Reasoning with Counterfactual Explanations for Code Vulnerability Detection and Correction

Anjana Wijekoon[1], Nirmalie Wiratunga[1]

[1]*Robert Gordon University, Aberdeen, Scotland*

Abstract

Counterfactual explanations highlight "actionable knowledge" which helps the end-users to understand how a machine learning outcome could be changed to a more desirable outcome. In code vulnerability detection, understanding these "actionable" corrections can be critical to proactively mitigate security attacks that are caused by known vulnerabilities. In this paper, we present the case-based explainer DisCERN for counterfactual discovery with code data. DisCERN explainer finds counterfactuals to explain the outcomes of black-box vulnerability detection models and highlight actionable corrections to guide the user. DisCERN uses feature relevance explainer knowledge as a proxy to discover potentially vulnerable code statements and then used a novel substitution algorithm based on pattern matching to find corrections from the nearest unlike neighbour. The overall aim of DisCERN is to identify vulnerabilities and correct them with minimal changes necessary. We evaluate DisCERN using the NIST Java SAR dataset to find that DisCERN finds counterfactuals for 96% of the test instances with $13 \sim 14$ statement changes in each test instance. Additionally, we present example counterfactuals found using DisCERN to qualitatively evaluate the algorithm.

Keywords

Counterfactual Explanations, Vulnerability Detection, Explainable AI

1. Introduction

Security attacks that exploit hidden software code flaws pose serious risks that compromise system performance and services. Therefore the ability to detect these vulnerabilities in a timely manner as well as being able to detect potential flaws is a desirable feature that can help to avoid disastrous financial and societal consequences. Application of AI for data-driven vulnerability detection has increased significantly in recent years [1, 2]. This is mainly due to the availability of large amounts of open-source code needed for training vulnerability detection models. Traditional classifiers such as SVM and Naive Bayes [3] as well as neural architectures for sequence modelling (e.g. LSTMs) have been successfully used for code vulnerability classification [4]. Given the textual nature of the data; these classifiers make use of text representation methods from information retrieval [3] as well as deep embedding techniques to represent software code and nodes in the abstract syntax tree [5].

Once vulnerabilities are detected or classified into flaw categories, the software needs to be fixed. Explainable AI (XAI) techniques are used for explaining AI model outcomes and to the

AI-CyberSec 2021: Workshop on Artificial Intelligence and Cyber Security, December 14, 2021, Cambridge, UK

a.wijekoon1@rgu.ac.uk (A. Wijekoon); n.wiratunga@rgu.ac.uk (N. Wiratunga)
0000-0003-3848-3100 (A. Wijekoon); 0000-0003-4040-2496 (N. Wiratunga)

best of our knowledge, only very little work has been done to assist with this code revision phase. Authors of [1] took a factual explanation approach by using their convolutional feature activations to highlight parts of the code that contributed most to the AI model decision. In this paper, we demonstrate how research in counterfactual explanations can be conveniently adapted to generate code correction operators to guide the fixing of vulnerable software segments that are detected by a classifier.

Counterfactual explanations for AI (CAI) have accrued benefits from counterfactual thinking research from psychology and GDPR guidelines for AI [6]. Unlike other forms of post-hoc XAI methods, CAI does not require exposure to the underlying AI model parameters. It instead reasons with the inputs, the outputs and the relationships between these to formulate a situationally (locally) relevant explanation to convey how a "better" or "more desirable" output (outcome) could have been achieved by minimally changing the inputs (situation). Which inputs to change and by how much to change, are interesting research questions that we address in this paper in relation to fixing code vulnerabilities. Here inputs relate to software code and the proposed change relates to the code correction operation. We adapt the CAI algorithm, DisCERN [7], to locate the specific area of vulnerability in a code segment, and to generate a correction at the statement-level using substitution operations. These substitutions are extracted from a code segment that is considered similar to the vulnerable segment. This similar segment needs to be threat-free; and therefore is referred to as the vulnerable segment's nearest unlike neighbour (here we say "unlike", because it differs in terms of the class label i.e. not vulnerable). The idea is to exploit similarities between pairs of code segments, where one of the pairs is considered to be vulnerable and the other is non-vulnerable; and to make use of the non-vulnerable segment to fix the vulnerability. Since we work with similar pairs (proximal property of CAI), we expect to identify minimal code corrections (sparsity property of CAI) to fix the detected vulnerability. Accordingly, this paper makes the following contributions:

- re-purpose feature relevance explainers like LIME [8] to find vulnerabilities in code;
- introduces a substitution algorithm based on pattern matching to correct vulnerable code statements; and
- present DisCERN counterfactual Explainer for code vulnerability correction which brings together knowledge from feature relevance explainers and the substitution algorithm.

