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.

Vulnerability Detection Trends

- Machine Learning Era (2016-2020)
- Generalization Crisis (2021-2023)
- The LLM Reasoning Failure Era (2023-2024)
- The Neuro-Symbolic & Structured Reasoning Era (2025)
- Specialized vulnerability-reasoning LLM + agent orchestration (-10 Days)

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

- 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.

Expectations for Vulnerability Detection Models

- General Expectations:
  - Predict vulnerabilities accurately regardless of transformations.
  - Remain robust to both semantic-preserving and label-inverting changes.
- Concerns:
  - Overfitting: Models depend on unrelated features in the training data.
  - Generalization Issues: Poor performance on out-of-distribution data.

How to evaluate the concern?

- Proposed Methodology:
  - Algorithm 1: Tests overfitting to unrelated features by using semantic-preserving transformations.
  - Algorithm 2: Assesses model ability to distinguish vulnerabilities from patches.

What is Data Augmentation?

- Application of code transformations to code snippets in a dataset.
  - 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;
}
```

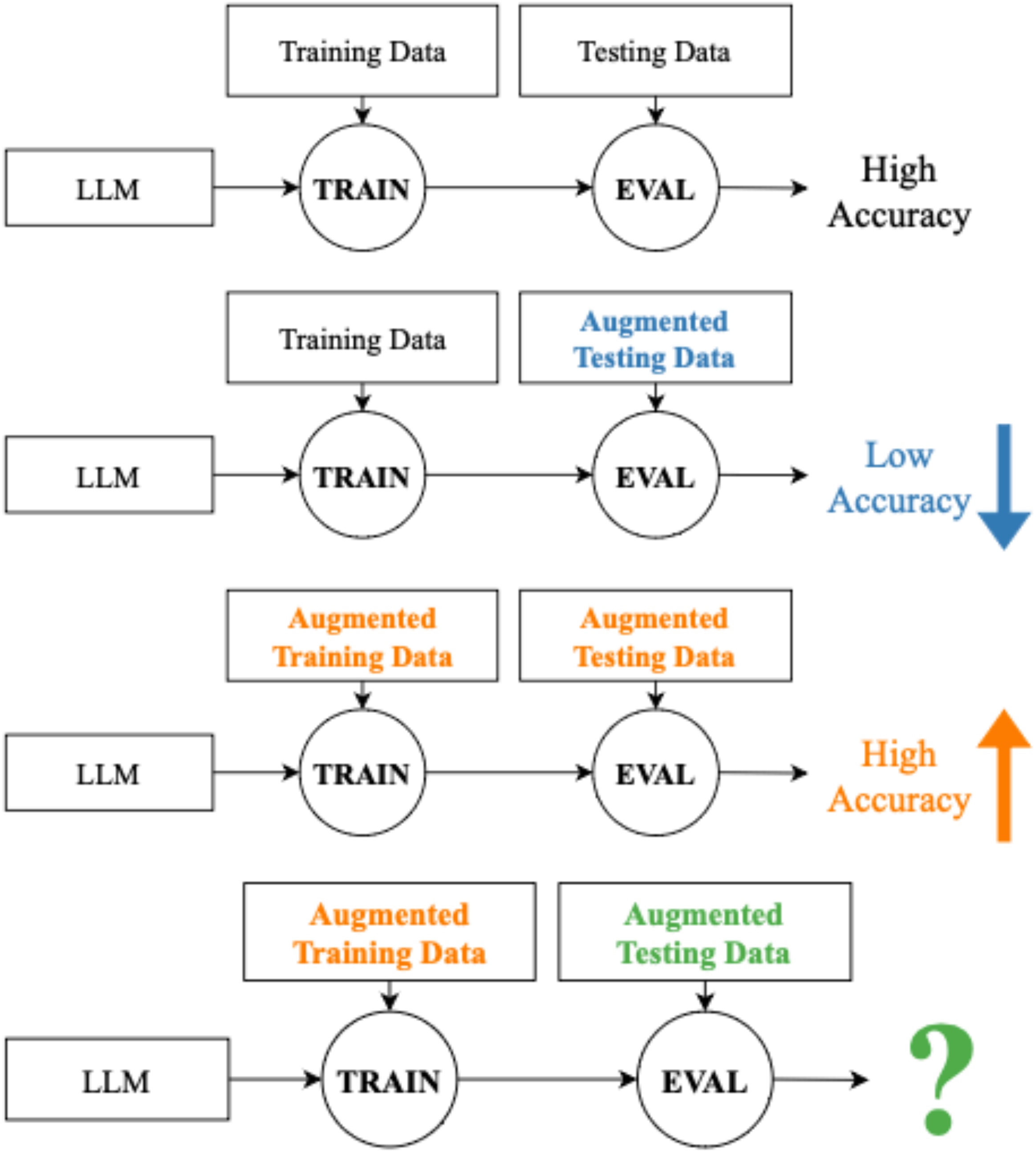
- Identifier Renaming:
  - a → firstParam, b → secondParam.
- Comment Insertion:
  - Added a comment describing the functionality.
- Key Point:
  - Ground Truth Label (e.g., vulnerable/non-vulnerable) remains the same.

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?

