Highlights

LocVul: Line-Level Vulnerability Localization based on a Sequence-to-Sequence Approach

Ilias Kalouptsoglou, Miltiadis Siavvas, Apostolos Ampatzoglou, Dionysios Kehagias, Alexander Chatzigeorgiou

- A Sequence-to-Sequence approach is proposed for line-level vulnerability localization.
- The Sequence-to-Sequence approach outperforms XAI-based methods in vulnerability detection.
- LLM hallucinations are mitigated by matching generated lines to real ones via cosine similarity.

LocVul: Line-Level Vulnerability Localization based on a Sequence-to-Sequence Approach

Ilias Kalouptsoglou[a,b,*], Miltiadis Siavvas[a], Apostolos Ampatzoglou[b], Dionysios Kehagias[a], Alexander Chatzigeorgiou[b]

[a] *Centre for Research and Technology Hellas/Information Technologies Institute, 6th km Charilaou-Thermi Rd, Thermi, 57001, Thessaloniki, Greece*
[b] *University of Macedonia/Department of Applied Informatics, Egnatia 156, Thessaloniki, 54636, Thessaloniki, Greece*

Abstract

Context: The development of secure software systems depends on early and accurate vulnerability identification. Manual inspection is a time-consuming process that requires specialized knowledge. Therefore, as software complexity grows, automated solutions become essential. Vulnerability Prediction (VP) is an emerging mechanism that identifies whether software components contain vulnerabilities, commonly using Machine Learning models trained on classifying components as vulnerable or clean. Recent explainability-based approaches attempt to rank the lines based on their influence on the output of the VP Models (VPMs). However, challenges remain in accurately localizing the vulnerable lines.
Objective: This study aims to examine an alternative to explainability-based approaches to overcome their shortcomings. Specifically, explainability-based methods depend on the type and accuracy of the file or function-level VPMs, inherit possible misleading patterns, and cannot indicate the exact code snippet that is vulnerable nor the number of vulnerable lines.
Method: To address these limitations, this study introduces an innovative approach based on fine-tuning Large Language Models on a Sequence-to-Sequence objective to directly return the vulnerable lines of a given function. The method is evaluated on the Big-Vul dataset to assess its capacity for fine-grained vulnerability detection.
Results: The results demonstrate that the proposed method significantly outperforms the explainability-based baseline both in terms of accuracy and cost-effectiveness.
Conclusions: The proposed approach marks a significant advancement in automated vulnerability detection by enabling accurate line-level localization of vulnerabilities.

Keywords: Software Security, Vulnerability Detection, Large Language Models, Sequence-to-Sequence Models, Self-Attention

*Corresponding Author.
Email addresses: iliaskaloup@iti.gr, iliaskaloup@uom.edu.gr (Ilias Kalouptsoglou), siavvasm@iti.gr (Miltiadis Siavvas), a.ampatzoglou@uom.edu.gr (Apostolos Ampatzoglou), diok@iti.gr (Dionysios Kehagias), achat@uom.edu.gr (Alexander Chatzigeorgiou)

1. Introduction

Security is an important characteristic of the Software Development Life-Cycle (SDLC) according to the International Standard on Software Quality ISO/IEC 25010 [1]. However, in practice it is usually considered an afterthought and is not addressed until the final stages of the overall SDLC. Software enterprises often neglect enhancing the security and the resilience of their products on the altar of fast delivery [2] or due to lack of security expertise [3]. As a result, the number of reported vulnerabilities continues to rise each year, according to the published Common Vulnerabilities and Exposures (CVE) records [4].

Software vulnerabilities are weaknesses within the source code that can be exploited by one or more external threats [5]. They are often caused by a small number of common programming errors made by developers during the coding phase, but their exploitation can lead to devastating consequences in terms of data privacy and system confidentiality, integrity, and availability as well as serious financial damage. For instance, one of the most notable critical vulnerabilities, the Log4Shell vulnerability was found in a widely-used open-source logging library called Log4j. It can allow an attacker to control the remote systems, execute malware and leak sensitive information (e.g., passwords), among others [6].

To avoid similar circumstances, it is important for software houses to adopt proper security assessment and vulnerability detection solutions from the early phases of the SDLC [7]. However, the continuous increase in the number of security breaches and vulnerabilities exploited, especially considering the fact that many vulnerabilities remain undetected for years (e.g., Heartbleed remained unnoticed for two years and Log4Shell for eight years) [8], demonstrates the insufficiency of the tools already in use. Hence, there is a strong need for innovative and accurate vulnerability detection techniques.

1.1. Background

Existing vulnerability detection techniques are divided in traditional and data-driven categories. The former include static and dynamic analysis whereas the latter leverage Machine Learning (ML) to perform vulnerability prediction (VP). Static analysis tools (e.g., SonarQube, Coverity, Checkmarx, etc.) [9] identify buggy lines of code based on rules and code patterns defined by experts. However, static code analyzers often miss certain types of vulnerabilities since not all vulnerabilities can be modeled by rules, and every analyzer has its own detection rules and supports different vulnerability categories [10]. Moreover, static analysis is too broad producing many false positives, overwhelming security teams with a large amount of false alarms [11],[12]. On the other hand, dynamic testing techniques, such as fuzzing and penetration testing, analyze running software and simulate attacks to identify weaknesses. However, dynamic testing still produces both false positives and false negatives, cannot be applied to every component, and has difficulty on tracing the vulnerability in the code [13].