The rest of the paper is organised as follows. The introduction of the NIST Dataset and detection of code vulnerabilities using Machine Learning methods is presented in Section 2. Section 3 presents the DisCERN algorithm which discovers counterfactuals for vulnerable code segments and thereby guide the user to correct the vulnerabilities. The evaluation methodologies, performance metrics with quantitative and qualitative results are presented in Section 4, followed by conclusions and future work in Section 5.

2. Vulnerability Detection with the NIST Java Dataset

NIST Software Assurance Reference Dataset (SARD) Project promotes the detection and correction of known security flaws in programming code. They have a publicly available repository of datasets from different programming languages that are labelled for flaws and possible

```java
public void method()
{
    int data;
    /* comment */
    data = (new SecureRandom()).nextInt();
    /* comment */
    int array[] = { 0, 1, 2, 3, 4 };
    /* comment */
    if (data >= 0)
    {
        IO.writeLine(array[data]);
    }
    else
    {
        IO.writeLine("Array index out of bounds");
    }
}
```

(a) Code Segment labelled ’Vulnerable’

```java
public void method()
{
    int data;
    /* comment */
    data = 2;
    /* comment */
    int array[] = { 0, 1, 2, 3, 4 };
    /* comment */
    if (data >= 0 && data < array.length)
    {
        IO.writeLine(array[data]);
    }
    else
    {
        IO.writeLine("Array index out of bounds");
    }
}
```

(b) Code Segment labelled ’Not Vulnerable’

Figure 1: Pre-processed Code Examples

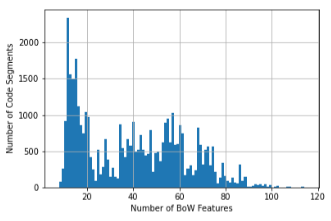
corrections. The flaws are standardised by the community maintained Common Weakness Enumeration (CWE) list which consists of software and hardware weaknesses. In this work, we selected the Java dataset from SARD [1]. Here Java files are grouped under their CWE code and each file contains a class object with one or more methods. For supervised learning we make use of two methods in the code: one with the code vulnerability and the other demonstrating the fix for the code vulnerability.

2.1. Pre-processing Code Segment Data

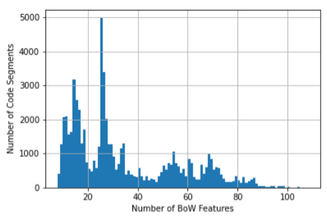
The supervised task involves a binary classification of ’Vulnerable’ or ’Not Vulnerable’ given some code segments. Forming the dataset involved the extraction of code segments and labelling them into one of the two classes and carrying out a masking step to ensure there is no target leak that can influence model training. The steps taken to organise this dataset for classification are as follows:

1. Split methods that contain a vulnerability and those that don’t into two separate files. Java methods were labelled as ’Vulnerable’, if they contain a comment that starts with either ’FLAW’ or ’POTENTIAL FLAW’; and a ’Not Vulnerable’ label is assigned when a comment that starts with ’FIX’ is found.
2. Entity obfuscation steps are applied to each file.
    a) Replace any method name with ’method’
    b) Replace any Parameter type with ’Parameter’
    c) Replace comments with ’/*comment*/’
    d) Change all method signatures to ’public void method()’

[1] https://samate.nist.gov/SARD/testsuite.php



(a) Vulnerable Dataset



(b) Not Vulnerable Dataset

Figure 2: BOW Distributions

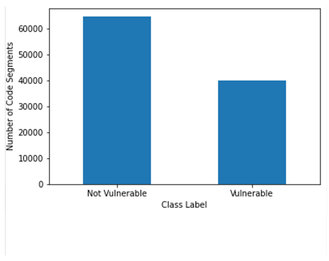


Figure 3: Class Distribution

An example of pre-processed code segments appear in Figures 1a and 1b. Here the "if" statement is used to check the size of data as a fix to the vulnerable code segment.

Using the Bag-of-Word (BoW) vectorisation method we analyse the distribution of the pre-processed dataset over the 1074 unique language tokens (see Figures 2a and 2b). The class distribution of the dataset (see Figure 3) suggests a class imbalance between the 'Vulnerable' and 'Not Vulnerable' data which indicates that not all files have a 'corrected' version of the code.

2.2. Code Vulnerability Classification

The most common Machine Learning (ML) pipeline for classification with text data is to use a Tokenizer followed by a Vectoriser to transform the text data into a vector representation and then apply a classification algorithm to learn from labelled data. Code data can be seen as a text that follows grammar rules defined by the Java Compiler. In this work, the tf-idf feature vector representation is used for classification purposes where each feature value is a real-valued number that denotes the importance of that token for a code-segment instance given the full corpus of code segments. The summary of the final dataset appears in Table 1.

