CE 815 – Secure Software Systems

ML-Based Vulnerability Detection Methods (Learning Limitations)

Mehdi Kharrazi
Department of Computer Engineering
Sharif University of Technology

Acknowledgments: Some of the slides are fully or partially obtained from other sources. A reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.

Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection, Niklas Risse, Marcel Böhme, Usenix Security 2024.

The Promise and Limitations of ML for Vulnerability Detection

- Current Achievements:
  - Machine Learning for Vulnerability Detection (ML4VD) models achieve up to 70% accuracy in identifying security flaws from source code.
  - Claims of outperforming traditional program analysis methods without hardcoded program semantics.
- Key Contradictions:
  - Models struggle to distinguish vulnerable functions from their patched counterparts.
  - High benchmark scores may give a false sense of security.
- Challenges Highlighted:
  - Overfitting: Models depend on unrelated features in the training data.
  - Generalization Issues: Poor performance on out-of-distribution data.

Proposed Solutions and Contributions

- Proposed Methodology:
    - Algorithm 1: Tests overfitting to unrelated features by using semantic-preserving transformations.
    - Algorithm 2: Assesses model ability to distinguish vulnerabilities from patches.
- Key Contributions:
    - Identification of critical flaws in current evaluation methods.
    - Introduction of a new dataset, VulnPatchPairs, featuring matched pairs of vulnerable and patched functions.
    - Empirical findings:
        - Severe overfitting to unrelated features during training.
        - Lack of generalization across vulnerability-related contexts.

Expectations for Vulnerability Detection Models

- General Expectations:
  - Predict vulnerabilities accurately regardless of transformations.
  - Remain robust to both semantic-preserving and label-inverting changes.
- Key Evaluation Criteria:
  - Semantic-Preserving: No change in prediction after transformation.
  - Label-Inverting: Prediction changes align with modified ground truth.
- Implications:
  - Robust models must handle diverse real-world code variations.

What is Data Augmentation?

- Definition:
  - Application of code transformations to code snippets in a dataset.
  - Ensures transformations preserve program semantics.
- Purpose:
  - Improve model robustness to variations in real-world code.
  - Test vulnerability detection models under diverse conditions.
- Core Concept:
  - Transformations should not change the ground truth vulnerability label, unless intended.

Types of Transformations

- Semantic-Preserving Transformations:
    - Changes that do not affect vulnerability status:
        - Identifier renaming.
        - Adding unused code or comments.
        - Reordering unrelated statements.
        - Replacing elements with equivalents.

Example: Semantic-Preserving Transformation

- Original Code:

```
int calculateSum(int a, int b) {
    int sum = a + b;
    return sum;
}
```

- Transformed Code (Semantic-Preserving):

```
int calculateSum(int firstParam, int secondParam) {
    // Calculate sum of two numbers
    int sum = firstParam + secondParam;
    return sum;
}
```

