm 2 requires a special Vulnerability-Patch testing dataset VPT. VPT also consists of code snippets $c_i \in C$ and vulnerability labels $v_i \in \{0, 1\}$, but for every code snippet c_j with label $v_j = 0$, it also contains 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 $t_{PATCH} : C \rightarrow C$, which maps code snippets c_j to their patched versions c_l.

What is computed? Algorithm 2 uses the Vulnerability-Patch testing dataset VPT to quantify the ability of the selected ML technique to generalize to the modified setting of distinguishing between vulnerabilities and their patches. Specifically, the algorithm computes this for (a) a model, that was trained on the standard training dataset Tr (Output o_1), and (b) for models that were trained on amplified data. For the models trained on amplified data, it computes both the average scores (Output o_2), and the average improvement compared to the model that was trained on unamplified data (Output o_3).

In the first two lines of the algorithm, a machine learning model $f[Tr]$ is trained on the training dataset Tr (line 1) and evaluated on the Vulnerability-Patch testing dataset VPT (line 2), resulting in the score $s[f[Tr], VPT]$. The score $s[f[Tr], VPT]$ quantifies the ability of $f[Tr]$ to distinguish between the vulnerable code snippets and their patches contained in VPT.

After that, the algorithm computes the effects of amplifying the training dataset Tr with the different transformations t_k on the performance of the model on the Vulnerability-Patch testing dataset VPT. The result is $e[Tr_k, VPT]$), the absolute difference in performance of $f[Tr_k]$ and $f[Tr]$ on the testing dataset VPT. In other words, $e[Tr_k, VPT]$ quantifies how many points in score are gained, if we amplify the training dataset with transformation t_k. The output o_3 aggregates this intermediate result over all transformations $t_k \in T$.

How can the results be used? Using Algorithm 2, researchers can effectively evaluate whether the high scores of ML techniques for vulnerability detection are specific to the testing datasets on which they were computed, or if they generalize to a slightly modified vulnerability detection setting. Specifically, for a selected technique researchers can answer the following questions:

(1) Does the performance of my model generalize to a modified setting, which requires to distinguish vulnerabilities from their patches? Answer: The model can distinguish between vulnerabilities and their patches with performance o_1.
(2) Does training data amplification using semantic preserving transformation help my model to generalize to a modified vulnerability detection setting? Answer: On average, o_3 score points are gained by training data amplification.

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 machine learning techniques for vulnerability detection. Specifically, we aim to answer the following research questions.

https://t.me/learningnets

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}	All	Random selection of t_1 to t_{10}.
ADV	Statement Insertion	Add adversarially optimized variable declaration (Adversarial training) [39].

RQ.1 How is the performance of ML techniques affected, if we amplify the input code snippets without affecting the vulnerability labels? (a) Can we measure a decrease in performance, if we amplify the testing data of ML techniques using semantic preserving transformations? (b) Does training data amplification using the same transformations restore the initial performance?

RQ.2 Do ML techniques over-fit to specific amplifications of their training data that do not affect the respective vulnerability labels? (a) Can we still restore the performance, if we amplify the training dataset with a different semantic preserving transformation than the testing dataset? (b) Can we restore the performance, if we replace the simple semantic preserving transformations with adversarial training?

RQ.3 Are the high scores of ML techniques specific to benchmark datasets or do they generalize to a modified vulnerability detection setting? (a) Are state-of-the-art machine learning techniques able to distinguish between vulnerable functions and their patches? (b) Does training data amplification using semantic preserving transformations have a positive effect on this ability?

4.2 Semantic Preserving Transformations

One of the central components of algorithms 1 and 2 are semantic preserving transformations.

The most common semantic preserving transformations that are used in the literature to investigate the robustness of machine learning techniques for source code related tasks are identifier renaming [17, 36, 38, 39, 41, 42], insertion of unexecuted statements [17, 36, 39, 41], replacement of statements with equivalent statements [20], reordering of unrelated statements [28], deletion of unexecuted statements (e.g. comments) [20], or combinations of the before mentioned [17, 36, 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. To investigate the effects of adversarial training for RQ.2 (see Section 5), we re-implemented the adversarial training method proposed by Yefet et al. [39] as one of our semantic preserving transformations (see *ADV* in Table 1).

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 the vulnerabilities were not formally classified, but based of 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 machine learning 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, that 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 machine learning techniques to distinguish between vulnerabilities and their patches (RQ3), we collected a new dataset, which we call *VulnPatchPairs*.

VulnPatchPairs is an extension of the CodeXGLUE/Devign vulnerability detection dataset [43], which consists of C functions from two popular open source repositories, FFmpeg [12] and Qemu [32]. 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 verified

https://t.me/learningnets

VulnPatchPairs by randomly selecting approx. 100 vulnerability-patch pairs from the dataset and checking, whether they actually contain vulnerabilities and their patches.

