Machine Learning (2026) 115:94
https://doi.org/10.1007/s10994-025-06902-1

Evaluating line-level localization ability of learning-based code vulnerability detection models

Marco Pintore[1] · Giorgio Piras[1] · Angelo Sotgiu[1,2] · Maura Pintor[1,2] · Battista Biggio[1,2]

Received: 25 February 2025 / Revised: 29 August 2025 / Accepted: 21 September 2025
© The Author(s) 2026

Abstract

To address the extremely concerning problem of software vulnerability, system security is often entrusted to Machine Learning (ML) algorithms. Despite their now established detection capabilities, such models are limited by design to flagging the entire input source code function as vulnerable, rather than precisely localizing the concerned code lines. However, the detection granularity is crucial to support human operators during software development, ensuring that such predictions reflect the true code semantics to help debug, evaluate, and fix the detected vulnerabilities. To address this issue, recent work made progress toward improving the detector's localization ability, thus narrowing down the vulnerability detection "window" and providing more fine-grained predictions. Such approaches, however, implicitly disregard the presence of spurious correlations and biases in the data, which often predominantly influence the performance of ML algorithms. In this work, we investigate how detectors comply with this requirement by proposing an explainability-based evaluation procedure. Our approach, defined as *Detection Alignment* (DA), quantifies the agreement between the input source code lines that most influence the prediction and the actual localization of the vulnerability as per the ground truth. Through DA, which is model-agnostic and adaptable to different detection tasks, not limited to our use case, we analyze multiple learning-based vulnerability detectors and datasets. As a result, we show how the predictions of such models are consistently biased by non-vulnerable lines, ultimately highlighting the high impact of biases and spurious correlations.

Keywords Software vulnerability · Vulnerability detection · Explainable AI · Interpretable AI

Editors: Annalisa Appice, Giuseppina Andresini, Przemyslaw Biecek, Christian Wressnegger.

Extended author information available on the last page of the article

Published online: 09 April 2026

1 Introduction

From the security perspective, the last decades' fast-paced development of software has been characterized by source code vulnerabilities getting exploited by attackers. Such software security issues, in practice, have been represented by a growing number of threats, such as privilege escalation and remote code execution. The situation may be worsened by the rising use of AI-based automated code generators: while these tools accelerate development, they can introduce vulnerabilities learned from their training data (Pearce et al. 2025). To prevent the occurrence of these issues, practitioners often resort to Static Application Security Testing (SAST), which is used during software development to identify potential vulnerabilities, thus enriching the development process by providing immediate feedback. Conversely, to ensure correct code functioning, Dynamic Application Security Testing (DAST) is often employed to analyze the code at runtime and identify potential vulnerabilities tied to the software's dynamic behavior. The effectiveness of both approaches, however, is mostly constrained to detecting well-known vulnerabilities, as they typically fail to detect *never-before-seen* vulnerabilities. Such limitation has led modern detection approaches to base their functioning on learning techniques, which can leverage large datasets of known benign and malicious code to build a predictive model for detecting new vulnerabilities (Chakraborty et al. 2022). Following the success of Machine Learning (ML) techniques across different domains, a relevant portion of source code vulnerability detection techniques has leveraged such models and notably improved state-of-the-art performances (Li et al. 2018; Zhou et al. 2019). More recently, vulnerability detection has also been extended to transformer-based architectures (Lu et al. 2021; Fu and Tantithamthavorn 2022), bringing further improvement in the detection performance. However, a now broad literature recognizes how the performance of learning-based models, which are predominantly trained in simulated or unrealistic scenarios, is often determined by the presence of spurious correlation or biases in the data, rather than generalizable patterns (Chakraborty et al. 2022). Besides limiting the real-world applicability of the models, this issue additionally emphasizes the potential of explainability techniques for studying on what evidence these models ground their decisions. In fact, it is of high relevance to ward off learning-based code vulnerability detectors to predict the presence of a vulnerability based on spurious correlations, such as less relevant syntactic or stylistic patterns not necessarily related to the actual vulnerable code portions (Arp et al. 2022). These misaligned patterns can, in fact, potentially reveal that the model is not learning semantically meaningful patterns relevant to the task, but instead it is exploiting artifacts present in the data. This becomes particularly evident when models are evaluated on unseen datasets that don't contain these artifacts and fail to generalize, highlighting the need for more robust and interpretable learning approaches. In this regard, despite the growing number of research papers proposing code vulnerability detectors and focusing on improving their localization ability (Li et al. 2021; Fu and Tantithamthavorn 2022), it still remains unclear, from a debugging perspective, how well their detection aligns with the ground truth and neglects the influence of non-causal patterns. We represent this scenario in Fig. 1, where a real code sample from Fan et al. (2020) is correctly classified as vulnerable by a CodeBERT model (Feng et al. 2020; Lu et al. 2021). Nevertheless, the prediction relies on non-vulnerable lines, leading the detector to incorrectly localize the vulnerability.

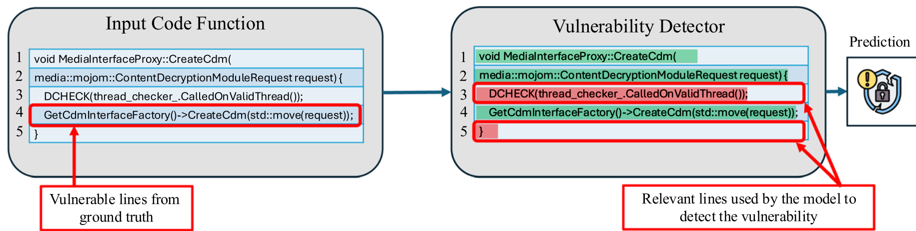

Fig. 1 Ground truth vulnerable lines against the lines indicated by the detector. We show on a sample from the BigVul dataset (Fan et al. 2020), correctly predicted as vulnerable by a finetuned CodeBERT detector from Fu and Tantithamthavorn (2022), how the localized vulnerable lines driving the prediction (3,5) are not aligned with the ground truth vulnerable line (4)

In this work, we tackle this problem by proposing an explainability-based evaluation procedure that quantifies this agreement. Our approach, which we define as Detection Alignment (DA), leverages the output of explainability techniques to assess which code lines mostly contributed to the vulnerability prediction from the ML model. Based on a careful design of two sets, DA quantifies how well the prediction aligns with the ground truth via Jaccard's index. Through our experiments, which we apply to state-of-the-art datasets and transformer-based detectors, we show how the DA metric highlights a significant influence of non-vulnerable code lines when predicting a given code function as vulnerable. We believe that our approach, besides being modular and adaptable to different applications outside of source code analysis, can serve as a stepping stone to improve the performance of future learning-based detectors and enhance the decision support for human operators.