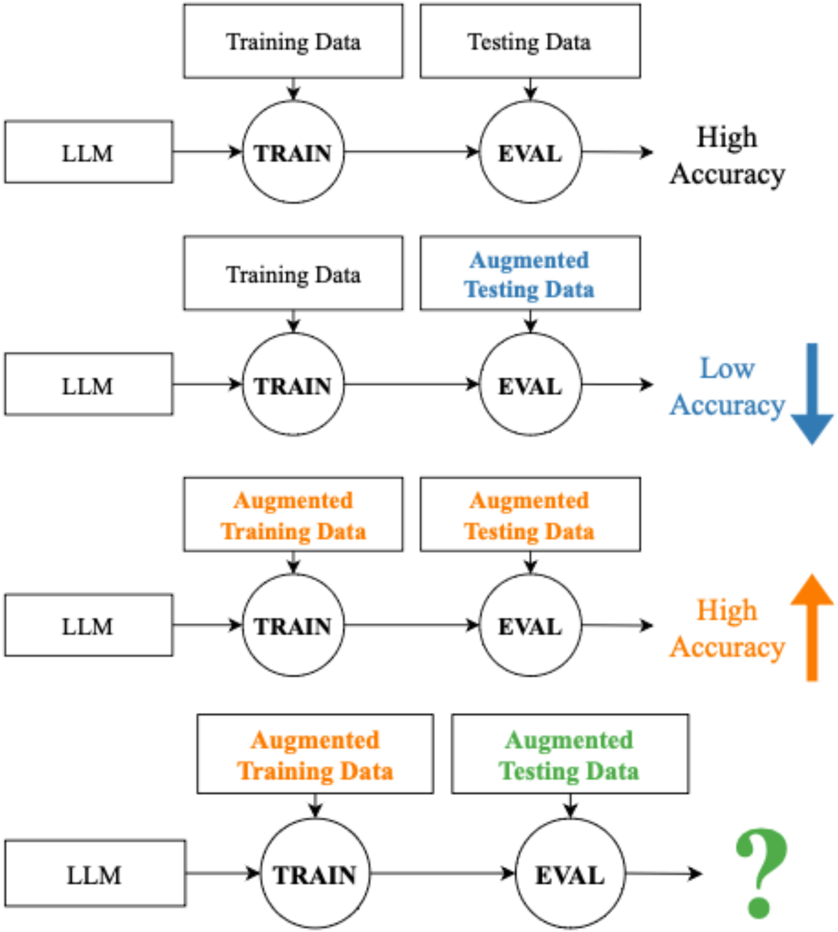
- Identifier Renaming:
  - a → firstParam, b → secondParam.
- Comment Insertion:
  - Added a comment describing the functionality.
- Key Point:
  - Ground Truth Label (e.g., vulnerable/non-vulnerable) remains the same.

Types of Transformations (con’t)

- Label-Inverting Transformations:
  - Changes that alter vulnerability status:
    - Adding a vulnerability to non-vulnerable code.
    - Removing a vulnerability from vulnerable code.
- Expected Behavior:
  - Models should:
    - Maintain predictions for semantic-preserving changes.
    - Adapt predictions accurately for label-inverting changes.

Goal of Algorithm 1 (Detecting Overfitting)

- Objective:
  - Assess if ML4VD models overfit to training data features unrelated to vulnerabilities.
  - Test if performance gains from training data augmentation generalize beyond specific transformations.
- Key Questions:
  - Does augmenting the testing data degrade performance?
  - Can augmenting the training data restore performance?
  - How does using different augmentations for training and testing affect performance?