As regards the data-driven approach, VP involves constructing vulnerability prediction models (VPMs), which are ML models trained on software attributes extracted from the source code of files or functions/methods in order to predict the existence of vulnerabilities in them. The VP-related research evolved from the construction of ML models based on software metrics (e.g., complexity, cohesion, etc.) [14],[15],[16],[17] to building Deep Learning (DL) models capable of learning patterns in the text of the

source code, similarly to Natural Language Processing (NLP) tasks [18],[19],[20]. Typically text mining attempts represent the source code as sequences of tokens, where each token is encoded with an embedding vector [21],[22], or as text-rich graphs, such as the Code Property Graphs (CPGs) [23], which are fed to Graph Neural Networks (GNNs) to extract a graph embedding of the source code [24],[25],[26]. Recently, with the breakthrough of the Transformer architecture [27], the focus of the VP-related literature has shifted to transfer learning approaches. Several studies employ pre-trained models to build Large Language Models (LLMs)-based VPMs [28],[29],[30],[31]. It can be argued that LLMs have dominated the VP field, but, despite being considered as a promising solution for addressing open issues in the literature [32], the LLM-based VP solutions still face important challenges [33].

1.2. Motivation for line-level vulnerability detection

A major challenge in the domain that hinders the adoption of ML-based VP solutions in practice is the level of granularity at which the predictions are made. Most studies in the literature are coarse-grained as they predict whether there are vulnerabilities at file or function level of granularity [32]. Such predictions could be useful for the development teams to prioritize their inspection and fortification efforts to specific software components, but are still far from achieving complete vulnerability detection, since they lack the ability to present the specific lines of the components containing the vulnerabilities, let alone the vulnerability categories [34]. Therefore, coarse-grained predictions discourage software engineers from using such tools. It can be argued that the lower the granularity, the more actionable the results and the easiest manual inspection, but it is more difficult to build accurate mechanisms.

To clearly demonstrate the benefit from fine-grained VPMs, Figure 1 illustrates the difference between the outputs of function-level and line-level approaches using a buffer overflow example. In this example, `processInput` function receives user input and copies it into a fixed-size buffer without verifying the input's length. This operation introduces a buffer overflow vulnerability, which can result in overwriting adjacent memory locations and potentially lead to arbitrary code execution. In Figure 1, the entire `processInput` function is highlighted with an outer, thicker red rectangle, representing the output of a typical function-level VPM, which flags a whole function as potentially vulnerable. This narrows the inspection focus (e.g., by a human reviewer) from the entire class to this specific function. In contrast, the inner, thinner red rectangle highlights only the lines directly responsible for the vulnerability, reflecting the output of a line-level prediction approach. This finer-grained localization provides developers with much more actionable guidance, allowing them to directly identify and remediate the exact code statements that are vulnerable, rather than needing to manually inspect every line in the function.

1.3. Motivation for the proposed LocVul approach to Localize Vulnerabilities

The vast majority of fine-grained solutions in the VP-related literature employ explainable artificial intelligence (XAI) to interpret function or file-level predictions and, thereby, localize vulnerabilities [33]. More specifically, current mechanisms that perform VP at a low granularity level focus on the adoption of XAI techniques to extract the features' importance in the predictions made by VPMs [35]. A prime example of such solutions is the LineVul [36] study, in which, Fu et al. [36] proposed fine-tuning a

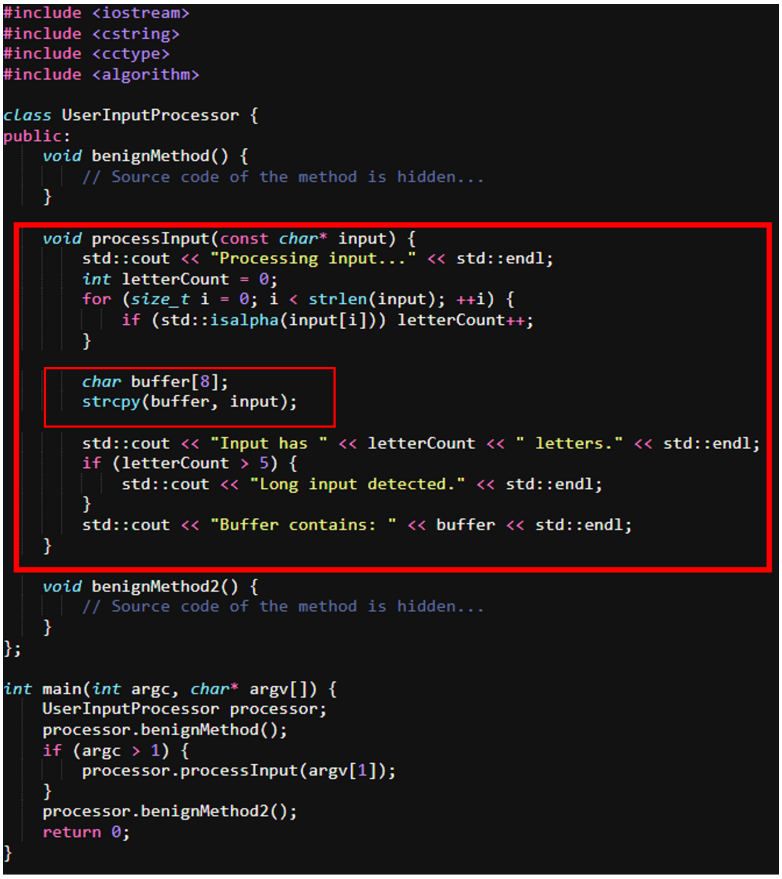


Figure 1: A motivating example of line-level vulnerability detection compared to coarser-grained predictions.

pre-trained on code LLM (i.e., CodeBERT[37]) for function-level predictions and then extracting its Self-Attention [27] weights to get line-level predictions, achieving quite accurate VP at line-level of granularity. From this point on, line-level VP will be referred to as Vulnerability Detection (VD).

