Broken Promises: Measuring Confounding Effects in Learning-based Vulnerability Discovery

Erik Imgrund
erik.imgrund@sap.com
SAP Security Research
Germany

Tom Ganz
tom.ganz@sap.com
SAP Security Research
Germany

Martin Härterich
martin.haerterich@sap.com
SAP Security Research
Germany

Lukas Pirch
lukas.pirch@tu-berlin.de
Technische Universität Berlin
Germany

Niklas Risse
niklas.risse@mpi-sp.org
Max-Planck Institute
Germany

Konrad Rieck
rieck@tu-berlin.de
Technische Universität Berlin
Germany

ABSTRACT

Several learning-based vulnerability detection methods have been proposed to assist developers during the secure software development life-cycle. In particular, recent learning-based large transformer networks have shown remarkably high performance in various vulnerability detection and localization benchmarks. However, these models have also been shown to have difficulties accurately locating the root cause of flaws and generalizing to out-of-distribution samples. In this work, we investigate this problem and identify spurious correlations as the main obstacle to transferability and generalization, resulting in performance losses of up to 30% for current models. We propose a method to measure the impact of these spurious correlations on learning models and estimate their true, unbiased performance. We present several strategies to counteract the underlying confounding bias, but ultimately our work highlights the limitations of evaluations in the laboratory for complex learning tasks such as vulnerability discovery.

CCS CONCEPTS

• **Security and privacy → Software and application security**;
• **Computing methodologies → Machine learning**.

KEYWORDS

Vulnerability Discovery, Confounding Effect, Overfitting, Causal Learning, Large Language Models

1 INTRODUCTION

The traditional approach to finding vulnerabilities in software relies on manual code review and extensive testing. This approach is time-consuming, resource-intensive, and prone to human error. Static program analysis, on the other hand, supports developers and security professionals in automatically identifying and locating potentially flawed areas without actually running the program. Unfortunately, such static application security testing (SAST) tools often report many false positive alerts, which consequently require expensive manual triage as well.

As a remedy, methods for learning-based vulnerability detection have been proposed to automatically derive rules from historical data to increase the detection rate while at the same time pertaining to a lower false-positive rate [31, 39]. These machine learning (ML) models have been shown to outperform rule-based SAST tools under laboratory settings [15, 65]. There is a vast number of learning-based detection methods available differing mainly in their model architecture, dataset, and the preprocessing techniques they use. For instance, there exist sequence-based solutions from the NLP domain [31, 39], graph neural networks for code analysis [7, 9, 50, 65] and more recently transformer-based models [8, 15, 46].

With large language models (LLMs) being on the rise, transformer networks have been trained on many code-centric tasks, achieving remarkable results on, for instance, code clone detection [61], code completion [33], code generation [45] and code summarization [51]. Naturally, this progress has also inspired LLM-based approaches for vulnerability detection and localization. However, since training an LLM requires a vast amount of data and resources, techniques categorized as one-shot and few-shot learning have been adopted to fine-tune pre-trained models in this setting. Novel advances like adapters [22] or low-rank adaption [23] yield possibilities to optimize pre-trained LLMs for very specific learning tasks. Similarly, it has been shown that the learned representation of pre-trained LLMs beneficially supports the performance on fine-tuned downstream tasks [24].

As a consequence of this development, recent works present an astonishing performance on the discovery and localization of defects in source code using fine-tuned LLMs, beating prior learning-based and rule-based methods by far and achieving more than 90% balanced accuracy under in-lab conditions [15]. A further benefit of these transformer networks is improved vulnerability localization through the interpretation of token attention scores as a measure of code importance [15, 40]. This success of transformer networks, however, is overshadowed by a notable weakness: The models fail to generalize to out-of-distribution samples [8]. That is, high performance is only achievable if the training and test data come from the same software project, which obviously undermines the practical utility of learning-based vulnerability discovery.

In this work, we explore the reason for this deficit and show that transformer networks suffer from spurious correlation, hindering

AISec '23, November 30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0260-0/23/11.
https://doi.org/10.1145/3605764.3623915

generalization and transferability. These correlations create a confounding bias in the learning models, which impacts the detection as well as localization of vulnerabilities. The networks not only fail to identify out-of-distribution data but also hint at irrelevant code when explaining their decisions. To characterize this problem, we propose a methodology to measure the impact of spurious correlations on learning-based vulnerability discovery using the framework of causal inference. To this end, we correlate the loss in performance with semantics-preserving transformations that gradually change the appearance of the code. We find that even minor tweaks in style, control flow, or variable naming are enough to render transformer networks unusable.

We propose three techniques to counter the underlying confounding effects: First, we observe that graph neural networks (GNNs) are less susceptible to artifacts in the dataset and hence offer an alternative architecture for vulnerability discovery. Second, we can mitigate some confounders by pre-tokenizing the input for the LLMs, and finally, we propose to normalize code to a canonical representation before passing it to LLMs. The latter achieves the best overall results. Although we cannot completely eliminate spurious correlations, their impact can be reduced significantly, enabling research to avoid confounding effects.

The rest of this paper is structured as follows: We begin with an introduction to LLMs and graph representations for vulnerability discovery in Section 2. Then, we detail our problem setting and introduce our methodology in Section 3. In Section 4, we present our empirical evaluation and discuss the results in Section 5, ending with the related work section and conclusion in Section 6 and Section 7, respectively.

2 VULNERABILITY DISCOVERY

Let us start by introducing the basic concepts of large language models (LLMs) and graph neural networks (GNNs) for the task of vulnerability discovery, before exploring their limitations and confounding effects.

2.1 Vulnerability Discovery

A vulnerability detection method aims to derive a single score indicating the vulnerability likelihood of a program based on a particular representation of it. This is expressed in Definition 1, which defines a decision function that takes a piece of code and maps it to the probability of it being vulnerable.

Definition 1. *A method for static vulnerability discovery is a decision function* $f_\theta \colon x \mapsto P(\textsc{vulnerable} \mid x)$ *that maps a code sample* x *to its probability of being vulnerable [17].*

Learning-based methods for vulnerability discovery utilize a parameterized classification function f_θ as depicted in Definition 1, whose weights θ are optimized during training on a dataset of vulnerable and non-vulnerable code samples [18]. We denote the classes with prediction probabilities as Vulnerable and Clean, where the former denotes a sample with a code bug present and the latter denotes a sample without bugs.

2.2 Large Language Models

Prior works apply models borrowed from the natural language processing domain to vulnerability discovery [31]. This includes the interpretation of code as a natural sequence of tokens. Models like recurrent neural networks (RNNs) or long short-term memorys (LSTMs) are naturally suited for this task [31, 39, 65]. With the rise of LLMs, which are in essence large transformer models, such networks are increasingly used and fine-tuned for the task of detecting and locating code defects.

Typically, a transformer model consists of either an encoder, a decoder, or both [32]. Each part is composed of multiple blocks consisting of bidirectional multi-head self-attention mechanisms and feed-forward neural networks. Compared to RNNs, transformer models are not limited by the Markov property, where the last hidden state of an RNN needs to store the latent representation of the entire program. Instead, attention matrices produce an attention vector for each token denoting the influence of each other token in the sequence [15]. Since the self-attention mechanism is the heart of LLMs, we define it formally using the original notation by Vaswani et al. [48]:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}V\right) \tag{1}$$

The matrix $Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ denotes a query containing the set of representations for the current tokens, which is then multiplied with the key matrix $K \in \mathbb{R}^{d_{\text{model}} \times d_k}$. The result is scaled by the inverse square root of the embedding size d_k and finally, after a softmax, used as an index to the value matrix $V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ yielding the attention vector. d_{model} denotes the size of the vocabulary and the Q, K and V matrices can be split into multiple attention-heads to capture richer semantics. Recent methods for vulnerability discovery, such as LineVul [15], use pre-trained transformer networks, like CodeT5 [52], BERT [12] or RoBERTa [34], and fine-tune them on vulnerable code.