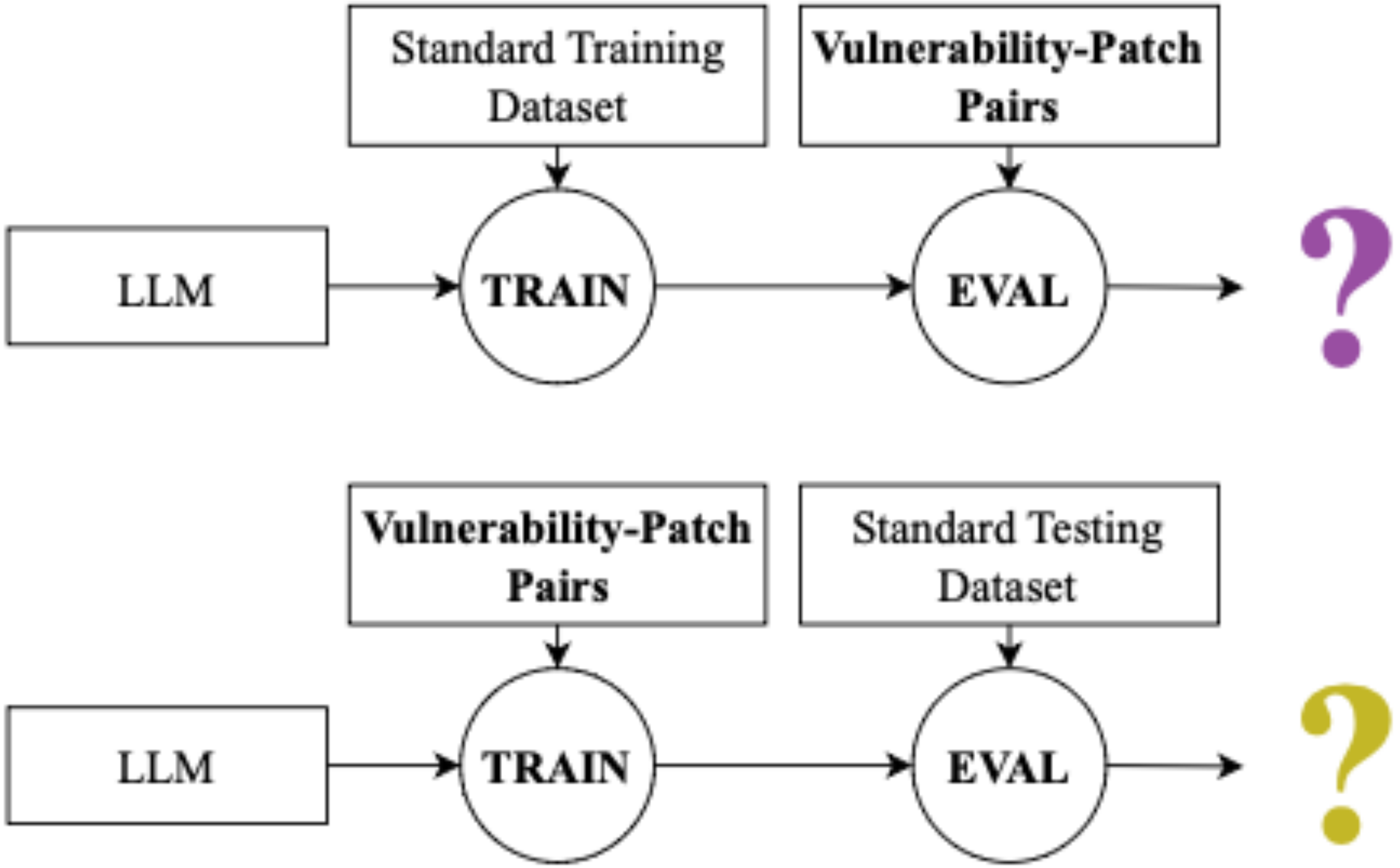

Training Data
Testing Data
LLM
TRAIN
EVAL
High Accuracy
Training Data
Augmented Testing Data
LLM
TRAIN
EVAL
Low Accuracy
Augmented Training Data
Augmented Testing Data
LLM
TRAIN
EVAL
High Accuracy
Augmented Training Data
Augmented Testing Data
LLM
TRAIN
EVAL
?

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?