Table 1
Dataset summary

Feature	Value
Number of data instances	104905
Classes	'Vulnerable' or 'Not Vulnerable'
Number of train and test instances (75/25)	78677 / 26226
Number of test instances labelled 'Vulnerable'	10019
Mean number of statements	40.56
Tf-idf features	500

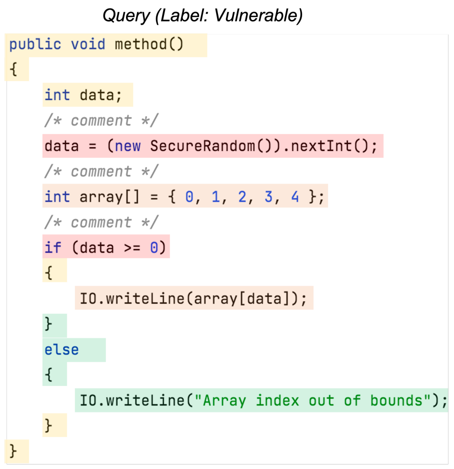
Results from a comparative study of classifiers on the dataset are presented in Table 2. Overall we can see that Random Forest performs extremely well on this dataset and therefore forms the AI model for our explainability evaluation studies. It is worth noting that the CAI method used here is model agnostic and we could use it with any classifier; however it makes sense to work with one that already has high accuracy so that explanations are underpinned by accurate predictions.

Table 2
Classification Algorithms and Performance

Tokenizer	Classifier	Accuracy	Macro-F1	Precision	Recall
Tf-idf	Naive Bayes	0.7206	0.6926	0.6625	0.5478
	k-Nearest Neighbour	0.9387	0.9340	0.9572	0.8788
	SVM	0.9574	0.9544	0.9706	0.9164
	Random Forest	**0.9722**	**0.9704**	**0.9730**	**0.9536**

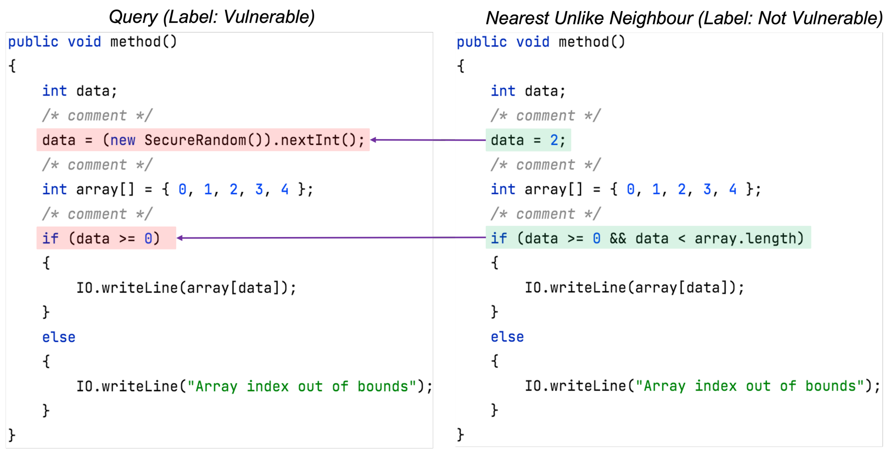
3. Counterfactual XAI for Vulnerability Detection

The use case of code vulnerability detection can benefit from different types of explanations. For example, given a code segment that is labelled "Vulnerable", a Factual Explanation will point to the part of the code segment that led the AI model to label it as "Vulnerable". An example factual explanation is shown in Figure 4 where text highlights indicate "Vulnerable" code and "Not Vulnerable" code in a Red to Green heat map scale. For an expert user this type of explanation should be sufficient given that they have the knowledge to correct the vulnerability. In contrast, a counterfactual explanation will compare the query with a similar yet 'Not Vulnerable' code segment and identify how to correct the vulnerable code segment. A counterfactual example is presented in Figure 5 where the algorithm has highlighted the substitution changes guided by the Nearest Unlike Neighbour. This type of explanation is informative for both expert and non-expert users. In this paper, we focus on generating counterfactual explanations and introduce DisCERN counterfactual algorithm specifically designed for code vulnerability correction. DisCERN was originally proposed in [7] which finds counterfactuals in tabular data.



```
public void method()
{
    int data;
    /* comment */
    data = (new SecureRandom()).nextInt();
    /* comment */
    int array[] = { 0, 1, 2, 3, 4 };
    /* comment */
    if (data >= 0)
    {
        IO.writeLine(array[data]);
    }
    else
    {
        IO.writeLine("Array index out of bounds");
    }
}
```