2.3 Graph Neural Networks

Since programs can be modeled as directed graphs [1, 5, 58], a different strain of research has explored graph representations for source code instead of flat token sequences [9, 50, 65]. We refer to the resulting program representation as a *code graph* and denote the underlying directed graphs as $G = G(V, E)$ with vertices V and edges $E \subseteq V \times V$. Moreover, nodes and edges of the graphs are attributed, that is, elements of V or E are assigned values in a feature space that characterize local properties of the code.

Different code graphs capture different syntactic and semantic features. A popular representation is the code property graph (CPG) by Yamaguchi et al. [58], which is a combination of the abstract syntax tree (AST), the control flow graph (CFG), and the program dependence graph. Other approaches use different combinations, for instance, combining the AST with the CFG and the data flow graph (DFG) [6]. Using such representations, research has started to focus on graph convolutional networks (GCNs) [65]. These networks are a class of deep learning models realizing a function $f \colon G(V, E) \mapsto y \in \mathbb{R}^d$ that can be used for the classification of graph-structured data [40].

GCNs can be viewed as a generalization of convolutional neural networks (CNNs), just as an image can be viewed as a regular grid graph where each pixel denotes a node in the graph connected by edges to its neighboring pixels [57]. A graph convolutional network needs two mandatory input parameters, that is, an initial feature matrix $X \in \mathbb{R}^{N \times F}$, with N being the number of nodes in the graph and F the number of features per node, and the topology commonly described by the adjacency matrix $A \in [0,1]^{N \times N}$. The most popular GCN types belong to so-called message passing networks (MPNs) where the prediction function is computed by iteratively aggregating and updating information from neighboring nodes. One of the simplest MPNs is the one defined by Kipf and Welling [26]:

$$h^{(l)} = \sigma(\hat{A}\, h^{(l-1)} W^{(l-1)}) \tag{2}$$

with $h^0 = X$ [26]. Here, the intermediate representations are linearly projected and sum-wise aggregated according to the normalized adjacency matrix $\hat{A}$ with self-loops followed by a non-linear activation function. These GCNs can be stacked to learn filters w.r.t. larger neighborhoods. Other GCN layers use different aggregation and update mechanisms, for instance, instead of an multilayer perceptron (MLP), gated graph neural networks (GGNNs) use gated recurrent unit (GRU) cells to update the hidden state of nodes [29], while graph attention network (GAT) layers use attention mechanisms [49]. We refer the reader to the overview article by Wu et al. [57] that discusses GNNs in detail.

Because of the fitting premise of GCNs, they have been widely adopted for representation learning on code graphs. The graph-based approaches in recent literature outperform classical SAST tools and older sequential models, such as VulDeepecker [31] or Draper [39]. Graph-based methods like Devign [65] and ReVeal [7] are currently among the best learning-based approaches for vulnerability discovery, though with lower performance than recent methods based on LLMs.

3 METHODOLOGY

We proceed to outline the problem setting and introduce our methodology for measuring the impact of spurious correlations.

3.1 Problem Setting

Currently, many popular learning-based vulnerability detectors exist with varying efficiency. Furthermore, previous works have shown, that although these approaches provide promising results, their ability to precisely pin down the root cause of a bug is lacking. It is questionable, how a security practitioner should respond when a model classifies a function as vulnerable if the model is unable to precisely locate the bug.

Some models come with an integrated explanation mechanism, for instance, LLMs, while others can be enhanced using model-agnostic explanation mechanisms [42], such as Class Activation Maps (CAM) or SHAP [35]. These explanation methods provide a more fine-grained view of the decision of the model and can be used for line-level or even statement-level bug localization. However, these methods provide vastly differing results [54, 66] and a fair comparison is generally non-trivial [3, 16]. It has been shown that vulnerability discovery models tend to focus on irrelevant artifacts in the provided data [16] and that their measured performance may be biased with respect to practice [3, 7].

A motivating example. Let us consider the example in Figure 1. Here, LineVul [15] correctly identifies a bug in the given C function. However, the model falsely claims that the root cause lies in line 2, that is, the instantiation of a matrix on the stack. The actual cause of the vulnerability, however, is a type confusion in lines 3 and 5. The variable `var` is pulled from a hash map and then, without further checks, converted to a double[1]. The matrix plays no role in this vulnerability and thus the explanation misleads a manual investigation of the finding.

```
1 ...
2 float matrix[3][3] = {{0,0,0}, {0,0,0}, {0,0,0}};
3 if (zend_hash_index_find(Z_ARRVAL_PP(var), (j), (void **)
      &var2) == SUCCESS) {
4     SEPARATE_ZVAL(var2);
5     convert_to_double(*var2);
6     matrix[i][j] = (float)Z_DVAL_PP(var2);
7  ...
```

Figure 1: Type confusion bug in the PHP Zend engine.

Interestingly, the tokens with the greatest attention scores from LineVul in line 2 consist of "float", "}}" and "{{". In the training set, these tokens co-occur 198 times for vulnerable functions and only 67 times for non-vulnerable functions, thus creating a spurious correlation. The model incorrectly learns this correlation as an indicator for a vulnerable function. Obviously, there must be more biases present in the training dataset such as the one identified here, which makes the model concentrate on irrelevant artifacts.

The problem becomes worse when we try to slightly change the coding style and obfuscate some lines as seen in Figure 2. Although the function is semantically equivalent and only minimally changed, LineVul now classifies this function as clean, despite the original vulnerability being still present.

```
1 ...
2 float matrix[3 & 0xF][3 & 0xF] = {
3   {0o0, 0o0, 0o0}, {0o0, 0o0, 0o0}, {0o0, 0o0, 0o0}};
4 if (zend_hash_index_find(Z_ARRVAL_PP(var), (j), (void **)&
      var2) ==
5     SUCCESS) {
6   SEPARATE_ZVAL(var2);
7   convert_to_double(*var2);
8   matrix[i][j] = (float)Z_DVAL_PP(var2);
9  ...
```

Figure 2: Obfuscated version of type confusion bug.

To visualize the effect of spurious correlation, we present Figure 3 which derives a simple causal model for a vulnerability discovery function f_θ [44]. Here, X is the input data and Y is the label, being either VULNERABLE or CLEAN. The learning goal of f_θ is to find a relationship between the learned representation R to the label Y. There is a relationship $C \leftarrow X \rightarrow A$ denoting that the causal features C and trivial or biased feature patterns A both influence the final latent representation R. Missing or different artifacts in unseen data then weaken the model performance.

[1] https://cve.report/CVE-2014-2020

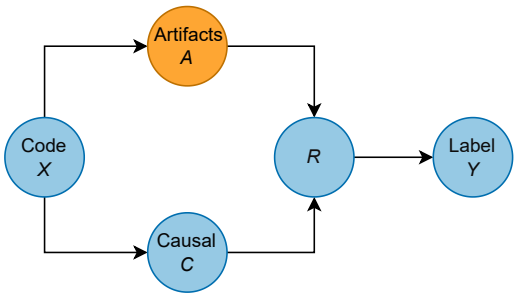


Figure 3: Simple causal model of machine learning for vulnerability discovery.