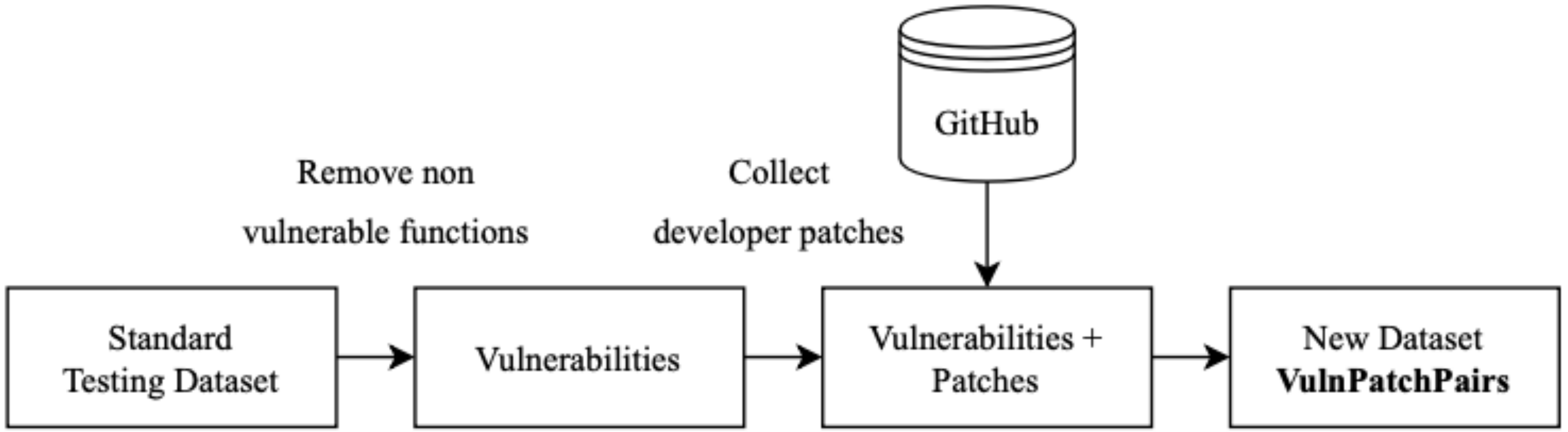

Standard Training Dataset
Vulnerability-Patch Pairs
LLM
TRAIN
EVAL
?
Vulnerability-Patch Pairs
Standard Testing Dataset
LLM
TRAIN
EVAL
?

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.

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

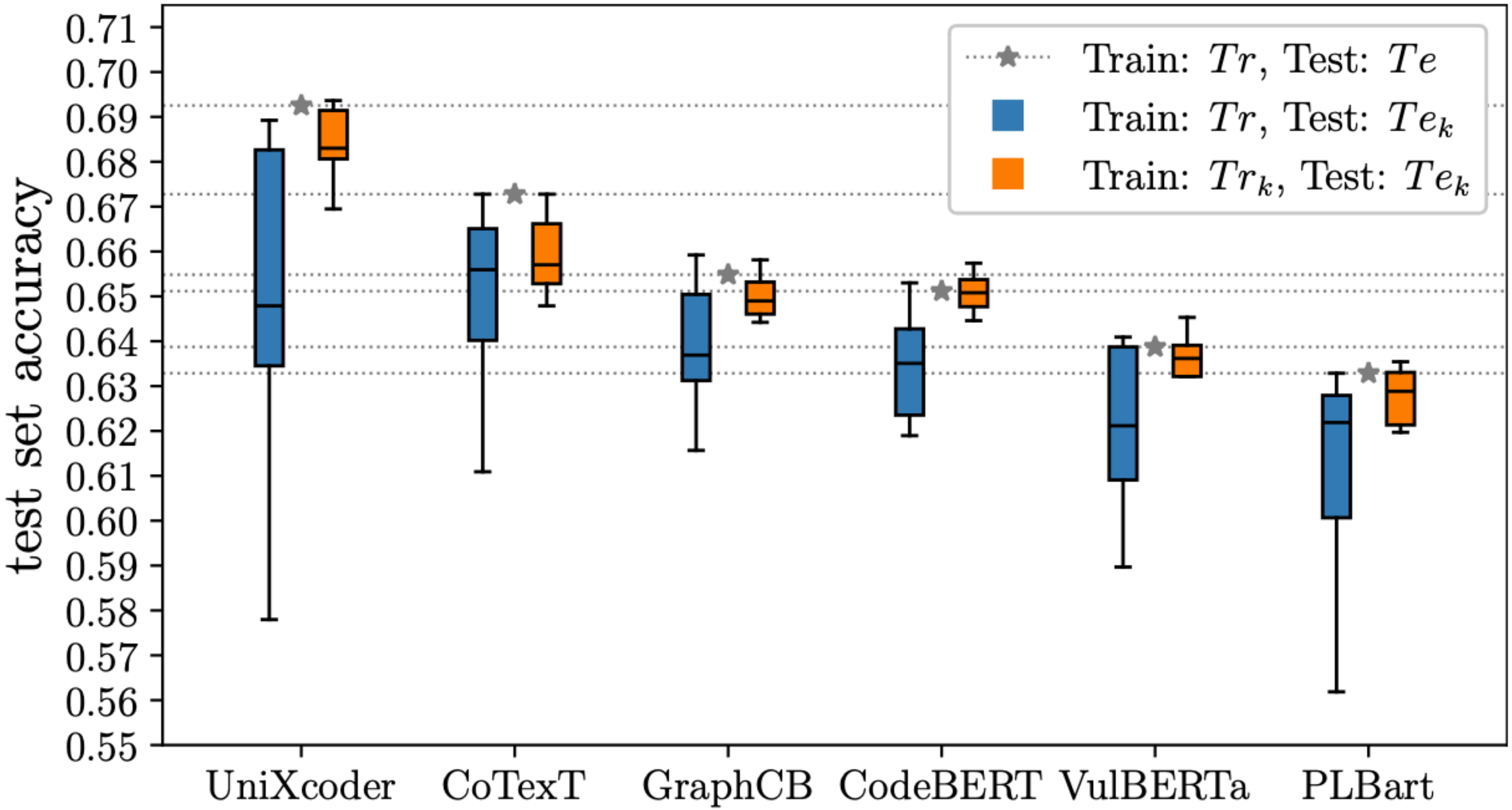
Identifier	Type	Description
t_1	Identifier Renaming	Rename all function parameters to a random token.
