WIJEKOON, A. and WIRATUNGA, N. 2023. A user-centred evaluation of DisCERN: discovering counterfactuals for code vulnerability detection and correction. *Knowledge-based systems* [online], In Press, article 110830. Available from: https://doi.org/10.1016/j.knosys.2023.110830

A user-centred evaluation of DisCERN: discovering counterfactuals for code vulnerability detection and correction.

WIJEKOON, A. and WIRATUNGA, N.

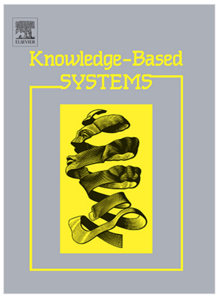
2023

OpenAIR @RGU

This document was downloaded from https://openair.rgu.ac.uk

CC BY

Journal Pre-proof



A user-centred evaluation of DisCERN: Discovering counterfactuals for code vulnerability detection and correction

Anjana Wijekoon, Nirmalie Wiratunga

PII: S0950-7051(23)00580-4
DOI: https://doi.org/10.1016/j.knosys.2023.110830
Reference: KNOSYS 110830

To appear in: *Knowledge-Based Systems*

Received date : 3 April 2023
Revised date : 13 June 2023
Accepted date : 20 July 2023

Please cite this article as: A. Wijekoon and N. Wiratunga, A user-centred evaluation of DisCERN: Discovering counterfactuals for code vulnerability detection and correction, *Knowledge-Based Systems* (2023), doi: https://doi.org/10.1016/j.knosys.2023.110830.

This is a PDF file of an article that has undergone enhancements after acceptance, such as the addition of a cover page and metadata, and formatting for readability, but it is not yet the definitive version of record. This version will undergo additional copyediting, typesetting and review before it is published in its final form, but we are providing this version to give early visibility of the article. Please note that, during the production process, errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

© 2023 Published by Elsevier B.V.

Journal Pre-proof

Credit Author Statement

Anjana Wijekoon: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data Curation, Writing - Original Draft, Writing - Review & Editing, Visualization
Nirmalie Wiratunga: Conceptualization, Methodology, Resources, Writing - Original Draft, Writing - Review & Editing, Supervision, Project administration, Funding acquisition

Journal Pre-proof

Journal Pre-proof

Revised Manuscript (Clean Version) Click here to view linked References

A User-centred Evaluation of DisCERN: Discovering Counterfactuals for Code Vulnerability Detection and Correction

Anjana Wijekoon, Nirmalie Wiratunga[a]

[a]School of Computing, Robert Gordon University, Aberdeen, UK

Abstract

Counterfactual explanations highlight *actionable knowledge* which helps to understand how a machine learning model outcome could be altered to a more favourable outcome. Understanding *actionable* corrections in source code analysis can be critical to proactively mitigate security attacks that are caused by known vulnerabilities. In this paper, we present the DisCERN explainer for discovering counterfactuals for code vulnerability correction. Given a vulnerable code segment, DisCERN finds counterfactual (i.e. non-vulnerable) code segments and recommends actionable corrections. DisCERN uses feature attribution knowledge to identify potentially vulnerable code statements. Subsequently, it applies a substitution-focused correction, suggesting suitable fixes by analysing the nearest-unlike neighbour. Overall, DisCERN aims to identify vulnerabilities and correct them while preserving both the code syntax and the original functionality of the code. A user study evaluated the utility of counterfactuals for vulnerability detection and correction compared to more commonly used feature attribution explainers. The study revealed that counterfactuals foster positive shifts in mental models, effectively guiding users toward making vulnerability corrections. Furthermore, counterfactuals significantly reduced the cognitive load when detecting and correcting vulnerabilities in complex code segments. Despite these benefits, the user study showed that feature attribution explanations are still more widely accepted than counterfactuals, possibly due to the greater familiarity with the former and the novelty of the latter. These findings encourage further research and development into counterfactual explanations, as they demonstrate the potential for acceptability over time among developers as a reliable resource for both coding and training.

Journal Pre-proof

Journal Pre-proof

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 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 structured 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 [5].

Once vulnerabilities are detected or classified into flaw categories, the software needs to be fixed. Feature attribution methods enhance the transparency of AI model decisions by revealing the underlying reasoning for classifying a code segment as vulnerable. It assigns a weight to each token of the code which indicates how much it contributed to the AI model prediction (See examples in Figure 5). For example, authors of [1] used the feature activation map of their convolutional neural model to highlight parts of the code that contributed most to the AI model decision. Similarly authors of [6] used LIME to highlight the contribution of code tokens towards vulnerability. The methods introduced in this paper address a gap in the current approaches by focusing not only on identifying vulnerabilities but also on providing corrections as a solution. Here we demonstrate how research in counterfactual explanations can be conveniently adapted to generate code correction operators to guide the fixing of vulnerable code segments that are detected by a classification model.

Counterfactual Explanations for AI have accrued benefits from counterfactual thinking research from Psychology and GDPR guidelines for AI [7]. Counterfactuals reason with the inputs, the outputs, and the relationships between these to formulate a locally relevant explanation to convey how a

Journal Pre-proof

better or *more desirable* output (AI model decision) could have been achieved by minimally changing the inputs. Questions concerning which part of the input to modify and the appropriate methods for implementing such changes to rectify code vulnerabilities are addressed in this paper. Here the input is code segments and the proposed change relates to the code correction operation. We present the DisCERN [8] algorithm, to locate the specific area of vulnerability in a code segment, and to generate statement-level corrections using substitution operations. In contrast to previous work where DisCERN was employed for identifying substitutions using similarity calculations on tabular data, in this paper, substitutions are derived from code snippets deemed similar but *non-vulnerable*. This is achieved by exploiting similarity-driven pattern matching of pairs of code segments.

The utility of explanations in code vulnerability detection and correction is best evaluated by the target users (i.e. developers). Accordingly, a user study is performed to compare the effectiveness of counterfactuals from DisCERN in comparison to feature attribution explanations from LIME. The goal is to understand how counterfactuals and feature attributions differ in the application of code vulnerability detection and correction in terms of shaping mental models, affecting cognitive load and explanation goodness and acceptability.

This paper makes the following contributions:

- introduces the DisCERN Counterfactual Explainer as a tool for code vulnerability correction leveraging knowledge from feature attribution explainers and pattern matching to make correction recommendations (Section4);
- demonstrates the generalisability of DisCERN across multiple programming languages in terms of validity and sparsity metrics (Section 5); and
- establishes the effectiveness of counterfactuals compared to feature attribution explanations for vulnerability detection and correction in a user study (Section 6).

The rest of the paper is organised as follows. Section 2 discusses the related work on vulnerability detection as a Machine Learning (ML) task and correction from the view of XAI. The introduction of the NIST Datasets and detection of code vulnerabilities using ML methods is presented in Section 3.

Journal Pre-proof

Section 4 presents the DisCERN algorithm which discovers counterfactuals for vulnerable code segments and thereby guides the user to correct these vulnerabilities. The empirical evaluation and performance metrics with quantitative and qualitative results are presented in Section 5. Section 6 presents the user study that compares the utility of counterfactual vs feature attribution explanations. Finally, we draw conclusions in Section 7.

2. Related Work

2.1. Code Vulnerability Detection

The conventional approach to Code Vulnerability Detection (CVD) involved software and security experts auditing a software system for potential security defects, bugs and weaknesses all of which are referred to as vulnerabilities [9]. Automation of vulnerability detection of code is an active applied research area where ML techniques are used for CVD [10, 11]. Early ML methods for CVD focused on optimising feature extraction techniques while neural network-based methods were used to learn semantic knowledge from unstructured code to detect vulnerabilities [11]. Most recently, recurrent networks [12], graph neural networks [13] and transformer-based language models [14, 15, 16] have been used for learning feature embeddings from code for CVD. Many reviews in this research area provide comprehensive overviews of ML techniques for CVD while emphasising the scarcity of explainability approaches [10, 17]. XAI can be harnessed to support CVD in multiple ways. For instance, it can help explain how the model works, identify the key features or variables that contribute to the detection process, and provide insights into how to improve code and reduce vulnerabilities by engaging humans in the loop. In this paper, we propose using the DisCERN algorithm as a credible approach to address these issues.

2.2. Code Vulnerability Correction

The conventional approaches to providing users with corrective feedback include rule-based [18, 19] and template-based approaches [20, 21]. Authors of [18] proposed to pre-configure corrections for specific vulnerabilities and reuse them as vulnerabilities are detected by their ensemble model in PHP code. Similarly, authors of [19] use pre-configured vulnerability matching rules and correction patterns for Java cryptography API code. Alternatively, sequence-to-sequence models have been trained to generate corrections [22].

However, they are limited to a single programming language (C/C++) and a vulnerability group (Buffer-overflow).

Our method is more closely related to work in [20] and [21] where the methodology makes use of vulnerable and non-vulnerable code pairs to find exemplar corrections. For each vulnerability in the code pair, they calculate edit operations and cluster them to find correction patterns. Discovered patterns are saved as templates to reuse on new vulnerable code segments. Their method captures a wider variety of corrections by identifying multiple correction patterns per vulnerability group. These methods share the same challenge as DisCERN which is once a correction example (in DisCERN) or a template(others) is found, how to adapt it to match the target code. DisCERN addresses this by selecting the corrections from the nearest unlike neighbour, which does not always guarantee perfect adaptation. Template-based methods apply knowledge-intensive post-processing steps (such as correcting variable names to match target code) that are not generalisable to different languages and vulnerabilities.

The main difference between existing work and ours is that DisCERN is generating the corrections to explain the prediction of an AI model (explaining the decision). Conversely, previous methods consider correction generation to be an independent task and require a detection model that classifies the exact vulnerability group. The difference is that DisCERN corrections are guided by the knowledge encapsulated in the AI model such as what features/tokens contributed to the decision. DisCERN is also not reliant on expert knowledge and heavily data-driven making it agnostic to the detection-model and the programming-language. It also simplifies the task of the detection AI model from a multi-class classification (up to 100+ classes) problem to a binary-classification problem as the explainer does not require the exact vulnerability group.

