Towards Causal Deep Learning for Vulnerability Detection

Md Mahbubur Rahman
mdrahman@iastate.edu
Iowa State University
Ames, IA, USA

Ira Ceka
ira.ceka@columbia.edu
Columbia University
New York, NY, USA

Chengzhi Mao
mcz@cs.columbia.edu
Columbia University
New York, NY, USA

Saikat Chakraborty
saikatc@microsoft.com
Microsoft Research
Redmond, WA, USA

Baishakhi Ray
rayb@cs.columbia.edu
Columbia University
New York, NY, USA

Wei Le
weile@iastate.edu
Iowa State University
Ames, IA, USA

ABSTRACT

Deep learning vulnerability detection has shown promising results in recent years. However, an important challenge that still blocks it from being very useful in practice is that the model is not robust under *perturbation* and it cannot generalize well over the *out-of-distribution (OOD)* data, e.g., applying a trained model to unseen projects in real world. We hypothesize that this is because the model learned non-robust features, e.g., variable names, that have *spurious correlations* with labels. When the perturbed and OOD datasets no longer have the same spurious features, the model prediction fails. To address the challenge, in this paper, we introduced *causality* into deep learning vulnerability detection. Our approach CausalVul consists of two phases. First, we designed novel perturbations to discover spurious features that the model may use to make predictions. Second, we applied the *causal learning* algorithms, specifically, *do-calculus*, on top of existing deep learning models to systematically remove the use of spurious features and thus promote causal based prediction. Our results show that CausalVul consistently improved the model accuracy, robustness and OOD performance for all the state-of-the-art models and datasets we experimented. To the best of our knowledge, this is the first work that introduces *do calculus based causal learning* to software engineering models and shows it's indeed useful for improving the model accuracy, robustness and generalization. Our replication package is located at https://figshare.com/s/0ffda320dcb96c249ef2.

CCS CONCEPTS

• **Software and its engineering** → **Empirical software validation**; • **Security and privacy** → **Software security engineering**.

KEYWORDS

vulnerability detection, causality, spurious features

ACM Reference Format:
Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. 2024. Towards Causal Deep Learning for Vulnerability Detection. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3597503.3639170

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639170

1 INTRODUCTION

A source code vulnerability refers to a potential flaw in the code that may be exploited by an external attacker to compromise the security of the system. Vulnerabilities have caused significant data and financial loss in the past [1, 2]. Despite numerous automatic vulnerability detection tools that have been developed in the past, vulnerabilities are still prevalent. The National Vulnerability Database received and analyzed a staggering number of 16,000 vulnerabilities in the year 2023 alone [1]. The Cybersecurity Infrastructure Security Agency (CISA) of the United States government reported that, since 2022, there have been over 850 documented instances of known vulnerabilities being exploited in products from more than 150 companies, including major tech firms such as Google, Microsoft, Adobe, and Cisco [2].

Due to the recent advancements in deep learning, researchers are working on utilizing deep learning to enhance vulnerability detection capabilities and have achieved promising results. Earlier models like Devign [27] and ReVeal [7] relied on architectures such as GNNs, while more-recent state-of-the-art (SOTA) models have moved towards transformer-based architectures. CodeBERT [13] uses *masked-language-model (MLM)* objective with a replaced token detection objective on both code and comments for pretraining. GraphCodeBERT [16] leverages semantic-level code information such as data flow to enhance their pre-training objectives. The more recent model UniXcoder [15] leverages cross-modal contents like AST and comments to enrich code representation.

However, an important challenge of deep learning tools is that the models learned and used *spurious correlations* between code features and labels, instead of using root causes, to predict the vulnerability. We call these features used in the spurious correlations *spurious features*. As an example, in Figure 1a, this code contains a memory leak. The SOTA model CodeBERT detected this vulnerability correctly with very high confidence (probability = 0.95). However, after we refactored the code and renamed the variables (Figure 1b), the model predicted this function as non-vulnerable. In Figure 2, we show that the change of the variable names caused its code representation to move from vulnerable to non-vulnerable clusters.

[1] https://nvd.nist.gov/general/nvd-dashboard
[2] https://www.cisa.gov/known-exploited-vulnerabilities-catalog

Apparently, the model did not use the cause of the vulnerability that "the allocated memory has to be released in every path" for prediction. In this example, `av_malloc` and `av_freep` are *causal features*—it is the incorrect use of `av_freep` API that leads to the vulnerability. However, the model associated `s`, `nbits` and `inverse` with the vulnerable label and associated `out1`, `dst0` and `out0` with the non-vulnerable label. Such correlations are spurious; those variable names are spurious features. We believe that spurious correlations are an important reason that prevent the models from being robust and being able to apply to unseen projects.

To address this challenge, in this paper, we introduce *causality* (a branch of statistics) into deep learning vulnerablity detection. We developed CausalVul and applied a causal learning algorithm that implements *do calculus* and the *backdoor criterion* in causality, aiming to disable models to use spurious features and promote the models to use causal features for prediction.

Our approach consists of two phases. In the first phase, we worked on discovering the spurious features a model uses. This task is challenging due to the inscrutable nature of deep learning models. To tackle this issue, we designed the novel perturbation method, drawing inspiration from adversarial robustness testing [24]. The perturbation changes lexical tokens of code but preserves the semantics of code like compiler transformations. In particular, we hypothesized that the models may use variable names as a spurious feature like Figure 1, and the models may also use API names as another spurious features. Correspondingly, we designed *PerturbVar* and *PerturbAPI*, two methods to change the programs and then observe if the models' predictions for these programs are changed. Through our empirical studies, we validated that the models indeed use certain variable names and API names as spurious features, and for vulnerable and non-vulnerable examples, the models use different sets of such names (Section 3).

To disable the models from using such spurious features, in the second phase, we applied causal learning, which consists of special training and inference routines on top of existing deep learning models. The causal learning puts a given model under *intervention*, computing *how the model would behave if it does not have spurious features*. This can be done via *backdoor criteria* from just the training data as well as our knowledge of spurious features. Specifically, we first train a model and explicitly encode a known spurious feature F into the model. At the inference time for an example x, we use the x's representation joint with a set of different spurious features, so that the model cannot use F to make the final decision for x.

We evaluated CausalVul using Devign [27] and Big-Vul [12], two real-world vulnerability detection datasets and investigated on three SOTA models – CodeBERT, GraphCodeBERT, and UnixCoder. Experimental evaluation shows that the causal model in CausalVul learns to ignore the spurious features, improving the overall performance on vulnerability detection by 6% in Big-Vul and 7% in the Devign dataset compared to the SOTA. The CausalVul also demonstrates significant improvement in generalization testing, improving the performance on the Devign dataset trained model up to 100% and Big-Vul dataset trained model up to 200%. Experimental results also show that our CausalVul is more robust than the current SOTA models. It improves the performance up to 62% on Devign and 100% on Big-Vul on the perturbed data we constructed for robustness testing.