The artifacts are called *confounders* in causal learning and in our example, the confounders originate from irrelevant features that cause the model to learn biased representations. A simple example for an artifact would be a common code style in all vulnerable samples and a different style for all clean samples. The ultimate goal in this work is not only to measure f and the actual influence of A but also to remediate this influence since these learned artifacts hinder the model from generalizing well over unseen or out-of-distribution data, that is, real-world code.

To this end, we identify three sources of bias that can manifest as artifacts in program code:

(1) *Coding style.* Every collected sample has an implicit coding style. Since many open-source projects use automatic linting, it is likely that samples from one project to another differ in their styles. If there are more vulnerable samples from one project than another, the coding style correlates with Y.

(2) *Control flow.* Projects often contain different calling hierarchies or indirections due to programming patterns, for instance, object-oriented design principles. Several projects and authors prefer one pattern over another which may introduce further confounding bias.

(3) *Naming.* Different samples from different projects naturally vary in their naming conventions. Hence, vulnerable samples may potentially differ in variable naming compared to clean samples. Although it is common to mask such symbols, recent works unfortunately desist from normalizing them.

This list is non-exhaustive since, potentially, there can be an infinitely large number of artifacts. Nonetheless, we state that a model is confounded if artifacts heavily influence its decision. Since other works do not account for this, we propose a more reliable evaluation methodology.

3.2 Evaluating Models

Current models for learning-based vulnerability discovery suffer from low transferability and generalizability. Yet, they pertain to a high true positive and true negative rate on test sets aligning with the training distribution [7, 8]. How a security expert can gain insight into how well the model performs in practice is an open research question. The performance can not be truly measured on the test set, as it is of the same distribution as the training set and might lack diversity compared to real-world code samples.

Training the model on one dataset and testing on another is a possible evaluation approach; however, it remains unclear how many datasets one has to evaluate. Worse still, vulnerability datasets are scarcely available [7]. Using more datasets improves the insights gained, but increases the amount of data and computing resources necessary for the evaluation. As a consequence of this situation, we propose a method that uses only one dataset.

To motivate our evaluation scheme, we briefly introduce the concept of causal inference that reveals influencing variables. If we inspect Figure 3, it is trivial to see that the representation learned by a model f directly influences the predicted label Y. But instead of using the sample under analysis to directly influence the representation, we model it so that X has a causal and a trivial part [44]. We call the latter the confounding variable, shortcut feature, or spurious correlation [44]. As a result, we have three relationships: $A \leftarrow X \rightarrow C$, $C \rightarrow R \leftarrow A$, and $R \rightarrow Y$. To measure the true causal correlation and to remove confounding variables in causal learning, it is common to calculate the influence of one variable affecting the other by *intervention*.

This can be done using do-calculus [36], that is, we can stratify the confounder by calculating the influence of $C \rightarrow Y$ given all possible artifacts from $a \in A$ [36]:

$$P(Y|do(C)) = \sum_{a \in A} P(Y|C, A = a)P(a) \tag{3}$$

We approximate the distribution of A by calculating the estimated likelihood of the code samples $X = (x_0, \ldots, x_n)$ with our different perturbations. As using all possible artifacts is not feasible, we use a subset $A' \subset A$ and define an artifact $a \in A'$ to be a variant of the code samples $k_a(X) = (k_a(x_0), \ldots, k_a(x)_n)$ obtained by one of our perturbations $a \in A'$. Equation 3 then becomes:

$$P(Y|do(C)) \approx \sum_{a \in A'} P(Y|C, A = a)\hat{P}(a). \tag{4}$$

We estimate $\hat{P}(a) = \frac{P_\theta(a)}{\sum_{a' \in A'} P_\theta(a')} \approx P(a)$ empirically by calculating the likelihood of a particular variant of the code sample $P_\theta(a)$ utilizing a generative LLM with weights θ. We calculate the likelihood of each token of the code sample dependent on the previous tokens. Since calculating the likelihood of the entire sequence by multiplying the individual token likelihoods is numerically infeasible, we instead average the log-likelihoods over the entire sequence to obtain the approximate likelihood, similar to the calculation of the perplexity, a popular metric for generative LLMs [37].

Further, we can measure the impact of the artifacts on the model by calculating the average relative difference between the original model decision compared to the decision when every artifact is marginalized. We call this difference the *confounding effect*,

$$c = \frac{\sum_{a \in A'} P(Y|C, A = a)\hat{P}(a) - P(Y|C, A)}{P(Y|C, A)} \tag{5}$$

As an intuition, consider $c = 0$, meaning that $P(Y|C, A) = P(Y|do(C))$. However, the more c deviates from 0, the greater the influence of the artifacts and $P(Y|C, A) \neq P(Y|do(C))$.

The application of different perturbations to the code should resemble a causal intervention. A non-confounded model should perform equally on semantically equivalent but perturbed code since the decision should solely depend on the causal feature part.

Let us define the model predictions on the code samples under different perturbations as $\forall a \in A' : f(k_a(x))$. Consequently, this intervention provides insight into how the model behaves under different artifacts and yields a more robust basis for model evaluation and comparison. In a more practical sense, suppose that we have a metric $\mathcal{M}: f \to \mathbb{R}$ assessing the quality of a learning-based vulnerability discovery model, such as the accuracy. We then have

$$c = \frac{\sum_{a \in A'} \mathcal{M}(f(k_a(x))\hat{P}(a) - \mathcal{M}(f(x))}{\mathcal{M}(f(x))} \tag{6}$$

and can measure the influence of the confounder using the *confounding effect* as defined above.

3.3 Reducing Confounder

The measurement of the confounding effect of artifacts on the model is one side of the coin, but on the other side, we also want to reduce the influence of such trivial patterns on the model. We propose three methods that can be applied to either LLMs or GNNs for vulnerability discovery that mitigate the effects in practice.

LLMs with normalized code. To remove the effect of style artifacts on LLMs, one naive solution is to normalize the code. Code normalization is the modification of code so that it conforms to a given style guide, which reduces, but does not remove, the impact of the personal style on the code [21]. As a code normalization method, we apply one code style to all methods and use the uniformly formatted code as training data. This has the same effect of normalizing the coding style, and thus removing style artifacts while preserving closeness to real-world code and thus making better use of the pretraining than the next method. We abstain from normalizing the variable naming by masking the variable names during training so that the effect of the naming artifacts can be measured as part of our evaluation.

LLMs with pre-tokenized code. Another solution follows from the work of Roziere et al. [38], who propose to tokenize the code before applying the byte-pair encoding using a programming language-specific lexer. They feed the resulting tokens as space-separated plain text into the model, a process we refer to as *pre-tokenization*. We adopt the same methodology, but as our models were trained with untokenized code, we expect a performance drop from the different data distributions obtained, as the code samples now lack all newlines and other style practices common to real-world code. Retraining the LLM completely on tokenized code is impractical since the main benefit of LLMs is the adaptability to several tasks using fine-tuning.

Causal graph learning. A more principled approach arises from the work of Sui et al. [44] in the domain of GNNs: By applying an intervention directly to the learning model, they can mitigate the impact of confounding variables. This is done by conditioning the causal input features, in this case, a code graph, per sample with all possible trivial subgraphs obtained during training.