However, such XAI-based solutions heavily depend on the function-level prediction models. Thus, they can only be applied on top of the VPMs, limiting the choices and possible improvements of VPMs of different kinds. In addition, VPMs often make a correct prediction based on irrelevant or even spurious correlations among tokens in the input sequence, which the VPMs erroneously associate with the existence of vulnerabilities [38],[39]. Therefore, XAI is often prone to fail in correctly identifying vulnerable lines. Moreover, such XAI-based solutions assign Attention scores to tokens and, based on them, assign importance scores to lines. Thus, although they can rank the lines prioritizing the reviewer's (e.g., developer) efforts to search for vulnerabilities starting from the higher ranked lines, they are not able to directly showcase which or how many of the lines are the vulnerable ones (i.e., there is no indication of when to stop the inspection).

Figure 2 is provided to illustrate the aforementioned limitations of the XAI-based VD as opposed to the proposed solution for **Loc**alizing **Vul**nerabilities (**LocVul**).

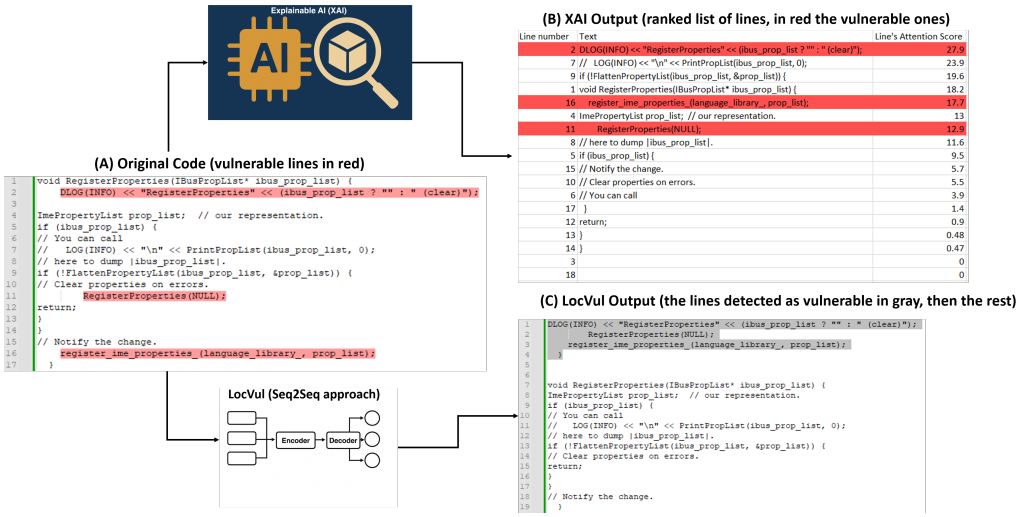


Figure 2: A motivating example of the proposed LocVul method compared to XAI-based methods.

Specifically, Figure 2 (A) presents the source code of a given function included in the dataset used, which contains a weak cryptography vulnerability. The exact location of the vulnerability is highlighted with red lines. As can be seen in Figure 2 (B), the explainability-based technique (e.g., Self-Attention mechanism) returns a list of the lines of the function sorted from the most likely to be vulnerable to the least likely. In contrast, as can be seen in Figure 2 (C), LocVul returns the exact set of lines it considers vulnerable, as highlighted in gray. LocVul response can then be filled in by the remaining lines of the function. Thus, LocVul's response facilitates the inspection process, since the code reviewer would have an indication of the number and the exact location of the vulnerable lines. Moreover, one can see in Figure 2 (B) that there are several lines before the vulnerable ones in the list proposed by the explainability-based method, which indicates that although the VPM at the function level has correctly identified the vulnerable function, it has also emphasized features that are not related to the vulnerability (i.e., spurious data associations).

1.4. Novelty and contribution

In this study, we propose an alternative approach named LocVul to perform line-level vulnerability prediction (i.e., vulnerability detection) that uses two distinct models for function-level and line-level prediction respectively, managing to overcome the aforementioned limitations. Specifically, in the proposed approach, the line-level model (i) is independent from the function-level VPM and, therefore, can be implemented on the top of metrics, text, graph-based or ensemble VPMs; (ii) does not propagate the error of the spurious patterns learned by the VPMs; and (iii) first returns a string with the specific lines it considers vulnerable and then returns a ranked list of lines so that the inspector has both an indication of the exact lines identified as vulnerable and, complementarily,

a list of all lines sorted by their likelihood of being part of the vulnerability. Moreover, LocVul achieves higher accuracy and better cost-effectiveness than XAI-based methods.

LocVul is based on a Sequence-to-Sequence (Seq2Seq) approach, where a Text-to-Text Transformer-based model [40] is trained on the task of extracting the vulnerable lines from functions that are predicted as vulnerable by a text mining-based VPM. More specifically, we first fine-tune a pre-trained model (i.e., CodeBERT) on the objective of classifying functions as vulnerable or not and, also, we fine-tune a pre-trained Sequence-to-Sequence model (i.e., CodeT5[41]) in the vulnerable lines extraction objective. Subsequently, we execute the classification model on unseen data to predict vulnerable functions and, then, we apply the Sequence-to-Sequence model on the predicted as vulnerable functions to get the vulnerable code snippets that are included in them.

To the best of our knowledge, LocVul is the first Sequence-to-Sequence VD approach, which manages not only to overcome the limitations of the XAI-based approaches, but also to clearly surpass their detection accuracy. Additionally, this study observes that the Sequence-to-Sequence model produces some imprecise outputs, which could be an important limitation of this approach. In particular, we observed that the developed model often generates similar but not identical vulnerable lines as those present in the analyzed function, which is considered an issue related to LLMs hallucinations [42]. Therefore, the present study proceeds by proposing also a solution that eliminates this problem by replacing the not-exact matches with the most similar lines in the original function (see Section 3.3). Overall, the contributions of the current study can be summarized as follows:

- A novel non-XAI-based VD mechanism is proposed based on a Sequence-to-Sequence model: We propose a two-stage approach where (i) a CodeBERT-based classifier is fine-tuned to identify vulnerable functions, and (ii) instead of explaining the function-level predictions, a CodeT5-based Sequence-to-Sequence model is fine-tuned to directly extract vulnerable lines from given vulnerable functions. The approach is trained and evaluated in the Big-Vul dataset.