In summary, this paper made the following contributions:

- We discovered and experimentally demonstrated that variable names and API names are used as spurious features in the current deep learning vulnerability detection models,
- We formulate deep learning vulnerability detection using causality and applied causal deep learning to remove spurious features in the models, and
- We experimentally demonstrated that causal deep learning can improve model accuracy, robustness and generalization.

2 AN OVERVIEW OF OUR APPROACH AND ITS NOVELTY

In Figure 3, we present an overview of our approach, CausalVul. CausalVul consists of two stages: (i) discovering spurious features and (ii) using causal learning to remove the spurious features.

Discovering Spurious Features. First, we work on discovering spurious features used in deep learning vulnerability detection models. Spurious features are the attributes of the input dataset, *e.g.,* variable names, that exhibit correlation with the target labels (*i.e.,* vulnerable/non-vulnerable) but are not the root causes of being (non)-vulnerable, and thus, may not generalize to a desired test distribution. To discover the spurious features, in our approach, we first hypothesize a spurious feature based on our domain knowledge. For instance, we assumed variable names can be spurious features for vulnerability detection tasks—models should not make a decision based on the names. Next, we design *perturbations* based on each hypothesized feature. Specifically, we transform the programs via a refactoring tool that changes the feature, but does not change the semantics of code (similar to compiler transformations). After we observe the model changes its prediction for many such examples, we conclude that the model likely relied on this spurious feature. Discovering spurious features is a very challenging task. As an instance, randomly altering variable names does not reveal its spurious nature. We need to carefully identify which variable(s) need to be changed by which name(s) to maximize its impact. To the best of our knowledge, we have not seen a systematic study on finding spurious features in vulnerability detection models.

Removing Spurious Features. Once we discovered spurious features, in this step we try to reduce the impact of these features on the model decision. One easy way to achieve that could be to augment the training data by randomizing the spurious features and then retrain the model. However, it will take much time for big, pretrained, code representations/deep models. So instead, we take existing representation and apply causal learning to disable the model to use spurious features at inference time. This ensures the model can rely mostly on the causal features for prediction.

We accomplish this goal as follows. First, as an input, we take the existing representation and the known spurious feature to be eliminated. We then train a model such that the representation learned in this step is especially biased towards the targeted spurious feature. Next, during inference, the model makes a decision based on the input representation while ignoring the bias representation learned in the previous step. In particular, we marginalized over a set of examples that contain different spurious features and prevent the inference from utilizing the targeted spurious feature while

```c
FFTContext *av_fft_init(int nbits, int inverse){
    FFTContext *s = av_malloc(sizeof(*s));
    if (s && ff_fft_init(s, nbits, inverse))
        av_freep(&s);
    return s;
}
// Prediction probability: 0.9493
```

(a) Vulnerable code - Correctly Predicted

```c
FFTContext *av_fft_init(int dst0, int out0){
    FFTContext *out1 = av_malloc(sizeof(*out1));
    if (out1 && ff_fft_init(out1, dst0, out0))
        av_freep(&out1);
    return out1;
}
// Prediction probability: 0.2270
```

(b) Perturbed Vulnerable code - Mispredicted

Figure 1: A vulnerable example predicted as vulnerable with 0.9493 but predicted as non-vulnerable with probability 0.2270 when names are perturbed by some of the spurious names from the opposite class.

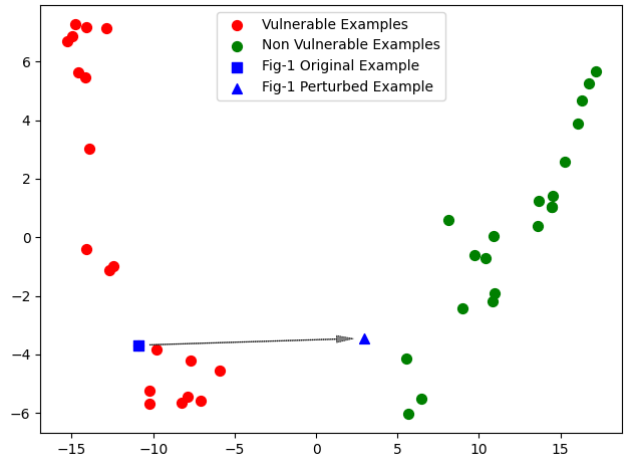

Figure 2: Visualization using Principle Component Analysis (PCA) of Figure 1's code representations generated by CodeBERT before and after perturbed the names.

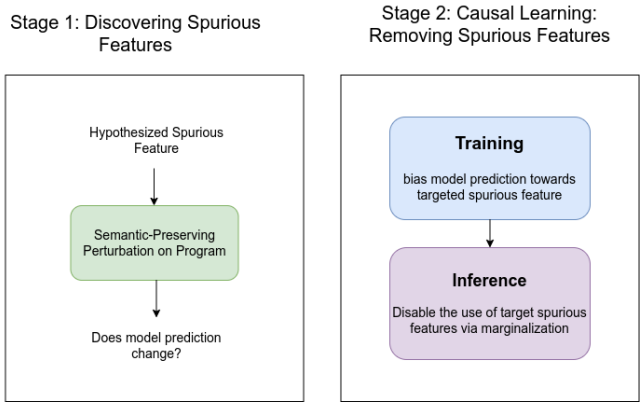

Figure 3: CausalVul: an overview

making the final decision. This approach relies on the principles of "do-calculus" and the "backdoor criterion," which we will explain in detail in Sections 4 and 5, respectively.

To the best of our knowledge, this is the first work to bridge causal learning with deep learning models for vulnerability detection. It helps researchers understand to what degree causality can help deep learning models remove spurious features and to what degree we can improve model accuracy, robustness and generalization after removing spurious features. Such causal learning approaches have been applied in computer vision [9]. A key difference is that the vision domain is *continuous* in that the pixel values are continuous real numbers, while program features are discrete. Thus, in the domain of program data, we can explicitly discover spurious features and then apply causal learning in a targeted manner.

3 DISCOVERING SPURIOUS FEATURES

In this section, we illustrate our technique for discovering spurious features through semantic-preserving perturbations.

Spurious features are features that spuriously correlate to the output. Those spurious correlations often only exist in training but not in testing, especially testing under perturbed data or unseen projects. That is because a feature that is spurious, as opposed to causal, will change across domains and is no longer useful for prediction, as shown in Figure 1. Thus, a standard way to determine whether one feature is spurious is to *perturb* the values of the feature and observe how the output changes.

Following this idea, we first hypothesized two spurious features, *variable names* and *API names* that the current deep learning models may use for vulnerability detection. We chose these two spurious features as a proof of concept to check if we perturb in lexical and syntactic characteristics respectively while keeping the overall semantic of the program the same, whether the model prediction would change. We applied an existing code refactoring tool, *NatGen* [3], to perturb the code in test dataset, and observe whether the model performance is changed between the original test set and the perturbed test set (see problem formulation in Section 3.1). Interestingly, we found that randomly changing the variable names and function names do not bring down the model performance. Thus we designed novel perturbation methods that can demonstrate the two spurious features. See Sections 3.2 to 3.4.