2 Background

We describe in this section the essential concepts needed to understand the proposed method. As we base our Detection Alignment (DA) approach on source code vulnerability detectors, we first provide the formulation of a general learning-based detector in Sect. 2.1. While our method is general enough to cover any type of vulnerability-detection model, we present a specialization of DA on transformer-based models, and provide the main architecture notions in Sect. 2.2, accordingly. We then discuss, in Sect. 2.3, the explainability concepts for the overall approach comprehension.

2.1 Learning-based source code vulnerability detection

Owing to their well-known capability of analyzing and classifying wide volumes of data, Machine Learning (ML) techniques have been dedicated over the years to the task of vulnerability identification, leveraging on Graph Neural Networks (Zhou et al. 2019; Li et al. 2021) or Recurrent Neural Networks (Li et al. 2018). More recently, this learning task has also been extended to transformer-based architectures (Fu and Tantithamthavorn 2022). We provide here a general formulation that covers models predicting a label to indicate the presence of a vulnerability.

General Formulation. Let $z \in \mathcal{Z}$ be an input sample containing source code. The goal of a general ML model f is to map $z \in \mathcal{Z}$ to the label space $\mathcal{Y} = \{0, 1\}$. Specifically, we

assign the class 1 to samples in which vulnerabilities are found, and 0 to non-vulnerable samples. However, as the provided source code is in a human-readable text-based format, the model is not able to process the text “as is". Therefore, to convert the input into numerical data, the code is preprocessed through a feature-mapping function $\phi : \mathcal{Z} \mapsto \mathcal{X}$. Such mapping transforms the source code file $\boldsymbol{z}$ to a d-dimensional feature vector $\boldsymbol{x} \in \mathbb{R}^d$. From the newly obtained representation, the model $f : \mathcal{X} \mapsto \mathcal{Y}$ can output a prediction for the input sample $\boldsymbol{z}$, obtained as $\hat{y} = f(\phi(\boldsymbol{z}); \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are the parameters learned by the model and the predicted label $\hat{y}$ indicates whether the input code file is predicted as vulnerable or not. We will refer to single code samples $\boldsymbol{z}$ as functions, which in turn will be described by a set of L code lines.

2.2 Transformer-based detectors

While our DA approach can be applied to any learning-based model, we focus our application on transformer architectures (Vaswani et al. 2017) dedicated to the task of source code vulnerability detection, which we describe here accordingly.

Tokenization Mapping. The transformer preprocessing starts by converting the source code files $\boldsymbol{z}$, consisting of raw text, into a d-dimensional input viable for the model. To perform such conversion, the common procedure involves specifying a mapping function ϕ known as tokenization. The goal of a tokenizer, in the context of source code detection, is to adapt the input code to the model while preserving syntactic and structural code properties. Most typically, tokenization is performed by (1) normalizing the input text; (2) splitting the input text into units (words, subwords, characters, bytes); (3) mapping tokens to unique integer indexes according to a precomputed vocabulary; (4) padding, truncation, and special token handling to reduce variability and allow batch processing. The tokenization strategy can be performed at different levels of granularity, depending on the representation level that is required for the application. While different tokenization strategies have been proposed, we focus here on the *Byte Pair Encoding* (BPE) (Sennrich et al. 2016) sub-word tokenization approach, which has been predominantly used by transformer-based detectors. BPE operates by first normalizing the input and then breaking it into individual characters. Then, frequently co-occurring character pairs are merged to form subwords, thus obtaining more expressive tokens while avoiding huge vocabulary sizes. To also improve efficiency on novel out-of-vocabulary words, BPE finally splits unseen words into the largest known subword units.

Attention mechanism. Traditional text models such as Recurrent Neural Networks (RNNs) (Hochreiter and Schmidhuber 1997) process text sequentially, and often fail to capture long-range dependencies, which are widespread in source code. Transformer architectures, on the other hand, leverage the attention mechanism to weigh the importance of different tokens in a sentence dynamically. Specifically, transformers leverage a set of multi-head attention used to compute attention scores for every token in a sequence relating to all other tokens. Given a head h and a d-dimensional input embedding $\boldsymbol{x}$, self-attention operates by transforming each input token into three Query ($\boldsymbol{Q}_h$), Key ($\boldsymbol{K}_h$), and Value ($\boldsymbol{V}_h$) vectors as follows:

$$\boldsymbol{Q}_h = \boldsymbol{x}\boldsymbol{W}_h^Q, \quad \boldsymbol{K}_h = \boldsymbol{x}\boldsymbol{W}_h^K, \quad \boldsymbol{V}_h = \boldsymbol{x}\boldsymbol{W}_h^V \tag{1}$$

where $\boldsymbol{W}_h^Q, \boldsymbol{W}_h^K, \boldsymbol{W}_h^V \in \mathbb{R}^{d \times d_k}$ are trainable parameter matrices, and d_k is the attention subspace dimension. Then, the attention mechanism involves obtaining a probability distribution on the dot-product between query and key, scaled by the subspace dimension d_k, obtaining an attention weight matrix as follows:

$$\boldsymbol{A}_h = \text{softmax}\left(\frac{\boldsymbol{Q}_h \boldsymbol{K}_h^T}{\sqrt{d_k}}\right) \tag{2}$$

where the elements $A_{i,j}$ of the attention matrix indicate how much attention token x_i puts on x_j. Finally, the output of the attention head is computed as the product between the attention weights and value matrix as $\boldsymbol{H}_h = \text{AttentionHead} = \boldsymbol{A}_h \boldsymbol{V}_h$. Therefore, transformers rely on multiple heads to learn different aspects of token relationships. This, among others, is the case of BERT (Devlin et al. 2019), an encoder-only architecture relying on 12 layers composed of 12 heads respectively, adapted to code vulnerability detection via its CodeBERT (Feng et al. 2020) version.

2.3 Explainable AI methods

Explaining a decision from ML models is useful to debug and understand what the model learns from the data. A typical subdivision of explainability techniques involves distinguishing between *local* and *global* methods. While local explanations provide a description of specific model decisions, global ones provide a broader understanding of the model's internal representations. We focus here on local methods, which, for instance, can explain which tokens are more relevant and thus contribute the most to the prediction of a source code segment as vulnerable. In detail, we first consider raw attention scores, which can be used to understand how input tokens attend to each other and identify the most relevant ones. However, as raw attention-based explainability methods can often be unreliable, we also consider more advanced transformer-specific approaches (Achtibat et al. 2024), as well as architecture-agnostic explanations computed through integrated gradients (IG) (Sundararajan et al. 2017).

Attention-based explainability. Given a transformer's encoder layer *m*, the goal of an attention-based explainability method is to determine the contribution of each token to the final prediction. Hence, considering an input sequence $\boldsymbol{x}$, to compute the relevance of an input sample $\boldsymbol{x}_i$ we are required to sum the attention scores over the different layers' heads *h* assigned to each token $\boldsymbol{x}_j$ when attending to the token $\boldsymbol{x}_i$. A detailed implementation of our attention-based relevance computation is shown in Sect. 3.1.