- Superior than state-of-the-art results are achieved in terms of accuracy and cost-effectiveness: Our method significantly improves various evaluation metrics over the Self-Attention XAI baseline in identifying vulnerable lines in vulnerable functions.

- A method to mitigate LLM hallucinations in VD is proposed: We propose a line replacement post-processing step using cosine similarity between the embedding vectors of the tokenized lines to match generated lines to those present in the function, mitigating "hallucinated" outputs from CodeT5.

- A replication package of our LocVul implementation is published publicly: We provide our source code, models, and guidelines to reproduce the results [43].

The rest of the paper includes Section 2, which summarizes the related work in the literature and Section 3 that provides a detailed description of the proposed approach. Subsequently, Section 4 sets up the experimental design, while the results of our experimental analysis are presented in Section 5. Section 6 discusses the insights and lessons learned from the analysis, and finally, Section 7 provides the conclusions of the study and suggestions for future work.

2. Related work

This section presents previous studies in the field of ML-based vulnerability prediction (VP). Initially, we describe notable studies regarding VP in file and function level, and subsequently, we discuss more fine-grained techniques.

2.1. Coarse-grained vulnerability prediction

Initial attempts at ML-based VP used software metrics as indicators of the existence of vulnerabilities. Shin and Williams [14],[44] were among the first to explore the potential of complexity metrics to differentiate between vulnerable and non-vulnerable functions. Chowdhury and Zulkernine [45] then presented a VP paradigm by using complexity, cohesion, and coupling metrics and training several ML models. Furthermore, Kalouptsoglou et al. [15] utilized a variety of statically extracted software metrics to predict vulnerable files using the Multi-Layer Perception (MLP) classifier.

Moreover, text mining-based studies were presented in the literature, which were initially based on the Bag of Words (BoW) text representation format, where each software component is a table, its words are the table's columns, and the frequencies of the words are the columns' values [18],[46],[47]. Comparative studies [8],[48] have shown that text mining-based VP is more accurate than software metrics-based VP, managing to capture more complex patterns in the source code. Text mining-based approaches evolved from the more simplistic BoW to the sequences of tokens representation format, which represents the source code of a file or function as a sequence of words (i.e., tokens). The tokens sequence is then transformed to a sequence of embedding vectors [22] using algorithms such as word2vec [49] and fastText [50]. In particular, Dam et al. [19] employed a Long Short-Term Memory (LSTM) model to automatically predict vulnerable files. In addition, Li et al. [21] proposed VulDeePecker that utilizes a Bidirectional LSTM to identify vulnerable code on tokens encoded with word2vec embeddings.

Furthermore, Bilgin et al. [24] proceeded with the extraction of the Abstract Syntax Tree (AST) of C/C++ methods and the utilization of ML models to classify them to vulnerable and non-vulnerable ones. Next, Zhou et al. [25] proposed the Devign model based on graphical source code representations (i.e., ASTs, control and data flow graphs) and leveraged a GNN to extract the graph embeddings of C/C++ functions, which then were classified to vulnerable and non-vulnerable ones. Nonetheless, Chakraborty et al. [26] revealed several limitations of the existing VP datasets and approaches and proposed a CPG [23] and GNN-based methodology managing to outperform Devign.

Recently, after the breakthrough of the Transformer architecture [27], several studies have employed transfer learning for VP leveraging the vast knowledge and the context-aware patterns encapsulated in the pre-trained LLMs [33]. To this end, Bagheri et al. [30] compared traditional embedding techniques (i.e., word2vec and fastText) to the embeddings extracted from the Bidirectional Encoder Representations from Transformers (BERT) [51] in their capacity to classify code snippets in vulnerable and non-vulnerable ones. Furthermore, Kalouptsoglou et al. [29] fine-tuned three popular LLMs and their pre-trained on code variants observing their superiority compared to traditional text mining-based methods and also the superiority of the code variants. They also observed an advantage of the encoder-only Transformer-based models in VP.

Moreover, several empirical studies conducted by researchers compared various pre-trained models in the task of VP [52],[53]. In addition, Kim at al. [54] fine-tuned BERT to

construct their VPM called VulDeBERT managing to outperform VulDeePecker, while Hanif et al. [55] pre-trained on code a RoBERTa [56] model, added an MLP and a Convolutional Neural Network (CNN), and fine-tuned it on vulnerability-related C/C++ data for VP, outperforming both Recurrent Neural Networks (RNN)-based and GNN-based approaches.

2.2. Fine-grained vulnerability detection

In an attempt to provide fine-grained vulnerability predictions, Hin et al. [57] employed supervised statement-level training. In particular, they presented a GNN-based model, namely LineVD, which classifies each line of the predicted as vulnerable functions into vulnerable or not. Instead of classifying every line, Li et al. [58] proposed the IVDetect model following an interpretable (i.e., explainable) ML-based approach. Specifically, they represented the source code in Program Dependency Graphs (PDGs) [59], performed function-level VP, and then interpreted the model's decisions using the GNNExplainer [60] method that derives interpretation sub-graphs. Those sub-graphs correspond to hotspots in the source code that are more relative to the identified vulnerabilities. Similarly, LineFlowDP [61], which employed PDGs to represent source code and GNNs to perform file-level prediction of defects (i.e., general bugs or malfunctions), leveraged GNNExplainer [60] to explain the predictions and localize defects.