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 publish the complete dataset as supplementary material in a publicly available GitHub repository (see Section 4.7).

4.5 Machine Learning Techniques

We selected three state-of-the-art machine learning techniques for our experiments.

Selection Criteria. In order to select techniques which represent the state-of-the-art of machine learning for vulnerability detection, we chose the Top-3 techniques from the CodeXGLUE leaderboard [25] for which the authors provide open-source implementations. Measured by citations[1] (289) and GitHub Stars[2] (972), 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 (253 citations[3]) and widely used for evaluating machine learning techniques for automatic vulnerability detection.

Selected Techniques. Based on the described criteria, we selected CoTexT [30], VulBERTa [14] and PLBart [2] for our experiments. At submission time of this paper, the three techniques hold rank 1 (CoTexT), rank 5 (VulBERTa), and rank 9 (PLBart) on the CodeXGLUE leaderboard for vulnerability detection [25]. The authors of all three techniques provide publicly available implementations of their techniques [26, 29, 37].

4.6 Model Training Pipeline

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

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 the Hanif et al. [14], 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 where taken as training, validation, and testing sets of VulnPatchPairs, augmented by their corresponding patches.

Pre-processing. We employed several pre-processing steps for the models trained with CoTexT. Since some special characters (e.g. "#") are out-of-vocabulary for the tokenizer of CoTexT, the authors replaced them with natural language tokens (e.g. "SHARP_TOKEN"). We implemented the same mapping as the authors of CoTexT [29] to address this issue for all three datasets. We also pre-pended the string "defect detection: " to each code snippet, since the authors used this approach to distinguish between the different tasks in their multi-task setup. Furthermore, we mapped all the labels of our datasets to either the tokens "positive" (vulnerable) or "negative" (not vulnerable) to match the text-to-text architecture of CoTexT. We verified our pre-processing by successfully reproducing the original scores reported by Phan et al. [30]. For the models trained with PLBart and VulBERTa, we did not employ additional technique-specific pre-processing steps.

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 pre-processing steps.

Hyperparameters. For all three ML techniques, we used the pre-trained models provided by the respective authors as starting points for our experiments. Similar to Hanif et al. [14], we noticed a relatively quick convergence of our performance metrics in our initial experiments (after 2-7 epochs), which is why we trained each model instance for 10 epochs. After each epoch, we collected the average loss and several performance metrics both for the training dataset and the testing dataset. For all hyperparameters except the batch size for CoTexT, we used the values that were reported in the original papers [2, 14, 30]. Due to limited VRAM of our hardware, we had to change the batch size for CoTexT from 128 to 8. To make sure that this change had no negative impact on the performance of CoTexT, we verified that we can still reproduce the reported scores of the original publication [30]. Consult our published training scripts (see Section 4.7) for the complete list of hyperparameter values that we used.

Tokenization. For all three ML techniques, we used the pre-configured tokenizers that the authors provide in their open-source implementations [26, 29, 37].

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 two metrics in this paper: Accuracy and F1-score.

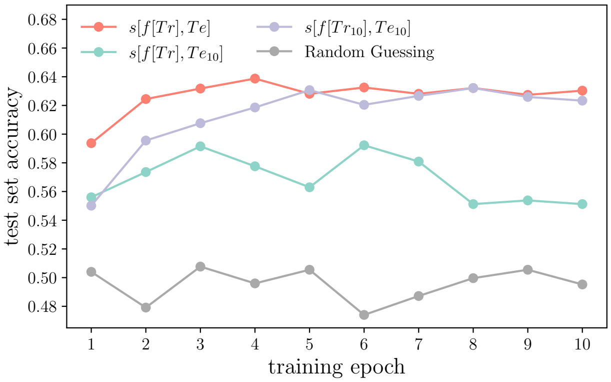
For CodeXGLUE/Devign as a balanced dataset (45.6% vulnerable functions), we use accuracy as the main performance metric. Accuracy is defined as the percentage of correct predictions for a particular testing dataset, and is the most popular metric for binary classification tasks in machine learning. Both in the CodeXGLUE benchmark [24] and on the leaderboard [25] accuracy is used as main performance measure.

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.

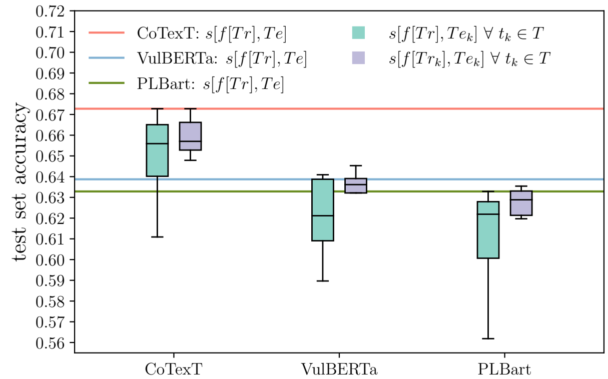
Hardware. We used a setup of five Nvidia A100 GPUs, each equipped with 40 GB HBM2 memory. One complete run of all our experiments takes approx. 45 days on a single A100 GPU, which equals approx. 45/5 = 9 days when all five GPUs are utilized.