Layer-wise relevance propagation for transformers. Despite being largely applied, it has been shown that attention alone is often not sufficient to fully capture the model behavior. We thus consider a more advanced approach, Attention-aware Layer-wise Relevance Propagation (AttnLRP) (Achtibat et al. 2024), that produces neuron-level relevance scores by rule-based backpropagation. Each network neuron is modeled as a function node that is individually decomposed, and the relevance values are distributed from the model output layer to its prior network neurons, one layer at a time. We refer to the paper for the detailed formulations.

Integrated Gradients. While the above-described explainability measures are clearly restricted to the transformer architecture, our DA approach can also be implemented with

architecture-agnostic explainability methods. To show this, in our work, we also consider the Integrated Gradients (IG) explainability method (Sundararajan et al. 2017). Considering a baseline input $\boldsymbol{x}'$ (e.g., a sequence of padding tokens) and the actual input $\boldsymbol{x}$, IG measures how each input token contributes to model output by integrating gradients along a path from $\boldsymbol{x}'$ to $\boldsymbol{x}$ as follows:

$$IG_i = (x_i - x_i') \int_{\alpha=0}^{1} \frac{\partial f(\mathbf{x}' + \alpha(\mathbf{x} - \mathbf{x}'))}{\partial x_i} d\alpha \tag{3}$$

where α is a scaling parameter interpolating between baseline and input tokens. The gradient $\frac{\partial f}{\partial \boldsymbol{x}_i}$ measures how sensitive the model output f is to changes in $\boldsymbol{x}_i$, effectively capturing the relevance of the input token based on how much it changes the model's decision compared to a "neutral" baseline token. In practice, this IG formulation is approximated, for instance, through variants of Riemann or Gauss-Legendre quadrature. We will now show, in Sect. 3, how we leveraged the presented explainability methods to compute three different token relevance and how, from such measure, we obtained the entire line relevance and the DA final metric.

3 Detection alignment

This section presents the Detection Alignment (DA) evaluation procedure. We subdivide our approach into two specific steps: relevance score computation (Sect. 3.1), where we start from the tokens and compute the relevance of each line with respect to the model's output prediction; and detection alignment measurement (Sect. 3.2), where we start from each line's relevance score and measure the overall detection alignment with respect to each input sample's ground truth.

3.1 Relevance score computation

Our DA approach starts from a pretrained code vulnerability detection model f, which, given a vulnerable input code function $\boldsymbol{z}$ (represented by L code lines and mapped to a sample $\boldsymbol{x}$), outputs a binary label indicating whether the function is vulnerable or not. We provide an overview of the vulnerability detection model in Fig. 2. Given the prediction of the vulnerable code function $\boldsymbol{z}$, DA aims to quantify how well it aligns with the sample's ground truth

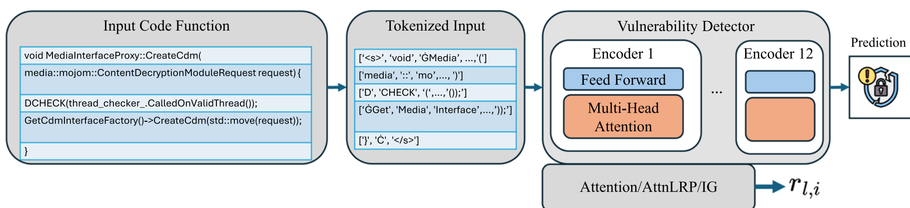

Fig. 2 A code vulnerability detector. Given an input code function, we tokenize the input and use, in this use case, a transformer-based vulnerability detector to predict the presence of a vulnerability in the source code. We then compute, with an explainability-based approach (via attention scores, AttnLRP or integrated gradients), a relevance score $r_{l,i}$ for each i-th token of the l-th line

by overlapping the relevant code lines that drove the prediction with the actual ground truth vulnerable lines. We describe in the following paragraphs how, starting from tokens, we finally compute code line relevance by adopting three different explainability-based measures. For all these measures, however, the DA procedure starts from the tokenization step, where the input z, represented by L lines, gets converted into a vector of tokens. Therefore, for each single line l, tokenization defines a set of N_l tokens $\boldsymbol{t}_l = [t_{l,0}, t_{l,1}, ..., t_{l,N_l-1}]$. Then, considering the i-th token of the l-th line $t_{l,i}$, we compute the *token relevance* using one of the three explainability-based approaches.

Attention-based relevance. Considering an embedded input token $t_{l,i}$, a transformer model computes a matrix of attention scores $\boldsymbol{A}^{(m)} \in \mathbb{R}^{H \times J \times J}$ at each encoder layer m, where H is the number of attention heads and J is the total sequence length (i.e., the total number of tokens in the input sequence $\boldsymbol{z}$). We compute the token-level relevance by aggregating the attention scores on a single specific layer m over each of the H heads as follows:

$$r_{l,i} = \sum_{h=1}^{H} \sum_{j=1}^{J} \boldsymbol{A}_{h,i,j}^{(m)} \tag{4}$$

where $r_{l,i}$ is the relevance score of the i-th token of line l, h the attention head, and j represent each of the other input tokens. Therefore, $\boldsymbol{A}_{h,i,j}^{(m)}$ represents the attention score assigned to token $r_{l,i}$ in layer m by the H heads when attending to the j-th token. Therefore, the higher the aggregated attention score $r_{l,i}$, the more that token $t_{l,i}$ contributed to the final prediction. However, while attention scores provide a straightforward way to interpret model decisions, they can often have limited reliability in our source code detection task due to biases in the code syntactic structure or positional dependencies (Achtibat et al. 2024). In turn, we do not limit our relevance computation to such a procedure but also extend it to the more advanced AttnLRP-based and architecture-agnostic IG-based computation as follows.

AttnLRP-based relevance. As the AttnLRP method is already able to provide a relevance score for each network's neuron, it is sufficient to take the produced score $r_{l,i}$ with respect to each input token.

Integrated gradients-based relevance. The third approach we use to compute token relevance instead, leverages layer integrated gradients (LIG) (Sundararajan et al. 2017). In detail, with respect to the base IG formulation from Eq. 3, we compute the gradients with respect to the input embeddings, effectively capturing the model attribution, approximating the integral with the Gauss-Legendre quadrature rule as follows:

$$r_{l,i} = \sum_{k=1}^{s} w_k \cdot \frac{\partial F(x' + \alpha_k (x - x'))}{\partial x_i} \tag{5}$$

where the values w_k weight the sum on a set of s steps, and $\alpha_k \in [0, 1]$ defines the interpolation between the baseline token $\boldsymbol{x}'$ to the actual input $\boldsymbol{x}$.

Line-based relevance score. We subsequently aggregate the token-based relevance scores $r_{l,i}$ into line-based relevance r_l as follows:

$$r_l = \sum_{i=1}^{N_l} r_{l,i}. \tag{6}$$

where N_l represents the number of tokens for line l. After repeating this process on the entire function, a line relevance score can be assigned to each line, resulting in a vector $\boldsymbol{r} = [r_0, r_1, ..., r_{L-1}]$. Since different explainability methods can produce relevance scores on substantially different scales, we rescale each line's relevance in [0, 1] to ensure consistency and proportionality across different methods.