Nonetheless, Fu et al. [36] emphasized on the limitation of graph-based solutions, such as IVDetect [58], in producing truly fine-grained interpretations, since sub-graphs contain a lot of lines of code. To this end, they proposed LineVul [36] by leveraging the capacity of the pre-trained CodeBERT [37] model. Specifically, they represented source code in token sequences, they fine-tuned CodeBERT for the binary classification task of VP, and then, they extracted the token scores through XAI (i.e., through Self-Attention). By summing the token scores per line, LineVul can rank lines from most likely to be vulnerable to least likely. In LineVul study, several XAI techniques were examined concluding that the Self-Attention mechanism of the Transformer model is the optimal explainability choice. It is also worth noting that all of them clearly outperformed the CppCheck [62] static code analyzer. Furthermore, DeepLineDP [63] used Recurrent Neural Networks (RNNs) to perform file-level defect prediction on sequences of source code tokens, and then, employed an Attention layer [64] to explain the predictions, and therefore, extract the tokens that contribute the most to the prediction of the defective files.

However, Cheng et al. [38] observed poor performance of several XAI-based approaches in localizing vulnerabilities due to the existence of irrelevant features learned by VPMs, the susceptibility of VPMs to perturbations not related to vulnerabilities, and the ineffectiveness of the explainers in choosing meaningful characteristics. In addition, Sotgiu et al. [39] investigated the capacity of explainability techniques to find the features that contribute most to ML model decisions. Their study used an XAI method called SHapley Additive exPlanations (SHAP) to highlight important issues in model construction that obstruct its ability to identify vulnerabilities. Specifically, they observed that a Transformer-based VPM tends to learn spurious correlations among data. For instance, the models often emphasize tokens, which should not have a discriminating role in prediction (e.g., special characters of programming languages). They further observed that the top-10 tokens, which led to a negative or positive decision, were the same for both classes. Therefore, VD through explainability often leads to incorrect findings.

2.3. Beyond state-of-the-art

In contrast to previous studies, we do not use an XAI-based technique but present a Sequence-to-Sequence approach to achieve line-level VD, which, in addition to a list of sorted lines to prioritize inspection efforts, can provide the inspector with a specific set of lines that it considers vulnerable. To our knowledge, we are the first to propose a Sequence-to-Sequence method for vulnerability detection, which, in fact, not only overcomes the limitations of XAI-based approaches (regarding the dependence on the function-level VPM as well as the failure due to spurious patterns learned by the VPM), but also achieves better results. Additionally, LocVul proposes a method to efficiently handle approximate line predictions (i.e., LLM hallucinations in VD).

3. Overview of the LocVul approach

In this section, we provide a detailed presentation of the proposed approach. Figure 3 illustrates the high-level overview of the entire methodology. First, there is the phase of training a function-level vulnerability prediction model. Subsequently, there is the training phase of the line-level vulnerability detection model, and, finally there is the inference phase where both models are executed to detect vulnerabilities in a function under test.

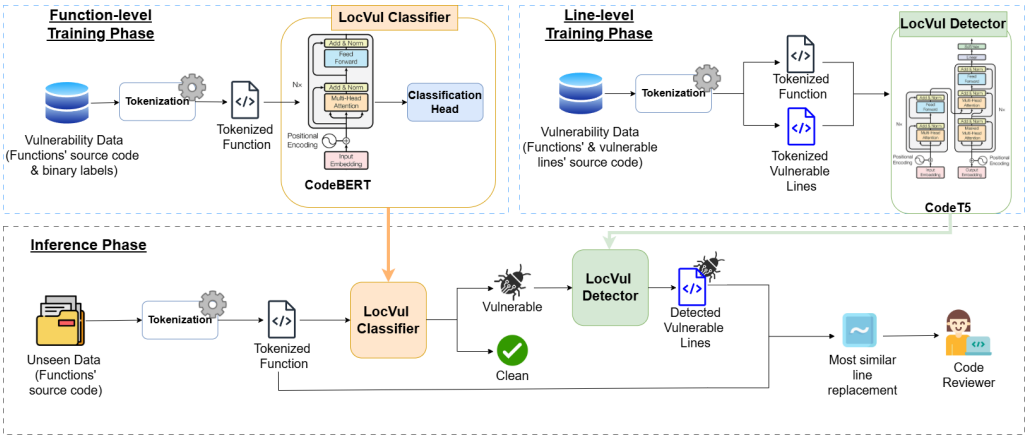


Figure 3: Overview of the overall approach.

3.1. Function-level vulnerability prediction model

For VP at the function level, we use the pre-trained Transformer-based CodeBERT model [37], which has demonstrated promising results in the literature [29],[34],[36],[39]. CodeBERT is a variant of the popular NLP model BERT, which has been pre-trained on the Masked Language Modeling (MLM) target [51]. More specifically, CodeBERT has the same encoder-only architecture as the optimized variant of BERT called RoBERTa [56], but it is bimodal, i.e., it is pre-trained on pairs of source code written in six different programming languages and documentations written in natural language [37].

To construct the function-level predictor, we fine-tune CodeBERT in the task of binary classification of software functions to vulnerable or clean ones. Initially, as can be seen in Figure 3, a large vulnerability dataset that consists of the source code of functions and their corresponding binary labels (i.e., vulnerable or clean function) is leveraged as historical knowledge to train the model. The source code is then tokenized by the CodeBERT tokenizer in order to derive the tokenized functions, which are the actual input to CodeBERT. This tokenizer utilizes subword-based tokenization to divide unknown words into smaller words or characters. A classification head (i.e., a classification layer) is placed on top of CodeBERT layers in order to facilitate its training in the VP objective. CodeBERT along with the classification head constitute the LocVul Classifier.