[1]Source: https://api.semanticscholar.org/CorpusID:231855531
[2]Source: https://github.com/microsoft/CodeXGLUE
[3]Source: https://api.semanticscholar.org/CorpusID:202539112

https://t.me/learningnets



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



(b) Extension of the results in Figure 2a, for all transformations $t_k \in T$, and for all three ML techniques. The boxplots represent distributions of the resulting accuracies. Amplifying the testing set Te with transformations $t_k \in T$ (green boxplots) decreases the accuracy, but applying the same transformation also to the training dataset Tr (purple boxplots) restores the accuracy back to previous levels.

Figure 2: Effects of amplifying the testing data and the training data using the same semantic preserving transformations.

4.7 Reproducibility

To facilitate open science and rigorous reproducibility, we publish all of our data, code and results. At the time of submission, they are available at the following URL.

★ `https://github.com/niklasrisse/LimitsOfML4Vuln`

In addition to all of the data and results, the repository contains Python scripts with deterministic settings, so that our experimental results can be exactly reproduced.

5 EXPERIMENTAL RESULTS

RQ.1 Testing- and Training Data Amplification

We investigate, whether (a) testing data amplification using semantic preserving transformation decreases the performance of state-of-the-art ML techniques for vulnerability detection, and (b) whether training data amplification using the same transformations restores the performance towards previous levels.

Methodology. We used Algorithm 1 to investigate both 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 2a shows the test set accuracy of different VulBERTa models measured after each of the ten training epochs. We can observe, that amplifying the testing dataset Te with transformation t_{10} leads to a substantial drop in accuracy, represented by the gap between the green and red graphs in the figure. We can also observe, that amplifying the training dataset Tr with the same transformation t_{10} as the testing dataset, restores the accuracy back to previous levels (purple graph).

Figure 2b extends the results of Figure 2a to all semantic preserving transformations $t_k \in T$, and to all three ML techniques. Instead of showing the accuracy for each training epoch, we only use the maximum accuracy across the full training. Across all three ML techniques we can observe, that amplifying the testing dataset Te with transformations $t_k \in T$ (represented by the green boxplots), on average, leads to a drop in accuracy compared to evaluations on the standard testing dataset Te (represented by the horizontal lines). In parallel to Figure 2a we can also observe that, on average, training data amplification using the same transformation as for testing data amplification leads to a restoration of the observed drops in accuracy (represented by the purple boxplots).

Table 2 summarizes the outputs of Algorithm 1 for all three ML techniques and both datasets. Specifically, it shows the average recorded drops in score, when only the testing dataset was amplified (fourth column, o_1), the training and testing datasets were amplified using the same transformation (fifth column, o_2), and the training and testing datasets were amplified using a different transformation (sixth column, o_3). We observe that, on average, amplifying the testing dataset leads to a drop in score (−0.020 for CodeXGLUE, −0.030 for VulDeePecker), and amplifying the training dataset using the same transformation restores that decrease towards previous levels. On average, approx. 59.8% (CodeXGLUE) and 54.1% (VulDeePecker) of the lost score is restored.

Across 2 datasets, 3 ML techniques, and 11 transformations, on average, (a) testing data amplification using semantic preserving transformations leads to a drop in score (CodeXGLUE: −0.020, VulDeePecker: −0.030), and (b) training data amplification using the same transformations restores 59.8% (CodeXGLUE) and 54.1% (VulDeePecker) of the lost performance.

RQ.1 has already been studied in the literature, for many different ML techniques, datasets, and tasks [5, 17, 21, 22, 36, 38, 39, 41, 42]. Based on our evidence, we can approve the findings of the literature.

https://t.me/learningnets

Table 2: Algorithm 1 Results: Average effects of amplifying only the testing data (o_1), training and testing data using the same (o_2), or a different transformation (o_3).