3.1 Problem Formulation

Given a perturbation p, a code sample s, and a trained model M, we discover a spurious feature if the following conditions are met: (1) the application of the perturbation, $p(s)$, does not alter the semantics of the function, (2) the model's prediction changes upon transformation, and (3) the candidates for a perturbation p are drawn from the training distribution.

We will use F1 score as a more-thorough evaluation metric for our imbalanced datasets and condition number (2). A degraded F1

score indicates the application of the perturbation $p(s)$ has resulted in mis-classifications (flipped labels).

3.2 *PerturbVar*: Variable Name as a Spurious Feature

To demonstrate that some variable names are indeed used by the models as spurious features, we design an extremely perturbed test set. We analyze the training dataset and sort the variable names based on their frequency of occurrences in vulnerable functions and non-vulnerable functions, respectively. When replacing existing variable names in the test set, we randomly select a name from the top-K most-frequent variable names of the *opposite-class labels*: for a non-vulnerable sample, we replace the existing variable names with a name from the vulnerable training set, but which does not occur in the non-vulnerable training set, and vice-versa. We apply this perturbation to every sample in the test set.

Table 1: PerturbVar: Impact of Variable Name Perturbation

Top-K (Freq.)	CodeBERT		UniXcoder		GraphCodeBERT	
	Devign F1	Big-Vul F1	Devign F1	Big-Vul F1	Devign F1	Big-Vul F1
Baseline	0.61	0.36	0.63	0.38	0.62	0.37
Random	0.61	0.32	0.63	0.36	0.62	0.35
Top 100	0.61	0.25	0.52	0.24	0.60	0.27
Top 50	0.55	0.25	0.55	0.22	0.59	0.26
Top 25	0.54	0.26	0.56	0.25	0.59	0.27
Top 20	0.55	0.25	0.56	0.24	0.59	0.28
Top 15	0.54	0.23	0.56	0.24	0.59	0.27
Top 10	0.54	0.26	0.53	0.23	**0.57**	0.28
Top 5	**0.52**	**0.21**	**0.52**	**0.18**	0.58	**0.26**

Observation. As shown in Table 1, we are able to degrade the F1 score as much as 11.5 % on the Devign dataset, and as much as 20 % on the Big-Vul dataset. We have observed the performance degradation across all the datasets and multiple architectures: CodeBERT, GraphCodeBERT, and UniXcoder. However, in the randomized setting, the performance almost does not change relative to the baseline. Introducing common vulnerable names into non-vulnerable code-samples causes the model to misclassify the sample as vulnerable. Conversely, introducing common non-vulnerable names into vulnerable code samples causes the model to misclassify the sample as non-vulnerable. The more common the variable names are used (i.e. the lower the Top-K), the more the performance degrades.

3.3 *PerturbAPI*: API Name as a Spurious Feature

Modern programs frequently use API calls. We conjecture that the models may establish spurious correlations between API names with vulnerabilities. Similar to the approach used in *PerturbVar*, we ranked the frequency of the API calls in the training data, for vulnerable examples and non-vulnerable examples respectively. We then insert API calls (that are ranked in the top-100 occurring calls in non-vulnerable examples, but which are not frequently-occurring in the vulnerable examples) into vulnerable examples and vice-versa. To preserve the semantics of code, we insert these API calls as "dead-code", *i.e.,* this code will never be executed.

We inject dead-code at n random positions within the code sample. The dead-code block is composed of m distinct function/API calls (m and n are configurable and we used $m = 5, n = 5$ to produce the results in Table 2). We guard the block of API-calls with an unsatisfied condition, ensuring the loop is never executed, as shown in Figure 4:

```
1    while ( _i_4 > _i_4 ) {
2    tcg_out_r(s , args[1]); help_cmd(argv[0]);
3    cris_alu(dc , CC_OP_BOUND , cpu_R[dc -> op2] , cpu_R[dc -> op2
         ] , 10 , 4);
4    RET_STOP(ctx); tcg_out8(s , args[3]); }
```

Figure 4: Dead-code composed of our spurious feature, API calls

Table 2: PerturbAPI: Impact of API Name Perturbation

Dead-code Type	CodeBERT		UniXcoder		GraphCodeBERT	
	Devign F1	Big-Vul F1	Devign F1	Big-Vul F1	Devign F1	Big-Vul F1
Baseline	0.61	0.36	0.63	0.38	0.62	0.37
Random	0.61	0.35	0.62	0.36	0.62	0.35
API	**0.52**	**0.10**	**0.34**	**0.11**	**0.47**	**0.09**

Observation. We compare our results to (1) baseline performance and (2) random dead-code transformation performance, shown in Table 2. Our results show that dead-code composed of API calls severely hurts model performance compared to the vanilla baseline and random dead-code transformation. Model performance degrades proportionally with the inclusion of increased API calls and increased injection locations. Performance in the vanilla model degrades as much as 28.73 % on Devign and 27.99 % on Big-Vul.

3.4 *PerturbJoint*: Combine Them Together

We hypothesize that the composition of the two spurious features will further degrade model performance. We setup the study, where for every sample constructed in the *PerturbAPI* dataset from Section 2, we replace existing variable names with a random selection from the top-K most-frequent variable names of the opposite class (using the same approach from Section 1).

Observation. In Table 3, our results show that the composition of API dead-code and variable renaming further degrades the model. The model degradation is more severe when applying the composition of the settings. In the combined setting, performance degrades as much as 41.37 % for Devign and 33.89 % for Big-Vul.

Summary. In this section, we investigated multiple datasets and multiple SOTA models, revealing that variable names and API names are spurious features frequently utilized by these models. Interestingly, the models associate different variable names and API names as spurious features for different labels. Consequently, it became evident that only meticulously designed perturbations, not random ones, showcased the usage of these spurious features, consequently leading to a decline in the models' performance on the perturbed datasets.

Table 3: PerturbJoint: Impact of Joint Perturbation

Top-K (Freq.)	CodeBERT		UniXcoder		GraphCodeBERT	
	Devign F1	Big-Vul F1	Devign F1	Big-Vul F1	Devign F1	Big-Vul F1
Baseline	0.61	0.36	0.63	0.38	0.62	0.37
Top 100	0.38	0.07	0.23	0.07	0.59	**0.06**
Top 50	0.37	0.07	0.25	0.06	0.60	0.07
Top 25	0.36	0.07	0.28	0.08	0.60	0.07
Top 20	0.38	0.08	0.27	0.08	0.48	0.07
Top 15	0.36	0.07	0.27	0.06	0.52	0.07
Top 10	0.37	0.08	0.24	0.07	**0.45**	0.08
Top 5	**0.33**	**0.06**	**0.22**	**0.05**	0.47	0.07

4 CAUSAL LEARNING TO REMOVE SPURIOUS FEATURES

In this section, we present how to apply causal learning to remove spurious features in the vulnerability models.

