Can explainability and deep-learning be used for localizing vulnerabilities in source code?

Alessandro Marchetto
University of Trento, Trento, Italy
Trento, Italy
alessandro.marchetto@unitn.it

ABSTRACT

Security vulnerabilities are weaknesses of software due for instance to design flaws or implementation bugs that can be exploited and lead to potentially devastating security breaches. Traditionally, static code analysis is recognized as effective in the detection of software security vulnerabilities but at the expense of a high human effort required for checking a large number of produced false positive cases. Deep-learning methods have been recently proposed to overcome such a limitation of static code analysis and detect the vulnerable code by using vulnerability-related patterns learned from large source code datasets. However, the use of these methods for localizing the causes of the vulnerability in the source code, i.e., localize the statements that contain the bugs, has not been extensively explored.

In this work, we experiment the use of deep-learning and explainability methods for detecting and localizing vulnerability-related statements in code fragments (named snippets). We aim at understanding if the code features adopted by deep-learning methods to identify vulnerable code snippets can also support the developers in debugging the code, thus localizing the vulnerability's cause. Our work shows that deep-learning methods can be effective in detecting the vulnerable code snippets, under certain conditions, but the code features that such methods use can only partially face the actual causes of the vulnerabilities in the code.

CCS CONCEPTS

• **Security and privacy** → **Vulnerability management**; **Systems security**; **Malware and its mitigation**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Cybersecurity, Vulnerability detection, Vulnerability localization.

ACM Reference Format:
Alessandro Marchetto. 2024. Can explainability and deep-learning be used for localizing vulnerabilities in source code?. In *5th ACM/IEEE International Conference on Automation of Software Test (AST 2024) (AST '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3644032.3644448

AST '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0588-5/24/04…$15.00
https://doi.org/10.1145/3644032.3644448

1 INTRODUCTION

Design flaws and software implementation bugs are weaknesses of software that can represent security vulnerabilities that can be exploited by malicious attackers, thus leading to security breaches. Security vulnerabilities can have a strong impact and compromise software and services in our everyday life. They can indeed be leveraged for forcing the software in malicious and unexpected states, those leading for instance to service interruptions and resource abuses. Recent experiences [8] and the growing market [10] show that there is an increasing attention toward security testing methods that can be integrated in software development processes for detecting software vulnerabilities.

Security testing studies specific methods and tools that aim at detecting software vulnerabilities that can be originated from specific types of software weaknesses and bugs such as the ones described in the Common Weakness Enumeration - CWE[1] (e.g., Buffer Overflow, Cross-site scripting XSS, SQL Injection, path traversal), and that can lead to security vulnerabilities, as classified by means of the Common Vulnerabilities and Exposures method (CVE[2]). Traditional security testing approaches that do not demand for the execution of the application under analysis for detecting vulnerabilities are based on static code analysis. For instance, taint analysis [5] uses data-flow analysis to discover vulnerabilities related to missing input validation checks. Code similarity [13] is used to detect vulnerabilities of cloned fragments of code. Pattern matching (e.g., Flawfinder[3], Sonarqube[4]) is used for identifying security patterns of known vulnerable code. Static analysis is effective but: (i) it produces a large number of false positive cases to be checked by developers; and (ii) it strongly focuses on specific types of vulnerability (e.g., depending on the used patterns).

Machine and deep-learning methods [4] have been presented to detect vulnerable code based on vulnerability-related patterns learned from datasets of source code. For instance, VulDeePecker [15] and SeVCs [14] use a token-based representation of source code and deep-learning to detect vulnerabilities in code snippets (fragments of source code). Devign [25] and BGNN4VD [3] represent the source code as a graph that is then fed to graph neural networks for vulnerability detection. JavaBert [17, 22] uses a transformer-based architecture for managing long sequences of source code. These approaches have been investigated to detect vulnerable code snippets (e.g., code functions, code gadgets, programs) but their capability of localizing the cause of the vulnerability in the source code, i.e., localize the statements that contain the bug(s) (hereafter: *vulnerability-related code statements*), has not been yet extensively

[1] https://cwe.mitre.org
[2] https://www.cve.org
[3] https://dwheeler.com/flawfinder
[4] https://www.sonarsource.com

explored. Indeed, only few existing works in the related literature deal with the identification of the vulnerability causes (e.g., [9], [7], and [6]) by designing ad-hoc deep-learning models to locate the vulnerable code statements.

In this work, we investigate the use of relevant code features identified by leveraging deep-learning methods and explainability techniques to localize the *vulnerability-related code statements*. Our goal is to understand if these features, used by the deep-learning methods to decide if a code snippet is vulnerable, according to explainability techniques can also support the developers in debugging the code and localizing the bug(s). To this aim, we apply a model-agnostic explainability method, named *SHapley Additive exPlanations* (SHAP [16]), on top of two deep-learning methods *VulDeePecker* and *JavaBert*, for detecting vulnerable code snippets and localizing its vulnerability-related code statements. We opted for such methods by considering (1) the tools' availability, (2) the need of applying the tools to the same dataset (after some required tuning and implementation customization), and (3) since they are quite representative of the existing literature for vulnerability detection (see Section 6). The achieved results show that these methods and techniques cannot be satisfactory adopted for code debugging, but we observed some potentiality and we notice that there is room for improvement: (i) such methods are effective in detecting vulnerable code snippets, under certain conditions; but (ii) such methods seem to use code features that are only partially related to the causes of the vulnerabilities in the code.

The rest of the paper is organizes as follows: Section 2 briefly introduces relevant concepts about VulDeePecker, JavaBert and SHAP and how to use them for vulnerability detection and localization; Section 3 reports the conducted experimentation, while Section 4 and Section 5 report and discuss the observed results; and, finally, Section 6 and Section 7 conclude the paper with the analysis of the existing literature and with final remarks.