t_2	Statement Reordering	Reorder all function parameters.
t_3	Identifier Renaming	Rename the function.
t_4	Statement Insertion	Insert unexecuted code.
t_5	Statement Insertion	Insert comment.
t_6	Statement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

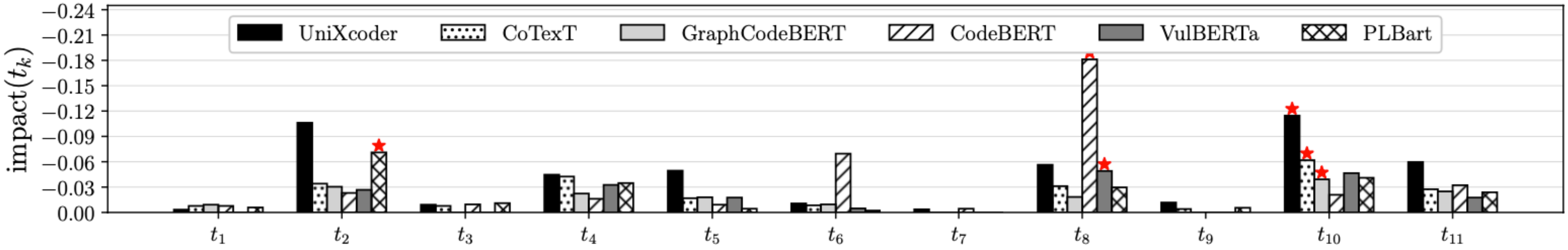
Experimental Design

- 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)

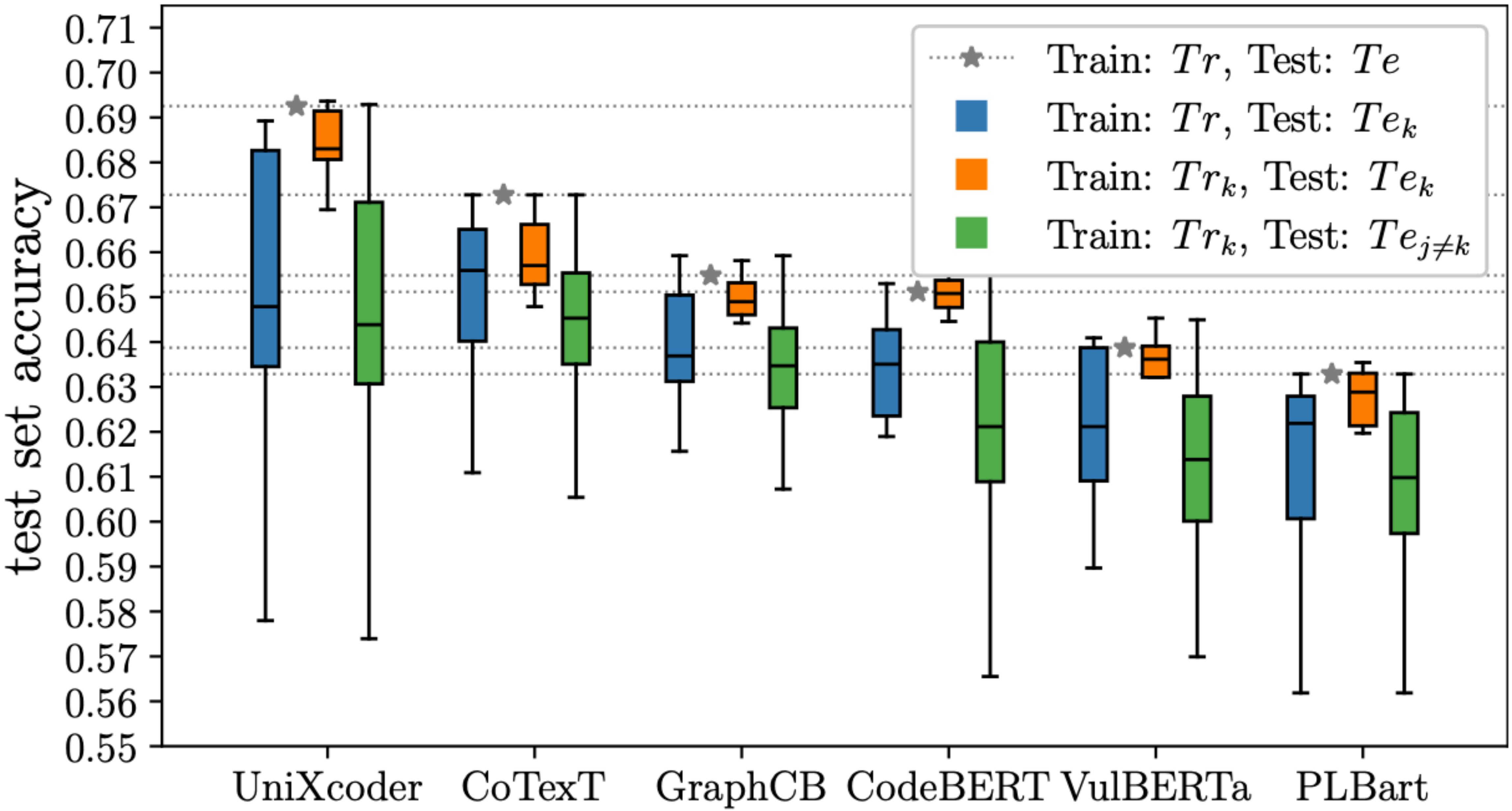
- Applying semantic-preserving transformations to testing data reduces performance (average drop):
    - CodeXGLUE: 2.5% accuracy.