4.1 Causal Graph for Vulnerability Detection

To apply causal learning, our first step is to construct a *causal graph* to model how vulnerability data is generated from a statistics point of view, shown in Figure 5 (a). A causal graph visually represents the causal relationships between different variables or events. In the graph, nodes represent random variables, and directed edges between nodes indicate causal relationships. Thus, A→B means variable A directly influences variable B. The absence of an edge between two nodes implies no direct causal relationship between them. In Figure 5 (a), X represents a function in the program. Whether there is a vulnerability or not in the code is directly dependent on the function. So we add an edge from X to Y, where Y is the label of vulnerability detection, 1 indicates vulnerable, and 0 indicates not vulnerable.

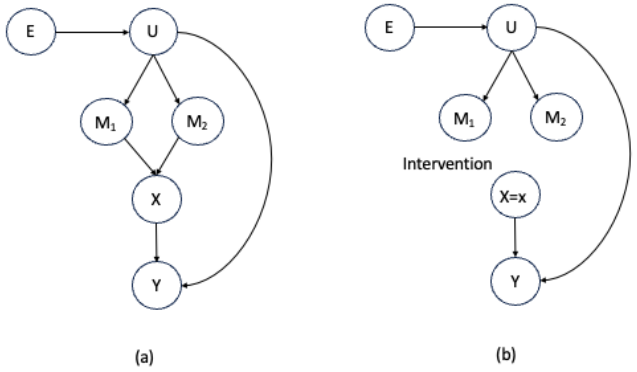

Figure 5: Causal Graph Before and After Do Calculus

In the causal graph, we use E, namely *environment*, to model the domain where the dataset is generated. For example, at training time, namely $E = 0$, this environment indicates e.g., the code is written by certain developers and for certain software applications. At testing or deployment time of the model, namely $E = 1$, the code may link to different developers and applications. Such environment-specific factors are modeled using U. It is a *latent* variable and not directly observed in the training data distribution. For example, U can denote the expertise or coding style of the programmer or the type of software application. Because of U, there can be different spurious features in the code, like the two we showed in Section 3, denoted as M_1 and M_2.

M_1 represents variable naming styles, as developers may like to use certain formats of variable names in their code. Similarly, M_2 represents API names, as certain developers or applications may more likely use a particular set of API calls. These basic (text) features of M_1 and M_2 influence the code. Thus, we add the edges X to M_1 and M_2, respectively. Note that the causal graph can be expanded to integrate more of such spurious features. We leave this to future work.

In Figure 5 (a), we also have the edge U to Y, indicating that U can impact Y. For example, a junior developer may more likely introduce vulnerabilities; similarly, certain APIs may more likely introduce vulnerabilities, e.g., SQL injection.

4.2 Applying Causality

When we train a deep learning model, using the code X as input, to predict the vulnerability label Y, we learn the correlation $P(Y|X)$ in the training dataset. However, when we deploy this trained model in a new environment, e.g., handling perturbed datasets or unseen datasets., the domain will be changed. Due to the shifting of E from 0 to 1, the correlation $P(Y|X)$ will be different, denoted as $P(Y|X, E = 0) \neq P(Y|X, E = 1)$. From the causal graph point of view, X and Y are both the descendants of E and therefore will be affected by its change.

To improve the models' performance in generalizing to new environments beyond the training, our goal is to learn the signal that is invariant across new environment and domains. Applying causal learning, we *intervene* on the causal graph so that the correlation between X and Y is the same no matter how the environment E changes. In causality, such intervention is performed via *do-calculus*, denoted as $P(Y|do(X = x))$.

The key idea is that we imagine to have an oracle that can write perfect, unbiased code x; we then use such code to replace the original code X for vulnerability detection. Doing so, we created a new causal graph by removing all the incoming edges to the node X and put the new code here $X = x$. See Figure 5 (b). In this new causal graph, E only can affect Y, not X, and thus the correlation of X and Y will no longer be dependent on E. In other words, the correlation of x and Y in the new causal graph then becomes the *invariant* signal across different domains, e.g., from training to testing. When we learn the correlation in this new causal graph, the learned model can generalize to new environments since this relationship also holds at testing. In the next section, we explain how we compute $P(Y|do(X = x)$ from the observational data.

4.3 Estimating Causality through Observational Data

Figure 5 (a) models the joint distribution of vulnerability data from a data generation point of view. Figure 5 (b) presents the causal graph after performing do calculus on X, allowing us to generalize across the domains. The challenge is that performing intervention and obtaining the perfect, unbiased code is almost impossible. To

estimate the causal effect without actually perform the intervention, we apply the *backdoor criterion* [21] to derive causal effect $P(Y|do(X))$ from the given observational data of X and Y.

Backdoor criterion teaches how to block all the non-causal paths from X to Y in the causal graph so we can compute the causal effect. For example, in Figure 5 (a), node U can cause X, U can cause Y; as a result, there can exist correlations between X and Y in the observational data. However, such correlation do not necessarily indicate the causal relation from X to Y. For example, the junior developers may like to write their code with the variable name `myvar` and the junior developers may more likely introduce the vulnerability into their code. In this case, we might often see the code with `myvar` is detected with vulnerability in the dataset. However, `myvar` is not the cause for the vulnerability. The correlation of `myvar` and vulnerability is spurious. To remove such spurious correlations, the backdoor criterion states that we need to condition X on M_1 and M_2; as a result, we can remove the incoming edges to X and thus impact from U to X. Mathematically, we will have the following formula for Figure 5 (b).

$$P(Y|do(X = x)) = \sum_{M_1, M_2} P(Y|X = x, M_1, M_2)P(M_1)P(M_2) \quad (1)$$

To compute Eq.(1), we adopted an algorithm from [9]. First, we train a model that predicts Y from R and M (we denote M_1, M_2 as M for simplicity), denoted as $P(Y|R, M)$, where R is the representation of X computed by an existing deep learning model, and M is the representation of X that encodes the spurious features. For example, if we encode each token of X into M, we will have variable names and API names encoded in M. Since R is a representation learned by models, the algorithm assumes R encoded both causal and spurious features. The goal of taking R and M jointly for training is to especially encode targeted spurious feature(s) in the M component of the model $P(Y|R, M)$.

At inference, we use the model $P(Y|R, M)$ to compute $P(Y|do(X = x))$ via Eq. (1). The backdoor criterion instructs that we first randomly sample examples from X that contains different spurious features. We will then marginalize over the spurious features through weighted averaging, that is, $\sum_{M_1, M_2} P(Y|X = x, M_1, M_2)P(M_1, M_2)$. This can be computed using the model trained above $P(Y|R, M)$. Intuitively, the spurious features in the data have been cancelled out due to the weighted averaging (please refer to [22] to understand why this is the case). The model will make predictions based on the remaining signals, which are the causal features left in R.