3.2 Detection alignment measurement

After obtaining a set of relevance scores for each line $\boldsymbol{r}$, as we preliminary show in Fig. 3, we can now effectively compute the alignment between the lines that drove the model prediction and the ground truth for the vulnerable input sample $\boldsymbol{z}$, represented as a set $\mathcal{G} \subseteq \{l \mid 0 \leq l < L\}$, which contains the indices of the actual vulnerable code lines. To compute DA, we take inspiration from fuzzy-set theory (Petkovi et al. 2021), which enables the concept of partial membership in a set through a membership degree function $\mu \in [0, 1]$. Therefore, in more practical terms, μ indicates the degree of belonging of the element to the fuzzy set.

We implement such a concept in a first fuzzy-set $\mathcal{R}$, where each element is represented by the line index l along with its line relevance score r_l, representing the elements' membership degree $\mu_{\mathcal{R}}(l) = r_l$. This amounts to designing a first set $\mathcal{R} = \{(l, \mu_{\mathcal{R}}(l)) \mid 0 \leq l < L\}$. We then apply the same logic to create a second fuzzy-set $\mathcal{G}'$, which is the result of converting $\mathcal{G}$ into $\mathcal{G}' = \{(l, \mu_{\mathcal{G}'}(l)) \mid 0 \leq l < L\}$, where $\mu_{\mathcal{G}'}(l) = \mathbb{1}_{\mathcal{G}}(l)$ is the membership function of $\mathcal{G}'$, which simply assigns ones when the line is vulnerable, as per the ground truth. After creating the two sets, we compute the detection alignment DA through the Jaccard's index (i.e., intersection over union) between the fuzzy sets $\mathcal{R}$ and $\mathcal{G}'$ as follows:

$$DA = \frac{|\mathcal{R} \cap \mathcal{G}'|}{|\mathcal{R} \cup \mathcal{G}'|} = \frac{\sum_l \min(\mu_{\mathcal{R}}(l), \mu_{\mathcal{G}'}(l))}{\sum_l \max(\mu_{\mathcal{R}}(l), \mu_{\mathcal{G}'}(l))}, \tag{7}$$

where $0 \leq DA \leq 1$, with 1 as optimal value. Such DA computation, leveraging the Jaccard's index while extending on fuzzy sets, allows for capturing the extent to which both prediction and ground truth align on line importance (intersection) while accounting for all

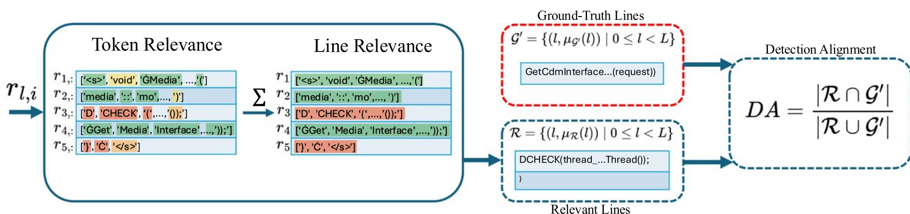

Fig. 3 A DA approach overview. We start from the token relevance scores $r_{l,:}$, which we aggregate into line relevance scores r_l. Then, we design the two sets: $\mathcal{G}'$, derived from the ground truth $\mathcal{G}$; and $\mathcal{R}$, derived from the line relevance scores r_l. We finally compute DA through the Jaccard's index computed between the two sets

lines that are considered relevant by one of ground truth or prediction (union). To measure the detection alignment, Eq. 7 is applied only on samples that are predicted as vulnerable by the model, and assign $DA = 0$ to vulnerable samples predicted as benign. Overall, our DA metric increases when the model assigns high relevance to ground truth lines and decreases when, in contrast, the model assigns high relevance to the lines that do not correspond to the vulnerability. Abstracting from a single code function z, we average the metric over multiple input samples $z \in \mathcal{Z}$ and obtain a global DA estimate.

We point out that the DA can also be generalized to diverse input representations other than tokens (e.g., graphs), with the only requirement of being able to map back the attributions produced with respect to the input representation onto the relative source code portions. Furthermore, the metric can be applied with different levels of granularity than code lines (e.g., group of tokens corresponding to source code entities), assuming to have a coherent ground truth.

4 Experiments

We present here the experiments conducted to evaluate our Detection Alignment (DA) metric. We start from a set of state-of-the-art vulnerability detectors and datasets, which we describe in Sect. 4.1. Then, in Sect. 4.2, we show and discuss the results obtained by our DA approach along with the other standard evaluation metrics, highlighting the misalignment of the predictions from such vulnerability detectors.

4.1 Experimental setup

We validate our approach on 3 datasets and 3 transformer-based vulnerability detectors. In the next paragraphs, we first discuss the details concerning both models and datasets. Then we provide an overview of the adopted experimental setup, concluding by presenting the DA computation settings.

Models. We evaluate 3 transformer-based vulnerability detection models: LineVul (Fu and Tantithamthavorn 2022), CodeBERT (Lu et al. 2021), and CodeT5-Small (Wang et al. 2021). We describe here the three detection models:

- **CodeBERT** is a pretrained language model for programming languages (Feng et al. 2020). However, in our specific use case, we refer to the version finetuned on a vulnerability detection task (Lu et al. 2021). Like other BERT architectures (Devlin et al. 2019), it is based on a total of 12 encoder-only layers.
- **LineVul** is a model designed for fine-grained vulnerability prediction (Fu and Tantithamthavorn 2022). It is built on the pretrained CodeBERT language model, which follows the BERT architecture (Devlin et al. 2019) with 12 encoder-only layers.
- **CodeT5-Small** is a unified encoder-decoder transformer model devised for multiple tasks. In our case, we use the model in its defect detection finetuned version (Wang et al. 2021), based on the T5 architecture consisting of 6 encoder layers and 6 decoder layers (Raffel et al. 2020).

Among the used models, LineVul proposes to analyze the attention mechanism to improve the granularity of the detection, thus aiming to localize the vulnerable code lines. CodeBERT and CodeT5 are instead limited to a coarse-grained vulnerability prediction, thus simply flagging the input source code function as vulnerable or not.