2 EXPLAINABLE AI FOR VULNERABILITY LOCALIZATION

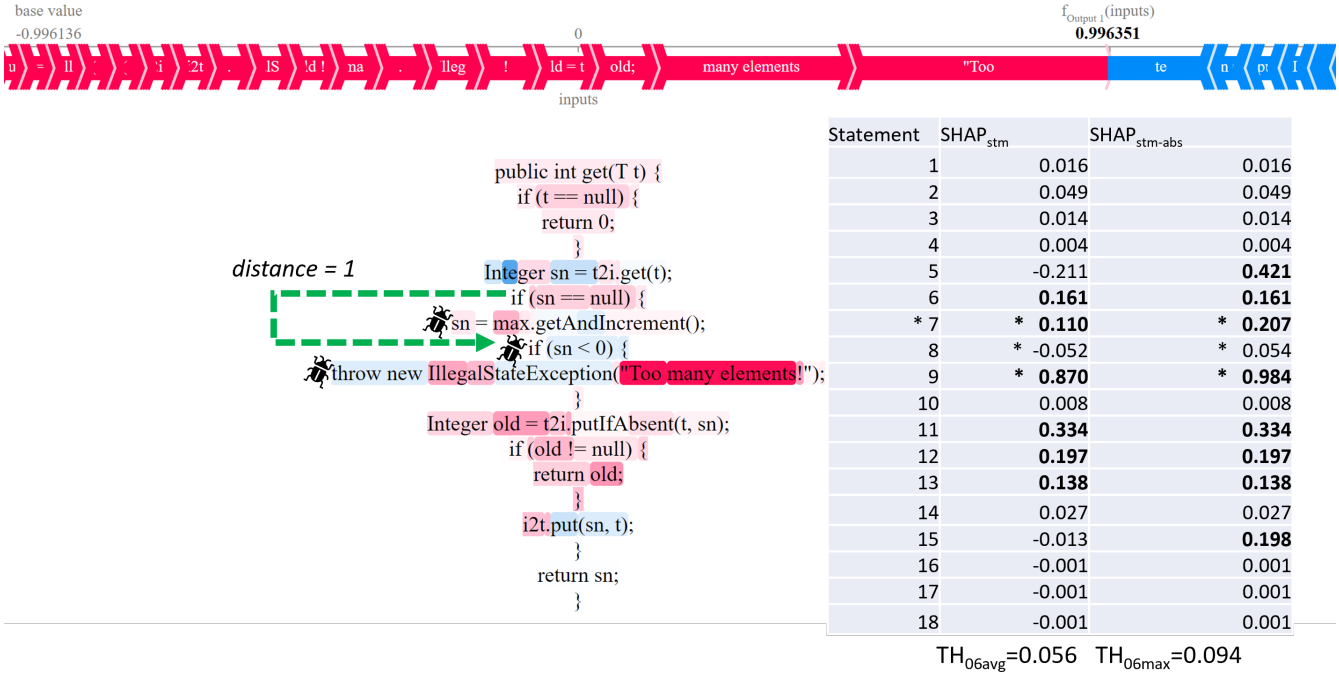
This section introduces the use of deep-learning and explainability for vulnerability detection and localization in source code. We briefly present the methods selected and how they are used in this work.

VulDeePecker [15] uses deep-learning to detect vulnerabilities in code *gadgets*. A gadget is a fragment of code composed of (not consecutive) code statements that are semantically related to each other. Such gadgets are tokenized and encoded in fixed-length vectors of symbols, by using Word2Vec[5] for tokenizing, and Code2Vec[6] for token embeddings. The obtained vectors are fed to a Bidirectional Long Short-Term Memory (BLSTM) neural network that classifies the target code gadgets as vulnerable and non-vulnerable, depending on the model learned from the labeled dataset.

JavaBert [22] leverages the Bidirectional Encoder Representation from the Transformer architecture (BERT), which is often applied in the natural language processing (NLP) field, to detect code vulnerabilities. The vulnerability detection problem is hence modeled as a NLP problem in which the text is the source code under analysis. The source code is provided to the transformer as a long sequence of text (string). The JavaBert transformer is based on a series of transformer blocks that implement a sequence-to-sequence transformation with an encoder structure and attention mechanisms. This implies that the input text (source code) is splitted in words by a tokenizer, and each word is coded into a representation vector by the embeddings. The JavaBert's tokenizer uses special tokens, such as $[MASK]$, $[UNK]$, $[PAD]$ to manage: (i) masked tokens (used in token-prediction tasks conducted during the learning phase to improve the model prediction performance); (ii) tokens uncovered by the vocabulary; and (iii) tokens used for short sequences, while longer sequences are split into chunks before being fed to the model. The representation vectors are then passed to the 12-layer structure of the JavaBert's transformer and the attention mechanism of each transformer is used to account for the context of each considered word. JavaBert provides a pre-trained model [22] that is then fine-tuned for the specific classification task of interest, i.e., detection of vulnerable code, in our case. For instance, VDT [17] is a fine-tuned JavaBert-based model for vulnerable code detection.

Explainable AI [2] aims at increasing the user's confidence in AI systems by improving their comprehension of the decision function (i.e., the features) used by the AI models. Explainability methods allow indeed for capturing relevant features (in our case, code token) used by models to make decisions (in our case, to classify code snippets) [20, 21]. These methods can be model-agnostic, if they work for any models by approximating the decision function with a surrogate function, or model-specific, if they leverage the knowledge of the model itself. Explainability can be applied locally, if explanations are focused on a specific case, or globally, if explanations consider the entire dataset. *SHAP* is a model-agnostic explainability method based on a game theoretic approach in which SHAP values (ranging from -1 to 1) are computed for each feature used by the AI model. Such values estimate how much each feature contributes to the model's decisions. While features with positive SHAP values positively impact the model's decisions, features with negative values show a negative impact. The absolute SHAP value indicates the overall effect of a feature, so that by prioritizing the features according to their SHAP value, we can identify the most relevant ones. In few words, SHAP computes a base SHAP value by masking all code tokens with the tag $[MASK]$ and obtaining the value without features. Then, it iteratively adds features again and computes the SHAP values depending on the contribution of each feature to the observed value changes. Figure 1 shows an example of a Java function code in which tokens (as identified by the tool) are highlighted according to their SHAP value (the color intensity depends on SHAP values: a more intense color represents higher SHAP value). For instance, the features "*Too*" and "*many elements*" (high positive SHAP values - red) have a positive impact on the decision of classifying the code as vulnerable, while "$(sn < 0)$" (high negative SHAP value - blue) has a negative impact. By considering the relevant features, we could try to comprehend the mechanism underlying the decision function adopted by the AI model to classify the code as vulnerable or non-vulnerable. For instance, the code in Figure 1 is classified as vulnerable mainly due to the presence of features such as "*Too*" and "*many elements*" (high positive SHAP value). In this paper, we use such code features to localize the vulnerability-related code statements (e.g., in the figure, the