5 THE ALGORITHMS OF CAUSAL VULNERABILITY DETECTION

In this section, we present the algorithms that compute the terms Eq. (1). In Algorithm 1, we will train a model of $P(Y|R, M)$. In Algorithm 2, we show how to apply $P(Y|R, M)$ and use backdoor criteria to undo the spurious features during inference.

5.1 Training P(Y|R,M)

Algorithm 1 takes as input the training dataset D as well as the spurious feature(s) we aim to remove, namely *targeted spurious feature(s) t*. For example, it can be variable names and/or API names, as presented in Section 3.

The goal of Algorithm 1 is to learn $P(Y|R, M)$ from the training data. We use the embedding (r) of the original input x computed by an existing deep learning model. At line 4, the training iterates through n epochs. For each labeled data point (x, y), we first obtain r of x at line 6.

At line 7, we select x' that shares the targeted spurious feature t with x but differ in root cause r. This means (1) x' should have the same label as x; as shown in Section 3, spurious features are label specific; (2) x' should share t with x, so that the model will more likely encode t in the M component of $P(Y|R, M)$; and (3) x' should not be x so that training with $(r, M_{x'}, y)$ at line 9, the model will rely on the spurious feature in $M_{x'}$ instead of r to make prediction. Our final goal is to encode in the model the root cause of x in the R component, and the targeted spurious feature t shared across x and x' in the M component.

In Table 4, we designed different ways of selecting x' at line 7, *Settings Var1-Var2* for removing the spurious feature of variable name, *Settings API1-API3* for API names, *Setting Var+API* is for both variable and API names.

The first idea is to select x' from the training dataset such that it shares the maximum number of *spurious* variable or API names with x. See the rows of *Var1* and *API1*. The second idea is to construct an example x' such that x and x' share k variable/API names. See the rows of *Var2* and *API2*. Similarly, Setting *API3* constructs x' to have the top k spurious API names. We have also tried this approach for variable names, but it does not report good results. In Setting *Var+API* for removing both spurious features, we take x' from Setting *Var1* and then perform transformation using Setting *API3*. This achieved the best results in our experiments compared to other combinations.

Algorithm 1: Causal Learning Model Training

```
1  Input: Training dataset D over (X, Y); Targeted spurious feature t
2  Phase 1: Compute P̂(R|X) by the SOTA code representation model
3  Phase 2:
4  for i ← 1 to n epochs do
5      foreach (x, y) in D do
6          Extract r for x using P̂(R|X).
7          Select x' by one of the selection procedures in Table 4
8          Encoding x' to M_x'
9          Train P(Y|R, M) using (r, M_x', y) via minimizing the
             classification loss
10     end
11 end
12 Output: Model P̂(R|X) and P(Y|R, M)
```

5.2 Undo Spurious Features during Inference

In Algorithm 2, we explain our inference procedure. To predict the label for a function x, we first extract r at line 2. Existing work directly predict the output y using r. Since there are both spurious features and core features in r, both of them are going to be used to predict the vulnerability. Here, we will use our causal algorithm to

Table 4: Selection Procedures for x' in Algorithm. 1

Spurious Feature	Setting	x': same label of x
Variable name	Var1	select x' which shares the maximum number of spurious variable names with x.
	Var2	random select x', replace at most k variable names of x' with the variable names from x
API name	API1	select x' which shares the maximum number of spurious API names with x
	API2	random select x', randomly select k APIs from x and insert them in x' as dead code
	API3	random select x', pick k APIs from the top 10% most frequent spurious APIs and insert them in x' as deadcode
Variable and API names	Var+API	select x' based on setting *Var1*; then insert dead code according to setting *API3*.

remove the spurious features in our inference. The key intuition is to cancel out the contribution of the spurious features by averaging over the prediction from all kinds of spurious features. To do so, at line 4, we randomly sample different x' K times from the training dataset D. At line 8, we then compute Eq (1). We assumed uniform distribution for $P(M'_x)$. Finally, at line 9, we make the prediction. *argmax_y* means select the label that has the better probability among the vulnerable/nonvulnerable classes.

Algorithm 2: Causal Learning Model Inference

1 **Input:** Query x, training dataset D over (X, Y), models $\hat{P}(R|X)$ and $\hat{P}(Y|R,M)$
2 Extract r for x using $\hat{P}(R|X)$.
3 **for** $i \leftarrow 1$ **to** K **do**
4 Randomly select x' from the training set D.
5 Extract spurious features $M_{x'}$ from x'.
6 Compute $\hat{P}(y_i|r, M_{x'_i})$
7 **end**
8 Calculate the causal effect $P(y|do(X=x)) = \hat{\sum}_i \hat{P}(y_i|r, M_{x'_i})P(M_{x'_i})$
9 **Output**: Class $\hat{y} = argmax_y P(y|do(X=x))$.

6 EXPERIMENTAL SETUP

6.1 Implementation

We use Pytorch 2.0.1[3] with Cuda version 12.1 and transformers[4] library to implement our method. All the models are fine-tuned on single NVIDIA RTX A6000 GPU, Intel(R) Xeon(R) W-2255 CPU with 64GB ram. The pretrained weights and tokenizers of the transformer models were obtained from the link provided by the original authors [5]. We used Adam Optimizer to fine-tune our models. The models were trained until 10 epochs while the batch size is set to 32. The learning rate is set to 2e-5. We trained the models with our training data and the best fine-tuned weight is selected based on the f1 score against the validation set. We used this weight during the evaluation of our test set. We set K=40 (Algorithm 2 line 3), as it reported the best results among the values we tried.

[3]https://github.com/pytorch/pytorch
[4]https://github.com/huggingface/transformers
[5]https://github.com/microsoft/CodeBERT

6.2 Datasets and Models

We considered two vulnerability detection datasets: Devign [27] and Big-Vul [12]. Devign is a balanced dataset consisting of 27,318 examples (53.22% vulnerable) collected from two different large C programming-based projects: Qemu and FFmpeg. As Zhou et al. [27] did not provide any train, test, and validation split, we used the split published by CodeXGLUE authors [19]. Big-Vul dataset is an imbalanced dataset consisting of 188,636 examples (5.78% vulnerable) collected by crawling the Common Vulnerabilities and Exposures (CVE) database. For this dataset, we used the partitions published by the LineVul authors [14].

We evaluated the three SOTA models, CodeBERT, GraphCodeBERT and UniXcoder as the model $P(R|X)$ in Algorithm 1. The representation R is extracted from the output embedding of the last hidden layer of these models. To construct the network $\hat{P}(Y|R,M)$, at first, we pass x' through the first four encoder block of these transformer models. The obtained output embedding is considered as M. We used the first fourth encoder block (empirically it is the best layer we found) from the twelve blocks to compute M because the early layers learn the low-level features and spurious features tend to be the low-level features [20]. M is then concatenated with R. Finally, a 2-layer fully-connected network is used to predict Y.