    - VulDeePecker: 4.3% F1-score.
- Augmenting both training and testing data with the same transformations restores most performance:
    - ~69.0% of lost accuracy (CodeXGLUE).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


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
Train: Tr, Test: Te
Train: Tr, Test: Te_k
Train: Tr_k, Test: Te_k
UniXcoder
CoTexT
GraphCB
CodeBERT
VulBERTa
PLBart


impact(t_k)
−0.24
−0.21
−0.18
−0.15
−0.12
−0.09
−0.06
−0.03
0.00
UniXcoder
CoTexT
GraphCodeBERT
CodeBERT
VulBERTa
PLBart
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.



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
test set accuracy
Train: Tr, Test: Te
Train: Tr, Test: Te_k
Train: Tr_k, Test: Te_k
Train: Tr_k, Test: $Te_{j \neq k}$
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.

TO ERR IS MACHINE: Vulnerability Detection Challenges LLM Reasoning, B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, H. Tong, S. Das, E. T. Barr, W. Le, arXiv 2025.

Motivation: Why This Paper Matters

- Large language models excel at code generation and explanation.
- Vulnerability detection requires understanding runtime behavior, memory safety, and missing checks.
- This paper evaluates whether LLM capabilities transfer to this harder reasoning task.

Vulnerability Detection as a Reasoning Problem

- Detecting vulnerabilities is fundamentally different from writing code.
- It requires reasoning about what happens during execution, including failure cases.
- Many vulnerabilities stem from missing validations rather than explicit errors.

Task Definition

- Given a single C or C++ function, the model must classify it as vulnerable or not.
- No execution, symbolic analysis, or additional context is provided.
- The decision must be made purely from the source code.

The SVEN Benchmark

- SVEN contains 772 real-world C/C++ functions.
- Each function has a vulnerable and a patched version.
- The dataset is manually validated to reduce noise.

Models Evaluated

- Fourteen modern large language models are evaluated.
- These include GPT-4-Turbo, Gemini, Code LLaMA, StarCoder, Mixtral, and others.
- All models are evaluated via prompting without fine-tuning.

Model	Can't Distinguish	Distinguished Both Correct	Distinguished Both Wrong
StarChat	86.1%	7.9%	6.1%
DeepSeek	82.5%	6.3%	11.2%
StarCoder	82.1%	12.5%	5.4%
GPT-3.5-turbo	80.9%	11.3%	7.8%
LLAMA 2	76.5%	15.6%	8.0%
MagiCoder	75.2%	11.9%	12.9%
Mixtral	67.8%	18.3%	13.9%
GPT-4-turbo	67.4%	18.9%	13.7%
Gemini	64.4%	19.1%	16.5%
Mistral	61.8%	20.6%	17.6%
StarChat2	61.4%	21.0%	17.6%
StarCoder2	57.5%	19.0%	23.5%
Code LLAMA	57.3%	22.3%	20.4%
WizardCoder	55.0%	23.8%	21.1%
Average	69.7%	16.3%	14.0%

Prompting Strategies

- Different strategies
- Zero-shot classification and few-shot examples are tested.
- Contrastive prompts and chain-of-thought reasoning are also evaluated.
- The goal is to determine whether prompting improves vulnerability reasoning.

Zero-shot Prompting

- Zero-shot prompting is used as the most basic evaluation setting in the paper.
- The model is given the source code of a single C/C++ function and asked directly whether it contains a vulnerability.
- No examples, explanations, or additional hints are provided.
- This setup tests whether vulnerability reasoning is already latent in the model without any external guidance.

Few-shot Prompting

- Few-shot prompting extends the zero-shot setup by providing a small number of labeled examples.
- The prompt includes one or more functions labeled as vulnerable or non-vulnerable before the target function (random or based on embedding).
- The target function is then classified using the same question as in zero-shot prompting.
- This tests whether the model can generalize vulnerability patterns from examples rather than reasoning about semantics.

Contrastive (Paired) Prompting

- Contrastive prompting presents both the vulnerable and the patched versions of the same function in a single prompt.
- The two versions differ only by a small fix, such as adding a NULL or bounds check.
- The model is asked to identify which version is vulnerable or whether either version is safe.
- This setup isolates the semantic effect of the patch and minimizes superficial differences.

CoT from CVE descriptions

- Chain-of-thought prompting instructs the model to explain its reasoning step by step before classification.
- In some prompts, the reasoning is guided by short vulnerability descriptions similar to CVE summaries. The final vulnerability decision is made after the explanation.