2.3. Explainable AI in Vulnerability Detection

Research literature and regulatory guidelines emphasise the necessity for explanations of ML model decisions, as ML methods have increasingly become more opaque and difficult to interpret [23, 24]. This applies to code vulnerability detection and specifically towards prevention and or mitigation. Feature attribution explainers have been explored as a way to pinpoint code lines or segments that may have contributed to a *vulnerable* prediction by an ML algorithm. Authors of [25] describe the design of a human-in-the-loop

XAI system for vulnerability mitigation, whereby model predictions are explained to forensic experts by way of feature attributions to enable them to make necessary corrections. Authors of [26] explore the explanation needs of target user groups of a code analyser to recognise two: a global explanation where the common behaviours of the tool are explained; and a local explanation where feature attribution explains why a specific code snippet is predicted to be vulnerable. Both explanations are targeted towards a knowledgeable audience of ML engineers. There are other works in similar areas such as malware labelling in Android applications [27] and predicting phishing URLs [28] that also make use of feature attribution explanations. Authors of [6] used LIME to explain vulnerability detection in C/C++ code when using the Bidirectional LSTM model named VulDeePecker [12]. This paper addresses a key gap in the literature by proposing the use of counterfactuals not only for explaining detection but also for correcting vulnerabilities. Accordingly, [6] is the most directly linked previous work we compared against DisCERN in our user study.

2.4. Explainable AI Techniques

Although there exists a broad range of explanation techniques and types [29] our main emphasis is on factual and counterfactual explanations. The factual explanation often answers the "what" or "why" questions by providing empirical evidence to support a particular AI model outcome based on the input provided [30]. This evidence can take the form of feature attribution where each input feature is assigned an attribution towards the outcome or example-based explanations where nearest neighbours are used to support the outcome. In contrast, counterfactuals answer "Why-not" or "How-to" questions by formulating a hypothetical scenario that has a *more desirable* outcome [30]. In code vulnerability detection and correction, a factual explanation would highlight where the vulnerabilities exist within the code, while a counterfactual explanation would help to demonstrate how to correct said vulnerabilities. In this study, we investigate the use of the DisCERN algorithm for discovering counterfactual explanations and evaluate its effectiveness through a user study. The user study involves participants with varying levels of expertise in code vulnerability detection and correction, allowing us to assess the utility of the algorithm in a range of contexts.