Figure 4: Factual Explanation


Query (Label: Vulnerable)
Nearest Unlike Neighbour (Label: Not Vulnerable)

```
public void method()
{
    int data;
    /* comment */
    data = (new SecureRandom()).nextInt();
    /* comment */
    int array[] = { 0, 1, 2, 3, 4 };
    /* comment */
    if (data >= 0)
    {
        IO.writeLine(array[data]);
    }
    else
    {
        IO.writeLine("Array index out of bounds");
    }
}
```

```
public void method()
{
    int data;
    /* comment */
    data = 2;
    /* comment */
    int array[] = { 0, 1, 2, 3, 4 };
    /* comment */
    if (data >= 0 && data < array.length)
    {
        IO.writeLine(array[data]);
    }
    else
    {
        IO.writeLine("Array index out of bounds");
    }
}
```

Figure 5: Counterfactual Explanation Example

3.1. DisCERN Counterfactual Explanations

Consider a query code segment x, with m statements where the i^{th} statement is denoted by s_i. If the vulnerability detection pipeline used to predict the code vulnerability consists of a Tokeniser, t, and a classification model, f, the outcome predicted for x is y.

$$\begin{aligned} x &= [s_1, s_2, ..., s_m] \\ y &= f(t(x)) \end{aligned} \tag{1}$$

For a given query x, there are five steps to discovering counterfactuals with DisCERN:

1. find the Nearest Unlike Neighbour (NUN), $\hat{x}$;
2. find the feature relevance weights for the query, x, using the Feature Relevance Explainer LIME [8];
3. given a token, z, in x, find statements pair, i.e. a list of statements in x and a list of candidate statements in $\hat{x}$ as potential vulnerability corrections;
4. create a perturbed code segment, x' by adapting a vulnerability correction and check x' for outcome change using the vulnerability detection pipeline; and
5. repeat steps 3 and 4 until desired outcome is achieved.

Once the perturbed code segment achieves the desired outcome (i.e. not vulnerable), it is identified as the counterfactual of the query. Next, we will explore each of these steps in detail.

3.1.1. Finding the Nearest Unlike Neighbour

Given a query x, the NUN, $\hat{x}$, is the nearest instance found in the train data with a different outcome. In theory, by selecting the NUN as the starting point of counterfactual discovery, we expect to minimise the actionable changes needed to flip the classifier decision. As in Equation 2, $\hat{x}$ has n number of statements and the predicted outcome is $\hat{y}$. Importantly, $\hat{x}$ and x have different number of statements (i.e. $n \neq m$) and different prediction outcomes (i.e. $\hat{y} \neq y$).

$$\begin{aligned}\hat{x} &= [\hat{s}_1, \hat{s}_2, ..., \hat{s}_n] \\ \hat{y} &= f(t(\hat{x})) \mid \hat{y} \neq y\end{aligned} \tag{2}$$

Any encoder which transforms text to a vector representation can be used to model the feature space to find the NUN. We use the Sentence Transformers library [2] to encode code segments. Sentence Transformers Library is a publicly available collection of state-of-the-art sentence encoder models that are based on the Sentence-BERT architectures [9]. The Sentence-BERT architecture uses BERT encoder in a Siamese architecture such that it is trained for similarity comparison. Sentence-BERT can be trained using data from different domains to suit different tasks, however, there is no encoder trained for encoding Java code. Accordingly, we use the generic pre-trained *all-MiniLM-L6-v2* encoder which is trained using a large and diverse dataset training pairs to support multiple domains.

Given a code segment, x, the encoder E generates a vector representation, v, of size l. From the train data set $\mathcal{X}$, we filter data instances for which $y_i \neq y$ and create the subset $\mathcal{X}'$. Each data instance in $\mathcal{X}'$ is encoded using the same encoder E to obtain the set of vectors $\mathcal{V}'$. The cosine similarity between the query x, and any other instance, x_i can be calculated as in Equation 3.

$$cosine(x, x_i) = \frac{\sum_{j=1}^{l} v_{ij} v_j}{\sqrt{\sum_{j=1}^{l} v_{ij}} \sqrt{\sum_{j=1}^{l} v_j}} \tag{3}$$

Once pair-wise similarity is computed (between x and each x_i in $\mathcal{X}'$), we select the train instance x_i from the pair with the highest similarity as the NUN of x. In the rest of this paper, this function is referred to as nn which given, x, $\mathcal{X}'$ and the similarity metric returns $\hat{x}$ as the output.

3.1.2. Finding Feature Relevance Weights

In DisCERN we hypothesise that if parts of the code that contributed most to the current outcome is substituted, we can find the minimal changes needed to flip the class outcome. Here Feature Relevance Explainers can provide the knowledge needed to identify important parts in the code. Accordingly, in this section, we use LIME Feature Relevance Explainer to find the feature relevance weights of the query to identify which parts of the code contributed to the current outcome.

LIME is a model-agnostic feature relevance explainer that creates an interpretable model around a data instance to estimate how each feature contributed to the black-box model outcome [8]. LIME creates a set of perturbations within the instance's neighbourhood and labels them using the black-box model. This newly labelled dataset is used to create a linear interpretable model (e.g. a weighted linear regression model). The resulting surrogate model is interpretable and only locally faithful to the black-box model (i.e. correctly classifies the input instance, but not all data instances outside its immediate neighbourhood). The new interpretable

[2]https://www.sbert.net/

model is used to predict and explain the classification outcome of the data instance. The explanation of the current outcome is formed by obtaining the weights that indicate how each feature contributed to the outcome.