[5] https://code.google.com/archive/p/word2vec
[6] https://code2vec.org



Statement	$SHAP_{stm}$	$SHAP_{stm-abs}$
1	0.016	0.016
2	0.049	0.049
3	0.014	0.014
4	0.004	0.004
5	-0.211	**0.421**
6	**0.161**	**0.161**
* 7	* **0.110**	* **0.207**
8	* -0.052	* 0.054
9	* **0.870**	* **0.984**
10	0.008	0.008
11	**0.334**	**0.334**
12	**0.197**	**0.197**
13	**0.138**	**0.138**
14	0.027	0.027
15	-0.013	**0.198**
16	-0.001	0.001
17	-0.001	0.001
18	-0.001	0.001

TH_{06avg}=0.056 TH_{06max}=0.094

Figure 1: Example of SHAP applied to a Java function code

ones with bugs and '*' in the table). The idea is to compute the SHAP value for each statement of a code snippet, and then select the statements that contain features with high positive values as vulnerability-related ones. We applied the following algorithms to identify the **selection threshold** (TH), i.e., the threshold used to select the vulnerability-related statements (the ones having equal or higher SHAP value). In detail:

- TH_{06avg}: (1) for each statement of a code snippet, we compute the statement-level SHAP value (named $SHAP_{stm}$) as the sum of the SHAP values of all tokens that compose the statement divided by the number of tokens; (2) then, we compute the average value of all measured $SHAP_{stm}$ related to the statements of a code snippet (named $SHAP_{stm-avg}$); and (3) then, we select as vulnerability-related statements of the snippet, the ones with a SHAP value higher than or equal to the 60% of the $SHAP_{stm-avg}$ value (this threshold is named TH_{06avg}).
- TH_{06max}: (1) for each statement of a code snippet, we identified the statement-level SHAP value (named $SHAP_{stm-abs}$) as the maximum absolute SHAP values among the ones of all tokens that compose the statement; (2) then, we compute the average value of all measured $SHAP_{stm-abs}$ related to the statements of a code snippet (named $SHAP_{stm-abs-max}$); and (3) then, we select as vulnerability-related statements of the snippet, the ones with a SHAP value higher than or equal to the 60% of the $SHAP_{stm-abs-max}$ value (this threshold is named TH_{06max}.).

Figure 1 shows the values computed for $SHAP_{stm}$ and $SHAP_{stm-abs}$, the obtained thresholds, and, in bold in the table, the statements selected to be classified as vulnerability-related by using the two thresholds: TH_{06avg} and TH_{06max}. The figure additionally shows an example of "minimal distance" between an actual, but not detected, vulnerability-related statement (statement nr.8) and the closest selected vulnerability-related statement (statement nr.6). As shown in the example, the "minimal distance" is defined as the number of statements between a statement and the closest vulnerability-related statement.

We are conscious that different algorithms for identifying the **selection threshold** (TH) could be applied and lead to different results. We opted for these algorithms since, on the one hand, we aim at using flexible values of *TH* tuned on the obtained SHAP values that allow us to select statements among the ones of the code snippets under analysis, and on the other hand, after some trials we noticed that variants of the algorithms achieve comparable results.

3 EXPERIMENT

We conducted an experiment in which SHAP[7] has been applied with VulDeePecker[8] and JavaBert[9] to a dataset of Java code snippets. We aim at evaluating the adoption of explainability and deep-learning methods for detecting and localizing vulnerability-related code statements. The obtained raw data are available online[10].

3.1 Research Questions and Metrics

3.1.1 RQ1. What is the capability of deep-learning methods to detect vulnerable code snippets?

[7] https://github.com/slundberg/shap
[8] https://github.com/tmv200/ml4code-Java
[9] https://github.com/TQRG/VDET-for-Java
[10] https://tinyurl.com/55mnd7ev

RQ1 aims at evaluating the capability of deep-learning methods to detect vulnerable code snippets. To answer RQ1, we measured the performance of VulDeePecker and JavaBert in the classification of code snippets. In our case, a code snippet is a Java function extracted from the Java code of the object dataset (see below). We compared the classification obtained from deep-learning methods with the gold standard (ground truth), in which the vulnerable code snippets are known in advance (information provided within the dataset). We then compute the number of true positives (TP, vulnerable code snippets that have been correctly classified), false positives (FP, non-vulnerable code snippets that have been wrongly classified), true negatives (TN, non-vulnerable code snippets that have been correctly classified), and false negatives (FN, vulnerable code snippets that have been wrongly classified). We hence measured the following metrics:

- $bAcc = \frac{(Sensitivity+Specificity)}{2}$ $bAcc$ is the balanced accuracy: it indicates how many times the classified correctly classifies the analyzed code snippets. $bAcc$ is often used in the case of imbalanced dataset (i.e., when one class is more frequent than the other) and it is computed starting from sensitivity and specificity.
- $Sensitivity = \frac{TP}{TP+FN}$ indicates the fraction of vulnerable code snippets the classifier is able to detect.
- $Specificity = \frac{TN}{TN+FN}$ indicates the fraction of non-vulnerable code snippets the classifier is able to detect.
- $Prec = \frac{TP}{TP+FP}$. Precision $Prec$ indicates the fraction of actually vulnerable code snippets correctly classified among the ones retrieved as vulnerable.
- $Rec = \frac{TP}{TP+FN}$. Recall Rec indicates the fraction of vulnerable code snippets that were correctly identified.
- $F1 = 2 * \frac{Precision*Recall}{Precision+Recall}$. F1-score $F1$ is the harmonic mean of precision and recall.

Deep-learning methods aim at achieving high accuracy and F1-score when detecting vulnerabilities in source code. Even if RQ1 is not new in the literature [15, 17], in our experiment, the performance achieved by the methods in this step is of interest for the subsequent RQs. In fact, the explainability method adopted to localize vulnerability-related code statements is built on top of such deep-learning methods, hence, knowing the capability of detecting vulnerable code of such methods is a key aspect to be considered for avoiding misleading interpretations of the achieved results.