Journal Pre-proof

```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) Label: 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) Label: Non-vulnerable

Figure 1: Pre-processed code segments from the Java dataset

3. Vulnerability Detection with NIST SAR Datasets

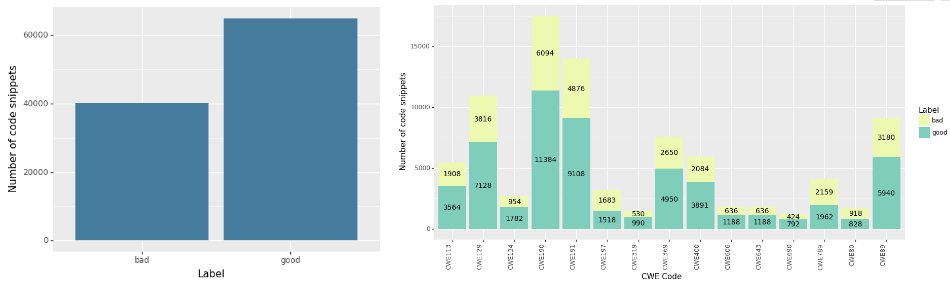
NIST Software Assurance Reference Dataset (SARD) Project promotes the detection and correction of known security flaws in programming code. The project maintains a publicly available repository of datasets from different programming languages that are labelled for flaws and possible corrections. The flaws are standardised by the Common Weakness Enumeration (CWE) list which consists of software and hardware weaknesses. In this work, we consider three datasets in Java, C and C# programming languages from the NIST test suite [1].

3.1. Preprocessing and Dataset Creation

In each dataset, code files are grouped under their CWE code and each file contains one or more functions (or methods in Java and C#). One function is *vulnerable* and often the remaining function is a proposed correction (i.e. non-vulnerable). We apply the following pre-processing steps to prepare each dataset for a binary-classification task:

1. Split functions in a file that are *vulnerable* and those *non-vulnerable* into individual data instances. An instance (i.e. function) was labelled

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



(a) Class distribution

(b) Class distribution by CWE code

Figure 2: Java dataset statistics

vulnerable if it contains one or more comments that start with either *FLAW* or *POTENTIAL FLAW* and labelled as *non-vulnerable* if only contains comments that start with *FIX*.

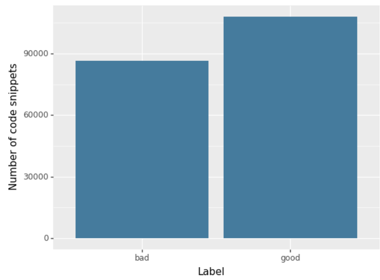
2. Apply the following entity obfuscation steps to each function with the aim to prevent target leakage:
    (a) replace all comments with /*comment*/; and
    (b) change all function signatures to public void method() (or language appropriate alternative).

Figure 1 presents two code segments from the Java dataset that were similar, one labelled as vulnerable and the other as non-vulnerable.

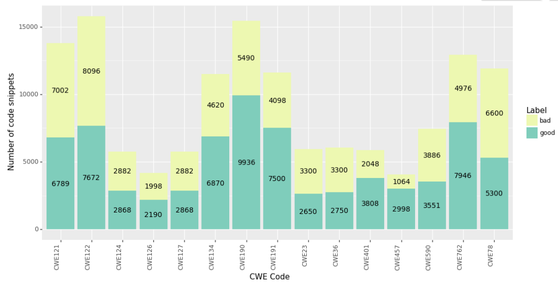
We present a detailed analysis of the class distribution of each dataset in Figures 2, 3 and 4. The left figure (Figure a) of each dataset shows that there are more *non-vulnerable* instances compared to *vulnerable* instances. Figure b on the right provides further analysis, examining the most frequent CWE codes (top 15) and the proportion of *vulnerable* and *non-vulnerable* instances for each code. Notably, there are no *non-vulnerable* examples for some CWE codes (example C# codes CWE313 and CWE94).

3.2. Vulnerability Classification

Code data can be seen as a text that follows grammar rules defined by the respective Compiler. The most common Machine Learning (ML) pipeline for classification with text data is to use a Tokenizer (t) to transform the text data into a vector representation and then apply a classification algorithm (f) to learn from labelled data. In this work, we consider several standard vector representations and classifier combinations to compare the performance

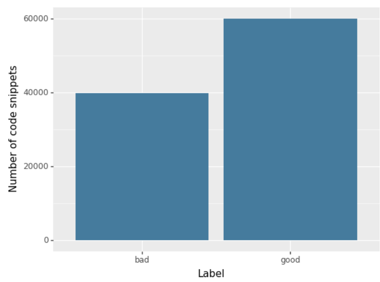


(a) Class distribution

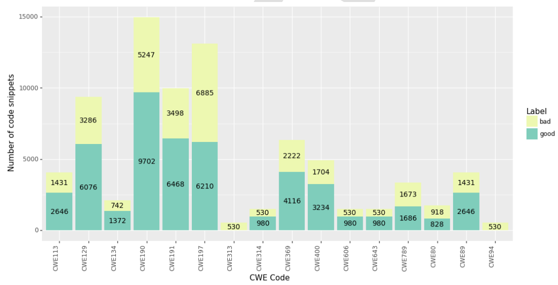


(b) Class distribution by CWE code

Figure 3: C dataset statistics



(a) Class distribution



(b) Class distribution by CWE code

Figure 4: C# dataset statistics

of commonly used black box models that detect vulnerabilities in code segments. We use 75/25 class stratified split to create 4 folds. For each fold, we train the model with 75% of the data and test with the remaining 25%. Table 1 presents the mean F1-score averaged across the four folds.

Overall we observe that *BoW + Random Forest* achieves the best performance for Java and C# datasets while CodeBERT classifier performs best for the C dataset. It is noteworthy that the contributions of this paper are model-agnostic, meaning that any combination of t and f should work with DisCERN, including the most recent encoders such as CodeBERT [31]. Accordingly, the focus of the paper is not on identifying the best classification model, but rather to identify a model that performs well for experimental purposes. Accordingly, XAI evaluations in Section 5 used the *BoW + Random Forest* as the detection pipeline for all three datasets. This allowed for fairness and consistency across experiments and helped to observe the impact

Table 1: Classification Algorithms and Performance

Tokenizer (t)	**Classifier** (f)	**Dataset** **Java**	**C**	**C#**
Tf-idf	Naive Bayes	0.7206	0.7284	0.7783
	kNN	0.9387	0.8457	0.9494
	SVM	0.9574	0.8839	0.9723
	Random Forest	0.9722	0.8734	0.9844
BoW	Random Forest	**0.9761**	0.8790	**0.9889**
CodeBERT-base Tokerniser	CodeBERT classifier	0.9469	**0.9484**	0.9880

of classification performance on the counterfactual generation.

4. DisCERN Counterfactuals for Vulnerability Detection and Correction

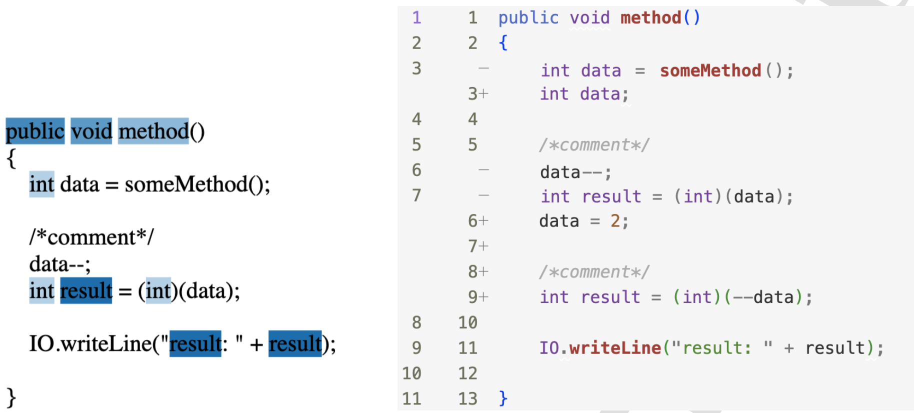
Code vulnerability detection decisions can be explained using different types of explanations. As discussed in Section 2, it is commonly explained using a factual explanation that uses feature attributions to explain the decision and it is often targeted to knowledgeable users. Given a code segment that is labelled *vulnerable*, a factual explanation will point to the part of the code segment which led the AI model to label it as *vulnerable*. An example factual explanation is shown in Figure 5a where text highlights indicate *vulnerable* and *non-vulnerable* tokens in a Blue to Orange heat map scale. For an expert, this type of explanation should be sufficient as they have the knowledge to correct the vulnerability. In contrast, a counterfactual explanation in Figure 5b will compare the given code segment with a similar yet *non-vulnerable* code segment and make recommendations on how to correct the vulnerability. Accordingly we argue that counterfactual explanations are more informative for both expert and non-expert users, and in support of this claim, we present the DisCERN algorithm for generating counterfactual explanations specifically for code vulnerability correction.

4.1. Problem Definition

Consider a query code segment x, with m number of statements where the i^{th} statement is denoted by s_i. If the vulnerability detection pipeline used to

Journal Pre-proof

Journal Pre-proof



(a) Factual Explanation (b) Counterfactual Explanation

Figure 5: Examples of feature attribution and counterfactual explanations

predict the code vulnerability consists of a Tokeniser, t, and a classification model, f, the decision 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, having prediction, $y =$*vulnerable*, there are four steps to discovering *non-vulnerable* counterfactuals with DisCERN:

1. find the Nearest Unlike Neighbour (NUN), $\hat{x}$ from the train dataset $\mathcal{X}$;
2. for each token z in x,find the attribution weights, using a feature attribution explainer (in this work we use LIME);
3. given a vulnerable token, z, in x, find statements pairs for correction, i.e. a list of statements in x and a list of candidate statements in $\hat{x}$ as a potential vulnerability correction;
4. create an updated code segment, x', by adapting the vulnerability correction and check x' for decision change using the vulnerability detection pipeline; and
5. repeat steps 3 and 4 until the detection pipeline predicts *non-vulnerable.*

Once the adapted code segment achieves the desired decision (i.e. non-vulnerable), it is identified as the counterfactual of the query. Next, we will explore each of these steps in detail.

4.2. 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 decision or label. In the context of counterfactual discovery, our query x is *vulnerable*. Selecting the NUN as the starting point, we expect 1) to minimise the actionable changes needed to flip the prediction i.e. with as few changes as possible; and 2) to preserve the original functionality of the code segment while correcting vulnerabilities. As in Equation 2, $\hat{x}$ has n number of statements and the prediction is $\hat{y}$. Importantly, $\hat{x}$ and x can have different number of statements (i.e. $n \neq m$) and should have different decisions (i.e. $\hat{y} \neq y$).

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

To find the NUN by similarity, it is necessary to use an encoder (E) to transform code segments into a vector representation. This work used CodeBERT [31] to encode code segments. CodeBERT is based on the BERT [32] architecture and is state-of-the-art for natural language code search and code generation. It supports multiple programming languages making it most suited for this task. More specifically, we use the pre-trained weights from *codebert-base* shared in the Hugging Face repository [2] which is trained using bi-modal data (consisting of the code and its natural language description as two modalities) from CodeSearchNet.

Given a code query, x, the encoder E generates a vector representation, v, where the standard *codebert-base* encoding length, l, is 768 (Equation 3). From the train data set $\mathcal{X}$, we filter data instances for which $y_i \neq y$ and create the subset $\mathcal{X}'$. $\mathcal{X}'$ represents all the *non-vulnerable* code segments that can be used to find a nearest-unlike-neighbour for x. Each data instance in $\mathcal{X}'$ is encoded using the encoder E to obtain the set of vectors $\mathcal{V}'$. We use cosine similarity to find the NUN due to its robustness in comparing high-dimensional data, and its output range of -1 to 1 allows for a clear interpretation of similarity scores. We compute the cosine similarity between the query x, and any other instance, x_i as in Equation 3.

[2]https://huggingface.co/microsoft/codebert-base

Journal Pre-proof

$$v = E(x) \text{ and } v \in \mathbb{R}^l$$
$$cosine(x, x_i) = \frac{\sum_{j=1}^{l} v_{ij} v_j}{\sqrt{\sum_{j=1}^{l} v_{ij}}\sqrt{\sum_{j=1}^{l} v_j}} \tag{3}$$

Once the 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, query, x, train subset, $\mathcal{X}'$ and the similarity metric, returns the NUN, $\hat{x}$.

4.3. Finding Feature Attribution Weights

Building upon counterfactual reasoning, DisCERN uses feature attribution to reveal the most important code tokens or segments that contribute to an outcome of *vulnerable*. By selectively substituting only these segments, DisCERN can then identify the minimum changes needed to reverse that decision. The feature attribution explainers can provide the knowledge needed for identifying the code segments that need to be substituted. Accordingly, without loss of generalisability, this section describes the use of LIME explainer to find feature attributions of the query to identify which parts of the code had contributed to it being labelled as *vulnerable*.

LIME is a model-agnostic feature attribution explainer that creates an interpretable model around a data instance to estimate how each feature contributed to the black-box model outcome [33]. LIME creates a set of perturbations within the query neighbourhood and labels them using the black-box model. This newly labelled dataset is used to create a linear interpretable model (e.g. a 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 model is used to explain the black-box model outcome of the query. The explanation is formed by obtaining the linear model coefficients that indicate how each feature contributed to the outcome.

Our selection of LIME as the feature attribution explainer is motivated by the evidence from the literature. Authors of [6] proposed the use of LIME in the code vulnerability detection domain. Their evaluation demonstrated that the attributions correctly identify tokens that cause vulnerabilities. When applying LIME in the context of code segment data, the *features* are the tokens identified by the Tokenizer, t, in the vulnerability detection pipeline. Accordingly, LIME can be used to understand the outcome of $f(t(x))$, by assigning an attribution, w, to each token which indicates how much the token contributes to the outcome.

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