3.2. Line-level vulnerability detection model

In the line-level training phase, we follow a Sequence-to-Sequence training paradigm. Commonly, Sequence-to-Sequence models convert an input sequence into an output sequence, which can be of different lengths. These models are trained to pair the input to the output and are particularly useful when understanding the context of the entire sequence is required (tasks such as translation, summarization, etc.) [65]. Since our purpose is to extract the vulnerable lines out of a given function, the task can be considered a Sequence-to-Sequence task, where the source code of the function is the input sequence and the source code of the vulnerable lines is the output sequence. Actually, the task of VD can be treated like a summarization task, where the function is the text to be summarized and the vulnerable lines correspond to the extractive summary.

To facilitate a Sequence-to-Sequence training, we employ an encoder-decoder model, in contrast to the encoder-only CodeBERT that we utilize for binary classification. The encoder-decoder architecture is naturally suited for tasks that require alignment between input and output sequences. Specifically, we employ the pre-trained Transformer-based model called Text-To-Text Transfer Transformer (T5) [66], which has learned to predict masked spans of text during its pre-training. In particular, we use the CodeT5 variant of T5, which has been pre-trained on programming languages [41]. As an encoder-decoder model, CodeT5 is a very appropriate LLM for a task that extracts specific lines from a given set of lines (i.e., functions). The encoder converts the input sequence into a vector representation, and the decoder generates the output sequence from this representation.

As can be seen in Figure 3, for the purposes of training and evaluating our line-level detection model, we leverage a large vulnerability database, which consists of source code pairs of functions and their lines that are labeled as vulnerable. The pairs of source code are then tokenized in order to produce pairs of tokenized functions and tokenized vulnerable lines. Both tokenized pairs are given as input to CodeT5 model, which is fine-tuned in the VD objective. The fine-tuned CodeT5 model constitutes the LocVul Detector.

3.3. Inference phase

Finally, there is the inference phase of the proposed methodology, as depicted in Figure 3. During inference, unseen functions are given for analysis, they are tokenized, and they are classified as vulnerable or clean by the LocVul Classifier (i.e., the fine-tuned CodeBERT model). The predicted as vulnerable functions are then fed to the LocVul Detector (i.e., the fine-tuned CodeT5 model), which extracts from the functions the lines it considers vulnerable.

In the methodology illustrated in Figure 3, after the vulnerable lines detection by the LocVul Detector, a process of most similar line replacement is taking place in order to address the problem of LLM hallucinations. This issue refers to cases where LLMs generate output that is syntactically correct and contextually plausible, but does not correspond to any actual line in the analyzed function [42]. In our case, we observed that the vulnerable lines produced by the LocVul Detector (based on CodeT5) are often almost but not entirely identical to actual lines in the function. For instance, instead of correctly returning the line `strcpy(buffer, input);`, the model might hallucinate a line such as `copyString(buffer, input);`, which does not exist in the original code. These discrepancies can hinder both automated and manual vulnerability remediation, as well as the reported localization accuracy.

To mitigate this, we implement a line replacement mechanism that, for each tokenized line generated by the LocVul Detector that is not present in the original function (i.e., is hallucinated), computes the cosine similarity [67] between its embedding and those of all the tokenized lines in the function. Subsequently, it replaces the generated line with the most similar one from the actual lines. In this way, every reported vulnerable line corresponds to a concrete line in the source code, substantially reducing hallucination-induced errors.

As a final outcome, the concerned user, who is a security expert or a code reviewer, receives, for each function flagged as vulnerable, the lines of the function that the LocVul methodology detects as vulnerable. Specifically, in contrast to the XAI-based methods that return all the lines of the functions ranked by a vulnerability likelihood, LocVul approach returns a list of the specific lines detected as vulnerable as a first recommendation, and then, the reviewer can continue inspecting the rest lines of the functions in the order in which they appear in the source code. We have to specify that as a code reviewer we consider a human that relies primarily on the output of the tool to guide the inspection process, without assuming additional prior knowledge about the specific vulnerability or function under review. In our evaluation, we simulate this reviewer by having them examine the lines returned by LocVul in the provided order until the true vulnerability is located.

4. Experimental design

This section initially expresses the Research Questions (RQs) that will guide the entire experimental design. Subsequently, it presents the studied dataset, the details of LocVul implementation, the scheme that we followed to evaluate our approach, and the selection of the baseline method.

4.1. Research questions definition

In order to formulate the objectives of the study, the following RQs are defined:

- **RQ**$_1$: How accurate is LocVul for line-level vulnerability detection?

  RQ$_1$ investigates whether LocVul can outperform existing approaches in detecting vulnerabilities that reside in source code. The accuracy of LocVul is measured through various evaluation metrics to determine whether it surpasses the explainability-based approaches for identifying line-level vulnerabilities.

- **RQ$_2$**: What is the cost-effectiveness of LocVul for line-level vulnerability detection?

  RQ$_2$ is responsible for evaluating the cost-effectiveness of LocVul, which is a critical factor for the actual employment of VD techniques. The analysis measures the inspection effort that one has to devote to find actual vulnerable lines using LocVul and compares it to existing explainability-based approaches.