7 EVALUATION

We studied the following research questions:
RQ1: Can CausalVul improve the accuracy of the model?
RQ2: Can CausalVul improve the robustness and generalization of the model?
RQ3: (ablation studies) How do different design choices affect the performance of CausalVul?

7.1 RQ1: Model Accuracy

Experimental Design. To answer this RQ, we implemented our causal approach on top of three state-of-the-art transformer-based vulnerability detection models - CodeBERT, GraphCodeBERT, and UniXcoder. We used the default (w/o causal) versions of these models as baselines. We address these default versions as *Vanilla models*.

We experimented with all the causal settings shown in Table 4, namely *Var1* and *Var2*, *API1*, *API2* and *API3*, and *Var+API*. We evaluated both the vanilla models and the causal models on the same *unperturbed* original test set and use the metrics of *F1*. We trained all the models three times with different random seeds and consider the average F1 score as our final score (this is done for all the RQs).

Table 5: The F1 score of CausalVul and the vanilla model on the test set of Devign and Big-Vul dataset.

Settings	CodeBERT		GraphCodeBERT		UniXcoder	
	Devign	Big-Vul	Devign	Big-Vul	Devign	Big-Vul
Vanilla	0.61	0.38	0.63	0.38	0.64	0.39
CausalVul						
Var1	0.65	0.40	0.63	0.38	0.66	0.40
Var2	0.64	0.40	0.63	0.40	0.65	**0.41**
API1	0.63	0.40	0.62	**0.41**	0.64	0.40
API2	0.65	0.40	0.64	0.40	0.66	0.40
API3	**0.66**	0.40	0.65	0.38	**0.68**	0.40
Var+API	0.65	**0.41**	**0.66**	0.39	0.66	0.40

Result: Table 5 shows the result. CausalVul outperforms the vanilla models for all the settings, all the datasets and all the models. For Devign data, CausalVul Var1, API3, and VAR+API show 4, 5, and 4 percentage points improvement respectively in terms of the F1-score against the CodeBERT vanilla model. In the UniXcoder models, the improvement for these three approaches are 2, 4 and 2 percentage points respectively. With the GraphCodeBERT model, our approaches API3 and Var+API show 2 and 3 percentage points improvement against the vanilla model. Overall, our approaches show 2-5 percentage points F1 score improvement against the vanilla model. For the Big-Vul dataset, our causal approaches with the CodeBERT model show a 2-3 percentage points improvement, so do the GraphCodeBERT and UniXcoder models.

To the best of our knowledge, these are the best-reported vulnerability detection results in these widely studied datasets [6]. One may suspect that while ignoring spurious features, a model may reduce some of the in-distribution accuracies, as the spurious features also contribute in benign settings. In contrast, to our surprise, we find that CausalVul is learning more significant causal signals, which compensate for the loss of spurious features and also improve in-distribution accuracy.

Result:RQ1. CausalVul outperforms other pre-trained models, suggesting causal learning focuses on the root causes of the vulnerabilities by learning to ignore spurious features. Overall, our causal settings show up to 6% improvement in F1 in Devign Dataset and 7% improvement in F1 in Big-Vul dataset.

7.2 RQ2: Model Robustness and Generalization

Experimental Design. For robustness evaluation, we compare the performance of the causal models with the vanilla model on the three perturbed datasets presented in Section 3. *Var1* and *Var2* run on the *PerturbVar* dataset, which has the worst performance on the corresponding vanilla model as per Table 1. For example, *PerturbVar* dataset perturbed with Top 5 and Top 10 most frequent variable names perform the worst in vanilla CodeBERT and GraphCodeBERT models, respectively. Hence, we select the Top 5 for comparison with CodeBERT and the Top 10 to compare with GraphCodeBERT. Similarly, *API1–API3* runs on the worst *PerturbAPI* dataset, and *Var+API* runs on the worst *PerturbJoint* dataset for the corresponding models.

To investigate the generalization performance of the models, we evaluated the model trained on the Devign dataset using the Big-Vul test dataset (excluding overlapped project FFMPEG). Similarly, we trained on Big-Vul and tested on the Devign dataset. For both experiments, we experimented all the settings in Table 4 and used the F1 as metrics.

Results for Robustness: Table 6 shows the robustness performance in three blocks: the upper block presents the results from running with the worst *PerturbVar* data, and similarly, the middle and lower blocks present the results from running with the worst *PerturbAPI* and with the worst *PerturbJoint* data respectively.

Between Var1 and Var2, Var1 performs better on Devign data and shows 6, 3, and 2 percentage points improvement in F1 with CodeBERT, GraphCodeBERT and UniXcoder model respectively. For the Big-Vul data, Var1 works better on the Codebert model with 1 percentage point improvement while Var2 is better for the other two models with 4 and 3 percentage points improvement respectively. Overall, both of Var1 and Var2 approaches work better than the vanilla model in terms of robustness. Among API1, API2 and API3, API3 works better in Devign data and demonstrates 10, 5, and 22 percentage points improvement with the three models respectively. For the Big-Vul data, API2 works better with the CodeBERT model and improves by 3 percentage points over CodeBERT vanilla. In GraphCodeBERT, both AP1 and API2 show similar performance and show 2 percentage point improvement. The VAR+API setting shows a similar improvement trend over vanilla performance.

In Figure 6, we show the predicted probability density for the Devign vulnerable data. In each subplot, X-axis is the predicted probability of being vulnerable, and Y-axis is the count of the examples whose predictions are that probability. The orange lines plot the causal model and the blue lines plot the vanila model. The figure demonstrates that the overall prediction probability for vulnerable data increases, which means the model is more confident in predicting vulnerabilities. Experiment result show that the overall difference of the probability density between vanilla and the causal approach is statistically significant with $p - value <<< 0.05$. with varying effect size, as documented at Table 7.

We also investigate how many examples from robustness data are predicted incorrectly in Vanilla models like Figure 1 and predicted correctly in CausalVul. Table 8 shows that CausalVul correctly predict a significant amount of data which are predicted incorrectly in Vanilla models.

Results for Generalization: We present the generalization results in Table 9. The Devign column defines the F1 score of the Big-Vul test set when evaluated on the model trained with the Devign train set. Similarly, the Big-Vul column presents the F1 score of the Devign test set evaluated on the model that is trained with the Big-Vul train set. Our results show that when the model is trained on the Devign train set and the Big-Vul test set is used as the out-of-distribuition (OOD), our causal approach shows 1-2 percentage points improvement for CodeBERT and UniXcoder models. But

[6] our setting keep the inputs in their original form without any perturbation or normalization.

Table 6: The Robustness performance of CausalVul and the vanilla models.