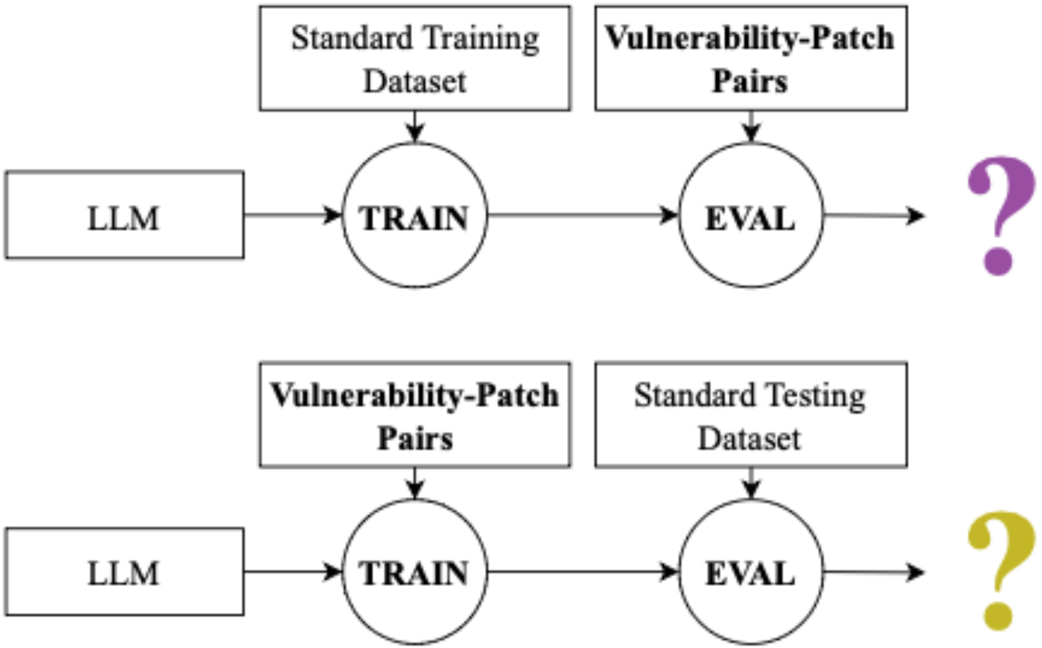

Training Data
Testing Data
LLM
TRAIN
EVAL
High
Accuracy
Training Data
Augmented
Testing Data
LLM
TRAIN
EVAL
Low
Accuracy
Augmented
Training Data
Augmented
Testing Data
LLM
TRAIN
EVAL
High
Accuracy
Augmented
Training Data
Augmented
Testing Data
LLM
TRAIN
EVAL
?

Key Insights from Algorithm 1

- Expected Results:
  - Testing augmentation without training augmentation reduces performance (outputA1.1>0).
  - Identical augmentations for training and testing partially restore performance (outputA1.2>outputA1.1).
  - Using different augmentations for training and testing causes performance drops (outputA1.3≪outputA1.2).
- Applications:
  - Identify overfitting to specific augmentations.
  - Assess model robustness across diverse data transformations.

Goal of Algorithm 2

- Objective:
  - Evaluate if ML4VD techniques can distinguish between vulnerabilities and their patches.
  - Test if models trained on one setting can generalize to another:
    - Standard vulnerability detection dataset.
    - Vulnerability-patch dataset.
- Key Questions:
  - Can models trained on standard datasets distinguish patched functions from vulnerable ones?
  - Can models trained on vulnerability-patch datasets perform well on standard datasets?


Standard Training Dataset
Vulnerability-Patch Pairs
LLM
TRAIN
EVAL
?
Vulnerability-Patch Pairs
Standard Testing Dataset
LLM
TRAIN
EVAL
?

Key Insights from Algorithm 2

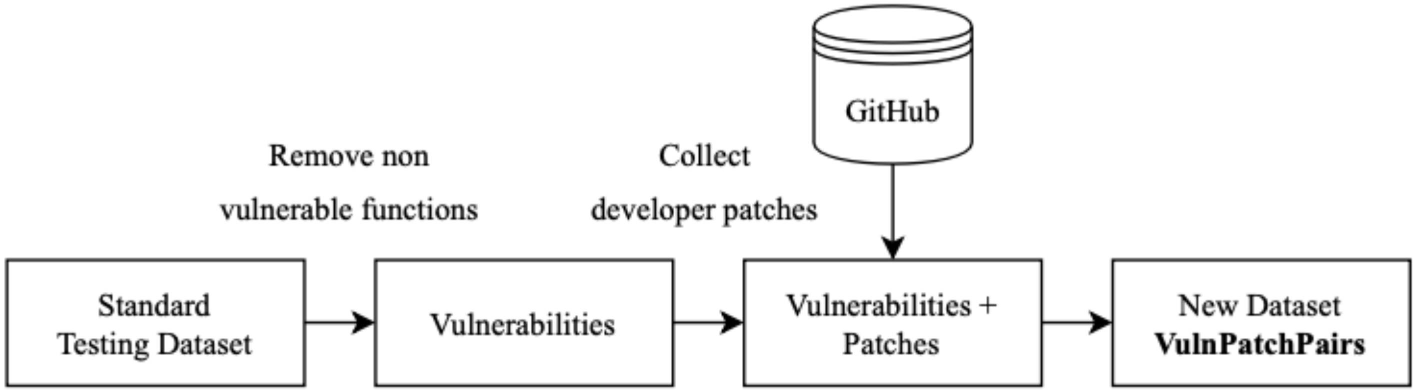
- Expected Results:
  - Models trained on standard datasets struggle with vulnerability-patch tasks (outputA2.2).
  - Models trained on vulnerability-patch tasks may generalize poorly to standard datasets (outputA2.4).
- Applications:
  - Evaluate real-world utility of ML4VD techniques.
  - Highlight gaps in generalization between standard and modified settings.