In the context of code segment data, the Tokenizer, t, used in the vulnerability detection pipeline considers language tokens in the code as "features". Accordingly, LIME assigns a weight for each token which indicates how much the token contributes to the classification outcome.

$$LIME(x,t,f) \rightarrow \{w(z) \mid w(z) \in \mathbb{R}, z \in Z\} \tag{4}$$

If the vocabulary of code segments is Z, LIME assigns a weight w for each token z in Z as in Equation 4. A positive weight ($w \geq 0$) indicates that the corresponding token contributes positively and a negative weight ($w < 0$) contributes negatively towards the prediction. The weights are sorted using the partial order condition in Equation 5 to obtain the list of tokens ordered by their contribution towards the current outcome as Z'.

$$z_i \preceq_{\mathcal{R}} z_j \iff \mathcal{R} :: w(z_i) \geq w(z_j) \tag{5}$$

3.1.3. Segment corrections with the substitution operation

Given a language token in the query, the goal of the substitution algorithm is to find a matching set of statements in the query and in the NUN to adapt the query such that it contributes towards a positive outcome change (i.e. from not vulnerable to vulnerable). A Pattern Matching (pm) algorithm is used here to find matching statement blocks as presented in Algorithm 1 for substitution. Since the Feature Relevance Explainers used with DisCERN identify important

Algorithm 1 substitute

```
Require: $x' = [s'_1, s'_2, ..., s'_m]$: perturbed query
Require: $\hat{x} = [\hat{s}_1, \hat{s}_2, ..., \hat{s}_n]$: NUN as a list of statements
Require: $z$: token in the query
1: $S' \leftarrow [s \in x' \mid z \in s]$          ▷ find the list of statements in $x'$ that include $z$
2: for $s_j \in S'$ do
3:     $s'_{ik}, \hat{s}_{gh} \leftarrow pm(s_j, [s'_1, s'_2, ..., s'_m], [\hat{s}_1, \hat{s}_2, ..., \hat{s}_n])$          ▷ find the suggested change
4:     $c_j = cosine(s'_{ik}, \hat{s}_{gh})$          ▷ similarity between current and suggested code
5: end for
6: $(s', \hat{s}') \leftarrow \underset{(s_{ik}, \hat{s}_{gh})}{\arg\max}\ c_j$          ▷ select maximum similarity pair i.e. similar yet corrected
7: $x'' \leftarrow adapt(x', s', \hat{s}')$          ▷ replace $s'$ in $x'$ with $\hat{s}'$
8: return $x''$          ▷ return the newly perturbed query
```

tokens, the first step involves finding the matching list of statements S' in the query that contains the important token z. We use a simple lookup function to identify all code statements in the perturbed query x', that contain the token z (Line 1). The next steps of finding the vulnerable statements and their replacements from NUN are based on the hypothesis that if a statement s_i in S' is *vulnerable*, it must be *corrected* in the NUN. Accordingly, for statements in S', we use the edit changes that are proposed by a pattern matching (pm) algorithm and select the change (based on statement-level similarity) as the most likely substitution to correct the code vulnerability.

A pm algorithm like the Gestalt Patten Matching or Levenshtein Edit Distance can find the minimum changes required to transform one string to another. The changes can be "replace", "insert" and "delete" [3]. We use pm at the statement granularity level to find what changes are proposed for a given statement, s_j. The pm algorithm will return a list of matching pairs of statements to the query (s_{ik}) and list of statements in the NUN ($\hat{s}_{vw}$) where i,k,v,w are start and end indices of statements and s_j is found within s_{ik} (Line 3).

Next, we calculate the similarity between the two lists of statements using Cosine similarity. Similar to Section 3.1.1 we use the *all-MiniLM-L6-v2* encoder to transform the statements to vector representations and calculate Cosine similarity. Once we have all the pairs for S', and their similarities, we select the pair, $(s', \hat{s}')$ that has the maximum similarity. We assume a vulnerable code segment and its corrected counterpart are different yet carry some similarities. Accordingly, by selecting the highest similarity we expect to discard any changes found by pm that are not vulnerability corrections. Finally in Line 7 we replace the list of statements s' in the perturbed query x' with the list of code statement $\hat{s}'$ to return the new perturbed query x''.