To encode the input graph, a graph isomorphism network (GIN) layer is used followed by two MLPs to calculate a relevance score for nodes and edges. For any node $v_i \in V$ we calculate their node attention and for any pair of nodes $(v_i, v_j) \in E$ their edge attention. Furthermore, the output dimension of the MLPs is halved to perform a latent space disentanglement, where the first half will later be optimized to contain only the relevant nodes that are causal for our task and the second half will be trained to only contain the trivial part of the graph, in this case, the artifacts.

A mean readout layer is applied last as a pooling strategy followed by a final MLP with softmax activation as the prediction head returning either VULNERABLE or CLEAN. Using the attention scores we can calculate attention masks for both, the causal and trivial nodes, and causal and trivial edges. We apply these masks to the adjacency matrix and feature matrix of the input graph resulting in the causal and trivial subgraphs respectively. The causal graph can be used to explain the prediction and track the cause of a vulnerability.

To train the model in a supervised fashion, we first apply a traditional negative log-likelihood (NLL) loss to our ground truth and the latent representation of the causal graph. Then, we take the representation of the trivial subgraph and optimize the model to separate trivial and causal features by fitting the softmax distribution using the trivial graph to a uniform distribution using the Kullback-Leibler divergence (KL). Finally, to stratify the confounder, another NLL loss is calculated between the ground truth and the prediction, while the causal graph is augmented with a random trivial subgraph from another graph in the dataset. During the training procedure, the model essentially learns to ignore trivial patterns.

4 EVALUATION

In this section, we describe the experimental setup and the results of our evaluation. The experiments are devised to give answers to the following research questions. We publish our code for easy experimental reproduction [2].

- RQ1: Are confounding effects measurable?
- RQ2: How do artifacts influence vulnerability localization?
- RQ3: Can the confounding effect be reduced?

4.1 Experimental Setup

We rely on *Fraunhofer-CPG* by Weiss and Banse [55] and *networkx* [19] as tools to generate code graphs. The graph-based models are implemented using *Pytorch Geometric* [14] and trained on AWS EC2 g4dn instances. We use the transformers library [56] to fine-tune the transformer-based models. The tokenization of code is calculated using *tree-sitter* as the parser. The hyperparameters of all models are documented in Table 1.

[2] https://github.com/SAP-samples/security-research-confoundingeffects

Table 1: Hyperparameters used for considered models.

	REVEAL	StackLSTM	CGIN	CodeT5+	LineVul
Optimizer	Adam			AdamW	
Learning Rate	$5 \cdot 10^{-4}$	10^{-4}	10^{-4}	10^{-5}	10^{-5}
Epochs	70	50	14	3	10
Batch Size	128	1	2	8	8
Warmup Steps	0	0	0	50	50
Weight Decay	10^{-4}	0	0	0.01	0.01
Number of parameters	719k	1.7M	222k	223M	249M

Table 2: Function-level accuracies with different augmentations. CodeT5+$_n$ and LineVul$_n$ are trained on normalized data and CodeT5+$_t$ is trained on tokenized data.

Transformation	ReVeal	CGIN	StackLSTM	CodeT5+	CodeT5+$_n$	CodeT5+$_t$	LineVul	LineVul$_n$	LineVul$_t$
None	63.57%	61.55%	62.79%	**94.62%**	83.15%	59.51%	*93.09%*	83.62%	65.21%
Chromium	63.62%	62.24%	62.79%	59.21%	*66.46%*	59.56%	58.76%	**68.68%**	65.25%
Mozilla	63.59%	61.39%	62.84%	59.18%	*66.54%*	59.56%	57.67%	**67.63%**	65.25%
Google	63.53%	63.42%	62.79%	59.16%	*66.20%*	59.56%	58.95%	**68.39%**	65.25%
LLVM	63.43%	60.37%	62.84%	59.01%	*66.54%*	59.56%	58.73%	**68.22%**	65.25%
Uglify	**62.29%**	58.02%	61.04%	50.67%	55.62%	55.32%	50.44%	55.77%	60.30%
Uglify (-Whitespace)	**62.86%**	58.47%	*61.04%*	54.15%	57.37%	55.35%	49.51%	55.88%	61.02%
Obfuscate	50.51%	59.41%	50.74%	53.61%	59.41%	*60.73%*	52.11%	57.29%	**60.74%**
Obfuscate (+Format)	50.46%	56.12%	50.71%	37.14%	58.13%	*61.19%*	40.56%	53.60%	**61.44%**
Obfuscate (-Whitespace)	49.76%	55.28%	53.12%	52.05%	*55.79%*	55.50%	51.02%	53.68%	**60.92%**
Causal Accuracy	64.18%	61.59%	64.75%	59.09%	*65.41%*	59.80%	58.71%	**67.46%**	65.02%

Dataset. We use *Big-Vul* [13] as the underlying dataset for all of our experiments, as it is one of the biggest currently available datasets with line-level defect information. The dataset consists of 26 635 vulnerable and 352 606 non-vulnerable functions from different code repositories.

Transformations. We apply transformations on the dataset that implicitly remove artifacts to obtain different variants of the dataset for training and evaluation. These transformations can be categorized into three classes:

(1) *Styling.* We apply style formatting using clang-format [21] with different popular predefined styles. We test the predefined styles *Chromium*, *Google*, *LLVM*, and *Mozilla* for a diverse range of style choices. As previously described, applying the formatting removes style-related artifacts.

(2) *Uglification.* We test two different kinds of "uglification". The first variant consists of removing comments and renaming all variables to a string of twelve randomly chosen lowercase letters and applying style normalization. The second variant is the same except for additionally removing all unneeded whitespace. This transformation removes artifacts in code style and naming whilst also partially removing causal information, as the variable names are typically chosen deliberately and essential to detect vulnerabilities.

(3) *Obfuscation.* The obfuscation consists of randomly renaming variables and functions, removing comments, adding unneeded statements, adding function definitions, and replacing numbers with an obfuscated equivalent number. The numbers are obtained by converting them to decimal, binary, octal, or hexadecimal. Additionally, we evaluate the models on the obfuscated code after applying a predefined coding style and after removing all unneeded whitespace. The obfuscation also removes control-flow artifacts, since statements are randomly inserted.

Models. We have chosen a diverse set of models for evaluation, both graph-based and text-based. For graph-based models, we train and evaluate ReVeal as a state-of-the-art GNN for vulnerability detection [7] and causal graph isomorphism network (CGIN) [44] as a causal graph model that mitigates the effect of artifacts. For text-based models, we choose LineVul [15] and fine-tune a CodeT5+ [52] model with 220 million parameters similar to Chen et al. [8] and Thapa et al. [46]. We fine-tune the transformer models on the original code available as part of the dataset. Additionally, we train the models on normalized and tokenized code.

Delétang et al. [11] show that transformer models cannot generalize well over different-sized input token lengths. The authors show that classical LSTMs with differentiable memory provide stronger generalization performance than transformer models on increasingly complex tasks. Since the number of tokens within samples can impose another bias, we extend VulDeepecker [31], a LSTM-based vulnerability discovery model, with a differentiable stack [27]. VulDeePecker is generally inferior to the other models [7]. However, an LSTM with access to a stack has been shown to provide advantageous results for regular and context-free tasks, similar to the capabilities of a real-world parser [11].