Settings	CodeBERT		GraphCodeBERT		UniXcoder	
	Devign	Big-Vul	Devign	Big-Vul	Devign	Big-Vul
Vanilla	0.52	0.22	0.58	0.26	0.52	0.19
Var1	**0.58**	**0.23**	**0.61**	0.28	**0.54**	0.21
Var2	0.55	0.22	0.58	**0.30**	0.53	**0.22**
Vanilla	0.52	0.10	0.47	0.09	0.35	0.09
API1	0.52	0.13	0.35	**0.11**	0.45	**0.14**
API2	0.56	**0.13**	0.39	**0.11**	0.49	0.13
API3	**0.62**	0.10	**0.52**	0.09	**0.57**	0.13
Vanilla	0.52	0.06	0.45	0.07	0.22	0.05
Var+API	**0.55**	**0.07**	**0.54**	**0.08**	**0.31**	**0.10**

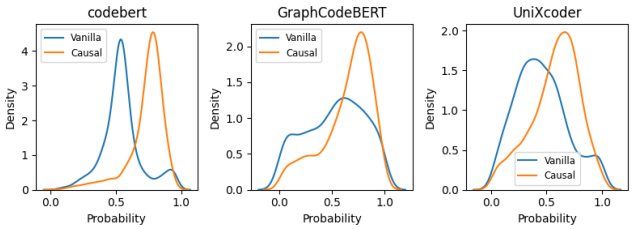

Figure 6: Predicted probability density of the Devign data from Vanilla and Causal approach.

Table 7: Cohen's d effect size of the difference of the probability density between Vanilla and Causal approach for vulnerable data.

Setting	CodeBERT		GraphCodeBERT		UniXcoder	
	Devign	Big-Vul	Devign	Big-Vul	Devign	Big-Vul
var	trivial	midium	trivial	small	trivial	small
api	large	midium	small	small	large	small
combine	large	small	large	small	small	midium

Table 8: Number of pertubed examples whose predictions are incorrect in Vanilla model (see Figure 1) but correct in CausalVul.

Dataset	CodeBERT	GraphCodeBERT	UniXcoder
Devign	298	255	368
Big-Vul	205	93	130

when the Devign test set is used as the OOD data on the model trained with Big-Vul data, CodeBERT model shows 7-10 percentage points improvement, GraphCodeBERT model shows at most 4 percentage points improvement and the UniXcoder model shows 6-14 percentage points improvement.

In this RQ, we show that causal learning has the potential of significantly improving the robust accuracy and generalization. In such a setting, since the spurious features may not be present in the evaluation data, learning to ignoring them helps CausalVul to significantly improve the performance.

Table 9: The generalization performance of CausalVul and the vanilla models.

Settings	CodeBERT		GraphCodeBERT		UniXcoder	
	Devign	Big-Vul	Devign	Big-Vul	Devign	Big-Vul
Vanilla	0.11	0.08	0.11	0.10	0.10	0.07
Var1	0.12	0.17	0.11	0.11	**0.12**	0.13
Var2	0.12	**0.18**	0.11	**0.14**	**0.12**	0.15
API1	0.12	0.17	0.11	0.12	**0.12**	0.11
API2	0.12	0.16	0.11	**0.14**	**0.12**	0.18
API3	**0.13**	0.15	0.11	0.10	**0.12**	**0.21**
Var+API	0.12	0.17	0.11	0.12	**0.12**	0.18

Result:RQ2. CausalVul shows up to 62% and 100% improvement in Devign and Big-Vul robustness data respectively. CausalVul also improves the generalization performance up to 100% and 200% for both datasets respectively.

7.3 RQ3: Ablation Studies

Experiment Design. In this RQ, we investigate the design choices for CausalVul. In the first setting, we use K=1 and set $x' = x$ in Algorithm 2. Here, we investigate if we don't use marginalization, how our approach performs. We evaluated the models on the robustness testing data. Due to space, we show the results for the setting of *Var+API*.

In the second setting, we investigated how our approach performs when using different early layers to represent M. Hence, we extract M from the first, second, third, and fourth layers and use that M in our causal approach respectively.

Results. In Figure 7, we used the probability density plots similar to Figure 6. The orange lines plot K=40, and the blue lines plot K=1. From the results, we can clearly see that for both vulnerable and non-vulnerable labels, K=40 learned better and reported more confident predictions towards ground truth labels.

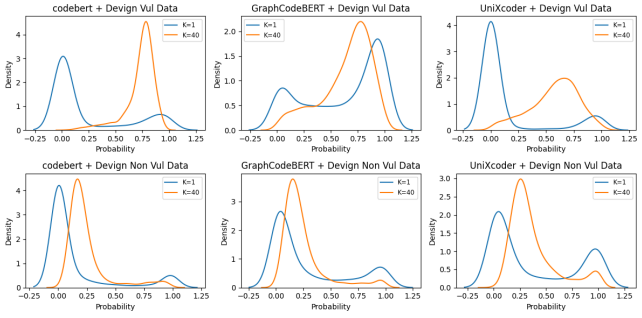

Figure 7: Prediction Probability Density of CausalVul for K=1 and K=40.

Table 10 demonstrates the result of using different early layers to extract M. We choose the *Var+API* settings for all models to present

Table 10: The performance of the Causal Approach when different early layer is used.

Model	Early Layer	Devign (Var+API)	Big-Vul (Var+API)
CodeBERT	1	0.6494	0.4034
CodeBERT	2	0.6528	0.4018
CodeBERT	3	0.6501	0.4026
CodeBERT	4	**0.6546**	**0.4055**
GraphCodeBERT	1	0.6156	0.3916
GraphCodeBERT	2	0.6170	0.3903
GraphCodeBERT	3	0.6363	0.3908
GraphCodeBERT	4	**0.6570**	**0.3927**
UniXcoder	1	0.5728	0.4049
UniXcoder	2	0.5720	0.4055
UniXcoder	3	0.5550	0.4056
UniXcoder	4	**0.6609**	**0.4058**

the result. For all the models and datasets, layer four reported the best performance.

Result:RQ3. Our results show that marginalization (backdoor criterion) helps the model to focus on the causal features instead of spurious features. On the other hand, we found Layer four is better to use to compute M from x' than the early three layers.

8 THREATS TO VALIDITY

To discover spurious features, our experiment design follows the literature [3] and ensures our perturbation follows consistency, naturalness, and semantic-preservation.

Deep learning models may report improved results due to a better random seed. All of our experiments have been run with three random seeds. Our causal models have consistently shown improvement across all the datasets and all the models. We have done a statistical test to show that our improvement is statistically significant. In addition to F1, our probability density plots shown in Figures 6 and 7 also strongly demonstrated our improvement.

The causal learning makes the assumption that the code representation R learned the causal features. Although we are not sure if that's the case, we see the improvement of our results in all settings.

Our evaluation worked on two real-world vulnerability datasets, including both balanced and imbalanced data, and the three SOTA models. In the future, we plan to experiment with more datasets and models.