3.1.2 *RQ2. What is the capability of explainability and deep-learning methods to localize source code statements related to vulnerabilities?*

RQ2 aims at evaluating the capability of explainability and deep-learning methods to localize vulnerability-related code statements inside the vulnerable code snippets. To answer RQ2, we observed that the results of explainability and deep-learning methods applied at statement-level can be seen as a fine-grained classification of each statement of a code snippet. A statement is classified as vulnerability-related if its SHAP value is higher or equal to the identified **selection threshold** (TH): in such a case, the code statement is selected by the explainability technique as vulnerability-related. We measured the performance of this (fine-grained) classification as done for RQ1 (by computing $bAcc$, $Prec$, Rec, and $F1$) but at statement-level. For RQ2, the gold standard has been inferred by the authors from the code commits provided in the object dataset and that document the security vulnerability observed in code changes. To obtain the statement-level gold standard (in which each code statement is labeled as vulnerable and non-vulnerable), we started from the existing literature (e.g., [12]) that suggests that all statements that are removed in vulnerability-related fixing commits have to be considered as vulnerable statements. Some works suggest also that others potential vulnerability-related statements can be identified by considering all statements that have control or data dependencies with the added statements. In this work, to be conservative in our results, we considered as vulnerability-related only the statements removed in vulnerability-related fixing commits in our dataset. To further investigate RQ2, we also compute the minimal distance between the non-detected vulnerability-related code statements and the ones selected as vulnerability-related by the explainability method, if any (in case of multiple non-detected vulnerable statements, we computed the average distance). For instance, in Figure 1 the minimal distance between the non detected vulnerability-related statement (statement nr.8) and the closest code statement identified as vulnerability-related by the explainability method (statement nr.6) is 1.

3.1.3 *RQ3. What is the "distraction degree" introduced by the explainability and deep-learning methods in the localization of vulnerability-related statements?*

RQ3 aims at evaluating how much "*distracting*" is the output provided by the explainability method for developers who have the objective of localizing the vulnerable code statements in a code snippet. To answer RQ3, we evaluated the effectiveness of the output produced by the explainability method by measuring the ($Selection$) percentage of vulnerability-related code statements selected by the explainability method with respect to the statements of the code snippet, and the ($Distraction$) percentage of wrongly selected code statements with respect to the snippet's statements.

- $Selection = \frac{sel}{vulns}$, where sel is the number of vulnerability-related code statements selected by the explainability, and $vulns$ is the number of code statements of the snippet.
- $Distraction = \frac{sel}{vulns}$, where sel is the number of vulnerability-related code statements wrongly selected by the explainability, and $vulns$ is the number of code statements of the snippet.

3.1.4 *RQ4. What are the most relevant features used by the explainability and deep-learning methods for the localization of vulnerability-related statements?*

RQ4 aims at identifying which features (Java code tokens) are used by explainability and deep-learning methods for detecting the vulnerable code snippets and localizing their vulnerability-related statements. To answer RQ4, we identified the most relevant code features that contributed to the decision made when classifying a code snippet and its statements. To this aim, we identified the top-10 most relevant code features used by the deep-learning methods [21]: (i) local-occurrence: we identified the first ten code features, based on their SHAP values, used for classifying each code snippet, we then counted how many times each feature is in the top-10 of the analyzed snippets, and, finally, we ranked the features accordingly; (ii) global-occurrence: we considered all code snippets as a whole and identified all code features, then we computed how many times

each feature has been used to identify the vulnerability-related code statements, and, finally, we ranked the features accordingly.

3.2 Object

We used the Java code provided by the Project KB[11]. Project KB is a vulnerability data knowledge-base. It provides a set of security-relevant commits extracted, and manually curated, from several open-source code repositories (real-world software and Github commits). In detail, it provides: information about the Java code that contains security vulnerabilities; the vulnerabilities' type, according to the National Vulnerability Database (NVD) and Common Vulnerabilities and Exposures (CVE); and the code used to fix such vulnerabilities. The used Java code set is composed of: (i) 3770 Java files in two versions each (before and after the vulnerability fix); (ii) 20155 Java functions, out of which 2467 (12%) are labeled as vulnerable, i.e., they contain at least one statement related to one of the 496 vulnerability CVEs considered in the dataset; and (iii) more than 1.3M lines of code (e.g., statements, comments) before the vulnerability fixes and more than 1.4M lines, after the fixes.

3.3 Experiment process

(1) We prepared the Java dataset to be used with VulDeePecker and JavaBert, so that each code snippet is a Java function of the dataset for which we know both vulnerability and fix.
(2) We randomly shuffled and then splitted the set of Java functions in training (80%), validation (10%), and test (10%) sets – each set contains about 12% of actually vulnerable Java functions.
(3) We provided both training and validation sets to VulDeePecker and JavaBert for building their AI models. In this step, we configured VulDeePecker and JavaBert as suggested in the existing literature and validated the obtained models in some ad-hoc preliminary executions. Table 1 reports the configuration setup applied in the experiment.
(4) We provided the 2014 Java functions (our code snippets) of the test set to the built AI models for classifying them. Note that in the case of JavaBert, we used the pre-trained model [22] provided via the Hugging Face Hub[12] and we considered two fine-tuned models on top of this: (i) the one tuned with our dataset (simply named *JavaBert* hereafter), and (ii) the one tuned with VDET [17] (named $JavaBert_{VDET}$ hereafter), as baseline for the comparison.
(5) On the obtained results, we computed the classification performance metrics (*bAcc*, *Prec*, *Rec*, and *F*1) for answering RQ1.
(6) We ran SHAP with VulDeePecker and JavaBert to obtain the SHAP values for each code statement of a subset of 500 (out of 2014) randomly selected Java functions of the test set.
(7) We applied the two algorithms on such SHAP values for identifying the *selection threshold* (see *TH* in Section 2) and we used it to select the vulnerability-related code statements, i.e., statements with SHAP values higher than or equal to *TH*.
(8) By considering the selected vulnerability-related code statements, we computed the metrics to answer RQ2 and RQ3.
(9) We repeated step 2, 3, and 4 three times for deep-learning method and considered epoch value, while step 6 and 7 have been repeated two times for method and one time per epoch.

3.4 Threats to Validity

We identify the threats that can hamper the validity of our results. The adoption of only one dataset as source of code and vulnerabilities clearly limits the generalization of the achieved results. Repeating the experiment with more datasets is an appealing option for future work. We however adopted a real-world dataset recognized as effective to train AI models for security purposes. The used dataset is imbalanced but it is obviously difficult to have a balanced dataset when we aim at working at statement-level. This limits the validity of our results, but it is the common setup in software development (few lines of code contain bugs). Another threat concerns the adoption of few AI methods; this could limit the generalization of the achieved results. About the AI methods, we opted for two well-known and representative methods that can achieve reasonable results according to the existing literature, as confirmed in our experiment. About the explainability method, we opted for SHAP since it is a well-known effective black-box method that can be applied to both considered deep-learning methods. We are conscious that adopting more methods could strengthen the results, and we plan to extend the experiment with more methods in our future work. Another threat concerns the algorithms defined for the selection threshold *TH*. Different TH values can lead to different results. However, after some preliminary runs, we opted for the two presented algorithms since they can identify reasonable thresholds for our case, different choices can be object of further studies.