Evaluation tasks and metrics. The models are evaluated based on two tasks: function-level binary classification with the classes vulnerable and clean and line-level classification of known-vulnerable functions. We use the balanced accuracy, as the mean between the true positive and true negative rate, for both tasks, due to the heavily imbalanced datasets. Additionally, we measure the causal accuracy as the balanced accuracy based on causal predictions according to Equation 4. Further, we use the balanced accuracy as metric $\mathcal{M}$ for measuring the confounding effect from Equation (6). For the line-level task, we use the top-1, top-3, and top-5 accuracy as introduced by Fu and Tantithamthavorn [15].

To obtain a ranking of the lines by each model, we use model-specific explainability methods. For the graph-based models we obtain node relevance scores and then propagate these scores to all lines included in the node to arrive at a line relevance [16]. We obtain node relevance scores for ReVeal by applying Grad-CAM [41] and for CGIN by utilizing the causal node attention scores. For the transformer-based models, we calculate the relevance of each line in the same way as proposed by Fu and Tantithamthavorn [15]. The relevance of each token in the line is summed to obtain the aggregate line relevance. The token relevance is similarly obtained as the attention to each token in the first layer of the encoder.

As our transformations can change the layout of the code and thus the locations of the vulnerable lines, we need to match the original lines to the transformed lines. As the matching is non-trivial and harder with additional transformations, we only test on the original dataset and the styled variants. We match the vulnerable lines to the formatted lines if one of the lines is an exact substring of the other line without considering whitespace. As a baseline, we also measure the expected top-k accuracy with random line orderings, which can be calculated based on the expected best rank of any vulnerable line, which is distributed according to a particular *negative hypergeometric* distribution. More formally, the expected rank is $E(X)$ for $X \sim NHG(|L|, |L_0|, 1)$ with the set of all lines L and the set of non-vulnerable lines L_0. By calculating the expected rank of each code sample in our test set, we can obtain the expected top-k accuracy for a random baseline.

4.2 Results

In this section, we provide our experimental results and use the outcomes to provide answers to our research questions.

RQ1: Are confounding effects measurable? In Table 2, we present our results for the function-level prediction scores for all eight models measured in their balanced accuracy. The best performance per transformation is bold and the second best value is italic. CodeT5+ and LineVul have an initial balanced accuracy of 94.62% and 93.09% on the test set, respectively. That is an approximate increase of 50% relative to ReVeal, StackLSTM, and CGIN. Interestingly, the detection performance of the transformer models shrinks to a detection rate lower than that of the other models when the test samples are transformed using different styles. With less than 60% balanced accuracy CodeT5+ and LineVul are performing worse than the non-transformer-based models having about 63% balanced accuracy on average. This is a clear hint that the transformer models overfit on artifacts in the train and test distribution aligning with the coding style. The discrepancy between the performances with and without augmentations for the other models is negligible, with CGIN having the overall worst results with approximately 61%, followed by StackLSTM with 62% and ReVeal with 63%.

The situation becomes worse when comparing the performances of the transformer models when provided with uglified code, that is, inlining functions, and removing whitespace and tabs. While the other models pertain to a comparable performance at around 60%, the performance of LineVul and CodeT5+ is not different from random guessing. The uglifier without the removal of whitespace leaves CodeT5+ with 54%, while LineVul is still comparable to a random guesser. Using an obfuscator is the most drastic augmentation transformation, as it even changes control flow and adds superfluous function calls. All models, except CGIN, are not significantly better than random guessing in this scenario.

Furthermore, when comparing the accuracy obtained based on causal-only features, it is obvious that the initial performance of CodeT5+ and LineVul is based on artifacts instead of causal features with only 59.80% and 58.71% causal accuracy, respectively. The graph-based models ReVeal and CGIN as well as StackLSTM on the other hand, obtain a causal accuracy near their initial performance, indicating that their predictive performance is based on causal features instead of artifacts.

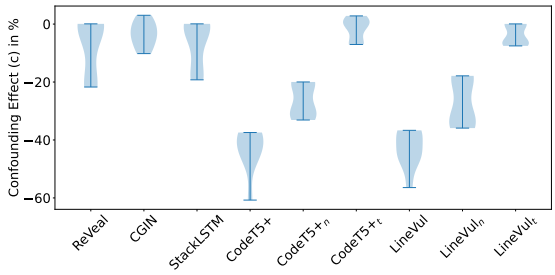


Figure 4: Confounding effect on function-level accuracy.

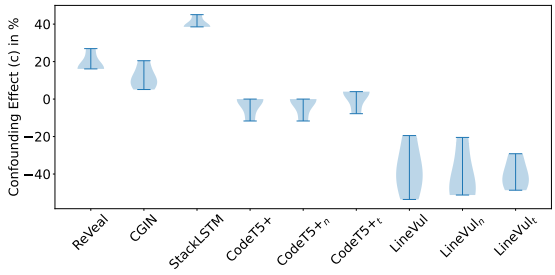


Figure 5: Confounding effect on Top-5 line-level accuracy.

We show that there are indeed artifacts encoded in the training set that negatively influence the models when transferring to semantically equivalent code. If the model already fails in predicting vulnerabilities in the test set with the same distribution as the training set but with minor changes, it is highly uncertain how it behaves on real data. The reported performance scores of the models are not suitable for a veracious comparison. While the initial highest performance is 94.62%, the true models' capabilities collectively level off at around 60%. In summary, the influence of the confounding artifacts distorts the reported performance by up to 60% measured by the discrepancy between the test and augmented test set. Figure 4 summarizes the confounding effect on different models visualizing the degrees of performance drop. The transformers are in fact the models most affected in this experiment.

RQ2: How do artifacts influence vulnerability localization? Table 3, Table 4 and Table 5 show the top-1, top-3 and top-5 line-level accuracy, respectively. The best scores per transformation are highlighted in bold. The first column denotes the performance of the random baseline. This is included as a reference for the performance of the models, as well as to show the limitations of our measurement method. Due to the fuzzy matching of formatted lines to the original lines, the number of total lines varies in the formatted code, and consequently, the expected performance changes slightly.

Although the performance of the transformer models for function-level prediction is greatly influenced by the application of code formatting, only much smaller differences for line-level localization can be observed. For LineVul, the top-1 accuracy drops from 40.33% to at most 38.67% while for CodeT5+ the performance increases from 38.67% up to 44.75%. For the top-3 accuracy, the same effect

Table 3: The Top-1 line-level Accuracy of the Transformer and Graph models with different code styles.

Codestyle	Random	ReVeal	CGIN	StackLSTM	CodeT5+	CodeT5+$_n$	CodeT5+$_t$	LineVul	LineVul$_n$	LineVul$_t$
None	26.86%	43.09%	**43.65%**	39.23%	38.67%	38.67%	38.67%	40.33%	39.78%	41.44%
Chromium	26.86%	44.20%	43.65%	**45.86%**	44.75%	44.75%	44.75%	37.02%	37.02%	37.02%
Mozilla	25.71%	43.65%	42.54%	**44.20%**	40.88%	40.88%	40.88%	38.67%	37.57%	39.23%
Google	28.57%	44.75%	44.20%	**45.30%**	44.75%	44.75%	44.75%	38.67%	37.57%	37.57%
LLVM	28.57%	44.20%	44.20%	**45.30%**	43.65%	43.65%	43.65%	35.91%	34.25%	34.81%

Table 4: The Top-3 line-level Accuracy of the Transformer and Graph models with different code styles.