4.2. Dataset

For the purposes of our experimental analysis, we leverage the Big-Vul dataset, which was originally provided by Fan et al. [68]. Big-Vul is one of the most established and widely-used datasets in the field [33],[36],[58],[69],[70]. In brief, Big-Vul is a large C/C++ vulnerability-related dataset that consists of open-source GitHub repositories labeled using information retrieved from the CVE database. Among various information (e.g., severity scores, vulnerability types, CVE summaries, etc.), it contains the code changes associated with the vulnerability that actually correspond to the vulnerability fixes. These code changes were retrieved from the commit history of the repositories. Overall, Big-Vul contains 188,636 functions, which constitute the samples of the dataset, gathered from 348 open-source C/C++ software repositories. In particular, the dataset includes 10,900 functions labeled as vulnerable and 177,736 functions that are considered clean or at least neutral (i.e., no vulnerability has been found for them yet). Therefore, Big-Vul is a dataset with a vulnerability ratio of 6.13 %.

We choose Big-Vul for our experiments, since it is not only a very large dataset, but it is also considered as a benchmark in the VP domain [71], contains data from a variety of projects, includes various vulnerability types, contains real vulnerabilities reported in the CVE database, has a realistic class balance ratio (i.e., 6.13 %) [72], and will therefore help to better position our study in the relevant domain. Moreover, it contains ground truths on line-level granularity as opposed to other popular vulnerability-related datasets that have only function-level labels (e.g., Devign [25], ReVeal[26], DiverseVul[52], etc.), which is a critical factor in our study.

Furthermore, after conducting a literature review of the studies described in Section 2, we observed that the most notable studies [36],[38],[57],[58], which deal with fine-grained VD, used the Big-Vul dataset. Only Sotgiu et al. [39] preferred the Devign dataset [25], which does not contain line-level labels, but their scope was not the detection of vulnerable lines. Instead, they investigated the existence of spurious correlations and bias in VPMs through XAI techniques. In addition, LineFlowDP [61] and DeepLineDP [63] studies addressed the problem of detecting defective lines in defective files and, therefore, their dataset contains bugs and general weaknesses in the source code, whereas our study focuses exclusively on security vulnerabilities, which are a specific type of weaknesses that threaten the security of software systems.

Before proceeding with training and evaluating the proposed models, we perform a step of dataset splitting. More specifically, the dataset is divided into 80 % training, 10 % validation, and 10 % testing data. The training set is used for fine-tuning the models (both function-level and line-level detection models), while the validation set is leveraged to choose the optimal hyperparameters, and the testing set constitutes the unseen data that we utilize for the final evaluation of the models.

4.3. Implementation details

To conduct our experimental analysis, we use the Hugging Face Transformers library [73] to load pre-trained Transformer-based models (i.e., CodeBERT and CodeT5) and efficiently manage tokenization. We also utilize the PyTorch framework [74] to construct our DL models. Furthermore, all experiments were carried out on the CUDA parallel computing platform of a couple of NVIDIA GeForce RTX 4080 graphic cards.

As regards the classification model that is trained to predict vulnerable functions, we first load the pre-trained CodeBERT model from Hugging Face and then fine-tune it for the downstream task of binary classification. The *codebert-base* model that we load from Hugging Face is an encoder-only model that consists of 12 layers with 768 hidden size each and 12 Attention heads in each Attention block, containing overall 125 million parameters (i.e., trainable weights).

In the fine-tuning phase, CodeBERT receives pairs of source code and binary labels (i.e., vulnerable and non-vulnerable functions) and is trained to predict which ones are vulnerable. During this process, all the layers and the weights of the CodeBERT model are updated to be adjusted to the downstream task. Furthermore, a classification head, which is a fully-connected layer, is appended to the model, and fine-tuning is performed using a learning rate (LR) of 0.00002. A linear scheduler is applied to progressively reduce the LR during training. The AdamW optimizer [75] is employed to optimize the gradient descent process. To prevent over-fitting, the Early Stopping technique is used, ensuring the optimal number of training epochs. Input sequences are limited to a maximum length of 512 tokens, which is the upper bound supported by the model. Additionally, zero padding is applied to standardize sequence lengths during the encoding process using the tokenizer of CodeBERT. For sequences exceeding the maximum length, truncation is performed to maintain consistency and compatibility with the model. Finally, the Cross-Entropy loss [76] is selected to compare predicted probabilities with true labels. The values of aforementioned hyperparameters are determined based on the values of common classification metrics (e.g., F_1-score) during experimentation on the validation data. A summary of the characteristics of the fine-tuned CodeBERT model is provided in Table 1.

As regards the Sequence-to-Sequence (Seq2Seq) model trained for line-level vulnerability detection, we employ the pre-trained CodeT5 model [41] from Hugging Face. Specifically, we load the *codet5-base* model, which comprises an encoder and a decoder, each with 12 layers, a hidden size of 768 neurons per layer, and 12 Attention heads per Attention block, resulting in approximately 223 million trainable parameters.

We then fine-tune it for the downstream task of aligning input sequences (i.e., vulnerable functions) with output target sequences (i.e. vulnerable lines). More specifically, the models learns to generate target sequences that correspond to vulnerable lines within the input functions. During this process, first, the input sequences are tokenized using the CodeT5 tokenizer, which employs subword-based tokenization to handle both known and unknown tokens, and then, all layers and weights of the model are updated to adapt to the downstream task of vulnerability detection. Regarding the selection of the configurable hyperparameters of the model, it is guided by performance on validation data, evaluated using the Recall-Oriented Understudy for Gisting Evaluation (ROUGE) metric [77]. ROUGE-L in particular, is commonly used for Sequence-to-Sequence NLP tasks to measure the Longest Common Subsequence (LCS) overlap among generated and reference texts [78], maintaining the structure of the sequences.

For the training process (i.e., fine-tuning), a LR of 0.00005 is selected, progressively reduced using a linear scheduler throughout the training. To optimize the gradient descent, we utilize the AdamW optimizer [75]. The Early Stopping technique is also applied to determine the number of training epochs to not exceed the number that the model no longer improves, preventing over-fitting. Additionally, zero padding is employed to standardize sequence lengths across batches, and truncation is used for sequences exceeding the maximum allowable length for this model (i.e., 512 tokens). The aforementioned configurations of the fine-tuned CodeT5 model are summarized in Table 1.