4 RESULTS

This section reports the results collected in the experiment with the aim of answering the RQs introduced in the previous section.

RQ1. Table 2 reports the macro average[13] and standard deviation obtained for the classification performance metrics (*bAcc*, *Prec*, *Rec*, and *F*1) measured for VulDeePecker, JavaBert, and $JavaBert_{VDET}$ to answer RQ1, investigating the capability of deep-learning methods to detect vulnerable Java functions. The table shows that both VulDeePecker and JavaBert achieve reasonable results for the considered performance metrics. In detail, VulDeePecker achieve 67% of accuracy and 60% of F1-score while JavaBert 95% of accuracy and 88% of F1-score. Conversely, $JavaBert_{VDET}$ achieves a very low performance, in particular, concerning the precision (*Prec* less than 15%). In terms of variability (*sd* in Table 2), we notice that the recall (*Rec*) for $JavaBert_{VDET}$ shows 19% of *sd* while in the other cases the variability is quite limited (less than 10%). We also note that, an increase of the *epochs* does not imply an increase of the performance. In fact, even if a trend exists, it is not always confirmed for VulDeePecker and JavaBert. Conversely, the increase of epochs improves the performance in all metrics for $JavaBert_{VDET}$.

[11] https://github.com/SAP/project-kb
[12] https://huggingface.co/CAUKiel/JavaBERT
[13] The *macro average* [11] has been computed as the average of the metrics in the different runs

Table 1: VulDeePecker and JavaBert setup

AI method	Input vector	Epoch length	Loss function	Dropout probability	Optimizer	Recurrent Activation
VulDeePecker	100	4,10	binary crossentropy	0.5	Adamax	sigmoid
JavaBert	512	4,10	BCE with logits loss	0.1	Adamax	sigmoid
JavaBert$_{VDET}$	512	10	BCE with logits loss	0.1	Adamax	sigmoid

AI method	Vector length	Vocabular Size	Training steps	Hidden Layers	Batch size	# Java functions		
						Training	Validation	Test
VulDeePecker	300	-	-	2	64	2004	235	228
JavaBert	768*4	30522	size(input)*epoch	12	12	2004	235	228
JavaBert$_{VDET}$	768*4	30522	size(input)*epoch	12	12	92485	11590	11528

RQ1 The experiment shows that both deep-learning methods can achieve very relevant results, in particular, when fine-tuned on a coherent dataset. Low performance can instead occur if third-party fine-tuned models are adopted.

RQ2. Table 3 reports the macro average and standard deviation obtained for the classification performance metrics (*bAcc*, *Prec*, *Rec*, and *F*1) measured for VulDeePecker and JavaBert to answer RQ2, investigating the capability of explainability and deep-learning methods to localize vulnerability-related code statements inside vulnerable code snippets. For this RQ, we did not consider JavaBert$_{VDET}$ due to the low results achieved in RQ1. The table shows that both deep-learning methods achieve from low to moderate results in the performance metrics. In detail, VulDeePecker achieves 51% of accuracy and 29% of F1-score, if we consider TH$_{06avg}$, and 53% of accuracy and 25% of F1-score, if we consider TH$_{06max}$. JavaBert, instead, achieves 56% of accuracy and 41% of F1-score, if we consider TH$_{06avg}$, and 51% of accuracy and 39% of F1-score, if we consider TH$_{06max}$. In particular, the observed precision tends to be low: less than 33% and 43% for VulDeePecker and JavaBert respectively. In terms of variability (*sd* in Table 3), we observe a high variability for the recall of VulDeePecker, while in the other cases the variability is limited. From the table, we also note that for both VulDeePecker and JavaBert, an increase of the epochs does not always imply an increase of the performance.

To further investigate these results, we also computed the minimal distance between the vulnerability-related code statements selected by the explainability method and the set of statements that are actually related to the security vulnerability, according to the gold standard. Table 4 reports the overall number of considered statements of the Java functions of our test set, the number of statements for which SHAP was able to compute meaningful SHAP values (e.g., excluding cases for which the SHAP value is very low values, e.g., 0.001 in Figure 1, and for very long Java code), and the number of statements that are related to security vulnerabilities. We observe that SHAP was able to compute values for most of the code, in particular, for 71% of the statements when working with VulDeePecker, and 57.5% of the statements when working with JavaBert. The Java functions analyzed contain, on average, 12% of vulnerability-related statements. The table also reports the average minimal distance computed for each vulnerability-related code statement with respect to the nearest statement selected as vulnerability-related by the explainability method. This distance is always lower than 9 statements, corresponding at maximum at 37% of statements of the considered Java functions.

RQ2 The experiment shows that explainability and deep-learning methods achieve low to moderate results. In particular, it shows a limited capability in precisely localizing the vulnerability-related code statements, even if we observe a positive trend in the identification of statements that are close to the actual vulnerable ones.

RQ3. Table 5 reports the metric to measure the distraction degree for VulDeePecker and JavaBert introduced by deep-learning and explainability methods in the localization of the vulnerability-related code statements. In detail, the table reports (i) the average (macro) percentage of vulnerability-related code statements identified by the explainability method with respect to the total number of statements of the code snippet (column "Selection"); and, out of these selected statements, the percentage of statements wrongly classified as vulnerability-related (column "Distraction"). For instance, 74% of the code statements have been selected and classified as vulnerability-related in the case of VulDeePecker with 4 Epochs and TH_{06avg}. The table shows that the percentage of code statements that have been identified as vulnerability-related ranges between 29% and 69% for VulDeePecker and around 48.5% for JavaBert. Furthermore, we can see that the distraction degree varies between 21% and 63% for VulDeePecker and around 29% for JavaBert. JavaBert shows a quite stable result, while a large variability affected the result of VulDeePecker.

RQ3 The experiment shows that explainability and deep-learning methods provide results with a moderate to high percentage of potentially distracting information. In particular, VulDeePecker can produce results with a high degree of distracting information, while results of JavaBert are quite reasonable and stable.