Codestyle	Random	ReVeal	CGIN	StackLSTM	CodeT5+	CodeT5+$_n$	CodeT5+$_t$	LineVul	LineVul$_n$	LineVul$_t$
None	46.86%	59.67%	58.01%	58.56%	**64.64%**	**64.64%**	**64.64%**	62.43%	63.54%	64.09%
Chromium	46.86%	60.77%	57.46%	60.77%	**64.64%**	64.09%	63.54%	56.91%	55.25%	56.35%
Mozilla	45.14%	61.88%	58.56%	60.77%	**64.09%**	63.54%	63.54%	61.88%	60.77%	60.22%
Google	47.43%	61.33%	60.77%	61.88%	**64.64%**	64.09%	64.09%	55.25%	55.25%	55.25%
LLVM	46.86%	61.33%	58.56%	61.88%	**64.64%**	64.09%	64.09%	56.91%	56.35%	55.80%

Table 5: The Top-5 line-level Accuracy of the Transformer and Graph models with different code styles

Codestyle	Random	ReVeal	CGIN	StackLSTM	CodeT5+	CodeT5+$_n$	CodeT5+$_t$	LineVul	LineVul$_n$	LineVul$_t$
None	57.71%	67.96%	68.51%	66.30%	**71.82%**	**71.82%**	**71.82%**	69.06%	68.51%	69.06%
Chromium	57.71%	69.61%	69.06%	69.61%	71.82%	71.82%	**72.38%**	64.64%	64.09%	64.64%
Mozilla	57.71%	**70.72%**	69.61%	70.17%	70.17%	70.17%	**70.72%**	66.85%	66.30%	65.75%
Google	57.71%	69.61%	70.72%	69.61%	71.82%	71.82%	**72.38%**	62.98%	62.98%	63.54%
LLVM	57.71%	69.61%	69.61%	69.61%	71.82%	71.82%	**72.38%**	64.64%	62.98%	64.09%

can be seen for LineVul with a drop from 62.93% to 55.25% and at most 61.88%, while CodeT5+ does not induce any big differences. The same can be seen in the top-5 accuracy. The graph-based models CGIN and ReVeal yield smaller differences in their performance and in general improve when a code style is applied. Their results closely follow the expected performance and the variations in performance can be explained by changes in the total number of lines and the matched number of flawed lines.

All models show better than random performance, even when applying the code formatting augmentation, implying they are not relying on styling artifacts for line-level localization. For the top-1 accuracy, the graph-based models and CodeT5+ are competitive and within reach of each other depending on the code style used with no clear best approach. The performance of LineVul on the other hand is worse than all the others, which is also seen for the top-5 accuracy and with mixed results for the top-3 accuracy. For the latter, ReVeal and CGIN show similar performance with ReVeal being better in all cases, while CodeT5+ is better than all other models. CodeT5+ is also better than the other models in all cases but one when considering its top-5 accuracy.

In summary, we see only a slight effect of artifacts on vulnerability localization for CodeT5+ and the graph-based models. LineVul is definitely affected by the removal of artifacts from the data and shows an overall weaker performance. Models that are less affected by the application of code formatting show an overall greater performance for vulnerability localization, indicating that generalizing code formatting changes are helpful to vulnerability localization. It is interesting that CodeT5+ does not see a performance drop in vulnerability localization, even though it was present in function-level prediction, indicating that the model is attending to the correct parts of the code but drawing wrong conclusions. Moreover, the performance difference between LineVul and CodeT5+ cannot be satisfactorily explained, as both models are trained similarly with the only difference being base architecture, indicating that the vulnerability localization performance of transformer models trained with artifacts is unpredictable. The graph-based models on the other hand are less affected in general and more predictable. Figure 5 visualizes the confounding effect calculated as the relative change of top-5 line-level accuracy normalized by subtracting the random baseline. All models suffer under the confounding effect, while ReVeal and CGIN suffer the least.

RQ3: Can the confounding effect be reduced? Considering Table 2 again, we can see that the adjustment of the training procedure for the transformer models significantly reduces the discrepancy between the test and augmented test performance. LineVul trained on normalized code achieves the best results on different styled code of around 68%, followed by CodeT5+ trained in normalized code with around 66%, beating the balanced accuracy scores from ReVeal, StackLSTM and CGIN by $\sim$ 3%. As opposed to the technique by Roziere et al. [38], pre-tokenization of the input code may decrease the artifact overfitting effect but its performance is still inferior to the other non-transformer-based models.

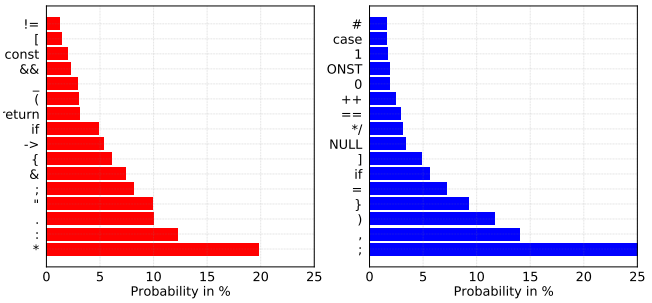
Tokenization and normalization for the LLMs also helped to reduce the bias measured by the *Uglify* transformation. However, ReVeal, StackLSTM, and CGIN outperform all transformer models. Surprisingly, reducing the confounder effect on the transformers helps to improve the performance on obfuscated code changes, slightly outperforming CGIN with up to 7%.

Moreover, tokenization and normalization also boost the causal accuracy of LineVul significantly. With a causal accuracy of 67.46%, an increase of nearly 15% from the unnormalized model, LineVul$_n$ performs best in our testing, beating even the graph-based models. CodeT5+$_n$ performs second-best with a causal accuracy of 65.41%, but CodeT5+$_t$ only achieves 59.80%. We attribute this only slight increase in performance of the tokenized model to the data distribution shift of tokenized data, which looks significantly different from the real-world data, that the model was pre-trained on, requiring further training. For LineVul$_t$ and CodeT5+$_t$ the causal accuracy is equal to the accuracy of the raw data, indicating that the artifacts initially present in the raw data are removed by tokenization.

It is also surprising that StackLSTM beats the transformer models and CGIN on uglified code. We conjecture that the StackLSTM learned to parse code despite syntactical differences. Considering Figure 6, we can see that the model learned to push opening brackets to the stack and pop closing ones from the stack if encountering them. Interestingly, StackLSTM learned to mimic a parser using the vulnerability discovery dataset.

We have shown that we can reduce the confounding influence of artifacts in the dataset on the detection models. GNNs are less influenced by code obfuscations than by control flow distortions as opposed to transformer models for which we observe the opposite effect. LSTMs are also less susceptible to style changes and CGIN is a viable approach to reduce the confounding influence. Transformer models and specifically LLMs are severely influenced by slight code transformations, however, we can mitigate this by rather normalizing the input code than tokenizing it.

In the end, LLMs prove their superiority to LSTMs and GNNs when correctly trained. Recall Figure 4, CGIN and CodeT5+ trained on pre-tokenized code reduced the confounding effect the most. While training on normalized code is also a viable strategy, in general, inferior to GNNs. The confounding effect on LineVul is



(a) Probability for pushing a token to the stack. **(b) Probability for popping a token from the stack.**

Figure 6: The learned stack policy for StackLSTM.