If the vocabulary of code segments is Z, LIME assigns a weight w for each token $z \in Z$ (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 outcome. We sort the weights using the partial order condition, $\mathcal{R}$, in Equation 5 to obtain the sorted list of tokens ordered from highest to lowest contribution towards the *vulnerable* outcome as Z'.

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

4.4. Substitution Algorithm

Given a token, z, in the query code segment, the goal of the substitution algorithm is to find a matching list of statements in the query and respective matches in the NUN to adapt the query such that it leads to a changed decision (i.e. vulnerable to non-vulnerable). To the best of our knowledge, existing feature attribution explainers identify the importance of tokens instead of code statements or segments. Instead of modifying the generic feature attribution explainers to operate at the statement level, we use a post-processing step to find the matching statements in the query that contains the token z, followed by a Pattern Matching (pm) algorithm to find matching lists of statements as presented in Algorithm 1. This allows for flexibility and compatibility of DisCERN with various existing attribution explainers.

We use a simple lookup function to identify all code statements (S') in the (adapted) query x', that contain the token z (Line 1). The next steps (Lines 2- 5) of finding the vulnerable statements and their replacements from NUN are based on the hypothesis that if a statement s_j in S' is *vulnerable*, it must be *corrected* in the NUN. Accordingly, for a statement, s_j, in S', first, we use a Pattern Matching algorithm to find a matching list of statements $s'_{[i:j]}$ from x' and $\hat{s}_{[v:w]}$ from $\hat{x}$. Here, the subscripts indicate the start and end indices of the list of statements and s_j is found within

Journal Pre-proof

Journal Pre-proof

Algorithm 1 substitute

Require: $x' = [s'_1, s'_2, ..., s'_m]$: (adapted) 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: $\quad s'_{[i:k]}, \hat{s}_{[v:w]} \leftarrow pm(s_j, [s'_1, s'_2, ..., s'_m], [\hat{s}_1, \hat{s}_2, ..., \hat{s}_n])$
4: $\quad c_j = cosine(E(s'_{[i:k]}), E(\hat{s}_{[v:w]}))$ ▷ calculate similarity
5: **end for**
6: $(s', \hat{s}) \leftarrow \underset{(s'_{[i:k]}, \hat{s}_{[v:w]})}{\arg\max}\ c_j$ ▷ select maximum similarity pair
7: $x' \leftarrow replace(x', s', \hat{s})$ ▷ replace s' in x' with $\hat{s}'$
8: **return** x' ▷ return the newly adapted query

$s'_{[i:k]}$ (Line 3). A pattern-matching algorithm like the Gestalt Pattern Matching or Levenshtein Edit Distance can find the changes required to transform one string to another where the types of edits are *replace*, *delete* and *insert*. This paper used the Gestalt Pattern Matching algorithm implemented by cdifflib Python package [3]. We consider consecutive lists of statements rather than individual statements to preserve the grammatical structure of the programming language as closely as possible.

Next, we calculate the similarity between the two lists of statements using Cosine similarity (Line 4). Similar to Section 4.2 we use the *codebert-base* encoder to transform the list of statements to a vector representation and calculate the cosine similarity. Once we have all the $(s'_{[i:k]}, \hat{s}_{[v:w]})$ pairs, and their similarities, c_j, we select the pair, $(s', \hat{s})$, that has the maximum similarity (Line 6). We assume a vulnerable code segment and its corrected counterpart are different yet carry some similarities. Accordingly, by selecting the pair with the highest similarity from the remaining, we expect to discard those suggested by *pm* that are not vulnerability corrections. Note that *pm* only returns edit operations, not exact matches, hence the similarity score between a pair is always < 1. Finally, in Line 7 we replace the list of statements s' in x' with the list of statement $\hat{s}$ to return the new adapted query.

[3] https://github.com/mduggan/cdifflib

Journal Pre-proof

4.5. DisCERN Algorithm

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: sim: similarity metric, default is cosine similarity
Require: $\mathcal{X}$: train dataset
Require: $y = f(t(x))$: black-box prediction for 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
3: $\{w(z)\} \leftarrow LIME(x,\ t,\ f)$ ▷ feature attributions
4: $Z' \leftarrow \mathcal{R}(\{w(z)\})$ ▷ tokens sorted by $\mathcal{R}$
5: **Initialise** $x' = x$ and $y' = y$
6: **for** $z \in Z'$ **do** ▷ for each token in the sorted list
7: $\quad x' \leftarrow substitute(x',\ \hat{x},\ z)$ ▷ Algorithm 1
8: $\quad y' = f(t(x'))$ ▷ predict decision for the adapted query x'
9: $\quad$ **if** $y' \neq y$ **then** ▷ check if the decision is changed
10: $\qquad$ **Break** ▷ stop substitutions if decision is changed
11: $\quad$ **end if**
12: **end for**
13: **return** x' ▷ return the adapted query as the counterfactual

DisCERN (Algorithm 2) brings together Sections 4.2 to 4.4 to discover counterfactuals for vulnerable code. Given the query x, and the train dataset $\mathcal{X}$, in Lines 1 and 2 we find the NUN as discussed in Section 4.2. Next, we find the LIME feature weights for the query and sort it to obtain the list of tokens that indicate which parts of the code contributed to the current decision (Line 3 and 4, Section 4.3). We iterate over the sorted list of tokens where for each token we find corresponding statements and substitutions (from Algorithm 1) until the prediction is changed (Line 8). Here the prediction for the adapted query x' is obtained using the original classification pipeline $f(t(.))$. The iteration is terminated when a prediction is changed and the algorithm returns the adapted query x' as the counterfactual for the query x. Compared to DisCERN for tabular data [8] the key novelty is the substitution algorithm that aims to preserve programme language syntax and original functionality while correcting the vulnerabilities. However, the outcome of, the substitution algorithm is dependent on the

Journal Pre-proof

Nearest-Unlike-Neighbour and does not always guarantee to find a counterfactual from the NUN. Accordingly, in the worst-case scenario, DisCERN iterates through all tokens in Z' and may fail to lead to a desirable decision change (of *non-vulnerable*) even after all corrections are actioned on the query.

5. Evaluation

This section presents the evaluation of the counterfactual DisCERN algorithm for vulnerable code correction. To the best of our knowledge, there are no existing algorithms in the literature for counterfactual discovery in the code vulnerability correction domain to compare performance with other methods.

5.1. Performance Metrics

DisCERN algorithm is evaluated using the three NIST datasets (Section 3); in each dataset, we only use *vulnerable* test data instances for the XAI evaluations. The following metrics are used to measure the performance.

- **Validity** measures the percentage of data for which the algorithm successfully finds a counterfactual [34, 35, 8]. At this stage, the requirement for a counterfactual discovered by an algorithm is to achieve a *positive* change of decision [4]. Given the set of test instances that were predicted *vulnerable* are X_v, and the subset for which the algorithm found a counterfactual is X_v^c, the validity is calculated as in Equation 6. A higher percentage of validity is desirable.

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

- **Sparsity** measures the mean number of statements that were changed (i.e. cost) for a change in decision [34, 35, 8]. Given the cost for each test instance in X_v^c is $[r_1, r_2, ..., r_N]$, where $N = |X_v^c|$, the sparsity is calculated as in Equation 7. In Algorithm 1, the number of statements changed for *replace*, *delete* and *insert* operations are calculated as $max(k-i, w-v)$,

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

$k - i$ and $w - v$ respectively. As such, the cost of a test instance is determined by aggregating the number of statement changes that correspond to the applied operations. In other domains, lower sparsity is preferred, however, in this domain, we hypothesise sparsity is not directly correlated to the algorithm performance as a vulnerability correction could require adding more statements. This will be discussed further with empirical results in Section 5.2.

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

There are other metrics used in counterfactual evaluations such as proximity (measures the difference between the original and the substitution code segments) [34, 35, 8] and diversity (measures the difference between multiple counterfactuals) [34] which we did not find to be transferable to the code vulnerability correction domain.

5.2. Results

Table 2 presents the performance evaluation results of DisCERN using the three NIST datasets. In addition to performance metrics, we also measure the mean number of statements in a query, nearest-unlike-neighbour and counterfactual which we found useful when discussing the performance of DisCERN.

Table 2: Validity and Sparsity of DisCERN

Dataset	Validity (%)	Sparsity	Mean no of statements in the		
			Query	NUN	CF
Java	96.49	13.88	44.62	51.81	50.93
C	85.50	8.40	24.78	28.26	26.08
C#	97.55	13.16	27.67	33.96	33.44

We observe that the validity is consistently below 100% across all datasets. The validity for the C dataset is significantly lower which means the C dataset queries were not able to find counterfactuals using DisCERN. This can be linked to a high ($\sim$ 21%) classification error seen in the vulnerability detection pipeline. For example, the query can be misclassified as vulnerable

Journal Pre-proof

Journal Pre-proof

or the adapted query can be continuously misclassified as *vulnerable*. It is further validated by the Java and C# datasets showing validity consistent with their classification pipeline performance.

Sparsity is measured as the number of changes that were required to get the decision changed from *vulnerable* to *non-vulnerable*. Considering the mean number of statements in the query (column 4), Java and C datasets make less number of changes compared to C#. It is noteworthy that these changes include *deletion* operations, thus it is not an indication of the length of the counterfactual. When generating counterfactuals for tabular data, a common goal is to minimise sparsity. However, when discovering counterfactuals for correcting code vulnerabilities we argue that lower sparsity is not always desirable. In general, correcting vulnerabilities can be costly; for example in Java, adding a *try-catch-finally* block surrounding a vulnerable statement can add up to 4-10 lines based on the formatting styles (Allman vs K&R).

Further analysis of the number of statements between NUN and the counterfactual shows the effectiveness of the DisCERN algorithm. The mean number of statements in a CF is consistently lower than that in the NUN indicating that DisCERN is in fact finding meaningful corrections instead of completely converting the query into its NUN. The consistently higher number of statements in CF compared to the Query further indicates the increased cost of correcting code vulnerabilities.

5.3. Qualitative Analysis

While DisCERN aims to maintain syntactic integrity and preserve the originally intended code functionality, sparsity, and validity metrics do not specifically measure these aspects. As a result, we examined a selection of the generated counterfactuals to determine whether the proposed code adaptations can effectively address code vulnerabilities and to what extent they implement reasonable modifications without compromising functionality.

Consider the two illustrative Java code examples in Figures 6a and 6b which were counterfactuals discovered by the DisCERN algorithm. In each figure, the first two columns indicate the line numbers of the query and the counterfactual; the third column uses addition and subtraction signs to indicate adaptation operations. In example 1, a replacement is proposed (i.e. replace query lines 5-6 with NUN lines 5-12). With Example 2, the counterfactual proposes an insertion (i.e. insert new lines 4-6) and a replacement (i.e. replace query lines 6-7 with NUN lines 9-13). Both sets of adaptations