9 RELATED WORK

Deep learning for Vulnerability: Deep learning vulnerability models can be separated into graph neural-network (GNN) based or transformer-based models. GNN-based models capture AST, control-flow, and data-flow information into a graph representation. Recent GNN-based models [6, 7, 17, 25, 27], have proposed statement-level vulnerability prediction.

In contrast, the transformer models are pre-trained in a self-supervised learning setting. They can be categorized by three different designs: encoder-based, decoder-based [8], and hybrid architectures [3, 4, 15] that combine elements from both approaches. Encoder-based models such as CodeBERT [13, 16] often employ the masked-language-model (MLM) pre-taining objective; some are coupled with a contrastive learning approach [5, 10], while others aim to make pre-training more execution aware [11], bi-modal [4] or naturalized [3]. Vulnerability detection has been one of the important downstream tasks for these models. In this paper, we used three recent SOTA transformer-based models: GraphCodeBERT, CodeBERT, and UniXcoder and show that causality can further improve their performance.

There have been also studies for vulnerability detection models regarding their robustness and generalization. In recent work, Steenhoek et al. [23] evaluated several SOTA vulnerability models to assess their capabilities to unseen data. Furthermore, they touch on spurious features and found some tokens such as "error" and "printf" are frequently used to make predictions, which can lead to mispredictions. In another systematic investigation of deep learning-based vulnerability detection, Chakraborty et al. [7] stated that vulnerability detection models did not pick up on relevant code features, but instead rely on irrelevant aspects from the training distribution (such as specific variables) to make predictions. Our work designed novel perturbations to confirm the hypothesized spurious features, and we found different names are used for different labels as spurious features. None of the existing work have conducted such studies.

Causal learning in SE: To our knowledge, applying causality is relatively new in SE. Cito et al. [9] is the most relevant recent work that investigates perturbations on source code which cause a model to "change its mind". This approach uses a masked language model (MLM) to generate a search space of "natural" perturbations that will flip the model prediction. These natural perturbations are called "counterfactual explanations." Our work also seeks for natural perturbations, and uses variable names and API names in programs to perform the perturbation. However, our work is different in that we require our perturbation to be semantic-preserving. Furthermore, our goal of promoting models to flip their decision is to discover spurious features, instead of explaining the cause of a bug. There has also been orthogonal work that uses counterfactual causal analysis in the context of ML system debugging [18, 26]. Unlike our work, these approaches do not use the backdoor criterion to remove spurious features.

Causal learning in Other Domains: Our work drew inspirations from Mao et al. [20]. This work addresses robustness and generalization in the vision domain using causal learning. They use "water bird" and "land bird" as two domains and show that by applying the backdoor criterion, the causal models can learn "invariants" of the birds in two domains and achieve better generalization. Their work does not discover spurious features, and their causal learning does not target spurious features.

10 CONCLUSIONS AND FUTURE WORK

This paper proposed the first step towards causal vulnerability detection. We addressed several important challenges for deep learning vulnerability detection. First, we designed novel perturbations to expose the spurious features the deep learning models have used for prediction. Second, we formulate the problem of deep learning vulnerability detection using *causality* and *do calculus* so that we

can apply causal learning algorithms to remove spurious features and push the models to use more robust features. We designed comprehensive experiments and demonstrated that CausalVul improved accuracy, robustness and generalisation of vulnerability detection. In the future, we will plan to discover more spurious features and explore the causal learning for other applications in software engineering.

11 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback. This research is partially supported by the U.S. National Science Foundation (NSF) under Award #2313054.

REFERENCES

[1] [n. d.]. Cybercrime To Cost The World $10.5 Trillion Annually By 2025, howpublished =https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/.

[2] [n. d.]. Microsoft Exchange Flaw: Attacks Surge After Code Published, howpublished =https://www.bankinfosecurity.com/ms-exchange-flaw-causes-spike-in-trdownloader-gen-trojans-a-16236.

[3] 2022. NatGen: Generative Pre-training by "Naturalizing" Source Code - Code and scripts for Pre-Training. https://doi.org/10.5281/zenodo.6977595

[4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *2021 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

[5] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations *(SIGIR '21)*. Association for Computing Machinery, New York, NY, USA, 511–521. https://doi.org/10.1145/3404835.3462840

[6] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh PA) *(ICSE '22)*. 1456–1468. https://doi.org/10.1145/3510003.3510219

[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), 1–1. https://doi.org/10.1109/TSE.2021.3087402

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[9] Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, and Satish Chandra. 2022. Counterfactual Explanations for Models of Code. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 125–134. https://doi.org/10.1145/3510457.3513081

[10] Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022. Towards Learning (Dis)-Similarity of Source Code from Program Contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 6300–6312.

[11] Yangruibo Ding, Ben Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2023. TRACED: Execution-aware Pre-training for Source Code. arXiv:2306.07487 [cs.SE]

[12] 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

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[14] 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)*. 608–620. https://doi.org/10.1145/3524842.3528452

[15] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. arXiv:2203.03850 [cs.CL]

[16] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

[17] David Hin, Andrey Kan, Huaming Chen, and M. Ali Babar. 2022. LineVD: Statement-Level Vulnerability Detection Using Graph Neural Networks. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh PA) *(MSR '22)*. 596–607. https://doi.org/10.1145/3524842.3527949

[18] Md Shahriar Iqbal, Rahul Krishna, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. 2022. Unicorn: Reasoning about Configurable System Performance through the Lens of Causality. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 199–217. https://doi.org/10.1145/3492321.3519575

[19] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664* (2021). https://arxiv.org/abs/2102.04664

[20] C. Mao, K. Xia, J. Wang, H. Wang, J. Yang, E. Bareinboim, and C. Vondrick. 2022. Causal Transportability for Visual Recognition. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 7511–7521. https://doi.org/10.1109/CVPR52688.2022.00737

[21] Judea Pearl. 2000. *Causality: Models, reasoning, and inference*.

[22] Judea Pearl and Elias Bareinboim. 2011. Transportability of causal and statistical relations: A formal approach. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 25. 247–254.

[23] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An Empirical Study of Deep Learning Models for Vulnerability Detection. arXiv:2212.08109 [cs.SE]

[24] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2022. ReCode: Robustness Evaluation of Code Generation Models. arXiv:2212.10264 [cs.LG]

[25] Wenbo Wang, Tien N. Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. 2023. DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2249–2261. https://doi.org/10.1109/ICSE48619.2023.00189

[26] Ziyuan Zhong, Zhisheng Hu, Shengjian Guo, Xinyang Zhang, Zhenyu Zhong, and Baishakhi Ray. 2022. Detecting Multi-Sensor Fusion Errors in Advanced Driver-Assistance Systems. 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, 493–505. https://doi.org/10.1145/3533767.3534223

[27] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems*, Vol. 32. 10197–10207.