Experiments

- Impact of Data Augmentation:
  - How does testing data augmentation affect ML4VD performance?
  - Does training data augmentation restore performance?
- Overfitting:
  - Do ML4VD techniques overfit to specific augmentations?
  - Can models generalize across different augmentations?
- Generalization to Vulnerability-Patch Tasks:
  - Can ML4VD distinguish between vulnerabilities and their patches?
  - Does training on patches improve standard task performance?

Datasets Used

- CodeXGLUE/Devign:
  - 26.4k C functions, ~46% vulnerable.
  - Common vulnerabilities: memory-related (e.g., buffer overflows, memory leaks).
- VulDeePecker:
  - 61.6k C/C++ code samples, ~28% vulnerable.
  - Focus: buffer and resource management errors.
- VulnPatchPairs (New Dataset):
  - 26.2k C functions:
    - 13.1k vulnerable functions from CodeXGLUE.
    - 13.1k patched versions extracted from FFmpeg and QEMU repositories.


GitHub
Remove non
vulnerable functions
Collect
developer patches
Standard
Testing Dataset
Vulnerabilities
Vulnerabilities +
Patches
New Dataset
VulnPatchPairs

Training Pipeline

- Training Process:
  - Models pre-trained on large source code datasets (e.g., 2.3M - 680M snippets).
  - Fine-tuned for 10 epochs on selected datasets.
- Performance Metrics:
  - CodeXGLUE: Accuracy (balanced dataset).
  - VulDeePecker: F1-score (imbalanced dataset).
  - Additional Metrics: Precision, Recall, False Positive Rate (FPR), False Negative Rate (FNR).
- Hardware Setup:
  - 5 NVIDIA A100 GPUs (40 GB RAM each).
  - Approx. 60 days of compute time per full experiment on one GPU.