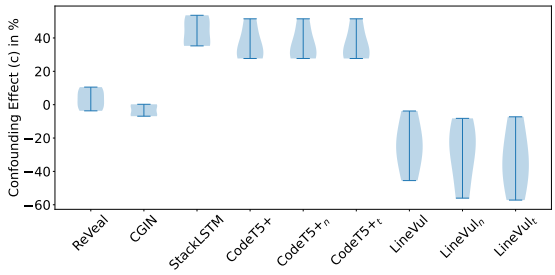


Figure 7: Confounding effect on Top-1 line-level accuracy.

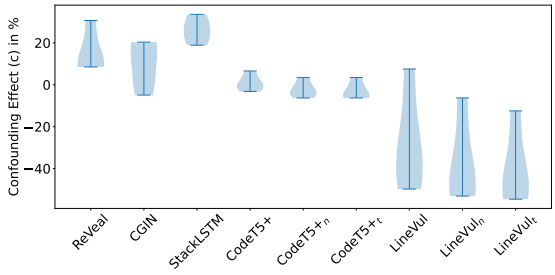


Figure 8: Confounding effect on Top-3 line-level accuracy.

largest on Top-3 line-level accuracy, considering Figure 8, while some models, especially CodeT5+, even benefit from the artifact removal as depicted by Figure 7 where they have a positive change of up to 50% detection improvement.

Interestingly, the shape and values of the confounding effect on the transformer models trained on the original data are nearly the same as those of the corresponding models trained on normalized and pre-tokenized data. As only the encoder input attention is used for generating line-level localization as per LineVul's method, this indicates that the encoder input attention is not affected by the presence of style artifacts in training while being greatly influenced by the presence of style artifacts at inference.

5 IMPACT

In Section 4, we have measured the confounding effect on both the function-level and line-level balanced accuracy for vulnerability discovery models. We have seen that their predictive performance and bug localization capabilities severely depend on artifacts present in the dataset. Furthermore, we demonstrate that we can not only measure but also reduce the confounding effect through our model or pre-processing choices. We provide here the most critical suggestions derived from our experimental study.

Tokenize the Code. The first insight is that using normalized code for fine-tuning the transformers yields the best function-level results on the augmented samples. However, there is still a measurable confounding effect of up to $c = 30\%$ for CodeT5+ and LineVul. That means that there are probably still artifacts remaining that normalizing code cannot account for. GNNs and StackLSTM provide better robustness in the first place but lose performance after

code obfuscation. CodeT5 + on tokenized code has the lowest scores but also the lowest confounding effect with around $c = 0\%$. Hence, given an LLM, pre-tokenizing the code may be the best option to receive unbiased performance results.

Compare against GNNs. ReVeal and CGIN have the best overall robustness against artifacts. CodeT5+, however, shows the strongest results in accurate line-level bug localization. Since a larger confounding effect hints that the model does not actually learn the underlying task, we argue that the graph-based models have better out-of-distribution performances and are thus better applicable to real-world cases. This is also in line with the results from Chen et al. [8]. By utilizing normalized or tokenized training data, the confounding effect of the transformer-based models is reduced such that their out-of-distribution performance is improved and competitive with GNNs.

Better Transferability. We argue that our evaluation provides a more faithful view of vulnerability discovery models. Obviously, if models tend to overfit to artifacts on one dataset, they lack generalization to another. Out-of-distribution transferability is an important property since this also mirrors the applicability to real-world cases. We tested LineVul on another dataset containing vulnerabilities from Chromium and Debian [7]. The new test set is disjoint from the original trainset. LineVul achieves 50%, tokenized LineVul 55%, and normalized LineVul 64% balanced accuracy on the unseen samples, underlining the usefulness of our evaluation and stratification approach.

6 RELATED WORK

In the following, we provide an outline of recent works that are tangent to this research area.

Vulnerability Discovery Models. There is a notable amount of research interest in developing novel vulnerability discovery models and comparing them to prior ones. With Vuddy [25] being a heuristic vulnerable clone detection model, Draper [39] and VulDeePecker [31] being one of the first token-based deep learning solutions, consecutive works started to benchmark their approaches against them. Slicing-based approaches [9, 30] and graph-learning-based approaches starting with Devign [65], have followed shortly after [7, 50, 65] reporting remarkable success even compared to traditional rule-based tools. The novel soft-attention mechanism from Vaswani et al. [48] has fostered research and approaches like VulSPG [64] and Cheng et al. [10] report even better results. Finally, LLMs like RoBERTa [34] or CodeT5 [53] have been applied to vulnerability discovery tasks [8, 15, 46] achieving currently the best performances on realistic datasets.

Explainable AI for Security. With the success of learning-based function-level vulnerability discovery models comes the problem with a lack of interpretability and defect localization [16]. Explainable AI helps to open up black boxes such as deep neural networks [4, 47]. This is even more critical to applications in a security context. Under the sheer number of explainability algorithms, finding the best suited for vulnerability discovery has been an active research question [54, 66]. Ganz et al. [16] investigate how a security practitioner can compare different localization of bugs, as a bug can be rarely pinned down to a single line [16, 64].

Fallacies in Vulnerability Discovery. Arp et al. [3] give rise to fallacies in developing ML models for security. For instance, they examine which metrics may induce a biased view of the performance, and how artifacts in the dataset can negatively impact the true performance. Chakraborty et al. [7] find that current datasets as the one from Zhou et al. [65] or Li et al. [31] are unrealistic and biased. They further show that most models lack transferability to out-of-distribution datasets by cross-evaluating popular models. Wang et al. [50] criticize datasets obtained through biased approaches like filtering commit messages by certain keywords. They propose to filter samples using a classifier identifying security-relevant patches. We leverage these insights in our experimental design as well as our metrics and dataset choice.

Code Transformations. In the experiments presented in this paper, we use specific types of transformations to investigate the confounding effects of artifacts on LLMs and graph-based models, specifically the application of predefined styles, uglification, and obfuscation. Applying transformations to code in order to investigate the limits of LLMs or graph-based models has received growing attention in the vulnerability discovery research community. Examples of such transformations that have been investigated are identifier renaming [20, 59, 60, 62, 63], insertion of unexecuted statements [20, 43, 60, 62] or replacement of code elements with equivalent elements [2, 28]. We continue this investigation by providing a novel implementation of transformations, by applying them to measure the confounding effects of artifacts, and by evaluating strategies to mitigate such effects.

7 CONCLUSION

In this work, we show that current vulnerability discovery models are severely influenced by artifacts such as code styles, variable naming, and common control flow patterns. The true performance of such models is hardly measurable and the reported ones from recent works can not be extrapolated to out-of-distribution code samples. We link the problem to spurious correlations in the dataset enabling models to shortcut decisions using unrelated information. We show that some models are less impacted by confounders and others more. Especially, large language models achieve remarkable results, but when provided with slightly modified code, their initial performance degrades. We propose three mitigations to drastically improve performance as a remedy to spurious correlation.

ACKNOWLEDGMENTS

The authors gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) under the grants IVAN (16KIS1165K) and BIFOLD (BIFOLD23B), from Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972, and from the European Research Council (ERC) under the consolidator grant MALFOY (101043410).

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. *CoRR* abs/1711.00740 (2017). arXiv:1711.00740 http://arxiv.org/abs/1711.00740

[2] 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

[3] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2020. Dos and Don'ts of Machine Learning in Computer Security. *CoRR* abs/2010.09470 (2020). arXiv:2010.09470 https://arxiv.org/abs/2010.09470

[4] Vaishak Belle and Ioannis Papantonis. 2021. Principles and Practice of Explainable Machine Learning. *Frontiers in Big Data* 4 (2021). https://doi.org/10.3389/fdata.2021.688969

[5] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. *CoRR* abs/1806.07336 (2018). arXiv:1806.07336 http://arxiv.org/abs/1806.07336

[6] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Inf. Softw. Technol.* 136 (2021), 106576.