- For example, CVE-2017-9211 Corporation (2024) describes the vulnerability, including the symptoms, attack surface, and variable involved

The crypto skcipher init tfm function in crypto/skcipher.c in the Linux kernel through 4.11.2 relies on a setkey function that lacks a key-size check, which allows local users to cause a denial of service (NULL pointer dereference) via a crafted application.

"Therefore, the example is buggy"

- This tests whether encouraging explicit reasoning improves vulnerability understanding.

Static-analysis-guided Prompting

- In the most guided setting, the model is provided with abstract reasoning hints derived from static analysis.
- These hints describe properties such as pointer flows or conditions under which a value becomes NULL.
- This is an example CoT response for a buffer overflow vulnerability:

1. A buffer buf of size 10 is allocated at line 1.
2. An index i is initialized to a value in the range [0, 100] at line 2.
3. The index i is used to access buf at line 3. This may exceed the bounds of buf.

"Therefore, the example is buggy"

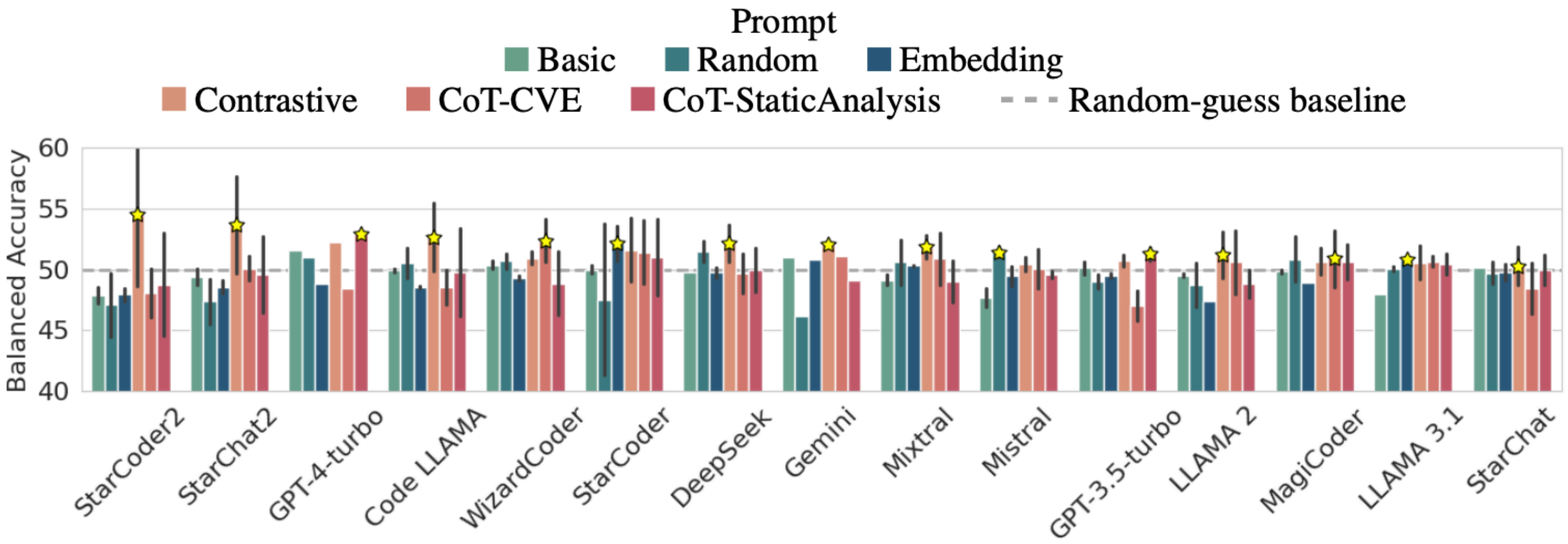
- The model is instructed to use this information when determining whether the code is vulnerable.
- This evaluates whether the model can correctly apply externally supplied semantic reasoning.

Evaluation Metric

- Balanced accuracy is used as the primary metric.
- This metric treats vulnerable and non-vulnerable classes equally.
- A score of 50 percent corresponds to random guessing.

Overall Performance Results

- Across all models, performance remains near random.
- The best models achieve only slightly above 50 percent balanced accuracy.
- Prompting strategies provide only marginal improvements.