Semantic preserving transformations used

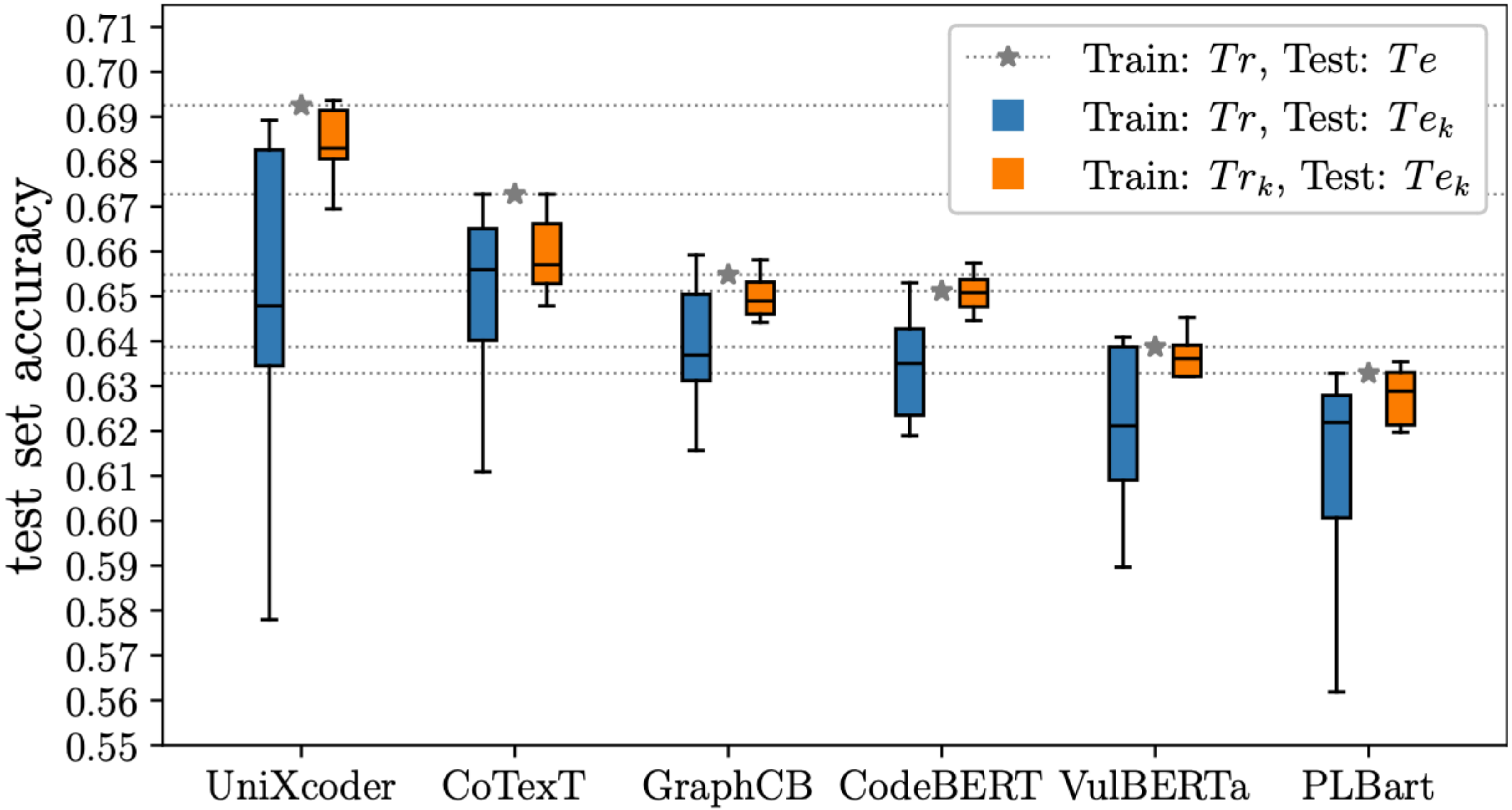
Identifier	Type	Description
t_1	Identifier Renaming	Rename all function parameters to a random token.
t_2	Statement Reordering	Reorder all function parameters.
t_3	Identifier Renaming	Rename the function.
t_4	Statement Insertion	Insert unexecuted code.
t_5	Statement Insertion	Insert comment.
t_6	Statement 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.

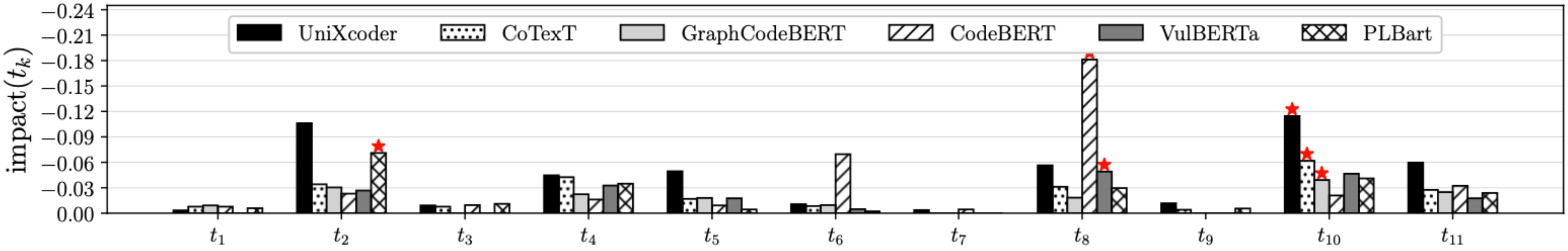
Experimental Design

- Algorithms Applied:
  - Algorithm 1: Detect overfitting to augmentations.
  - Algorithm 2: Test generalization to vulnerability-patch tasks.
- Transformations Used:
  - 11 semantic-preserving transformations (e.g., identifier renaming, statement reordering, comment removal).