Datasets. We use 3 different datasets to validate the effectiveness of our DA metric. In detail, we finetune each of the three models on BigVul (Fan et al. 2020), Devign (Zhou et al. 2019), and Primevul (Ding et al. 2025). The datasets can be described as follows:

- **BigVul** is built by collecting a large set of C/C++ code from 348 open-source GitHub repositories. The authors crawled the public Common Vulnerabilities and Exposures (CVE) database and related source code repositories (Fan et al. 2020). We split the dataset into benign and vulnerable samples following the same approach adopted in Fu and Tantithamthavorn (2022), ending up with a dataset consisting of 188.636 functions, of which 177.736 are benign and 10.900 are vulnerable.
- **Devign** is constructed by employing security-related keywords to filter out GitHub commits (Zhou et al. 2019). In addition to this automated dataset construction, the vulnerable commits have been further annotated by human operators. However, as pointed out by Ding et al. (2025), the annotations from Devign are highly inaccurate. We split the dataset following (Lu et al. 2021), which results in an overall of 27.318 functions, of which 14.858 are benign and 12.460 vulnerable.
- **PrimeVul** is a recently proposed benchmark that aims to overcome biases in existing datasets (Ding et al. 2025). It is built by merging security-related commits from several previously proposed datasets, including BigVul, while excluding Devign after discovering that many of its commits were unrelated to security issues. The authors applied several steps to address common biases like the presence of code duplicates, incorrect labeling, and chronological inconsistencies between train and test data that cause data leakage. It consists of 235,768 samples, with 228,800 non-vulnerable and 6,968 vulnerable functions.

We summarize the main characteristics of these datasets in Table 1. We base our choice of the datasets on their popularity among other work on vulnerability detection and their availability (Ding et al. 2025). Although we finetune the three models on each of these datasets, obtaining a total of 9 different models, to test each detector our approach requires a line-level ground truth $\mathcal{G}$ (see Sect. 3.2), which indicates which are the vulnerable lines within a vulnerable input function. In turn, we test each of the 9 models on the BigVul test set (Fan et al. 2020), composed of 18864 samples, of which 1005 are vulnerable. This dataset provides a field `vul_func_with_fix` (where, for each vulnerable function, the flawed lines are flagged), from which we extracted the ground truth set $\mathcal{G}$ following the same methodology as in (Fu and Tantithamthavorn 2022).

Finetuning settings. We finetune each of the three models with the three different datasets, adding up to a total of 9 different models. In detail, for all the model-dataset pairs, we finetune using a learning rate $\gamma = 2 \times 10^{-5}$, a number of epochs equal to 10, and a fixed

Table 1 Summary of datasets used for finetuning and evaluation

Dataset	Language	# Functions	# Vulnerable	# Benign
BigVul	C/C++	188,636	10,900	177,736
Devign	C/C++	27,318	12,460	14,858
PrimeVul	C/C++	235,768	6,968	228,800

Table 2 Summary of models used in the experiments

Model	Architecture	Granularity	Finetuned on	Tested on
CodeBERT	Encoder-only	Function-level	BigVul, Devign, PrimeVul	BigVul
LineVul	Encoder-only	Line-level	BigVul, Devign, PrimeVul	BigVul
CodeT5-Small	Encoder-Decoder	Function-level	BigVul, Devign, PrimeVul	BigVul

Table 3 Evaluation metrics used to assess model performance and alignment

Metric	Description
F1-score	Standard classification performance metric on function-level predictions
$DA\text{-}A^{(1)}$	Detection Alignment based on attention weights from the first encoder layer
$DA\text{-}A^{(M)}$	Detection Alignment based on attention weights from the last encoder layer
$DA\text{-}ALRP$	Detection Alignment using Attention Layer-wise Relevance Propagation
$DA\text{-}IG$	Detection Alignment computed via Integrated Gradients (architecture-agnostic)

number of processed tokens equal to 512, i.e., each input function is allowed to be represented by a maximum of 510 tokens, and special tokens marking the start and the end of the sequence are added. If the length of the tokenized source code is higher, it is truncated; otherwise, padding tokens are used to fill the input sequence until it reaches the required length. Among the 1055 samples labeled as vulnerable used to compute the DA metric, 426 samples are too long to be entirely represented in the input sequence and thus are truncated to 510 tokens. During the finetuning procedure, we save the models with the highest F1 scores based on validation. We summarize the setup and model-dataset combinations in Table 2.

Metrics. We compute the relevance scores on each of the 9 models following the approaches described in Sect. 3.1. In detail, we first use the attention-based relevance computation on two specific encoder layers out of M: the first ($m = 1$) and the last ($m = M$), for which we provide a motivation in Sect. 4.2. Based on these two different attention-based relevance measures, we define a $DA\text{-}A^{(1)}$ and $DA\text{-}A^{(M)}$ detection alignment metric. Then, we apply the AttnLRP method, obtaining a $DA\text{-}ALRP$ detection alignment metric. Finally, we use the architecture-agnostic integrated gradient approach to compute relevance, from which we derive a third $DA\text{-}IG$ detection alignment metric. These four DA metrics are computed on each of the 9 model/dataset pairs, on which we also compute F1-score values to provide a complete overview of the detectors' performances, which would otherwise be limited. We summarize all the employed metrics in Table 3.

4.2 Results

Based on the model and dataset pairs previously described, we create a set of 9 models that we test on the BigVul dataset. Overall, the results obtained from our DA approach highlight, in Table 4, how the line-level localization of such detectors is consistently misaligned with the ground truth. This implies that, despite obtaining acceptable F1-score values, the predictions provided by these models are mostly driven by code lines that are not vulnerable, suggesting the presence of spurious correlations or biases when correctly flagging a vulnerable code function. For completeness, we computed the metric also considering the absolute values of the relevance scores, i.e. $r_l = \sum_{i=1}^{N_l} |r_{l,i}|$, obtaining results very close of the ones without the absolute values.

Relevance level comparison. We compute four DA metrics based on four different relevance computations. We choose to compute DA on the first layer, DA -$A^{(1)}$, inspired by (and to enable comparison with) the line-level localization approach from Fu and Tantithamthavorn (2022), which explicitly uses the attention on the first layer to indicate the vulnerable lines of code. Then, following the rationale that growing in layers the models can provide more class-specific representations in different tasks and architectures (Zeiler and Fergus 2014), we additionally use attention-based relevance to compute DA -$A^{(M)}$ on the last encoder layer M and compare with the first layer DA metric. Overall, we find that the DA -$A^{(M)}$ metric is consistently lower compared to DA -$A^{(1)}$, suggesting that the first encoder layers can capture more meaningful patterns based on the attention scores. More broadly, this result shows that the first and last layers simply focus on different kinds of correlations, with a lower alignment on the last layer. However, independently of the layer, we find a consistently low DA metric, suggesting how, in general, such detectors fail in flagging the input code functions as vulnerable based on the actual vulnerable lines. A similar conclusion can also be drawn for the AttnLRP and integrated gradients based metrics DA -*ALRP* and DA -*IG*, which in some specific cases also suggests worse alignment than the attention-based metrics and assumes higher relevance owing to its improved reliability, as mentioned in Sect. 2.3. Comparing the F1-score and the various DA metrics through Spearman's rank correlation, we observe a moderate negative correlation ($\rho \approx -0.61$) across all metrics. However, this trend lacks statistical significance (p-value > 0.05) and is further