Dataset	Technique	$s[Tr, Te]$	$o_1 = \sum_k \frac{e[Tr, Te_k]}{N}$	$o_2 = \sum_k \frac{e[Tr_k, Te_k]}{N}$	$o_3 = \sum_k \sum_{j \neq k} \frac{e[Tr_k, Te_j]}{(N(N-1))}$
CodeXGLUE	VulBERTa	0.639	-0.017 ↓	-0.004 ↑	-0.025 ↓
CodeXGLUE	CoTexT	0.673	-0.022 ↓	-0.014 ↑	-0.030 ↓
CodeXGLUE	PLBart	0.633	-0.021 ↓	-0.007 ↑	-0.026 ↓
			-0.020 ↓	**-0.008** ↑	**-0.027** ↓
VulDeePecker	VulBERTa	0.875	-0.067 ↓	-0.011 ↑	-0.057 ↓
VulDeePecker	CoTexT	0.873	-0.007 ↓	-0.005 ↑	-0.018 ↓
VulDeePecker	PLBart	0.865	-0.014 ↓	-0.007 ↑	-0.035 ↓
			-0.030 ↓	**-0.007** ↑	**-0.037** ↓

RQ.2 Over-fitting to Specific Transformations

We investigate, whether (a) the performance of ML techniques can still be restored if we amplify the training dataset with a different semantic preserving transformation than the testing dataset, and (b) whether the performance can be restored if we replace the training data amplification with adversarial training.

Methodology. We use Algorithm 1 to investigate RQ.2(a), with the same setup as for RQ.1. To investigate research question RQ.2 (b), we use an altered version of Algorithm 1, in which we train an additional model with adversarial training using the method proposed by Yefet et al. [39] (see Section 4.2) after the standard training in line 1 of the algorithm. We denote this adversarially trained model by $f[Tr_{ADV}]$.

Results. Figure 3a is similar to Figure 2a, 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 2a, Figure 3a shows the accuracies (gray lines) of models trained on data that was amplified using all transformations except t_{10}, which was used to amplify the testing dataset. We observe, that amplifying the training dataset Tr with different transformations $t_{k \neq 10}$ as the testing dataset does not restore the accuracy to previous levels.

Figure 3b visualizes the same results for all semantic preserving transformations $t_k \in T$ and for all three ML techniques. Again, the green and the purple boxplots represent the distributions of accuracies, when either only the testing dataset (green) or training and testing datasets were amplified using the same transformation (purple). The yellow boxplots represent the distribution of accuracies achieved by models that were trained on data, which was amplified using a different transformation than for the testing data. Across all three ML techniques we observe that, on average, amplifying the training dataset Tr with a different transformations $t_{k \neq j}$ as the testing dataset does not restore the accuracy back to previous levels.

Figure 3c visualizes the same results as Figure 3b, but using the VulDeePecker dataset. In this case, the y-axis measures the F1-score, since this is the preferred evaluation metric for the VulDeePecker dataset. Again, we observe that, on average, amplifying the training dataset Tr with a different transformations $t_{k \neq j}$ as the testing dataset does not restore the accuracy back to previous levels.

In addition to the results for RQ.1, Table 2 also shows the average recorded drops in score, when the training and testing datasets were amplified using different transformations (sixth column, o_3). We observe that, on average, the score drops by 0.027 (CodeXGLUE) and 0.037 (VulDeePecker). Across the three techniques, the decrease is on average 35.7% (CodeXGLUE) and 97.4% (VulDeePecker) stronger than for training on unamplified data. In other words, amplifying 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.

Figure 3d shows the results of extending Algorithm 1 to adversarial training. In parallel to Figure 3b and Figure 3c, the green and the purple boxplots represent the distributions of accuracies, when either only the testing dataset (green) or training and testing datasets were amplified using the same transformation (purple). The pink boxplots represent the distribution of accuracies achieved by adversarial training for testing datasets amplified using the different transformations $t_k \in T$. We observe, that adversarial training also does not restore the accuracy back to previous levels. Across all transformations $t_k \in T$, on average, adversarial training leads to 0.002 (VulBERTa), -0.014 (CoTexT), and 0.0 (PLBart) improvement in accuracy compared to unamplified training.

(a) Across 2 datasets, 3 ML techniques, and 11 transformations, amplifying the training dataset using a different transformation than for the testing dataset does not restore the performance back to previous levels. Instead of restoring, on average, the performance even drops by 35.7% (CodeXGLUE) and 97.4% (VulDeePecker) more compared to using standard training data. In other words, the ML techniques over-fit to the label-unrelated features introduced by semantic preserving transformations during training. (b) Replacing training data amplification with adversarial training also does not restore the performance back to previous levels.

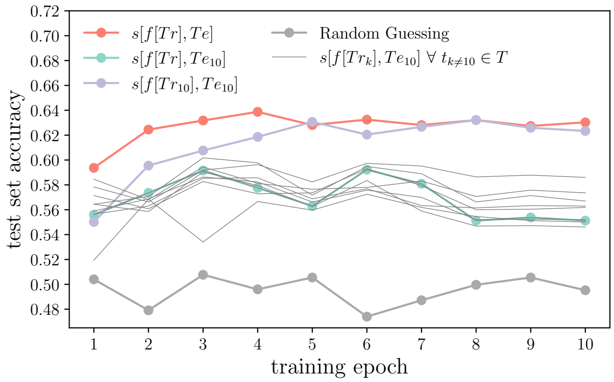
In summary, we can observe that across the tested ML techniques, transformations and datasets, training data amplification does only restore the performance back to previous levels when the testing dataset is amplified in a similar way than the training dataset.