[7] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[8] Yizheng Chen, Zhoujie Ding, Xinyun Chen, and David A. Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. *ArXiv* abs/2304.00409 (2023).

[9] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 38 (apr 2021), 33 pages. https://doi.org/10.1145/3436877

[10] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-Sensitive Code Embedding via Contrastive Learning for Software Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 519–531. https://doi.org/10.1145/3533767.3534371

[11] Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. 2023. Neural Networks and the Chomsky Hierarchy. arXiv:2207.02098 [cs.LG]

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[13] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 508–512. https://doi.org/10.1145/3379597.3387501

[14] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[15] 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.

[16] Tom Ganz, Martin Härterich, Alexander Warnecke, and Konrad Rieck. 2021. Explaining Graph Neural Networks for Vulnerability Discovery. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security* (Virtual Event, Republic of Korea) *(AISec '21)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/3474369.3486866

[17] Tom Ganz, Philipp Rall, Martin Härterich, and Konrad Rieck. 2023. Hunting for Truth: Analyzing Explanation Methods in Learning-based Vulnerability Discovery. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 524–541. https://doi.org/10.1109/EuroSP57164.2023.00038

[18] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (New Orleans, Louisiana, USA) *(CODASPY '16)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2857705.2857720

[19] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. Exploring network structure, dynamics, and function using networkx. (1 2008).

[20] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2022. Semantic Robustness of Models of Source Code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. https://doi.org/10.1109/saner53432.2022.00070

[21] Micha Horlboge, Erwin Quiring, Roland Meyer, and Konrad Rieck. 2022. I still know it's you! On Challenges in Anonymizing Source Code. arXiv:2208.12553 [cs.CR]

[22] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-Efficient Transfer Learning for NLP. arXiv:1902.00751 [cs.LG]

[23] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL]

[24] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. 2022. Transformers in Vision: A Survey. *ACM Comput. Surv.* 54, 10s, Article 200 (sep 2022), 41 pages. https://doi.org/10.1145/3505244

[25] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

[26] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* abs/1609.02907 (2016). arXiv:1609.02907 http://arxiv.org/abs/1609.02907

[27] Guillaume Lample, Miguel Ballesteros, Kazuya Kawakami, Sandeep Subramanian, and Chris Dyer. 2016. Neural Architectures for Named Entity Recognition. In *Proc. NAACL-HLT*.

[28] 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

[29] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2017. Gated Graph Sequence Neural Networks. arXiv:1511.05493 [cs.LG]

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. 2018. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *CoRR* abs/1807.06756 (2018). arXiv:1807.06756 http://arxiv.org/abs/1807.06756

[31] 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. *CoRR* abs/1801.01681 (2018). arXiv:1801.01681 http://arxiv.org/abs/1801.01681

[32] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. 2021. A Survey of Transformers. arXiv:2106.04554 [cs.LG]

[33] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2021. Multi-Task Learning Based Pre-Trained Language Model for Code Completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 473–485. https://doi.org/10.1145/3324884.3416591

[34] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL]

[35] Scott Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. arXiv:1705.07874 [cs.AI]

[36] Judea Pearl. 2009. *Causality* (2 ed.). Cambridge University Press. https://doi.org/10.1017/CBO9780511803161

[37] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[38] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1730, 11 pages.

[39] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. *CoRR* abs/1807.04320 (2018). arXiv:1807.04320 http://arxiv.org/abs/1807.04320

[40] Benjamin Sanchez-Lengeling, Jennifer Wei, Brian Lee, Emily Reif, Peter Wang, Wesley Qian, Kevin McCloskey, Lucy Colwell, and Alexander Wiltschko. 2020. Evaluating Attribution for Graph Neural Networks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 5898–5910. https://proceedings.neurips.cc/paper/2020/file/417fbbf2e9d5a28a855a11894b2e795a-Paper.pdf

[41] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*. 618–626.

[42] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning Important Features Through Propagating Activation Differences. *CoRR* abs/1704.02685 (2017). arXiv:1704.02685 http://arxiv.org/abs/1704.02685

[43] 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

[44] Yongduo Sui, Xiang Wang, Jiancan Wu, Min Lin, Xiangnan He, and Tat-Seng Chua. 2022. Causal Attention for Interpretable and Generalizable Graph Classification. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 1696–1705. https://doi.org/10.1145/3534678.3539366

[45] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode Compose: Code Generation Using Transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1433–1443. https://doi.org/10.1145/3368089.3417058

[46] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-Based Language Models for Software Vulnerability Detection. In *Proceedings of the 38th Annual Computer Security Applications Conference* (Austin, TX, USA) *(ACSAC '22)*. Association for Computing Machinery, New York, NY, USA, 481–496. https://doi.org/10.1145/3564625.3567985

[47] Erico Tjoa and Cuntai Guan. 2019. A Survey on Explainable Artificial Intelligence (XAI): Towards Medical XAI. *CoRR* abs/1907.07374 (2019). arXiv:1907.07374 http://arxiv.org/abs/1907.07374

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. arXiv:1706.03762 [cs.CL]

[49] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. *International Conference on Learning Representations* (2018). https://openreview.net/forum?id=rJXMpikCZ

[50] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. 2021. Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection. *IEEE Transactions on Information Forensics and Security* 16 (2021), 1943–1958. https://doi.org/10.1109/TIFS.2020.3044773

[51] Wenhua Wang, Yuqun Zhang, Zhengran Zeng, and Guandong Xu. 2020. TranŜ3: A Transformer-based Framework for Unifying Code Summarization and Code Search. arXiv:2003.03238 [cs.SE]

[52] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[53] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[54] Alexander Warnecke, Daniel Arp, Christian Wressnegger, and Konrad Rieck. 2020. Evaluating Explanation Methods for Deep Learning in Security. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Genoa, Italy, 158–174. https://doi.org/10.1109/EuroSP48549.2020.00018

[55] Konrad Weiss and Christian Banse. 2022. A Language-Independent Analysis Platform for Source Code. arXiv:2203.08424 [cs.CR]

[56] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[57] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). arXiv:1901.00596 http://arxiv.org/abs/1901.00596

[58] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

[59] 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

[60] 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

[61] Aiping Zhang, Liming Fang, Chunpeng Ge, Piji Li, and Zhe Liu. 2023. Efficient transformer with code token learner for code clone detection. *Journal of Systems and Software* 197 (2023), 111557. https://doi.org/10.1016/j.jss.2022.111557

[62] 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

[63] 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*.

[64] Weining Zheng, Yuan Jiang, and Xiaohong Su. 2021. VulSPG: Vulnerability detection based on slice property graph representation learning. *CoRR* abs/2109.02527 (2021). arXiv:2109.02527 https://arxiv.org/abs/2109.02527

[65] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *CoRR* abs/1909.03496 (2019). arXiv:1909.03496 http://arxiv.org/abs/1909.03496

[66] Deqing Zou, Yawei Zhu, Shouhuai Xu, Zhen Li, Hai Jin, and Hengkai Ye. 2021. Interpreting Deep Learning-Based Vulnerability Detector Predictions Based on Heuristic Searching. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 23 (mar 2021), 31 pages. https://doi.org/10.1145/3429444