Table 4 F1 score and DA computed on the BigVul (Fan et al. 2020) test set with the attention from the first attention layer (DA -$A^{(1)}$), the last layer *m* (DA -$A^{(M)}$), through AttnLRP (DA -*ALRP*) and Integrated Gradients (DA -*IG*). We denote with - the models for which we could not obtain acceptable performances (F1$<$ 0.01)

Training set	Model	F1	DA-$A^{(1)}$	DA-$A^{(M)}$	$DA-ALRP$	$DA-IG$
BigVul	CodeBERT	0.40	0.1241	0.0938	0.1140	0.1142
	LineVul	0.91	0.1208	0.0992	0.0850	0.0958
	CodeT5	0.93	0.1224	0.1118	0.0782	0.0777
Devign	CodeBERT	0.13	0.1181	0.1012	0.1093	0.1047
	LineVul	0.11	0.1278	0.0906	0.1160	0.1116
	CodeT5	0.10	0.1343	0.0922	0.1314	0.1381
PrimeVul	CodeBERT	0.17	4.33×10^{-5}	4.21×10^{-5}	2.14×10^{-5}	0.0001
	LineVul	–	–	–	–	–
	CodeT5	0.29	0.0045	0.0046	0.0107	0.0062

limited by the small sample size, which prevents us from drawing any strong scientific conclusions.

Model & dataset level comparison. Independent of the model-dataset combination, we constantly find a low DA. In general, through our setup, we train and test models with the same dataset on the first three BigVul models and additionally analyze the generalization capabilities by training the models on Devign and PrimeVul while still testing on BigVul. Interestingly, by analyzing the F1-score, we find that the performance on the Devign dataset is consistently low for all models. We believe that this condition also occurs due to the low label accuracy affecting Devign (Zhou et al. 2019). In fact, previous work has shown that the label accuracy for vulnerable functions on Devign is limited to 24%, which inevitably impairs the performance of the trained detectors. However, for both Devign and PrimeVul, the low F1-score inevitably highlights a general lack of detectors to generalize to different datasets. Also, we find the LineVul model trained on PrimeVul unable to reach appropriate performances, having an almost negligible F1 score. Models trained on BigVul instead clearly hold higher F1. Notably, though, the DA obtained by such models is extremely low, which further indicates how, despite having an extremely high F1 score, this metric alone is not entirely descriptive of the detectors' capabilities. In fact, most of the correct vulnerability flags given to the input code functions are actually due to non-vulnerable lines. Comparing across the different models instead, we likewise find constantly lower DA, without any highly relevant deviance from such a trend.

Sample-wise analysis. To provide a more detailed analysis of the effectiveness of our approach, we show in Figs. 4 and 5 a qualitative analysis of two samples, considering the CodeBERT model and attention-based relevance at the first layer as the attribution method. Figure 4 shows a vulnerable sample potentially related to a *buffer overflow* attack, localized in line 14 according to the ground truth provided by the BigVul dataset. Although the sample is correctly labeled as vulnerable by the model, the computed line relevance scores show how the prediction was mostly driven by non-vulnerable lines misaligned with the ground truth. In detail, as the `strcpy` function allows copying strings of arbitrary length into the memory region pointed by `m`, if the `name` length exceeds the allocated space, this can potentially lead to a buffer overflow. However, the model assigns high relevance to pointer arithmetic and, in particular, to low-level memory operations such as the `memcpy` function on line 18. Interestingly, among the 10,018 vulnerable samples in the Devign dataset, `strcpy` appears in only 36 samples, whereas `memcpy` occurs 766 times. This disparity suggests

```
1  char *path_name(const struct name_path *path, const char *name)
2  {
       ...
13     m = n + len - (nlen + 1);
14     strcpy(m, name);
15     for (p = path; p; p = p->up) {
16         if (p->elem_len) {
17             m -= p->elem_len + 1;
18             memcpy(m, p->elem, p->elem_len);
19             m[p->elem_len] = '/';
20         }
21     }
22     return n;
23 }
```

(a)

```
1  char *path_name(const struct name_path *path, const char *name)
2  {
       ...
13     m = n + len - (nlen + 1);
14     strcpy(m, name);
15     for (p = path; p; p = p->up) {
16         if (p->elem_len) {
17             m -= p->elem_len + 1;
18             memcpy(m, p->elem, p->elem_len);
19             m[p->elem_len] = '/';
20         }
21     }
22     return n;
23 }
```

(b)

Fig. 4 A vulnerable sample from the BigVul dataset. **a** depicts the ground truth vulnerable line (14), while **b** shows the relevant lines (DA -$A^{(1)}$) as conceived by a CodeBERT model trained on the Devign dataset. The detector mostly assigns high relevance to non-vulnerable code lines (1, 15, 17, 18, and 19)

```
1  void *arm_dma_alloc(struct device *dev, size_t size, dma_addr_t *handle,
2                      gfp_t gfp, struct dma_attrs *attrs)
3  {
4      pgprot_t prot = __get_dma_pgprot(attrs, pgprot_kernel);
5      void *memory;
6
7      if (dma_alloc_from_coherent(dev, size, handle, &memory))
8          return memory;
9
10     return __dma_alloc(dev, size, handle, gfp, prot, false,
11                        __builtin_return_address(0));
12 }
```

(a)

```
1  void *arm_dma_alloc(struct device *dev, size_t size, dma_addr_t *handle,
2                      gfp_t gfp, struct dma_attrs *attrs)
3  }
4      pgprot_t prot = __get_dma_pgprot(attrs, pgprot_kernel);
5      void *memory;
6
7      if (dma_alloc_from_coherent(dev, size, handle, &memory))
8          return memory;
9
10     return __dma_alloc(dev, size, handle, gfp, prot, false,
11                        __builtin_return_address(0));
12 }
```

(b)

Fig. 5 A vulnerable sample from the BigVul dataset. **a** depicts the ground truth vulnerable line (4), while **b** shows the relevant lines (DA -$A^{(1)}$) as conceived by a CodeBERT model trained on the Devign dataset. The detector assigns high relevance to non-vulnerable code line (8)

that the model may associate the presence of `memcpy` with vulnerability, highlighting the influence of spurious correlations learned from training data. Coherently, the $DA{-}A^{(1)}$ of this sample is equal to 0.0421, being $|\mathcal{R} \cap \mathcal{G}'| = 0.3538$ and $|\mathcal{R} \cup \mathcal{G}'| = 8.3921$. In contrast, the model fails to correctly predict the vulnerable sample in Fig. 5. In this case, the ground truth in Fig. 5a indicates the vulnerability in line 4, where an executable DMA (Direct Memory Access) mapping might lead to undesired privilege escalation. However, the resulting relevance attribution in Fig. 5b suggests that the model's prediction is mostly led by line 8, hence implying no intersection with the ground truth and resulting in $DA{-}A^{(1)}$ of 0 ($|\mathcal{R} \cap \mathcal{G}'| = 0$ and $|\mathcal{R} \cup \mathcal{G}'| = 1$).