RQ4. Table 6 reports the top-10 code features used by VulDeePecker and JavaBert to detect vulnerable code snippets and their vulnerability-related statements. The most relevant features are listed as derived from both local and global-occurrence analysis. Moreover, the table reports their type (e.g., symbols, Java keywords, Java types and classes, operation - e.g., call, identifier) and their occurrence in percentage. The table shows that the most relevant features are mainly Java symbols and language keywords for JavaBert, while the set of VulDeePecker's features varies: besides symbols and keywords it also contains identifiers, types and classes. Moreover, the table shows that the occurrence of the VulDeePecker's top-10 features is higher than the ones of JavaBert (29% to 99% and 6% to 35%, respectively). This indicates that, differently than JavaBert, VulDeePecker tends to classify code snippets based on a quite limited set of features, i.e., few code tokens strongly influence

Table 2: RQ1: Performance metrics for vulnerability detection for code snippets

Method	Epoch	*bAcc*		*Prec*		*Rec*		*F1*	
		avg	sd	avg	sd	avg	sd	avg	sd
VulDeePecker	4	**0.67**	0.02	**0.64**	0.06	**0.57**	0.16	**0.59**	0.07
	10	**0.68**	0.03	**0.63**	0.02	**0.60**	0.08	**0.61**	0.04
	4&10	**0.67**	0.02	**0.64**	0.04	**0.58**	0.12	**0.60**	0.06
JavaBert	4	**0.94**	0.03	**0.80**	0.02	**0.89**	0.06	**0.84**	0.02
	10	**0.95**	0.02	**0.91**	0.04	**0.91**	0.04	**0.91**	0.04
	4&10	**0.95**	0.02	**0.86**	0.07	**0.90**	0.05	**0.88**	0.05
JavaBert$_{VDET}$	4	0.49	0.06	0.09	0.02	0.46	0.09	0.15	0.04
	10	**0.56**	0.04	0.15	0.05	**0.79**	0.10	0.24	0.08
	4&10	**0.53**	0.06	0.12	0.05	**0.65**	0.19	0.20	0.08

Table 3: RQ2: Performance metrics for localization of vulnerability-related code statements

Method	Epoch	*bAcc*		*Prec*		*Rec*		*F1*	
		avg	sd	avg	sd	avg	sd	avg	sd
TH$_{06avg}$									
VulDeePecker	4	0.49	0.04	0.25	0.04	**0.70**	0.09	0.29	0.08
	10	**0.53**	0.05	0.28	0.04	**0.63**	0.48	0.29	0.07
	4&10	**0.51**	0.04	0.26	0.04	**0.66**	0.28	0.29	0.06
JavaBert	4	**0.60**	0.09	0.41	0.1	**0.61**	0.13	0.43	0.10
	10	**0.52**	0.05	0.43	0.28	**0.55**	0.06	0.40	0.23
	4&10	**0.56**	0.08	0.42	0.2	**0.58**	0.1	0.41	0.17
TH$_{06max}$									
VulDeePecker	4	**0.56**	0.09	0.33	0.13	0.34	0.04	0.28	0.09
	10	**0.50**	0.02	0.25	0.11	0.27	0.04	0.22	0.07
	4&10	**0.53**	0.06	0.29	0.11	0.31	0.05	0.25	0.07
JavaBert	4	**0.55**	0.1	0.40	0.07	**0.55**	0.18	0.42	0.07
	10	0.45	0.09	0.38	0.2	0.32	0.24	0.29	0.18
	4&10	**0.51**	0.1	0.39	0.14	0.45	0.23	0.39	0.14

Table 4: RQ2: The average (minimal) distance between the actual vulnerability-related code statements and the nearest one selected by the explanability method

Method	Epoch	Java function			Distance (%)			
		Statements	Statements with SHAP value	Vulnerability-related Statements avg	TH$_{06avg}$ Statements		TH$_{06max}$ Statements	
					avg	sd	avg	sd
VulDeePecker	4	1029	669	102	0.08	0.21	7.19	9.16
	10	759	609	134	2.94	4.82	7.22	6.77
	4&10	1788	1278	236	1.77	3.96	7.21	7.79
JavaBert	4	1325	837	138	8.56	17.86	8.70	18.05
	10	863	423	184	1.54	2.40	3.43	3.37
	4&10	2188	1260	322	5.05	10.13	6.06	10.71

the result. Furthermore, the lists of the top-10 features support the preliminary findings of state-of-the-art works [21] according to which these deep-learning methods mostly rely on special characters of programming languages, that is learning and using spurious correlations to make decisions.

RQ4 The experiment shows that the code features used by deep-learning methods are mainly related to language symbols and keywords. VulDeePecker's decisions, in particular, are based on a quite limited set of features, while JavaBert tends to use a larger set of features.

5 DISCUSSION

We observed that deep-learning methods could achieve interesting results in the detection of vulnerable code snippets, in particular, if: (i) the target code is coherent with the code used to fine-tune the model; and (ii) the setup of the learning method is adequate. This seems to suggest that this kind of methods are particularly suitable to test subsequent versions of a software. For instance, we can assume that they can be successfully applied in regression testing. In our experience, JavaBert overcomes Vuldeepecker in terms of detection performance. This is expected since JavaBert is more effective in managing longer sequences of code and a larger vocabulary of symbols, as well as in abstracting from code entities without loosing their characteristics. Furthermore, JavaBert seems

Table 5: RQ3: the table reports the average (macro) percentage of selected vulnerability-related code statements and out of these statements, the average percentage of wrongly classified ones

Method	Epoch	Selection (%)		Distraction (%)	
		avg (%)	sd	avg (%)	sd
TH_{06avg}					
VulDeePecker	4	0.74	0.27	0.70	0.33
	10	0.64	0.29	0.56	0.30
	4&10	0.69	0.28	0.63	0.32
JavaBert	4	0.50	0.18	0.30	0.24
	10	0.48	0.21	0.27	0.23
	4&10	0.49	0.19	0.29	0.23
TH_{06max}					
VulDeePecker	4	0.28	0.22	0.20	0.16
	10	0.30	0.21	0.23	0.20
	4&10	0.29	0.21	0.21	0.18
JavaBert	4	0.51	0.17	0.30	0.30
	10	0.42	0.42	0.27	0.27
	4&10	0.48	0.48	0.29	0.29

to be less sensitive to the parameter values, although finding the most adequate setup is not an easy task for both methods.