3.1.4. DisCERN Counterfactual Discovery

Algorithm 2 DisCERN Algorithm

```
Require: $x = [s_1, s_2, ..., s_m]$: query as a list of statements
Require: $f(t(.))$: vulnerability detection pipeline
Require: $X$: train dataset
Require: $y = f(t(x))$: predicted outcome of the query
 1: $\mathcal{X}' \leftarrow \{x_i \in \mathcal{X} \mid y_i \neq y\}$            ▷ filter the train dataset
 2: $\hat{x} \leftarrow nn(x,\ \mathcal{X}',\ sim)$            ▷ find the NUN using function $nn$
 3: $\{w(z)\} \leftarrow LIME(x,\ t,\ f)$            ▷ the set of weighted tokens from LIME in Equation 4
 4: $Z' \leftarrow \mathcal{R}(\{w(z)\})$            ▷ list of tokens ordered using $\mathcal{R}$ in Equation 5
 5: Initialise $x' = x$ and $y' = y$            ▷ Initialise counterfactual as query
 6: for $z \in Z'$ do
 7:     $x' \leftarrow substitute(x',\ \hat{x},\ z)$            ▷ from Algorithm 1
 8:     $y' = f(t(x'))$            ▷ predict outcome of the perturbed query $x'$
 9:     if $y' \neq y$ then            ▷ check if the outcome is changed
10:         Break            ▷ stop perturbing if outcome is changed
11:     end if
12: end for
13: return $x'$            ▷ return the perturbed query as the counterfactual
```

DisCERN (Algorithm 2) brings together methods from Sections 3.1.1 to 3.1.3 to discover counterfactuals. Given the query x, and the train dataset X, in Lines 1 and 2 we find the NUN

[3]In this paper we used the Gestalt Pattern Matching algorithm implemented by cdifflib Python package https://github.com/mduggan/cdifflib

as discussed in Section 3.1.1. Next, we find the LIME feature weights for the query and use these to sort the list of tokens that indicate which parts of the code contributed to the current outcome (Line 3 and 4). We iterate over the list of tokens where for each token we consider a substitution correction until a prediction outcome is changed. In each iteration for a given token z, Algorithm 1 finds the best matching adaptation to create a new perturbed query x'. It then obtains a prediction to check the outcome class for the adapted x' using the original classification pipeline $f(t(.))$. The iteration is terminated when a change in the outcome is observed and the algorithm returns the perturbed query x' as the counterfactual.

4. Evaluation

This section presents the preliminary evaluation of the counterfactual DisCERN algorithm for vulnerable code correction. To the best of our knowledge, there are no other counterfactual algorithms in the literature for counterfactual discovery for this application. Accordingly, we are not able to compare any performance metrics with other algorithms in this work. Instead, we compare DisCERN algorithm with an ablated version of DisCERN as listed below.

DisCERN-rand considers a random order of statements in Algorithm 2. This is instead of using LIME feature relevance as in Lines 3 and 4. This ablated version evaluates the impact of using feature relevance knowledge to guide efficient counterfactual discovery.

DisCERN: is presented in the Algorithm 2.

4.1. Quantitative Evaluation

DisCERN algorithm is evaluated using the NIST Java dataset. We only use 10019 test data instances that were classified as 'Vulnerable' by the vulnerability detection pipeline in the DisCERN evaluation. Following metrics are used to measure the performance of the algorithm on the NIST Java dataset.

- **Validity:** measures the percentage of data for which DisCERN successfully finds a counterfactual. At this stage, the only requirement for a counterfactual discovered by the DisCERN algorithm is to achieve a "positive" change of outcome [4]. Given the set of test instances that were predicted 'Vunerable' are X_v, and the subset for which DisCERN found a counterfactual is X_v^c, the validity is calculated as in Equation 6. A higher percentage of validity is desirable and unlike in DisCERN for tabular data, with unstructured data (like code), we cannot yet guarantee the validity of the generated code data.

$$Validity = \frac{|X_v^c|}{|X_v|} \times 100 \tag{6}$$

- **Sparsity:** measures the mean number of statements that are changed for a change in outcome. Given the number of statements changed in each test instance in X_v^c is

[4]A more stringent metric would be to evaluate if the change conforms to grammar rules of the Java Compiler, which we will explore in future work.

$[r_1, r_2, ..., r_N]$, where $N = |X^c_v|$, the sparsity is calculated as in Equation 7. In Algorithm 1, number of statement changes for operations "replace", "delete" and "insert" are calculated as $max(k - i, h - g)$, $k - i$ and $h - g$ respectively. Lower value of sparsity is desirable.

$$Sparsity = \frac{1}{N} \sum_{j=1}^{N} r_j \tag{7}$$

In addition to Validity and Sparsity, we also report the mean number of statements in the counterfactual discovered by each algorithm. This is to show the contrast between the mean number of statements in the test instances which is 44.62.