The robustness gained by data amplification only applies to the specific transformations used during training of the model.

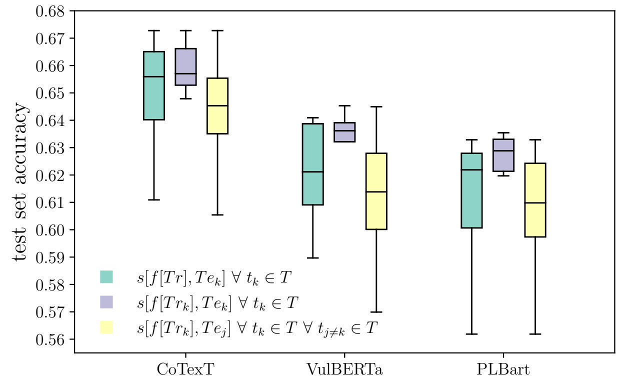
RQ.3 Generalization to VulnPatchPairs

We investigate, whether (a) ML techniques are able to generalize from typical vulnerability detection training datasets to a modified setting, in which they have to distinguish between vulnerabilities

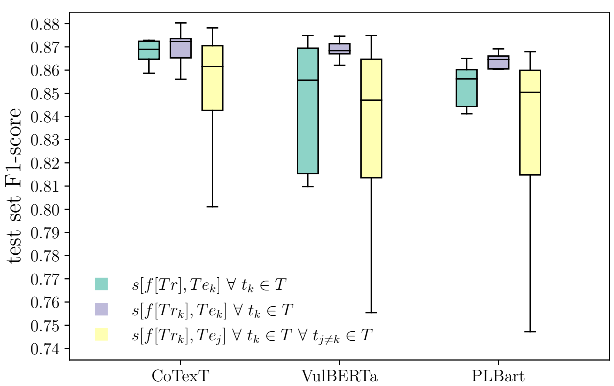
https://t.me/learningnets



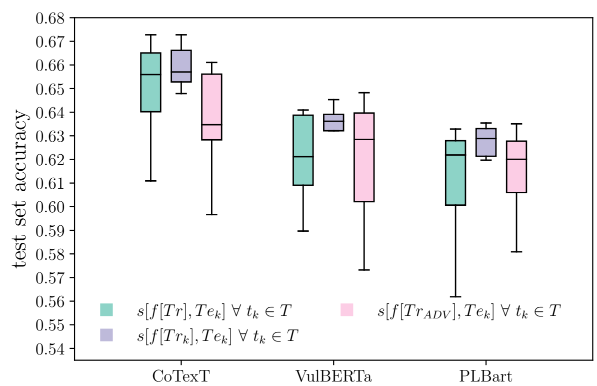
(a) Test set accuracy over ten training epochs of different models trained with VulBERTa on the CodeXGLUE/Devign dataset. Amplifying the training set Tr with different transformations $t_{k\neq 10}$ than the testing dataset (gray lines) does not restore the accuracy back to previous levels.



(b) Extension of the results in Figure 3a, for all transformations $t_k \in T$ and for all three ML techniques. The boxplots represent distributions of the resulting accuracies. Amplifying the training set Tr with different transformations $t_k \neq j$ than the testing dataset (yellow boxplots) does not restore the accuracy back to previous levels.



(c) Same setup as for Figure 3b, but using the VulDeePecker dataset. The boxplots represent distributions of the resulting F1-scores. Amplifying the training set Tr with different transformations $t_k \neq j$ than the testing dataset (yellow boxplots) does not restore the F1-score back to previous levels.



(d) Similar setup as for Figure 3b, but using adversarial training to amplify the training dataset. Amplifying the training set Tr with adversarial training (pink boxplots) does not restore the accuracy back to previous levels.

Figure 3: Effects of amplifying the training data with different semantic preserving transformations than the testing data.

and their patches, and (b) whether training data amplification leads to an improvement in performance in this modified setting.

Methodology. We used Algorithm 2 to investigate both questions. We ran the algorithm for all three ML techniques seperately, and used the training subset of the CodeXGLUE/Devign dataset (see Section 4.3) as training dataset Tr. We used the testing subset of VulnPatchPairs (see Section 4.4) as the vulnerability-patch testing dataset VPT.