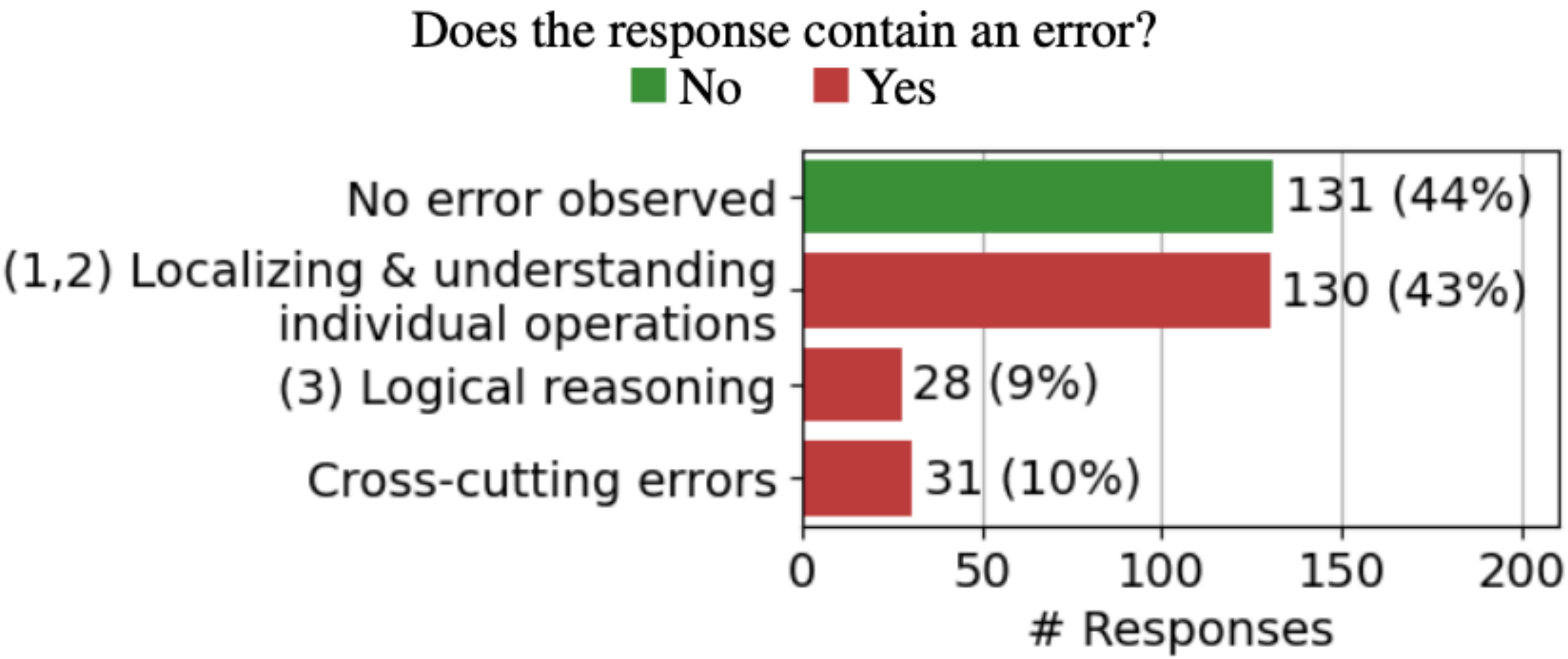
Paired Vulnerable–Patched Evaluation

- Each function appears in both vulnerable and patched form.
- Models should classify these differently if they understand the vulnerability.
- This evaluation directly tests semantic reasoning.

		Distinguished	
Model	**Can't Distinguish**	**Both Correct**	**Both Wrong**
StarChat	86.1%	7.9%	6.1%
DeepSeek	82.5%	6.3%	11.2%
StarCoder	82.1%	12.5%	5.4%
GPT-3.5-turbo	80.9%	11.3%	7.8%
LLAMA 2	76.5%	15.6%	8.0%
MagiCoder	75.2%	11.9%	12.9%
Mixtral	67.8%	18.3%	13.9%
GPT-4-turbo	67.4%	18.9%	13.7%
Gemini	64.4%	19.1%	16.5%
Mistral	61.8%	20.6%	17.6%
StarChat2	61.4%	21.0%	17.6%
StarCoder2	57.5%	19.0%	23.5%
Code LLAMA	57.3%	22.3%	20.4%
WizardCoder	55.0%	23.8%	21.1%
Average	69.7%	16.3%	14.0%

Manual Error Analysis

- The authors manually analyze 300 model predictions.
- They identify recurring patterns of reasoning failure.



Manual Error Analysis

Reasoning step	Error	Count
(1,2) Localizing and understanding statements related to vulnerability	Misunderstood Bounds/NULL check	80/159 (50%)
	Misunderstood string operation	3/29 (10%)
	Misunderstood arithmetic operation	8/96 (8%)
	Misunderstood pointer operation	9/147 (6%)
	Misunderstood alloc/free operation	4/81 (5%)
	Misunderstood index operation	1/60 (2%)
	Misunderstood execution order	11
	Improper assumption	8
	Misunderstood syntax	6
	Total	125
(3) Logical reasoning	Faulty implication ($\Rightarrow$)	14
	Inconsistent ($\perp$)	14
	Total	28
Cross-cutting errors	Hallucination	15
	Memorization	11
	Repetition	5
	Total	31

Missed Bounds/NULL check

Source code (label: non-vulnerable, prediction: vulnerable)

```c
static int changedline (const Proto *p, int oldpc, int newpc) {
 if (p->lineinfo == NULL)   /* no debug information? */
  return 0;
 while (oldpc++ < newpc) {
  if (p->lineinfo[oldpc] != 0)
   return (luaG_getfuncline(p, oldpc - 1) != luaG_getfuncline(p,
↪   newpc));
 }
 return 0;
}
```

Model response (StarChat)

The function dereferences the `p->lineinfo` pointer without checking if it is null. This can cause a crash or other unexpected behavior if the pointer is null. . .