```
1   1  public void method()
2   2  {
3   3      float data = dataArray[2];
4   4      /*comment*/
5   -      int result = (int)(100.0 % data)
6   -      IO.writeLine("");
    5+     if (Math.abs(data) > 0.000001)
    6+     {
    7+         int result = (int)(100.0 % data)
    8+         IO.writeLine("");
    9+     }
   10+     else{
   11+         IO.writeLine("");
   12+     }
7  13  }
```

(a) Example 1: Successful Adaptation

```
1   1  public void method()
2   2  {
3   3      int data = dataArray[2];
    4+     /*comment*/
    5+     int array[] = null;
    6+     /*comment*/
4   7      if (data > 0)
5   8      {
6   -          /*comment*/
7   -          IO.writeLine((short)data);
    9+         array = new int[data];
   10+     }
   11+     else
   12+     {
   13+         IO.writeLine("");
8  14      }
9  15  }
```

(b) Example 2: Unsuccessful Adaptation

Figure 6: DisCERN Counterfactual Examples

have maintained the grammatical structure of the Java language, however, Example 1 is better at preserving functionality, because it ensures that the original functionality of writing an empty line (originally line 6) even after having introduced an *if* condition. In Example 2, DisCERN fails to preserve the intended functionality in the original query line 7 (by failing to treat *data* as an array).

Both examples corroborate findings in Table 2 that code vulnerability correction can increase sparsity due to the insertion of additional statements. Overall, both evaluations indicate that DisCERN is a promising approach to discovering counterfactuals, however, to ensure comprehensive validity, further adaptation heuristics are needed to verify counterfactuals maintain the original functionality (e.g., apply unit testing if available).

6. User Evaluation

The primary objective of this user study is to assess the effectiveness of factual and counterfactual explainers in addressing code vulnerabilities, specifically examining their utility for both experienced and novice developers. While existing literature [25, 26, 27] highlights a focus on factual explanations (such as feature attributions) for knowledgeable users in the XAI research, our hypothesis posits that counterfactual explanations may prove more informative for both skilled and trainee developers aiming to correct code vulnerabilities. Table 3 presents the user study protocol; enumeration

Journal Pre-proof

Journal Pre-proof

indicates the order in which the questions were presented; Green colour indicates content presented to the participant (code segment or explanation) and the protocol is grouped by different intents (Blue). The questionnaire was prepared to capture users' mental models before and after receiving explanations, as well as to evaluate the quality and acceptability of the explanations provided by the system for detecting and correcting code vulnerabilities.

Table 3: User Study Protocol

Present code snippet	
A priori mental model for detecting code vulnerabilities	
Q1. Do you think the code snippet contains code vulnerabilities?	*Yes, No, Maybe*
A priori mental model for correcting code vulnerabilities	
Q2. If you answered yes, which lines would you change to correct code vulnerabilities?	*Free text*
Q3. If you listed any lines, why do you think these lines contain code vulnerabilities?	*Free text*
Present explanation (annotated or modified code snippet)	
A posterior mental model for correcting code vulnerabilities	
Q4. After seeing the explanation, which lines would you change to correct code vulnerabilities?	*Free text*
Q5. If you changed your answer from before viewing the explanation, please mention why?	*Free text*
Measure goodness of the explanation for detection and correction	
Q6. Did the explanation help you detect vulnerabilities?	*Yes, No*
Q7. Did the explanation help you to identify the lines you would change to correct code vulnerabilities?	*Yes, No*
Measure acceptability of the explanation	
Q8. Did the explainer correctly annotate the parts of the code that contain vulnerabilities?	*Yes, No, Partially*

The questionnaire was repeated with three different code snippets of different lengths (11, 33 and 21 lines of code) to minimise bias. Snippets were selected from the Java dataset over *C* and *C#* languages considering the wider usage and familiarity within the target user group. All snippets contained a variant of the CWE-191:Integer Underflow vulnerability. To prioritise the evaluation of the explanation over participant proficiency in detecting various types of vulnerabilities, only one type of vulnerability was included in the user study.

The hypothesis was evaluated with independent groups of participants recruited through Amazon Mechanical Turk. One group received the questionnaire together with DisCERN counterfactual explanations and the other with LIME feature attribution explanations. From here on we will refer to the two groups as DisCERN and LIME. The inclusion criteria for recruitment were set as *Employment Industry* is *Software and/or IT Services* and *Job function* is *Information Technology* to ensure the participants have a working knowledge of programming languages. In 40 days, 95 and 103 submissions were received for DisCERN and LIME groups respectively from which 78 and 68 were accepted. These submissions met the minimum requirements where they attempted to answer at least one free-text question in addition to all multiple choice questions (There were only 9 and 12 submissions for DisCERN and LIME groups where participants answered all questions).

6.1. A priori mental model - detecting code vulnerabilities

Q1 measures the a priori mental model for understanding how to detect code vulnerabilities. There are 438 responses (78 + 68 participants responded to 3 code snippets each) considered in total. Figure 7a plots the percentage of *Yes*, *No* and *Maybe* responses from the two groups. The percentages between the groups are comparable which suggests that the a priori knowledge and understanding levels are similar. However, the LIME group demonstrates higher accuracy and more confidence in their decision choices evidenced by the lower percentage in *Maybe* responses.

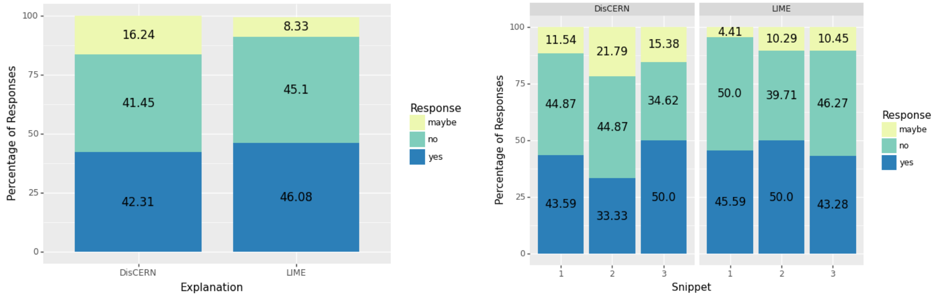
Figure 7b plots the percentage of responses received for each snippet. The DisCERN group identifies Snippet 2 as the most complex, as evidenced by their higher percentage of *Maybe* responses. Additionally, we observe that the high confidence of the LIME group stems from the least complex Snippet 1. Both observations imply that the responses are not arbitrary, lending credibility to the utilisation of Q1 responses as an indicator of the group's a priori mental model.

6.2. A priori mental model - correcting code vulnerabilities

Q2 measures the a priori mental model for correcting code vulnerabilities. Participants answered Q2 with line numbers or code lines which they considered to be *vulnerable*. Few example responses were *3,4,5*, *int data = method();* and *3rd line*. After pre-processing, Table 4 plots the number of responses for the three snippets across the two groups against corresponding code lines. Here the number of responses relates to the number of times

Journal Pre-proof

Journal Pre-proof



(a) Overall response

(b) Response by snippet

Figure 7: Q1 analysis on a priori mental model - detecting code vulnerabilities

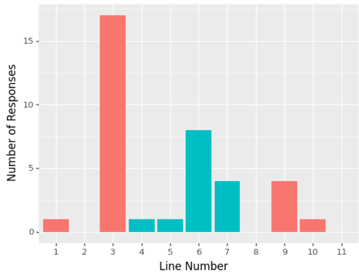
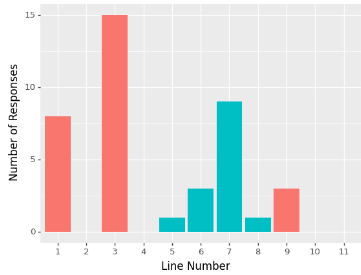
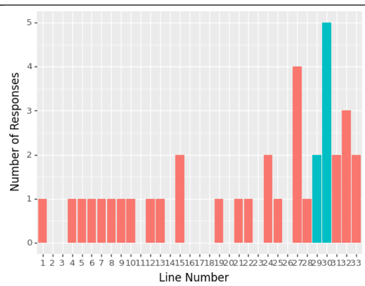
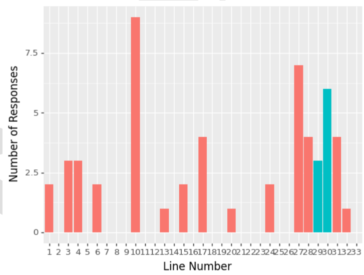
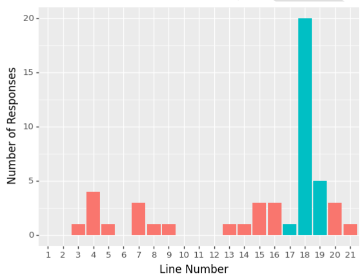
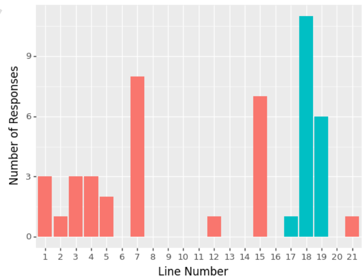
a specific line was identified as vulnerable. We then analyse these against the actual vulnerable lines (the ground truths). The plots use a two-way colour coding to distinguish between lines that are correctly identified as vulnerable (in blue) and those that are incorrectly identified as vulnerable (in red). Although we wouldn't anticipate participants who answered *No* (or to a lesser extent *Maybe*) in Q1 to respond to Q2, we have still included their Q2 responses in the graphs if they chose to provide them.

We calculate response accuracy as a percentage of correct responses compared to ground truth. DisCERN group demonstrated 37.8%, 18.9%, 53.1% response accuracy while LIME group achieves 35.0%, 16.7%, 38.3%. Overall accuracy for DisCERN and LIME groups were 36.6% and 30.0%. Snippet 2 was the most challenging for both groups indicated by the lowest accuracy, The wide variety of responses suggests that the increased complexity made participants uncertain and led to guessing. Overall, guessing or random responses are expected from those who did not detect vulnerability in Q1.

We observe that the code segment length has some correlation to the number of errors. Accordingly, we further normalise the accuracy values by the "difficulty of predicting vulnerable code lines in a code segment" using inspirations from document length normalisation which alleviates the "term-frequency-bias". Given the number of lines of code in the segment is α out of which β number of lines are vulnerable, the difficulty is calculated as $1 - \beta/\alpha$. If all lines were vulnerable $\beta = \alpha$ then $difficulty = 0$ and vice versa. The weighted accuracy values are 20.4%, 17.8% and 45.6% for DisCERN group (mean is 27.93%) and 18.9%, 15.7% and 32.9% for the LIME