5 Related work

We describe here the most relevant related work, which we subdivide based on work focusing on the detectors' interpretability (Sect. 5.1), and on studies analyzing biased decisions in vulnerability detection (Sect. 5.2).

5.1 Interpretable vulnerability detectors

The majority of proposed approaches only produce a binary label indicating the presence of a vulnerability in the analyzed source code input. Existing literature focuses on developing models that could provide fine-grained predictions, e.g. at the line level, in order to provide more meaningful feedback for human operators in charge of analyzing vulnerabilities. VulDeeLocator (Li et al. 2022) uses intermediate code representations, human-defined rules, and a Bidirectional Recurrent Neural Network (BRNN) enhanced with additional layers to produce line-level outputs. Similarly, VulChecker (Mirsky et al. 2023) exploits a GNN trained on program dependency graphs produced from intermediate code representation, while LineVD(Hin et al. 2022) combines a transformer model and a GNN to achieve the same objective. LineVul (Fu and Tantithamthavorn 2022) and IVDetect (Li et al. 2021) leverage the attributions produced with attention scores from a transformer-based model and explainable AI algorithms from a GNN, respectively, to obtain and output the most important lines of code for each model's prediction. First, we highlight that in our work the focus is slightly different. We do not focus on human interpretability, but rather on exposing the model's reliance on spurious correlations that hinder generalization to unseen

data. In our work, we compute the Detection Alignment on LineVul (in addition to other models providing coarse-grained outputs), as we restricted the analysis to transformer-based approaches. This gives us the possibility to test our metric with both model-specific (i.e., attention scores) and architecture-agnostic (i.e., Integrated Gradients) attribution methods. We note that our approach is agnostic of both considered models and attribution methods and thus can be easily applied to the other aforementioned detectors. Furthermore, for those models, the Detection Alignment can be computed relying either on the produced line-level outputs or on an attribution method.

5.2 Bias in vulnerability detectors

Several works inspected vulnerability detection models through explainable machine learning techniques, trying to determine how they produce a certain output to understand if their decisions were the result of a real capture of the meaning of the analyzed code, or if they were affected by bias and exploited shortcuts to label the data. (Chakraborty et al. 2022), applied attribution methods on some input samples to extract the most important features used to classify them from 4 different detectors, showing that in most cases they focused on code segments not correlated with the vulnerability present in the code. (Warnecke et al. 2020) showed how an RNN-based vulnerability detector (Li et al. 2018) relies on artifacts present in the dataset that are irrelevant to the discovery of vulnerabilities in the code. The same problem was later resumed and analyzed in detail in (Arp et al. 2022). (Sotgiu et al. 2022) extended these sample-wise analyses to the dataset level, considering also transformer-base detectors. Although all these works share the use of explainability methods to investigate how much the decisions of the models are taken based on an understanding of the code and on the real location of the vulnerabilities, all of them require the intervention of a human operator who inspects the outputs produced, to evaluate and quantify this phenomenon. Contrarily, we provide a metric that allows us to have an immediate vision and a quantification of the problem. (Imgrund et al. 2023) presented a method to quantify how much detectors are influenced by some categories of artifacts in the code (e.g., identifier names, coding styles, etc.), but they do not consider how much a model is able to focus on the relevant parts of the code. (Warnecke et al. 2020) also proposed metrics to evaluate, under multiple perspectives, the quality of explanations produced by different algorithms on models applied to security-related tasks. (Ganz et al. 2021) extended this analysis by specifically considering GNNs for vulnerability detection. Following the same line of work, (Ganz et al. 2023) proposed other criteria to compare different explainability techniques by specifically focusing on vulnerability detection and considering different detectors. More generally, a body of research (Faber et al. 2021; Arras et al. 2022; Guidotti 2021; Zhou et al. 2022) investigated approaches to evaluate the quality of xAI methods, leveraging ground truth on synthetic or real-world data. These works are orthogonal to ours, as they do not focus on models but rather on techniques to evaluate the quality and utility of attributions. These techniques can, however, be used to select the best attribution methods to use to compute our metric, helping to disentangle whether observed discrepancies with the ground truth truly reflect model misalignment and not shortcomings in the explanation methods themselves.

6 Conclusions

Despite ML-based source code vulnerability detectors reporting promising results, their performance is often overestimated, especially when they rely on learning feature correlations that are not relevant with respect to the presence of a vulnerability. Recent work has explored this phenomenon through the lens of empirical qualitative analysis, but having a complete overview of it requires an inspection by a human expert.

In this work, we propose the Detection Alignment (DA) metric, which quantifies how much a model is able to focus on the input portions (e.g., code lines) that are actually related to the considered problem when it predicts a sample as vulnerable. We implement the metric by extracting the most relevant lines influencing the classifier's decision with explainable AI algorithms and comparing them with a ground truth representing the lines associated with the vulnerabilities. We test the metric on the BigVul dataset, considering a combination of three transformer-based detectors trained on three different datasets, showing that all of them struggle to recognize the really important parts for the presence of a vulnerability in the code. This is also confirmed by the lack of generalization capabilities across different datasets. We believe DA can provide a fast and easy-to-interpret estimate of the vulnerability detectors' trustworthiness, which can help debug and improve models during their development. Moreover, the metric design allows practitioners to adapt it to a wide range of diverse settings (models, datasets, programming languages) and extend it to several tasks beyond vulnerability detection (code quality assessment, bug finding), and domains (natural language processing, and with slight adaptations, even computer vision).

In future work, we plan to extend the experimental evaluation in order to consider different attribution methods and detectors, possibly able to handle larger inputs and avoid truncating the input functions, which represent a limitation of current approaches. Beyond evaluation, a key direction is to integrate the DA metric directly into the training process. In particular, we propose leveraging DA as a regularization signal that encourages the model to focus its predictive attention on semantically relevant code regions.

Acknowledgements This work has been partly supported by the EU-funded Horizon Europe projects ELSA (GA no. 101070617) and Sec4AI4Sec (GA no. 101120393); and by projects SERICS (PE00000014) and FAIR (PE00000013, CUP: J23C24000090007) under the MUR NRRP funded by the European Union - NextGenerationEU.