Research Question 1 (Impact of Data Augmentation)

- Applying semantic-preserving transformations to testing data reduces performance (average drop):
    - CodeXGLUE: 2.5% accuracy.
    - VulDeePecker: 4.3% F1-score.
- Augmenting both training and testing data with the same transformations restores most performance:
    - ~69.0% of lost accuracy (CodeXGLUE).
    - ~66.2% of lost F1-score (VulDeePecker).
- Most Impactful Transformations: Adding comments, reordering statements, and inserting unused functions.

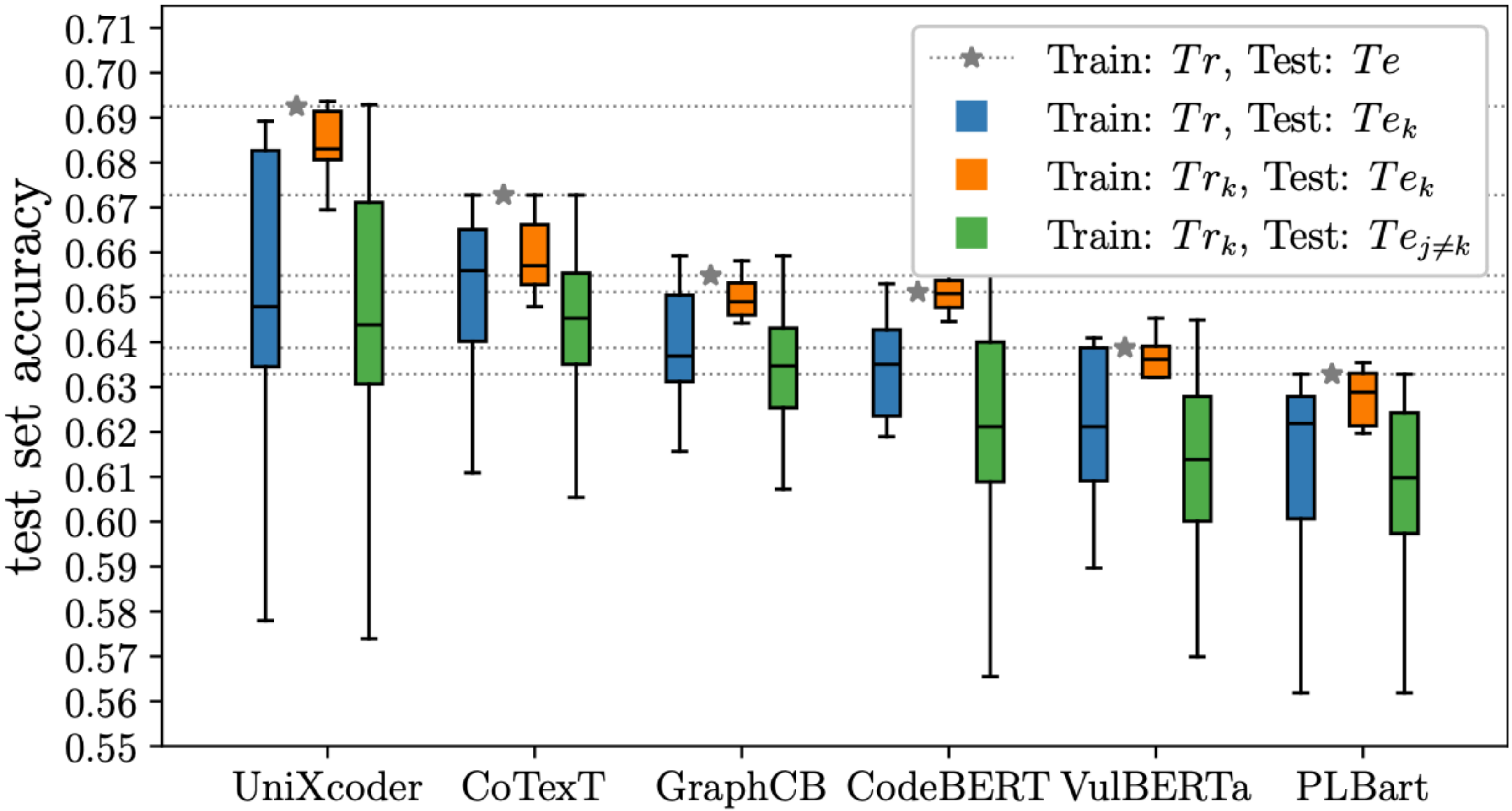

Train: Tr, Test: Te
Train: Tr, Test: Te_k
Train: Tr_k, Test: Te_k
test set accuracy
0.71
0.70
0.69
0.68
0.67
0.66
0.65
0.64
0.63
0.62
0.61
0.60
0.59
0.58
0.57
0.56
0.55
UniXcoder
CoTexT
GraphCB
CodeBERT
VulBERTa
PLBart



UniXcoder
CoTexT
GraphCodeBERT
CodeBERT
VulBERTa
PLBart
impact(t_k)
−0.24
−0.21
−0.18
−0.15
−0.12
−0.09
−0.06
−0.03
0.00
t_1
t_2
t_3
t_4
t_5
t_6
t_7
t_8
t_9
t_{10}
t_{11}

Research Question 2 (Overfitting to Specific Transformations)

- Training on transformations different from the testing data:
  - Performance restoration fails.
  - Results in an additional performance drop (30.2% for CodeXGLUE, 77.5% for VulDeePecker).
- Using a meta-transformation (combining various transformations except one):
  - Partially restores performance but does not fully mitigate the drop.
- Conclusion: ML4VD models overfit to specific augmentations and fail to generalize to unseen transformations.



Train: Tr, Test: Te
Train: Tr, Test: Te_k
Train: Tr_k, Test: Te_k
Train: Tr_k, Test: $Te_{j \neq k}$