Journal Pre-proof

Table 4: Q2 analysis on a priori mental model - correcting code vulnerabilities

Snippet	DisCERN	LIME
Snippet 1		
Snippet 2		
Snippet 3		

group(mean is 22.5%). The difference between the two groups is influenced by two factors: the number of responses for Snippet 2 from the DisCERN group was significantly lower than LIME group (37 vs 54) which contributed to the 2.1% difference, and for Snippet 3 DisCERN group responses were significantly more accurate (45.6% over 32.9%) although the number of responses was comparable (49 vs 47). This analysis aids in determining the groups' initial mental models, which is valuable for assessing the subsequent changes in their mental models a posteriori. We recognise the marginally higher (approximately 5%) performance of the DisCERN cohort and will consider this

Journal Pre-proof

in our subsequent analysis when we focus on a posteriori evaluations.

6.3. A posteriori mental model for correcting code vulnerabilities

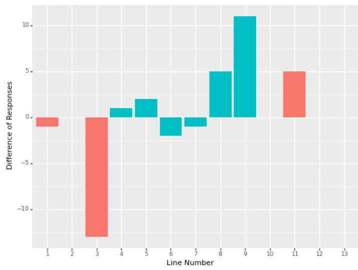
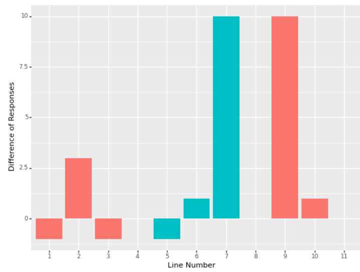
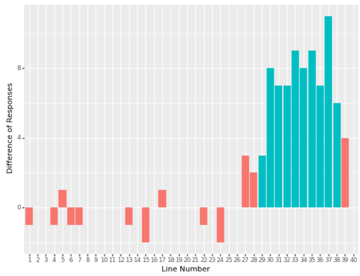
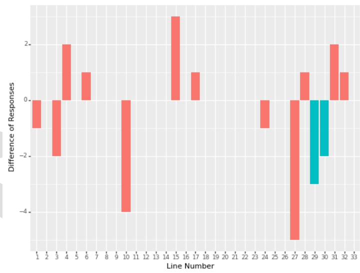
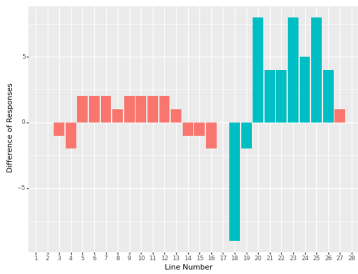
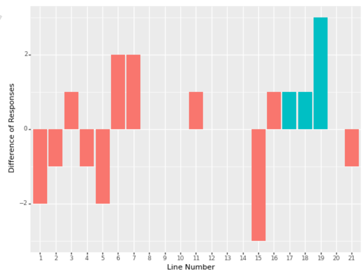
Q4 measures the a posteriori mental model for addressing vulnerabilities after participants have been exposed to the explanation. This implies that participants have been informed about the snippet's vulnerability and are presented with an explanation—either a counterfactual from DisCERN or feature attribution from LIME. The explanations were presented as code-diff for DisCERN and heat maps for LIME. To minimise the possibility of misinterpretations, we have provided supporting text alongside both explanations, detailing how to interpret them effectively.

Following pre-processing of the participants' responses, we analysed any changes in knowledge among each user group after exposure to the explanation for the three code snippets, as shown in Table 5. Here, we anticipate that changes to their mental model will be evident in at least two ways: 1) withdrawing their belief for lines that were incorrectly identified as vulnerable in Q2, and 2) recognising new lines that are necessary to address the vulnerability having seen the explanation in Q4. For example, if the change in response for a code line is denoted by -3, it means that the number of responses for that line after participants saw the explanation (Q4) decreased by three compared to before (Q2), indicating a shift in their belief about the vulnerability of that line. Here, the reductions observed with the Orange lines represent a positive change that was achieved a posterori. Unlike LIME, DisCERN not only identifies vulnerabilities but also provides hints on how to correct them by displaying counterfactuals. As a result, participants can access additional lines from the counterfactual that were not available in Q2. This is seen in Table 5 for DisCERN, where a relatively larger number of blue lines can be observed on the x-axis, indicating a notable difference.

Overall Table 5 observations strongly indicate that participants found counterfactuals more informative to correct vulnerabilities compared to feature attributions. The DisCERN group exhibited some errors, as misidentified lines on either side of the vulnerability boundary were observed. For instance, in Snippet 2, lines 28 and 39 were considered worthy of change, despite not being vulnerable. Similarly, in Snippet 3, lines 18 and 19 were not recognised as vulnerable, representing another error. The boundary cases observed with DisCERN and the errors observed with the LIME group both suggest that some participants are likely to either misinterpret or disagree with the explanations.

Journal Pre-proof

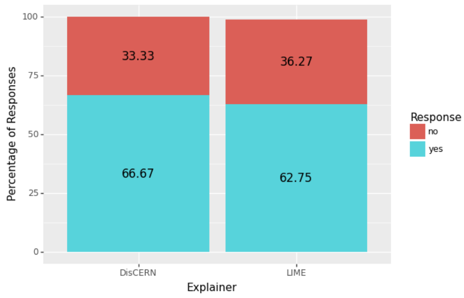
Table 5: Q4 analysis on a posterior mental model - correcting code vulnerabilities

Snippet	DisCERN	LIME
Snippet 1		
Snippet 2		
Snippet 3		

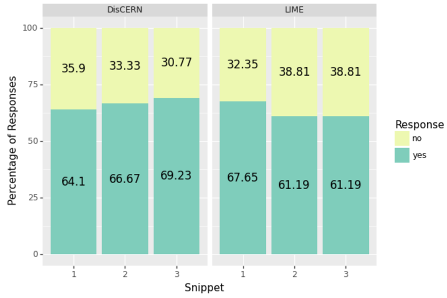
6.4. *Goodness of explanations for vulnerability detection and correction*

Q6 and Q7 aim to measure the overall goodness of the explanation to detect and correct vulnerabilities. Both questions are further analysed in relation to Q1 to examine the utility of the explanations to different cohorts: knowledgeable participants who responded *Yes* in Q1; and trainee participants who responded *No* or *Maybe* in Q1.

Q6 results across the two groups are plotted in Figure 8. The positive response rate from DisCERN and LIME groups were 66.7% and 62.7% respectively when asked about the utility of explanations for vulnerability detection. This indicates a slight preference towards counterfactual explanations. Furthermore, Figure 8b indicates that the counterfactual explanations were found to be useful for more complex snippets (2 and 3) and feature attributions useful for the smallest snippet (1). This suggests that using counterfactual explanations may result in a lower cognitive load for detecting errors when compared to feature attributions.



(a) Overall response



(b) Response by snippet

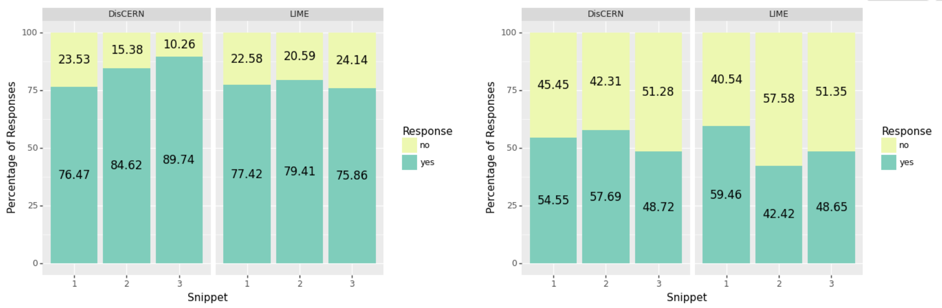
Figure 8: Q6 analysis on the goodness of explanations - detecting code vulnerabilities

Figures 9a and 9b present an in-depth analysis of the Q6 responses with respect to Q1. Figure 9a shows that participants with prior knowledge of vulnerability detection found both types of explanations useful. The improved positive response rates of 83.61% and 77.56% from their baselines for DisCERN and LIME indicate that knowledgeable users found both types of explanations helpful. However, counterfactuals have been significantly more helpful than feature attributions, especially for complex code snippets. Figure 9b shows that trainee cohorts struggle with types of explanations. It is indicated by the decreased positive response rate from their baselines to 53.7% and 50.2% for DisCERN and LIME groups. However, trainee cohorts found counterfactuals significantly helpful for the most complex snippet whereas feature attribution helped with the simplest snippet. These observations further verify that counterfactuals reduced the cognitive burden of vulnerability detection in complex code snippets.

Q7 measures the utility of the explanation to **correct** vulnerabilities and we plot similar graphs to Q6. Figure 10a shows that the overall positive response rates from DisCERN and LIME groups were 66.24% and 63.24% respectively. Similar to detection (Q6), the responses for Q7 indicate a preference for the counterfactuals for more complex snippets. In contrast to

Journal Pre-proof

Journal Pre-proof

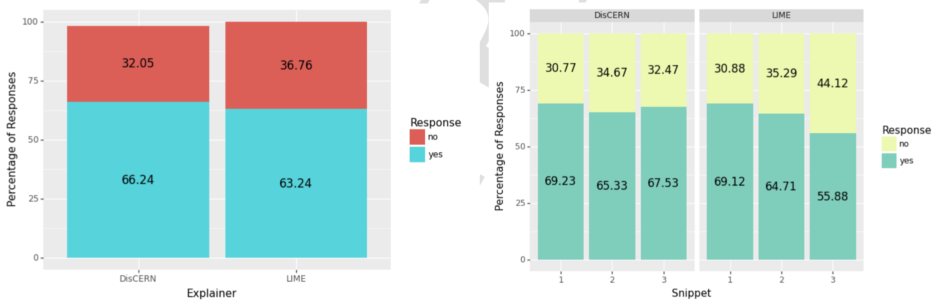


(a) Responses from knowledgeable cohorts

(b) Responses from trainee cohorts

Figure 9: Q6 analysis on the goodness of explanations - detecting code vulnerabilities by knowledgeable and trainee cohorts

detection, counterfactuals are found to be comparably helpful for the correction of vulnerabilities in simpler snippets which evidence an overall preference towards counterfactual explanations.



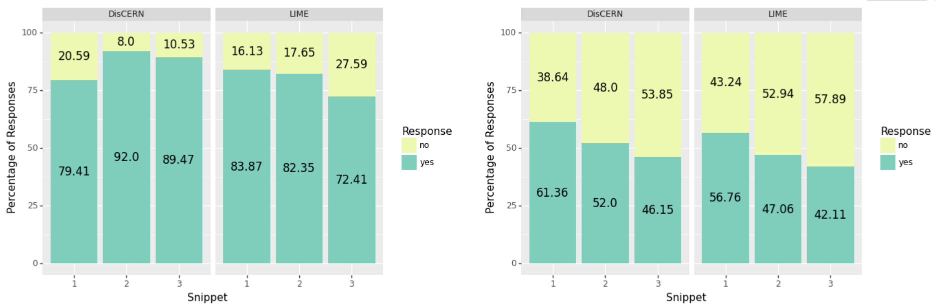
(a) Overall response

(b) Response by snippet

Figure 10: Q7 analysis on the goodness of Explanations - correcting code vulnerabilities

Figure 11 presents the analysis of the Q7 response with respect to cohorts identified in Q1. Similar to Q6, the positive response rate for both explanations have improved from the knowledgeable cohort and decreased from the trainee cohort. The preference for counterfactuals over feature attributions by both cohorts for complex snippets remains significant for vulnerability correction. While trainee cohorts consistently find counterfactuals to be more helpful, knowledgeable cohorts find feature attributions are sufficient for correcting vulnerabilities in simple snippets.

Journal Pre-proof



(a) Responses from knowledgeable cohorts (b) Responses from trainee cohorts

Figure 11: Q7 analysis on the goodness of explanations - correcting code vulnerabilities by knowledgeable and trainee cohorts

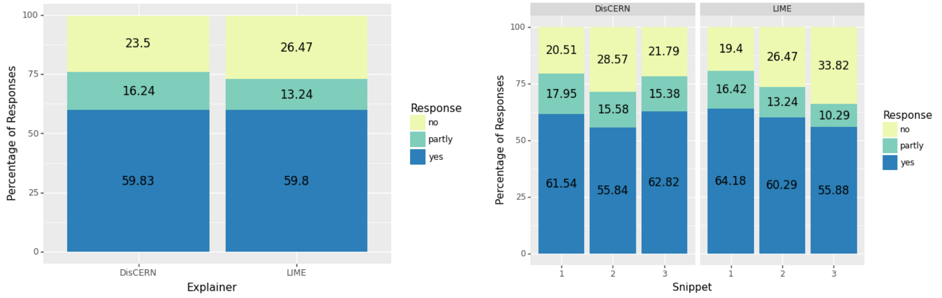
We acknowledge that a significant number of trainee users did not find either type of explanation helpful for both detection and correction. This was clearly seen from responses to both Q6 and Q7 having less than 50% positive responses for Snippets 2,3 in the LIME group and Snippet 3 in DisCERN group. However, the overall preference was for DisCERN counterfactuals. This feedback is useful for future research to further improve counterfactual explanations to assist trainee developers to learn about vulnerability detection and correction.

6.5. Acceptability of the explanations