4.1.1. Results

Table 3 presents the quantitative evaluation results of DisCERN using the NIST Java dataset. DisCERN finds counterfactuals for 96.49% of the test instances while the random version only finds counterfactuals for 83.48% (13% less). Sparsity is comparably similar in both algorithms, this means that they both have a comparable number of statement changes $12 \sim 14$ needed to flip the class of the query. DisCERN-rand records a slightly lower sparsity, and this could be explained by the 13% of counterfactuals additionally found by DisCERN having longer code segments (thereby increasing the number of statement changes).

Both DisCERN and DisCERN-rand record comparable values for the mean number of statements in the counterfactuals. In comparison to the mean number of statements in the query, both algorithms find counterfactuals with $4 \sim 6$ more statements. Interestingly, the mean number of statements in the NUNs was 51.81, however, this did not mean that DisCERN completely changed each test instance to its NUN in order to achieve outcome change. We explore this further in qualitative evaluation (Section 4.2).

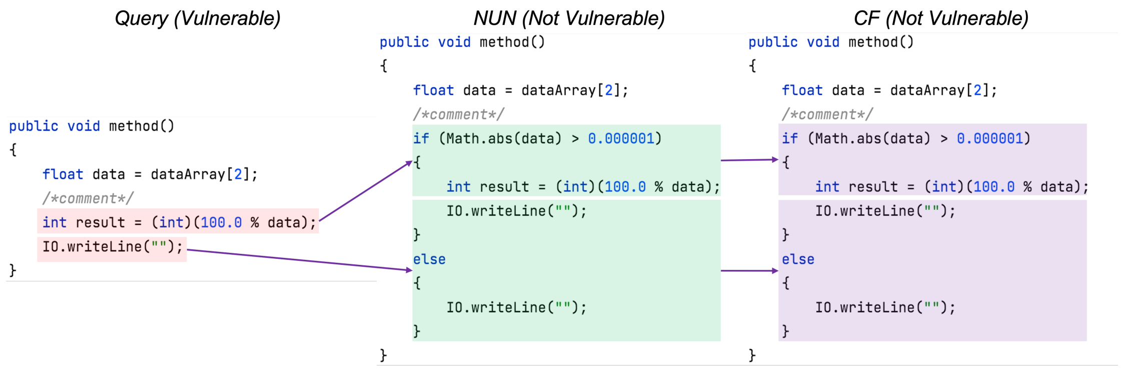
Table 3
Quantitative Results

Algorithm	**Validity (%)**	**Sparsity**	**Mean Number of Statements in Counterfactual**
DisCERN-rand	83.48	12.44	49.28
DisCERN	96.49	13.58	50.62

4.2. Qualitative Evaluation

In the qualitative evaluation we present few examples of counterfactuals found by the DisCERN algorithm. In this preliminary work, we do not check if the counterfactual based code corrections follow grammar rules defined by the Java Compiler. However, we examined a sample of the generated code to closely examine if they are indeed correcting code vulnerabilities and are sensible modifications.

Consider Figures 6a and 6b which presents two counterfactual examples. In each, the query is on the left, the NUN is on the middle and the counterfactual (CF) is on the right. The

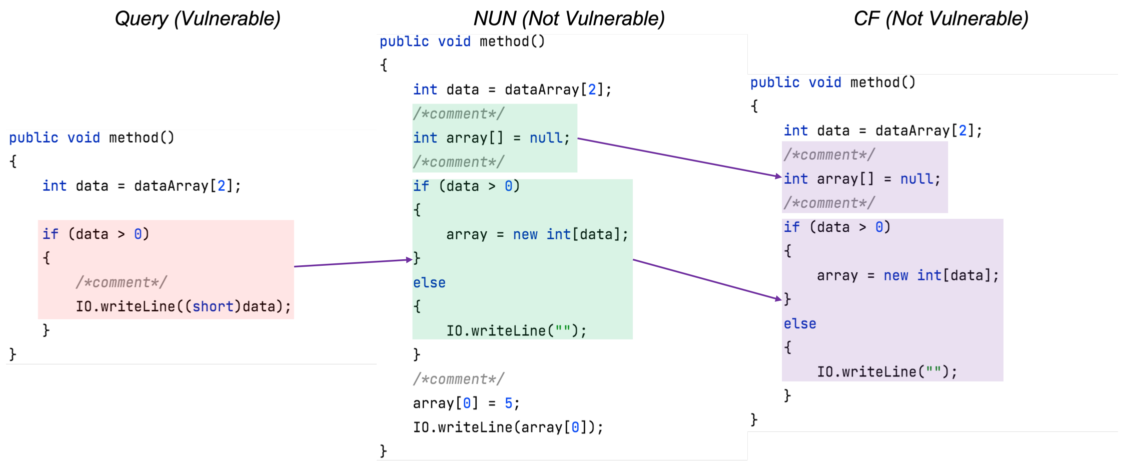


```
public void method()
{
    float data = dataArray[2];
    /*comment*/
    int result = (int)(100.0 % data);
    IO.writeLine("");
}
```

```
public void method()
{
    float data = dataArray[2];
    /*comment*/
    if (Math.abs(data) > 0.000001)
    {
        int result = (int)(100.0 % data);
        IO.writeLine("");
    }
    else
    {
        IO.writeLine("");
    }
}
```

```
public void method()
{
    float data = dataArray[2];
    /*comment*/
    if (Math.abs(data) > 0.000001)
    {
        int result = (int)(100.0 % data);
        IO.writeLine("");
    }
    else
    {
        IO.writeLine("");
    }
}
```