test set accuracy
0.71
0.70
0.69
0.68
0.67
0.66
0.65
0.64
0.63
0.62
0.61
0.60
0.59
0.58
0.57
0.56
0.55
UniXcoder
CoTexT
GraphCB
CodeBERT
VulBERTa
PLBart

Research Question 3 (Generalization to Vulnerability-Patch Tasks)

- Standard to Patch Generalization:
  - Models trained on standard datasets performed worse than random guessing on vulnerability-patch tasks.
- Patch to Standard Generalization:
  - Models trained on vulnerability-patch data performed poorly on standard datasets, with a significant performance drop.
- Implications: ML4VD models cannot generalize across vulnerability-related contexts without task-specific training.

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**

Key Insights Across Experiments

- Testing data augmentation exposes dependence on unrelated features.
- Training on specific transformations limits generalization capability.
- Algorithm 1 reveals overfitting to label-unrelated features.
- Algorithm 2 demonstrates inability to generalize between vulnerabilities and patches.
- Impact on Real-World Use: Current ML4VD techniques are highly context-dependent and unsuitable for real-world vulnerability detection without targeted improvements.

Acknowledgments

- [Risse] Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection, Niklas Risse, Marcel Böhme, Usenix Security 2024.
- [VulChecker] VulChecker: Graph-based Vulnerability Localization in Source Code, Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno, and W. Lee, Usenix Security 2023.
- [Alves] Program Slicing. SwE 455, Alves, E., Federal University of Pernambuco, 2015.