Author contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by M.P., G.P. and A.S.. The first draft of the manuscript was written by A.S. and G.P., and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding Open access funding provided by Università degli Studi di Cagliari within the CRUI-CARE Agreement. This work has been partly supported by the EU-funded Horizon Europe projects ELSA (GA no. 101070617) and Sec4AI4Sec (GA no. 101120393); and by projects SERICS (PE00000014) and FAIR (PE00000013, CUP: J23C24000090007) under the MUR NRRP funded by the European Union - NextGenerationEU.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare no Conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Achtibat, R., Hatefi, S.M.V., Dreyer, M., Jain, A., Wiegand, T., Lapuschkin, S., & Samek, W. (2024). Attnlrp: attention-aware layer-wise relevance propagation for transformers. In: Proceedings of the 41st international conference on machine learning. ICML'24

Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., & Rieck, K. (2022). Dos and don'ts of machine learning in computer security. In: 31st USENIX security symposium (USENIX Security 22), pp. 3971–3988. USENIX Association, Boston, MA

Arras, L., Osman, A., & Samek, W. (2022). Clevr-xai: A benchmark dataset for the ground truth evaluation of neural network explanations. *Information Fusion, 81*, 14–40. https://doi.org/10.1016/j.inffus.2021.11.008

Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2022). Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering, 48*(09), 3280–3296. https://doi.org/10.1109/TSE.2021.3087402

Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding, pp. 4171–4186 https://doi.org/10.18653/V1/N19-1423

Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., Wagner, D., Ray, B., & Chen, Y. (2025). Vulnerability detection with code language models: How far are we?. In: 2025 IEEE/ACM 47th international conference on software engineering (ICSE), pp. 469–481. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/ICSE55347.2025.00038

Faber, L., K. Moghaddam, A., & Wattenhofer, R. (2021). When comparing to ground truth is wrong: On evaluating gnn explanation methods. In: Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining. KDD '21, pp. 332–341. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3447548.3467283

Fan, J., Li, Y., Wang, S., & Nguyen, T.N. (2020) A c/c++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th international conference on mining software repositories. MSR '20, pp. 508–512. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3379597.3387501

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020) CodeBERT: A pre-trained model for programming and natural languages, pp. 1536–1547 https://doi.org/10.18653/v1/2020.findings-emnlp.139

Fu, M., & Tantithamthavorn, C. (2022). Linevul: A transformer-based line-level vulnerability prediction. In: 2022 IEEE/ACM 19th international conference on mining software repositories (MSR), pp. 608–620. https://doi.org/10.1145/3524842.3528452

Ganz, T., Härterich, M., Warnecke, A., & Rieck, K. (2021). Explaining graph neural networks for vulnerability discovery. In: Proceedings of the 14th ACM workshop on artificial intelligence and security, pp. 145–156

Ganz, T., Rall, P., Härterich, M., & Rieck, K. (2023). Hunting for truth: Analyzing explanation methods in learning-based vulnerability discovery. In: 2023 IEEE 8th European symposium on security and privacy (EuroS &P), pp. 524–541. https://doi.org/10.1109/EuroSP57164.2023.00038

Guidotti, R. (2021). Evaluating local explanation methods on ground truth. *Artificial Intelligence, 291*, Article 103428. https://doi.org/10.1016/j.artint.2020.103428

Hin, D., Kan, A., Chen, H., & Babar, M.A. (2022). Linevd: statement-level vulnerability detection using graph neural networks. In: Proceedings of the 19th international conference on mining software repositories. MSR '22, pp. 596–607. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3524842.3527949

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Imgrund, E., Ganz, T., Härterich, M., Pirch, L., Risse, N., & Rieck, K. (2023). Broken promises: Measuring confounding effects in learning-based vulnerability discovery. In: Proceedings of the 16th ACM workshop on artificial intelligence and security. AISec '23, pp. 149–160. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3605764.3623915

Li, Y., Wang, S., & Nguyen, T.N. (2021). Vulnerability detection with fine-grained interpretations. ESEC/FSE 2021, pp. 292–303. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3468264.3468597

Li, Z., Zou, D., Xu, S., Chen, Z., Zhu, Y., & Jin, H. (2022). Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing, 19*(04), 2821–2837. https://doi.org/10.1109/TDSC.2021.3076142

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

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., & Tang, D. (2021) Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664

Mirsky, Y., Macon, G., Brown, M., Yagemann, C., Pruett, M., Downing, E., Mertoguno, S., & Lee, W. (2023). VulChecker: Graph-based vulnerability localization in source code. In: 32nd USENIX security symposium (USENIX Security 23), pp. 6557–6574. USENIX Association, Anaheim, CA. https://www.usenix.org/conference/usenixsecurity23/presentation/mirsky

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2025). Asleep at the keyboard? assessing the security of Github copilot's code contributions. *Communications of the ACM, 68*(2), 96–105. https://doi.org/10.1145/3610721

Petkovi, M., Škrlj, B., Kocev, D., & Simidjievski, N. (2021). Fuzzy Jaccard index: A robust comparison of ordered lists. *Applied Soft Computing, 113*, Article 107849. https://doi.org/10.1016/j.asoc.2021.107849

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research, 21*(140), 1–67.

Sennrich, R., Haddow, B., & Birch, A. (2016). Neural machine translation of rare words with subword units. In: Erk, K., Smith, N.A. (eds.) Proceedings of the 54th annual meeting of the association for computational linguistics (Volume 1: Long Papers), pp. 1715–1725. Association for Computational Linguistics, Berlin, Germany. https://doi.org/10.18653/v1/P16-1162

Sotgiu, A., Pintor, M., & Biggio, B. (2022). Explainability-based debugging of machine learning for vulnerability discovery. In: Proceedings of the 17th international conference on availability, reliability and security. ARES '22. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3538969.3543809

Sundararajan, M., Taly, A., & Yan, Q. (2017). Axiomatic attribution for deep networks. In: Proceedings of the 34th international conference on machine learning - Volume 70. ICML'17, pp. 3319–3328

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. In: Proceedings of the 31st international conference on neural information processing systems. NIPS'17, pp. 6000–6010. Curran Associates Inc., Red Hook, NY, USA

Wang, Y., Wang, W., Joty, S., & Hoi, S.C.H. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Moens, M.-F., Huang, X., Specia, L., Yih, S.W.-t. (eds.) Proceedings of the 2021 conference on empirical methods in natural language processing, pp. 8696–8708. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic. https://doi.org/10.18653/v1/2021.emnlp-main.685

Warnecke, A., Arp, D., Wressnegger, C., & Rieck, K. (2020). Evaluating explanation methods for deep learning in security. In: 2020 IEEE European symposium on security and privacy (EuroS &P), pp. 158–174. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/EuroSP48549.2020.00018

Zeiler, M.D., & Fergus, R. (2014). Visualizing and understanding convolutional networks. In: Computer vision–ECCV 2014: 13th European conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13, pp. 818–833. Springer

Zhou, Y., Booth, S., Ribeiro, M. T., & Shah, J. (2022). Do feature attribution methods correctly attribute features? *Proceedings of the AAAI Conference on Artificial Intelligence, 36*(9), 9623–9633. https://doi.org/10.1609/aaai.v36i9.21196

Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019) Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Marco Pintore[1] · Giorgio Piras[1] · Angelo Sotgiu[1,2] · Maura Pintor[1,2] · Battista Biggio[1,2]

✉ Angelo Sotgiu
angelo.sotgiu@unica.it

Marco Pintore
marco.pintore@unica.it

Giorgio Piras
giorgio.piras@unica.it

Maura Pintor
maura.pintor@unica.it

Battista Biggio
battista.biggio@unica.it

[1] University of Cagliari, Cagliari, Italy

[2] CINI, Rome, Italy