Q8 is aimed to measure the acceptability of the explanations. Similar to Q6 and Q7 we plot overall responses and responses by snippets in Figure 12. In both groups, approximately 60% of the participants agreed with the explanations provided. However, the disagreement is significantly lower in the DisCERN group where 3% more partially accepted the counterfactual explanation. Figure 12b shows that the acceptability of LIME is significantly lower for Snippet 3 which has affected the overall acceptance. Otherwise, agreement with feature attributions is similar to or greater than that of counterfactuals which is inconsistent with the previous observations on change in mental model and goodness. LIME is a well-established explanation method in various domains for several years, which may have influenced the observed results, to further verify, we perform a more in-depth analysis.

The first analysis of acceptability is with respect to the cohorts recognised in Q1. Figure 13 plots the acceptability by knowledgeable and trainee co-

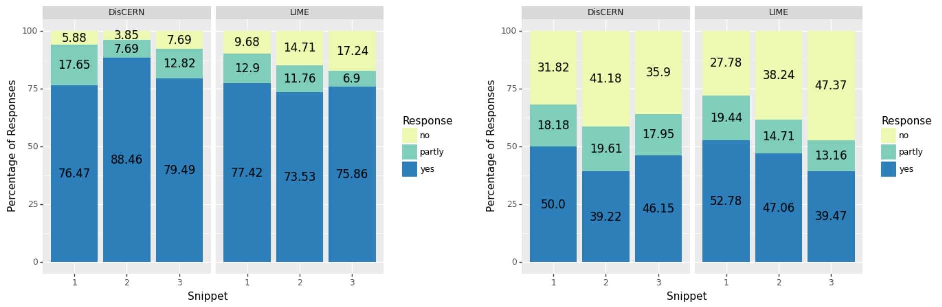
Journal Pre-proof



(a) Overall response (b) Response by snippet

Figure 12: Q8 analysis on the acceptability of explanations

horts. The knowledgeable cohorts found counterfactuals more agreeable than feature attributions, indicated by the accumulated positive response rates of 81.47% and 75.60%. The most significant difference is that the counterfactual explanation for the most complex snippet is found to be more agreeable which aligns with previous observations. We failed to observe a majority of trainee cohorts agreeing with either explanation, however, we observe partial agreement rates of 63.70% and 62.20% respectively for DisCERN and LIME. These findings reinforce the overall utility of counterfactuals over feature attribution and also highlight the need to improve the counterfactuals to build trust among trainee developers as an effective learning tool.



(a) Responses from knowledgeable cohorts (b) Responses from trainee cohorts

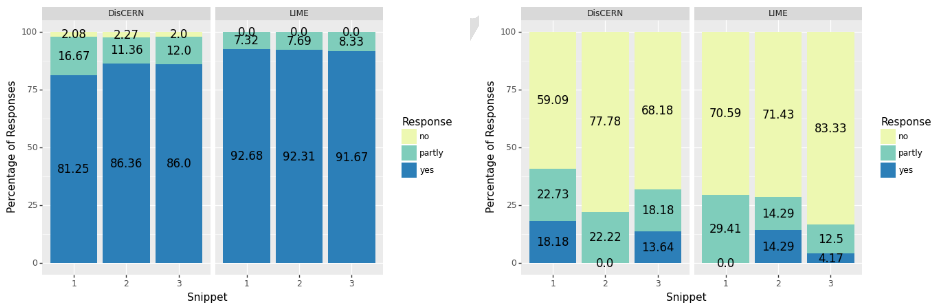
Figure 13: Q8 analysis on the acceptability of explanations by knowledgeable and trainee cohorts

The second analysis of acceptability is with respect to the explanation goodness observed by Q6 and Q7. We recognise two cohorts from Q6 and

Journal Pre-proof

Q7, the ones who found explanations helpful and others who did not for both vulnerability detection and correction. Figure 14a plots the Q8 responses for those who found explanations helpful. Results show that the participants who found feature attribution helpful overwhelmingly agreed with the explanation (0% *no* responses). However, not all who found counterfactuals useful agreed with it indicated by 2.11% disagreeing and 13.34% only partially agreeing. Figure 14b plots the Q8 responses for those who found explanations not helpful. Those who found feature attribution not helpful for small snippets completely disagreed with the explanation and those who found counterfactuals not helpful for complex snippets, also completely disagreed with the explanation. It is noteworthy that both of these cohorts are the minority when determining goodness. Overall, 31.63% and 24.89% at least partially agreed with counterfactuals and feature attributions respectively.

These observations conclude that the higher overall agreement with feature attributions seen in Figure 12b for Snippets 1 and 2 is influenced by the cohorts who found counterfactuals helpful (Q6 and Q7) but did not fully agree with them. What is unknown and needs to be established in the long term is if this acceptance of feature attribution is influenced by familiarity with LIME explanations. The need for this is supported by the results in Section 6.3 which clearly showed that counterfactuals influence a positive mental model change compared to feature attributions.



(a) Responses from Q6, Q7 = yes cohorts

(b) Responses from Q6, Q7 = no cohorts

Figure 14: Q8 analysis on the acceptability of explanations by goodness cohorts

6.6. Implications and Limitations of the User Study

Overall, counterfactual explanations encourage positive mental model changes and were perceived as more helpful than feature attributions for

Journal Pre-proof

detecting and correcting code vulnerabilities. However, feature attributions exhibit comparable or higher acceptability, possibly due to their widespread use. These conclusions should be made with the limitations of the study in mind. The key limitations of the above user study are three-fold: 1) incompleteness of heuristics used to identify the knowledgeable and trainee cohorts; 2) inclusion and exclusion criteria of participants; and 3) representations and interpretation of explanations.

The a priori mental model for detecting vulnerabilities was based on Q1 which recognised two cohorts as knowledgeable and trainee. However, we did not account for those who recognised the vulnerabilities incorrectly by collating them with answers to Q2. As seen in Section 6.2 only 36% and 30% of the two groups identified the lines correctly and it includes all participants. We found it challenging to filter participants by both Q1 and Q2 because many of those who answered *Yes* in Q1 were able to partially identify vulnerable lines. A more strict filter would have resulted in no knowledgeable participants. Accordingly, we relied solely on Q1 to categorise participants into the two cohorts.

The recruitment of participants for the user study was limited to Amazon Mechanical Turk (AMT), which placed constraints on the inclusion criteria. Accordingly, the inclusion criteria for selecting participants were constrained to those possible in the AMT platform. Ideally, a more comprehensive study would include participants in various career stages with Java software development skills.

The explanations generated by LIME and DisCERN are significantly different in their presentation. LIME highlights the original query using a heat-map scale and DisCERN presents counterfactual in a code-diff view. With LIME explanations where the individual tokens are highlighted, it may mislead the participants. An example scenario is if two tokens in a statement were highlighted as *vulnerable* and *non-vulnerable*, the participant can consider the statement as *vulnerable*, *non-vulnerable* or *have no impact* on the vulnerability detection. DisCERN code-diff view has two columns with query line numbers and counterfactual line numbers. A participant who is unfamiliar with code-diff may mistakenly use the line numbers from the inappropriate column when responding to the questionnaire.

All three limitations are well-founded, however, they do not invalidate the findings, rather, they provide enhancing user studies in this particular domain and in Explainable AI in general.

The user study was conducted with three code segments, all of which be-

Journal Pre-proof

longed to the same vulnerability code (CWE-191). Our main reasoning was to prioritise the evaluation of the utility of different types of explanation while keeping other variables constant. Additionally, it allowed the user study not to be biased by the proficiency of the participant in detecting various types of vulnerabilities. However, there can be implications for this approach if some vulnerabilities were better explained using feature attributions over counterfactuals. This can be linked to our observations in Section 6.4 where there was no significant preference between feature attribution and counterfactual explanations when the code segment was simple. The generalisability of DisCERN to different vulnerability classes and languages (as seen in Section 5) provides an opportunity to evaluate this in the future.

7. Conclusion

The DisCERN algorithm finds counterfactual explanations for correcting code vulnerabilities using pattern matching to find corrections to a code segment from its nearest-unlike neighbour. DisCERN was evaluated using three NIST datasets in different programming languages and the results showed that it finds counterfactuals in 85% - 96% of the cases with $8 \sim 14$ statement corrections needed. A qualitative analysis revealed that some of the counterfactuals generated by DisCERN did not preserve the original functionality of the code. This highlights the need for comprehensive heuristics in the future to ensure plausible code corrections. We conducted a user study to assess the utility of counterfactual explanations compared to the more commonly used feature attribution explanations for correcting vulnerabilities. The user study showed that counterfactuals facilitated a positive mental model change towards correcting vulnerabilities. Counterfactuals were specifically preferred over feature attributions when dealing with complex code segments, indicating a reduction in cognitive burden. However, despite being less helpful for vulnerability correction, feature attribution explanations received higher acceptance than counterfactuals, possibly due to the trust built around their familiarity. These findings provide evidence for the utility of counterfactual explanations over feature attribution explanations. Nonetheless, they also emphasise the importance of conducting long-term evaluations to determine if counterfactuals can establish trust with developers as a reliable tool for vulnerability detection and correction.

Journal Pre-proof

Acknowledgments

This research is funded by the iSee project [5]. iSee is an EU CHIST-ERA project [6] which received funding for the UK from EPSRC under the grant number EP/V061755/1.

Appendix A. Code snippets from the user study