Results. Table 3 shows the computed outputs of running Algorithm 2. Specifically, it shows the accuracy of different models evaluated on the standard CodeXGLUE/Devign testing dataset Te (second column, $s[f[Tr], Te]$), evaluated on the vulnerability-patch testing dataset VPT (third column, o_1), and the average accuracy of models trained on amplified data and evaluated on the vulnerability-patch testing dataset VPT (fourth column, o_2). We observe, that the accuracy of all three ML techniques evaluated on the vulnerability-patch testing dataset VPT (o_1) is far worse than the accuracy on the standard CodeXGLUE/Devign testing dataset Te ($s[f[Tr], Te]$). Since VPT is perfectly balanced (50% vulnerable, 50% not vulnerable), a random guesser (coin flip) would be expected to achieve an accuracy of 0.5. On average, the three ML techniques achieve an accuracy of 0.518, which is only 0.018 higher than the expected accuracy of a random guesser.

From the fourth and the fifth columns of Table 3 (o_2 and o_3) we can also observe that, on average, training data amplification leads to no improvement of the accuracy compared to the models trained on unamplified training data.

(a) All three ML techniques are not able to distinguish between vulnerabilities and their patches. On average, the performance is only 0.018 better than the expected accuracy of a random guesser. In other words, the ML techniques are unable generalize from their training data to a slightly modified vulnerability detection setting. (b) Training data amplification has no effect on the ability of the models to generalize to the modified setting.

https://t.me/learningnets

Table 3: Algorithm 2 Results: Accuracy on VulnPatchPairs (o_1) and the effects of training data amplification (o_2, o_3).

Technique	$s[f[Tr], Te]$	$o_1 = s[f[Tr], VPT]$	$o_2 = \frac{\sum_k s[f[Tr_k], VPT]}{N}$	$o_3 = \frac{\sum_k e[f[Tr_k], VPT]}{N}$
VulBERTa	0.639	0.527	0.522	-0.005
CoTexT	0.673	0.503	0.501	-0.001
PLBart	0.633	0.524	0.528	0.004
	0.648	**0.518**	**0.517**	**-0.001**

6 THREATS TO VALIDITY

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

Internal validity. There are several potential sources of systematic error in our experiments.

A common source of systematic error in empirical studies on machine learning 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 ML 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. [27].

Even the choice of random seed can be a source of systematic error in machine learning training pipelines. We addressed this by randomly selecting and then fixing random seeds prior to carrying out the particular experiments.

External validity. The degree to which our results generalize to other machine learning techniques, vulnerability detection 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 three state-of-the-art techniques, two datasets, two performance metrics, and 11 semantic preserving transformations. However, all evaluated ML techniques happen to be token-based large language models (LLMs). As our selection criterion, we defined the top-performing ML techniques on the most widely known ML vulnerability detection benchmark CodeXGLUE [24, 25] that are available as open source. This gave us ranks 1, 5, and 9 of the leaderboard, all of which are token-based LLMs. In fact, 8 of the Top-10 solutions on the leaderboard are token-based LLMs. Investigating, whether our results generalize to other types of techniques, e.g. to graph-based neural networks or path-based sequential models, would be an interesting avenue for future work. 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.

The concern of generalization is also valid for our results on adversarial training (see Figure 3d), which we computed by using only one particular technique [39] to compute the adversarial transformations.

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 robustness gained by training data amplification only applies to the specific transformations used for training and that the techniques that we investigated over-fit to the label-unrelated features introduced by these transformations. For a new technique, they could be replaced by a different set of transformations.

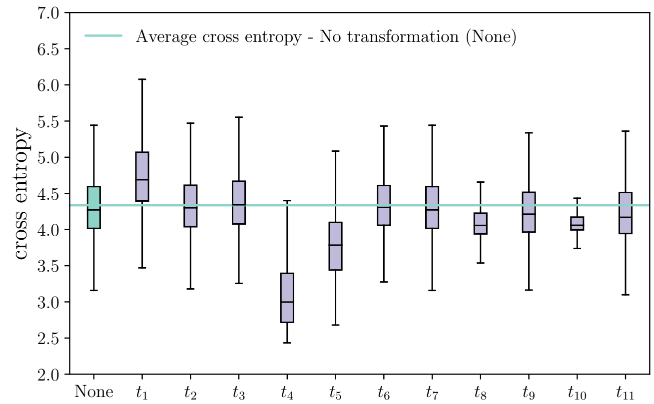


Figure 4: Naturalness of the semantic preserving transformations, that we used in our experiments. Lower cross entropy means higher naturalness. All transformations except t_1 (identifier renaming) lead to lower or equal cross entropy than no transformation (None).

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 which 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. [18] and present the results in Figure 4. Following the method of Hindle et al., we first trained a 2-gram markov model on the CodeXGLUE training dataset. Using the trained model, 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. [18]). Using this approach, we computed the cross entropy for all code snippets in the CodeXGLUE testing dataset (green boxplot), and for transformed versions of the CodeXGLUE testing dataset (purple boxplots). The green 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.