We also observed that, concerning the localization of the vulnerability related code statements inside vulnerable code snippets, JavaBert still overcomes VulDeePecker but the result is only partially satisfactory. Focusing on the explainability technique applied on the top of these deep-learning methods in fact, we observed that the SHAP values we obtained for large parts of the analyzed code is low, while the base SHAP value tends to be high (e.g., around 0.9), indicating that the contribution of the features is actually quite limited. In particular, we observed that:

- the vulnerability-related code statements localization performance of the explainability is limited, even if our result is inline with the existing correlated literature, e.g., LineVD [9] and Line-DP [23];
- the statements suggested as vulnerability-related by the explainability are, however, quite close (in terms of statements) to the actual vulnerable statements;
- the amount of misleading information based on the SHAP explaination that can be source of "distraction" for the developers, who have to conduct bug hunting, is overall limited.

Overall, we can say that traditional static code analysis based on data-flow or pattern analysis is recognized as effective in detecting software security vulnerabilities but at the expense of a high human effort required for checking a large number of produced false positive cases. In this regard, (1) deep-learning methods can support the traditional static analysis by suggesting the portions of code where the attention of the testers should be focused since they can contain vulnerabilities. Furthermore, (2) the adoption of explainability techniques with such deep-learning methods can additionally indicate the point where to start analyzing the code to localize the vulnerability-related statements. In fact, the use of the explainability technique with deep-learning methods to localize vulnerability-related code statements shows some potentiality but it is only partially adequate and needs to be improved with further study and analysis.

According to our experience, the most relevant limitations we observed by using deep-learning methods to localize vulnerabilities seem to be related to the applied algorithms and, in particular, they concern:

- the use of a too coarse-grained granularity of the items used as input for training the deep-learning algorithms;
- the tokenization techniques adopted by the algorithms, inspired by the one used for natural-language processing;
- the code features used by the deep-learning algorithms to identify the portions of vulnerable code that seem not to be related to the actual vulnerability causes.

We believe that the improvement of these aspects of the adopted deep-learning algorithms could lead to a substantial improvement of their vulnerability detection and localization performance. Our future work will focus on the improvement of the existing deep-learning methods by:

- training the AI models with input at statement-level rather than at function-level, thus driving the models to work at a fine-grained granularity;
- adopting more programming-language oriented encoding and embedding techniques;
- extending the sources of information in input for training the deep-learning algorithms, thus avoiding to analyze only the source code as a (plain) text, and using additional knowledge extracted from other sources, e.g., from the commits in the code version system, source code data-flow information.

In this last respect, however, we can also consider the adoption of a white-box explainability technique that could work together with the deep-learning method rather than on top of it. We believe that this could lead to more precise results for the identification of meaningful and relevant features useful to detect the vulnerable portions of code. We also believe that these aspects can represent directions for future research in the field.

6 RELATED WORK

Software fault detection and localization is a relevant topic for the community, several works have been proposed and surveys exist that document and analyze them (e.g., [24, 26]). Some of these work adopt machine learning and deep-learning methods. For instance, [18] presents a tool that uses statistical causal inference and a machine-learning algorithm (random forests) to automatically localizing faults in software.

In this work, however, we focus on specific kinds of faults that are known to lead to security vulnerabilities. Some static approaches use machine and deep-learning methods to detect vulnerable code snippets [4]. For instance, VulDeePecker [15], then evolved in SeVCs framework [14], uses deep-learning to detect vulnerabilities in code gadgets (i.e., semantically related code statements). Such gadgets are encoded in fixed-length vectors of symbols and passed to a BLSTM neural network. μVulDeePecker [27] extends VulDeePecker to use multiclass classification for detecting vulnerabilities. Devign [25] represents code snippets with a graph structure built from multiple representations (e.g, Abstract Syntax Tree) and it uses graph neural networks (GNN) for graph-level classification. BGNN4VD [3] uses bidirectional graph neural networks (BGNN) for vulnerability detection. IVDetect [12] uses also a graph-based neural network

Table 6: RQ4: Top-10 code features according to their local/global-occurrence

Method	Local-Occurrence			Global-Occurrence		
	Code feature	Type	Occurrence	Code feature	Type	Occurrence
VulDeePecker	public	keyword	0.99	public	keyword	0.99
	(	symbol	0.99	boolean	type	0.52
	)	symbol	0.69	attachments	var	0.49
	boolean	type	0.52	return	keyword	0.49
	{	symbol	0.49	Source	interface	0.49
	arg0	var	0.29	(	symbol	0.49
	Attachment	class	0.29	arg0	var	0.29
	attachments	var	0.29	toStreamSource	operation	0.29
	return	keyword	0.29	throws	keyword	0.29
	fileName	var	0.29	StreamSource	class	0.29
JavaBert	(	symbol	0.35	(	symbol	0.35
	public	keyword	0.31	;	symbol	0.31
	)	symbol	0.27	public	keyword	0.31
	.	symbol	0.24	)	symbol	0.27
	}	symbol	0.22	.	symbol	0.24
	{	symbol	0.2	{	symbol	0.2
	=	symbol	0.18	=	symbol	0.18
	void	keyword	0.08	}	symbol	0.15
	new	keyword	0.07	void	keyword	0.08
	throws	keyword	0.06	new	keyword	0.07

to predict vulnerabilities at the function-level. JavaBert [17, 22] adopt the BERT architecture for managing long text (source code) sequences and accounting for the context of each identified text word. Empirical experiences show contrasting results concerning the adoption of such methods [4]. In fact, these methods can suffer of limitations which have been only partially alleviated by: (i) capturing syntax and semantics characteristics of code and vulnerabilities, (ii) managing long code sequences, (iii) abstracting from code entities without loosing their characteristics; and (iv) managing a large vocabulary of symbols.

Explainable AI has been used for identifying the features used by such methods to make decisions [20, 21]. Deep-learning methods, however, seem to detect vulnerabilities by only partially capturing the cause of the vulnerabilities in the code. Indeed, they tend to learn artifacts that are dataset or language specific [21], e.g., variable identifiers, specific symbols. Moreover, several problems can affect the results, e.g., imbalanced datasets, label inaccuracy, spurious correlations on the data used to make a decision, biased parameter selection [1]. In particular, [20] demonstrated that: (i) explainability provides meaningful information to developers about code features, and (ii) the most relevant features are often project-specific.