(a) Example 1: Successful Adaptation



```
public void method()
{
    int data = dataArray[2];

    if (data > 0)
    {
        /*comment*/
        IO.writeLine((short)data);
    }
}
```

```
public void method()
{
    int data = dataArray[2];
    /*comment*/
    int array[] = null;
    /*comment*/
    if (data > 0)
    {
        array = new int[data];
    }
    else
    {
        IO.writeLine("");
    }
    /*comment*/
    array[0] = 5;
    IO.writeLine(array[0]);
}
```

```
public void method()
{
    int data = dataArray[2];
    /*comment*/
    int array[] = null;
    /*comment*/
    if (data > 0)
    {
        array = new int[data];
    }
    else
    {
        IO.writeLine("");
    }
}
```

(b) Example 2: Unsuccessful Adaptation

Figure 6: DisCERN Counterfactual Examples

adaptations are highlighted with coloured boxes and arrows. Example in Figure 6a shows how feature relevance weights have guided the algorithm to select correct adaptations to perturb the query into a counterfactual that is similar to NUN and achieved the class outcome change. In contrast, the example in Figure 6b shows that in some cases, the adaptation does not result in grammatically correct counterfactuals. In such cases, the vulnerability detection pipeline may recognise the perturbed query as “Not Vulnerable” but the adaptations applied to the code have lost some of the original functionality. Such examples suggests, that DisCERN is a promising approach to discovering counterfactuals, however it requires further adaptation heuristics and code correction knowledge to ensure accurate code corrections from counterfactuals.

5. Conclusion

In this paper, we presented a novel approach to finding counterfactual explanations for correcting vulnerabilities in code. We used feature relevance explainer knowledge as a proxy to discovering potentially vulnerable code statements and then used a novel substitution algorithm based on pattern matching to find corrections from the nearest unlike neighbour. Overall aim of our algorithm was to identify vulnerabilities and correct them with minimal necessary changes. We evaluated our algorithm using the NIST Java SAR dataset to find that DisCERN finds counterfactuals 96% of the time with $13 \sim 14$ statement changes to the query. Additionally, we presented example counterfactuals found using DisCERN to qualitatively evaluate the algorithm. These suggest that further correction operations and heuristics are needed to ensure plausible code changes. Future work will also expand upon our evaluation to include additional SAR datasets in different programming languages and the use of qualitative evaluation through crowd-sourcing techniques.

Acknowledgments

This research is funded by the iSee project (https://isee4xai.com) which received funding from EPSRC under the grant number EP/V061755/1. iSee is part of the CHIST-ERA pathfinder programme for European coordinated research on future and emerging information and communication technologies. Authors would like to thank Dr. Harsha Kalutarage and Ashish Upadhyay at the Robert Gordon University for their enlightening discussions.

References

[1] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, M. McConley, Automated vulnerability detection in source code using deep representation learning, in: 2018 17th IEEE international conference on machine learning and applications (ICMLA), IEEE, 2018, pp. 757–762.

[2] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, L. Karaçay, Vulnerability prediction from source code using machine learning, IEEE Access 8 (2020) 150672–150684.

[3] B. Chernis, R. Verma, Machine learning methods for software vulnerability detection, in: Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics, 2018, pp. 31–39.

[4] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, A. Ghose, Automatic feature learning for vulnerability prediction, arXiv preprint arXiv:1708.02368 (2017).

[5] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, H. Jin, A comparative study of deep learning-based vulnerability detection system, IEEE Access 7 (2019) 103184–103197.

[6] S. Wachter, B. Mittelstadt, C. Russell, Counterfactual explanations without opening the black box: Automated decisions and the gdpr, Harv. JL & Tech. 31 (2017) 841.

[7] N. Wiratunga, A. Wijekoon, I. Nkisi-Orji, K. Martin, C. Palihawadana, D. Corsar, Discern: Discovering counterfactual explanations using relevance features from neighbourhoods, arXiv preprint arXiv:2109.05800 (2021).

[8] M. T. Ribeiro, S. Singh, C. Guestrin, " why should i trust you?" explaining the predictions of any classifier, in: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, 2016, pp. 1135–1144.

[9] N. Reimers, I. Gurevych, Sentence-bert: Sentence embeddings using siamese bert-networks, arXiv preprint arXiv:1908.10084 (2019).