Misunderstood arithmetic operation

Source code (label: non-vulnerable, prediction: vulnerable)

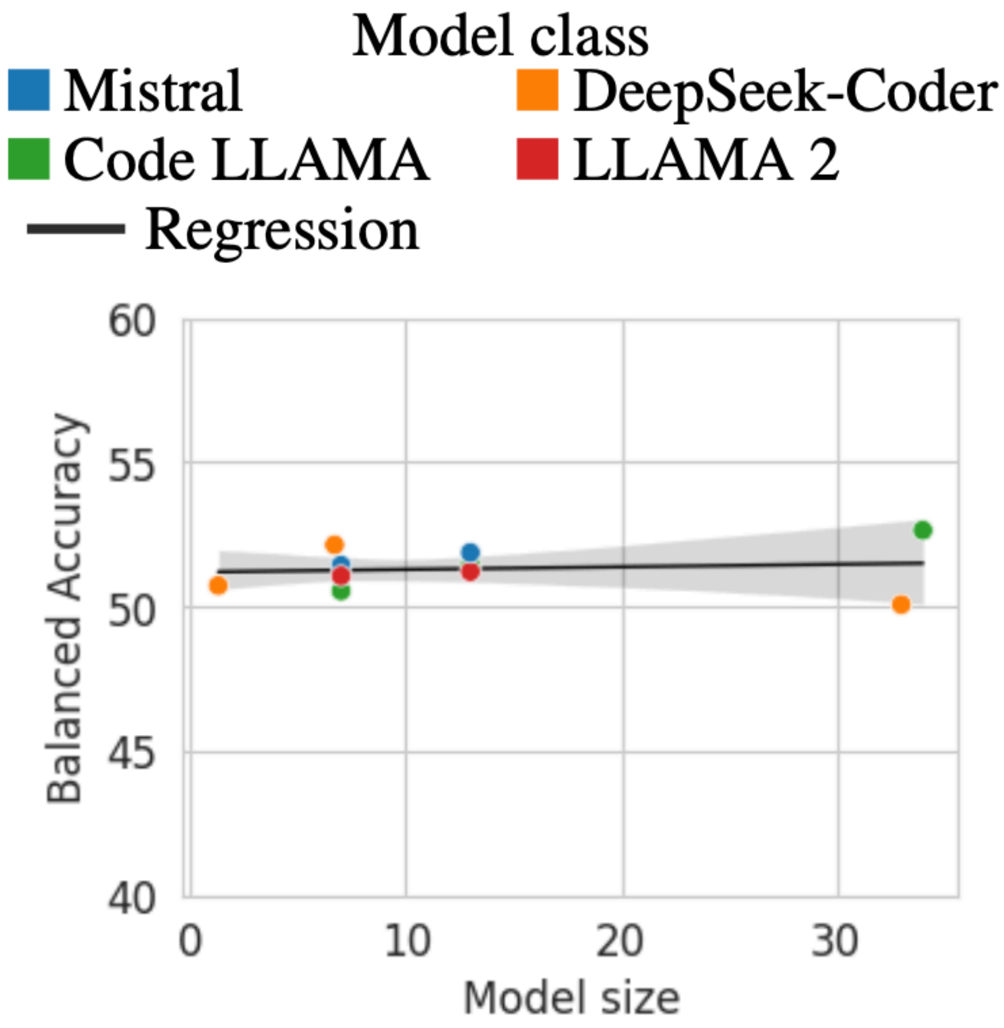
```c
void AllocateDataSet(cmsIT8* it8) {
  TABLE* t = GetTable(it8);
  if (t -> Data) return;  // Already allocated
  t-> nSamples = atoi(cmsIT8GetProperty(it8, "NUMBER_OF_FIELDS"));
  t-> nPatches = atoi(cmsIT8GetProperty(it8, "NUMBER_OF_SETS"));
  if (t -> nSamples < 0 || t->nSamples > 0x7ffe || t->nPatches < 0
↪   || t->nPatches > 0x7ffe)
    SynError(it8, "AllocateDataSet: too much data");
  else {
    t->Data = (char**)AllocChunk(it8, ((cmsUInt32Number)t->nSamples
↪   + 1) * ((cmsUInt32Number)t->nPatches + 1) * sizeof(char*));
    if (t->Data == NULL) {
      SynError(it8, "AllocateDataSet: Unable to allocate data
↪   array");
    }
  }
}
```

Model response (GPT-4)

Both `t->nSamples` and `t->nPatches` are user-controlled and can be large. When these values are incremented by 1 and then multiplied by each other and further multiplied by `sizeof(char*)`, the result can overflow a 32-bit integer if the values are sufficiently large...

Before performing the allocation, the function checks if `nSamples` and `nSamples` are within a certain range (`0` to `0x7ffe`). However, this check does not prevent overflow. For example, values close to the upper limit can still cause the multiplication result to wrap around, leading to a smaller allocation than required.

Does Model Size Matter?



Further Analysis

- Authors also study the effect of data on training models as well as fine-tuning and report minimal accuracy effect.
- They also discuss providing additional domain knowledge to the model with CoT annotations.

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.
- [Steenhoek] TO ERR IS MACHINE: Vulnerability Detection Challenges LLM Reasoning, B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, H. Tong, S. Das, E. T. Barr, W. Le, arXiv 2025.