Deep-learning methods have been applied to detect vulnerable code snippets but their capability of localizing the cause of the vulnerability in the code has not been extensively explored yet. Differently from the existing literature, we investigate the use of code features to localize vulnerability-related code statements. Similar to our work's goal, [9] presents LineVD to identify vulnerability-related code statements by leveraging control and data dependencies between statements in graph neural networks, and a transformer-based model to encode the code tokens. A transformer-based model is also used in LineVul [7]. Instead, [6] presents SedSVD that uses a graph structure to embed an enriched graph representation of code snippets and use graph neural networks (GNN) for detecting vulnerability-related code statements. Differently from these works, we do not present a new approach specifically designed to identify vulnerability-related code statements but we leverage existing deep-learning methods and explainability techniques to understand if relevant features can support the identification of the vulnerability cause in the code. [23] presents Line-DP that uses a model-agnostic explainability method, named LIME [19], with a logistic regression method to identify the defective statements, i.e., statements that contain tokens leading to software defects. Similarly to [23], we investigate the adoption of explainability methods for statement-level bugs localization. Differently from [23], however, we focus on security vulnerabilities and we use SHAP with well-known deep-learning methods.

7 CONCLUSIONS

We reported an experiment on the use of explainability and deep-learning methods for detecting and localizing vulnerable code. The results show that there is room for improvement since such methods: (i) are effective in identifying vulnerable code snippets, but under certain conditions; and (ii) seem to use code features that are only partially related to the causes of the vulnerabilities in the code.

In our opinion, the main limits of such methods are mainly related to: (i) the granularity of the information used in the training: (ii) the adopted encoding and embedding techniques; and (iii) the limited knowledge that they can learn by analyzing only the source code.

Our future work, hence, will be devoted to improve the adopted deep-learning methods in these three directions, thus increasing their capability of localizing vulnerable code statements.

ACKNOWLEDGMENTS

The work was supported by the European Union Horizon Europe Research and Innovation Programme under Grant 101120393 (Sec4Ai4Sec).

REFERENCES

[1] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and Don'ts of Machine Learning in Computer Security. In *31st USENIX Security Symposium (USENIX Security 22)*. 3971–3988.

[2] Nadia Burkart and Marco F. Huber. 2021. A Survey on the Explainability of Supervised Machine Learning. *Journal of Artificial Intelligence Research* 70 (jan 2021), 245–317.

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

[4] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48, 09 (sep 2022), 3280–3296.

[5] Xiarun Chen, Qien Li, Zhou Yang, Yongzhi Liu, Shaosen Shi, Chenglin Xie, and Weiping Wen. 2021. VulChecker: Achieving More Effective Taint Analysis by Identifying Sanitizers Automatically. In *20th Intern. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 774–782.

[6] Yukun Dong, Yeer Tang, Xiaotong Cheng, Yufei Yang, and Shuqi Wang. 2023. SedSVD: Statement-level software vulnerability detection based on Relational Graph Convolutional Network with subgraph embedding. *Information and Software Technology* 158 (2023), 107168.

[7] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-Based Line-Level Vulnerability Prediction. In *19th Inter. Conf. on Mining Software Repositories* (USA) *(MSR)*. ACM, 608–620.

[8] Harman Singh - Infosecurity magazine. 2022. The Top Security Vulnerabilities of 2022 and Their Workaround.

[9] David Hin, Andrey Kan, Huaming Chen, and M.Ali Babar. 2022. LineVD: Statement-Level Vulnerability Detection Using Graph Neural Networks. In *19th International Conference on Mining Software Repositories* (USA) *(MSR)*. ACM, 596–607.

[10] Mordor Intelligence. 2023. Security Testing Market Size and Share Analysis - Growth trends and forecasts (2023 - 2028).

[11] Takahashi Kanae, Yamamoto Kouji, Kuchiba Aya, and Koyama Tatsuki. 2022. Confidence interval for micro-averaged F1 and macro-averaged F1 scores. *Applied Intelligence* 52, 5 (2022), 4961–4972.

[12] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability Detection with Fine-Grained Interpretations *(ESEC/FSE 2021)*. Association for Computing Machinery, 292–303.

[13] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis *(ACSAC '16)*. ACM, New York, NY, USA, 201–213.

[14] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 04 (jul 2022), 2244–2258.

[15] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, and Hai Jin. 2018. VulDeePecker: A Deep Learning-Based System forVulnerability Detection. In *Network and Distributed Systems Security (NDSS) Symposium*. USA.

[16] Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 4768–4777.

[17] Cláudia Mamede, Eduard Pinconschi, and Rui Abreu. 2023. A Transformer-Based IDE Plugin for Vulnerability Detection. In *37th Intern. Conf. on Automated Software Engineering* (USA) *(ASE)*. ACM, Article 149, 4 pages.

[18] Andy Podgurski and Yiğit Küçük. 2020. CounterFault: Value-Based Fault Localization by Modeling and Predicting Counterfactual Outcomes. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 382–393. https://doi.org/10.1109/ICSME46990.2020.00044

[19] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier *(KDD)*. ACM, 1135–1144. https://doi.org/10.1145/2939672.2939778

[20] Geanderson Santos, Eduardo Figueiredo, Adriano Veloso, Markos Viggiato, and Nivio Ziviani. 2021. Predicting Software Defects with Explainable Machine Learning. In *XIX Brazilian Symposium on Software Quality* (Brazil) *(SBQS)*. ACM, Article 18, 10 pages.

[21] Angelo Sotgiu, Maura Pintor, and Battista Biggio. 2022. Explainability-Based Debugging of Machine Learning for Vulnerability Discovery. In *17th Intern. Conf. on Availability, Reliability and Security* (Austria) *(ARES)*. ACM, New York, NY, USA, Article 113, 8 pages.

[22] N. De Sousa and W. Hasselbring. 2021. JavaBERT: Training a Transformer-Based Model for the Java Programming Language. In *36th Intern. Conf. on Automated Software Engineering Workshops (ASEW)*. IEEE, USA, 90–95.

[23] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2022. Predicting Defective Lines Using a Model-Agnostic Technique. *IEEE Transactions on Software Engineering* 48, 5 (2022), 1480–1496.

[24] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[25] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. Curran Associates Inc., USA.

[26] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2021. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering* 47, 2 (2021), 332–347. https://doi.org/10.1109/TSE.2019.2892102

[27] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2021. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2021), 2224–2236.