Class balance. He et al. [15] have shown that learning-based vulnerability detection techniques trained on fairly balanced datasets (such as CodeXGLUE and VulDeePecker) often fail to generalize to real-world code repositories, which usually contain a much smaller ratio of security vulnerabilities [19]. However, measured by citations, these balanced datasets are still by far the most popular datasets to evaluate machine learning techniques for vulnerability

https://t.me/learningnets

detection. To our current knowledge, there is no vulnerability detection dataset with sufficient size (more than 10k code snippets), high quality labels (provided by security experts), and a realistic distribution of vulnerable to non-vulnerable code snippets that is widely used in the research community to evaluate machine learning techniques (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

In summary, our results demonstrate that the robustness gained by data amplification only applies to the specific transformations used during training of the model, that state-of-the-art ML techniques still over-fit to the label-unrelated features introduced by semantic preserving transformations during training, and that they are not able to generalize to a modified setting, in which they have to distinguish between vulnerabilities and their patches.

Over-fitting of learned models is a well-known problem in the machine learning research field [10, 40]. The degree of over-fitting for a specific machine learning model is usually measured as the gap in score quantified by a chosen performance metric between the training dataset and a held-out test dataset. Our proposed Algorithm 1 is a novel way to measure over-fitting of ML techniques for vulnerability detection, that goes beyond the performance gap between training and testing datasets in a standard evaluation setup, and can even detect over-fitting if there is no gap in the standard setup at all. There are several common strategies to reduce over-fitting in the standard evaluation setup, e.g. early-stopping, dropout or large pre-training datasets [40], which are already integrated in the ML techniques that we used in our experiments. However, our experiments demonstrate that the techniques are still severely over-fitting to label-unrelated features introduced by semantic preserving transformations during training data amplification. Finding ways to robustify ML techniques without or with minimal over-fitting will be a central challenge of the machine learning for vulnerability detection research area. We hope that our proposed algorithms can be used to understand the problem, to develop new approaches and to track the progress in this direction.

Generalization. The results for RQ.3 (see Section 5) reveal, that state-of-the-art ML techniques for vulnerability detection 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 in 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 machine learning techniques has been recently identified (e.g. in the computer vision domain [16, 35]). 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 an interesting avenue for future work to try approaches that have been used to address out-of-distribution generalization in other domains (e.g. causal representation learning [34]) on the task of automatic vulnerability detection, and measure the success using our Algorithm 2.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. 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*. Association for Computational Linguistics, Online, 2655–2668. https://www.aclweb.org/anthology/2021.naacl-main.211
[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. 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*. Association for Computational Linguistics, Online, 2655–2668. https://www.aclweb.org/anthology/2021.naacl-main.211
[3] Leonhard Applis, Annibale Panichella, and Arie van Deursen. 2021. Assessing Robustness of ML-Based Program Analysis Tools using Metamorphic Program Transformations. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1377–1381. https://doi.org/10.1109/ASE51524.2021.9678706
[4] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. 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* (Athens, Greece) *(PROMISE 2021)*. Association for Computing Machinery, New York, NY, USA, 30–39. https://doi.org/10.1145/3475960.3475985
[5] Pavol Bielik and Martin Vechev. 2020. Adversarial Robustness for Code. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 84, 12 pages.
[6] Paul Black. 2018. A Software Assurance Reference Dataset: Thousands of Programs With Known Bugs. https://doi.org/10.6028/jres.123.005
[7] Harold Booth, Doug Rike, and Gregory Witte. 2013. The National Vulnerability Database (NVD): Overview. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=915172
[8] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. 2022. 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* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 464–468. https://doi.org/10.1145/3524842.3528482
[9] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. 2020. Exploring software naturalness through neural language models. *arXiv preprint arXiv:2006.12641* (2020).
[10] Tom Dietterich. 1995. Overfitting and Undercomputing in Machine Learning. *ACM Comput. Surv.* 27, 3 (sep 1995), 326–327. https://doi.org/10.1145/212094.212114
[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139
[12] FFmpeg 2023. *FFmpeg GitHub repository*. Retrieved March 8, 2023 from https://github.com/FFmpeg/FFmpeg
[13] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE.
[14] Hazim Hanif and Sergio Maffeis. 2022. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*. 1–8. https://doi.org/10.1109/IJCNN55064.2022.9892280
[15] Jingxuan He, Luca Beurer-Kellner, and Martin Vechev. 2022. On Distribution Shift in Learning-based Bug Detectors. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.). PMLR, 8559–8580. https://proceedings.mlr.press/v162/he22a.html
[16] 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. 2021. 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)*. 8340–8349.
[17] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2022. Semantic Robustness of Models of Source Code. In

https://t.me/learningnets

2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. https://doi.org/10.1109/saner53432.2022.00070