```
public void method()
{
    int data = someMethod();

    /*comment*/
    data--;
    int result = (int)(data);

    IO.writeLine("result: " + result);

}
```

Figure A.15: Snippet 1: LIME Explanation

```diff
 public void method()
 {
-    int data = someMethod();
+    int data;

     /*comment*/
-    data--;
-    int result = (int)(data);
+    data = 2;
+
+    /*comment*/
+    int result = (int)(--data);

     IO.writeLine("result: " + result);

 }
```

Figure A.16: Snippet 1: DisCERN Explanation

[5] https://isee4xai.com/

[6] https://www.chistera.eu/projects/isee

Journal Pre-proof

```
public void method()
{
    int data;
    if (IO.STATIC_FINAL_TRUE)
    {
        data = Integer.MIN_VALUE; /*comment*/
        /*comment*/
        /*comment*/
        {
            String stringNumber = System.getProperty("user.home");
            try
            {
                data = Integer.parseInt(stringNumber.trim());
            }
            catch(NumberFormatException exceptNumberFormat)
            {
                IO.logger.log(Level.WARNING, "Number format exception parsing data from string",
exceptNumberFormat);
            }
        }
    }
    else
    {
        /*comment*/
        data = 0;
    }

    if (IO.STATIC_FINAL_TRUE)
    {
        /*comment*/
        int result = (int)(data - 1);
        IO.writeLine("result: " + result);
    }
}
```

Figure A.17: Snippet 2: LIME Explanation

```
public void method()
{
    int data;
    if (IO.STATIC_FINAL_TRUE)
    {
        data = Integer.MIN_VALUE; /*comment*/
        /*comment*/
        /*comment*/
        {
            String stringNumber = System.getProperty("user.home");
            try
            {
                data = Integer.parseInt(stringNumber.trim());
            }
            catch(NumberFormatException exceptNumberFormat)
            {
                IO.logger.log(Level.WARNING, "Number format exception parsing data from string", exceptNumberFormat);
            }
        }
    }
    else
    {
        /*comment*/
        data = 0;
    }

    if (IO.STATIC_FINAL_TRUE)
    {
        /*comment*/
-       int result = (int)(data - 1);
-       IO.writeLine("result: " + result);
+       if (data > Integer.MIN_VALUE)
+       {
+           int result = (int)(data - 1);
+           IO.writeLine("result: " + result);
+       }
+       else
+       {
+           IO.writeLine("");
+       }
    }
}
```

Figure A.18: Snippet 2: DisCERN Explanation

Journal Pre-proof

```
public void method()
{
    byte data;
    if (IO.staticTrue)
    {
        /*comment*/
        data = Byte.MIN_VALUE;
    }
    else
    {
        /*comment*/
        data = 0;
    }

    if (IO.staticTrue)
    {
        /*comment*/
        byte result = (byte)(data - 1);
        IO.writeLine("result: " + result);
    }
}
```

Figure A.19: Snippet 3: LIME Explanation

```
 public void method()
 {
     byte data;
     if (IO.staticTrue)
     {
         /*comment*/
         data = Byte.MIN_VALUE;
     }
     else
     {
         /*comment*/
         data = 0;
     }

     if (IO.staticTrue)
     {
         /*comment*/
-        byte result = (byte)(data - 1);
-        IO.writeLine("result: " + result);
+        if (data > Byte.MIN_VALUE)
+        {
+            byte result = (byte)(data - 1);
+            IO.writeLine("result: " + result);
+        }
+        else
+        {
+            IO.writeLine("");
+        }
     }
 }
```

Figure A.20: Snippet 3: DisCERN Explanation

Journal Pre-proof

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] G. Tang, L. Zhang, F. Yang, L. Meng, W. Cao, M. Qiu, S. Ren, L. Yang, H. Wang, Interpretation of learning-based automatic source code vulnerability detection model using lime, in: Knowledge Science, Engineering and Management: 14th International Conference, KSEM 2021, Tokyo, Japan, August 14–16, 2021, Proceedings, Part III, Springer, 2021, pp. 275–286.

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

[8] A. Wijekoon, N. Wiratunga, I. Nkisi-Orji, C. Palihawadana, D. Corsar, K. Martin, How close is too close? the role of feature attributions in discovering counterfactual explanations, in: Case-Based Reasoning Research and Development: 30th International Conference, ICCBR 2022, Nancy, France, September 12–15, 2022, Proceedings, Springer, 2022, pp. 33–47.

Journal Pre-proof

Journal Pre-proof

[9] D. Votipka, R. Stevens, E. Redmiles, J. Hu, M. Mazurek, Hackers vs. testers: A comparison of software vulnerability discovery processes, in: 2018 IEEE Symposium on Security and Privacy (SP), IEEE, 2018, pp. 374–391.

[10] A. C. Eberendu, V. I. Udegbe, E. O. Ezennorom, A. C. Ibegbulam, T. I. Chinebu, et al., A systematic literature review of software vulnerability detection, European Journal of Computer Science and Information Technology 10 (1) (2022) 23–37.

[11] S. M. Ghaffarian, H. R. Shahriari, Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey, ACM Computing Surveys (CSUR) 50 (4) (2017) 1–36.

[12] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, Vuldeepecker: A deep learning-based system for vulnerability detection, arXiv preprint arXiv:1801.01681 (2018).

[13] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, Advances in neural information processing systems 32 (2019).

[14] X. Yuan, G. Lin, Y. Tai, J. Zhang, Deep neural embedding for software vulnerability discovery: Comparison and optimization, Security and Communication Networks 2022 (2022) 1–12.

[15] E. Mashhadi, H. Hemmati, Applying codebert for automated program repair of java simple bugs, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 505–509.

[16] N. Ziems, S. Wu, Security vulnerability detection using deep learning natural language processing, in: IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), IEEE, 2021, pp. 1–6.

[17] J. Senanayake, H. Kalutarage, M. O. Al-Kadri, A. Petrovski, L. Piras, Android source code vulnerability detection: a systematic literature review, ACM Computing Surveys 55 (9) (2023) 1–37.

Journal Pre-proof

[18] I. Medeiros, N. Neves, M. Correia, Detecting and removing web application vulnerabilities with static analysis and data mining, IEEE Transactions on Reliability 65 (1) (2015) 54–69.

[19] D. K. P. Newar, R. Zhao, H. Siy, L.-K. Soh, M. Song, Ssdtutor: A feedback-driven intelligent tutoring system for secure software development, Science of Computer Programming 227 (2023) 102933.

[20] S. Ma, F. Thung, D. Lo, C. Sun, R. H. Deng, Vurle: Automatic vulnerability detection and repair by learning from examples, in: Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22, Springer, 2017, pp. 229–246.

[21] Y. Zhang, Y. Xiao, M. M. A. Kabir, D. Yao, N. Meng, Example-based vulnerability detection and repair in java code, in: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022, pp. 190–201.

[22] A. Savchenko, O. Fokin, A. Chernousov, O. Sinelnikova, S. Osadchyi, Deedp: vulnerability detection and patching based on deep learning, Theoretical and Applied Cybersecurity 2 (1) (2020).

[23] D. Gunning, D. Aha, Darpa's explainable artificial intelligence (xai) program, AI magazine 40 (2) (2019) 44–58.

[24] M. Ebers, Regulating explainable ai in the european union. an overview of the current legal framework (s), An Overview of the Current Legal Framework (s)(August 9, 2021). Liane Colonna/Stanley Greenstein (eds.), Nordic Yearbook of Law and Informatics (2020).

[25] T. N. Nguyen, R. Choo, Human-in-the-loop xai-enabled vulnerability detection, investigation, and mitigation, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2021, pp. 1210–1212.

[26] S. Höhn, N. Faradouris, What does it cost to deploy an xai system: A case study in legacy systems, in: Explainable and Transparent AI and Multi-Agent Systems: Third International Workshop, EXTRAAMAS 2021, Virtual Event, May 3–7, 2021, Revised Selected Papers 3, Springer, 2021, pp. 173–186.

Journal Pre-proof

[27] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, C. Siemens, Drebin: Effective and explainable detection of android malware in your pocket., in: Ndss, Vol. 14, 2014, pp. 23–26.

[28] V. J. Sudhakar, S. Mahalingam, V. Venkatesh, V. Vetriselvi, Phishing url detection and vulnerability assessment of web applications using ivs attributes with xai, in: ICT Analysis and Applications, Springer, 2022, pp. 933–944.

[29] G. Schwalbe, B. Finzel, A comprehensive taxonomy for explainable artificial intelligence: a systematic survey of surveys on methods and concepts, Data Mining and Knowledge Discovery (2023) 1–59.

[30] T. Miller, Explanation in artificial intelligence: Insights from the social sciences, Artificial intelligence 267 (2019) 1–38.

[31] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, in: Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547.

[32] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, arXiv preprint arXiv:1810.04805 (2018).

[33] 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.

[34] R. K. Mothilal, A. Sharma, C. Tan, Explaining machine learning classifiers through diverse counterfactual explanations, in: Proceedings of the 2020 conference on fairness, accountability, and transparency, 2020, pp. 607–617.

[35] D. Brughmans, P. Leyman, D. Martens, Nice: an algorithm for nearest instance counterfactual explanations, Data Mining and Knowledge Discovery (2023) 1–39.

Journal Pre-proof