[18] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[19] Rafael-Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. In *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*. 573–577. https://doi.org/10.1145/3379597.3387491

[20] Yaoxian Li, Shiyi Qi, Cuiyun Gao, Yun Peng, David Lo, Zenglin Xu, and Michael R. Lyu. 2022. A Closer Look into Transformer-Based Code Intelligence Through Code Transformation: Challenges and Opportunities. https://doi.org/10.48550/ARXIV.2207.04285

[21] Yiyang Li, Hongqiu Wu, and Hai Zhao. 2022. Semantic-Preserving Adversarial Code Comprehension. In *Proceedings of the 29th International Conference on Computational Linguistics*. International Committee on Computational Linguistics, Gyeongju, Republic of Korea, 3017–3028. https://aclanthology.org/2022.coling-1.267

[22] Zhen Li, Jing Tang, Deqing Zou, Qian Chen, Shouhuai Xu, Chao Zhang, Yichen Li, and Hai Jin. 2021. Towards Making Deep Learning-based Vulnerability Detectors Robust. https://doi.org/10.48550/ARXIV.2108.00669

[23] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society. https://doi.org/10.14722/ndss.2018.23158

[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. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, J. Vanschoren and S. Yeung (Eds.), Vol. 1. https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/file/c16a5320fa475530d9583c34fd356ef5-Paper-round1.pdf

[25] Microsoft 2021. *CodeXGLUE leaderboards*. Retrieved March 8, 2023 from https://microsoft.github.io/CodeXGLUE/#LB-DefectDetection

[26] ML4CSec research group from the Department of Computing, Imperial College London 2022. *VulBERTa GitHub repository*. Retrieved March 8, 2023 from https://github.com/ICL-ml4csec/VulBERTa

[27] ML4CSec research group from the Department of Computing, Imperial College London 2022. *VulDeePecker Function-level dataset*. Retrieved March 8, 2023 from https://github.com/ICL-ml4csec/VulBERTa/tree/main/data

[28] Pedro Orvalho, Mikoláš Janota, and Vasco Manquinho. 2022. 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*. 1657–1661.

[29] Long Phan. 2023. *CoTexT GitHub repository*. Retrieved March 8, 2023 from https://github.com/justinphan3110/CoTexT

[30] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Annibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task Learning with Code-Text Transformer. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. Association for Computational Linguistics, Online, 40–47. https://doi.org/10.18653/v1/2021.nlp4prog-1.5

[31] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 383–387. https://doi.org/10.1109/MSR.2019.00064

[32] Qemu 2023. *Qemu GitHub repository*. Retrieved March 8, 2023 from https://github.com/qemu/qemu

[33] Md Rafiqul Islam Rabin, Nghi D.Q. Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (jul 2021), 106552. https://doi.org/10.1016/j.infsof.2021.106552

[34] B. Schölkopf, F. Locatello, S. Bauer, N. R. Ke, N. Kalchbrenner, A. Goyal, and Y. Bengio. 2021. Toward Causal Representation Learning. *Proc. IEEE* 109, 5 (2021), 612–634. https://doi.org/10.1109/JPROC.2021.3058954

[35] Zheyan Shen, Jiashuo Liu, Yue He, Xingxuan Zhang, Renzhe Xu, Han Yu, and Peng Cui. 2021. Towards Out-Of-Distribution Generalization: A Survey. https://doi.org/10.48550/ARXIV.2108.13624

[36] Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. 2021. Generating Adversarial Computer Programs using Optimized Obfuscations. In *International Conference on Learning Representations*. https://openreview.net/forum?id=PH5PH9ZO_4

[37] Ahmad Wasi. 2021. *PLBart GitHub repository*. Retrieved March 8, 2023 from https://github.com/wasiahmad/PLBART

[38] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural Attack for Pre-Trained Models of Code. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1482–1493. https://doi.org/10.1145/3510003.3510146

[39] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial Examples for Models of Code. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 162 (nov 2020), 30 pages. https://doi.org/10.1145/3428230

[40] Xue Ying. 2019. An Overview of Overfitting and its Solutions. *Journal of Physics: Conference Series* 1168, 2 (feb 2019), 022022. https://doi.org/10.1088/1742-6596/1168/2/022022

[41] Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. 2022. Towards Robustness of Deep Program Processing Models—Detection, Estimation, and Enhancement. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 50 (apr 2022), 40 pages. https://doi.org/10.1145/3511887

[42] Huangzhao Zhang, Zhuo Li, Ge Li, L. Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. In *AAAI Conference on Artificial Intelligence*.

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

https://t.me/learningnets