Table 1: Characteristics of the fine-tuned CodeBERT and CodeT5 models for LocVul Classifier and LocVul Detector, respectively.

Attribute	**CodeBERT**	**CodeT5**
Model Type	Encoder	Encoder-Decoder
Transformer Variant	RoBERTa	T5
Version	codebert-base	codet5-base
Transformer Layers	12	24
Hidden Size	768	768
Attention Heads	12	12
Number of Parameters	$\sim 125M$	$\sim 223M$
Learning Rate (LR)	0.00002	0.00005
Optimizer	AdamW	AdamW
Loss Function	Cross-Entropy	Cross-Entropy
Max Length	512	512

4.4. Evaluation scheme

To assess our approach on the testing set, we use several evaluation metrics, which have been established in the related literature [32],[36],[58],[61],[63]. We employ widely used classification metrics [32] to evaluate the function-level predictions, while the line-level performance of the model is measured through metrics commonly used in recommendation systems, since VD methods traditionally provide a list of lines sorted from the most likely to be vulnerable to the least ones, recommending that the top of the list be inspected first [36],[58],[61],[63]. Moreover, different metrics are used to measure the accuracy and the cost effectiveness/effort-awareness of the line-level detectors [36],[61],[63].

First, we assess the function-level predictions of the CodeBERT-based VPM (i.e., LocVul Classifier). Since the dataset is highly unbalanced (i.e., class balance ratio 6.13%), classic Accuracy computed as $\frac{TP+TN}{TP+TN+FP+FN}$, where TP, TN, FP, and FN stand for true positives, true negatives, false positives, and false negatives, is not sufficient in itself. Therefore, all of Accuracy, Recall ($\frac{TP}{TP+FN}$), Precision ($\frac{TP}{TP+FP}$), and F_1-score ($\frac{2\times \text{Precision}\times \text{Recall}}{\text{Precision}+\text{Recall}}$) are considered. Being the harmonic mean of Precision and Recall, F_1-score is considered the most critical metric in VP, providing a single metric capable of measuring both [26],[32].

Subsequently, to evaluate the accuracy of the LocVul Detector, we compute Top-K Accuracy (A@K), which is the most used measurement for line-level detection [36],[58],[61].

Top-K Accuracy is defined as $\frac{x}{N} \times 100\%$, where x is the number of vulnerable functions whose at least one line is vulnerable in the top-K of the list, and N the total number of vulnerable functions. It measures the percentage of functions for which the model detects at least one truly vulnerable line in the top-K lines it considers most likely to be vulnerable.

Similarly, we report the average value of the Top-K Precision (P@K) and Top-K Recall (R@K) metrics across all functions. The former stands for the actual vulnerable lines detected in the top-K lines proposed by the model and is computed as $\frac{\text{Number of vulnerable lines in } K}{\text{Total number of lines in } K}$. The latter, which is equal to $\frac{\text{Number of vulnerable lines in } K}{\text{Total number of vulnerable lines}}$, computes the truly vulnerable lines of the top-K lines proposed by the model among the total of vulnerable lines in a function [10].

Nonetheless, Top-K Precision and Top-K Recall are indifferent to the order of vulnerable lines in the top-K ranking. They depict the number of vulnerable lines within the top-K, regardless of their specific order. Therefore, we utilize Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) at K, which are two rank-aware metrics.

Specifically, MRR@K is a ranking quality metric, which is equal to the arithmetic mean of the Reciprocal Ranks (RR) across all functions, where a RR is the inverse of the position of the first vulnerable line. MRR@K is calculated as $\frac{1}{N}\sum_{i=1}^{N}\frac{1}{\text{rank}_i}$, where N refers to the total number of vulnerable functions and rank_i is the position of the first vulnerable line for function i in the top-K results [79]. If no vulnerable lines are found, MRR is equal to zero.

Moreover, MAP@K is the mean of the Average Precision (AP) at K of all functions analyzed. It considers the number of vulnerable lines in the top-K list and their position in the list. First, we compute the AP per function by averaging the precision at each position of vulnerable lines in the top-K ranking list. Particularly, MAP@K is calculated as $\frac{1}{N}\sum_{n=1}^{N}$ AP@K, where N presents the total number of vulnerable functions, and AP@K is calculated as $\frac{1}{M}\sum_{k=1}^{K} P@k \times rel(k)$, where M refers to the number of vulnerable lines in the top-K results for a specific function, K refers to the selected cutoff point, P@k is the Top-k Precision, and rel(k) equals 1 if the line at position k is vulnerable and 0 if not [79].

Since such a VD tool would only be useful if it managed to rank the vulnerable lines of the analyzed functions at the top of the recommended list of lines, we have to choose a relatively small value of K. Therefore, we choose $K = 10$ to compute the aforementioned metrics, following the example of prior studies [36],[58].

In addition, to measure the cost-effectiveness of the proposed approach, we use metrics indicative of the effort required to achieve a sufficient line-level VD performance, First, we employ one function-based measurement called Initial False Alarms (IFA) [36],[63], which counts how many false alarms (i.e., non-vulnerable lines) occur before the first truly vulnerable line in a function. The lower the IFA, the less effort and time is spent by the code reviewer in inspecting non-vulnerable lines.

Furthermore, we also use Effort@K%Recall and Recall@K%LOC to measure cost-effectiveness by considering the entire dataset under test as a whole [36],[63]. In particular, Effort@K%Recall measures how much effort, expressed in lines of code (LOC), is required to identify the $K\%$ of the actual vulnerable lines of the entire testing set. The lower the value of Effort@K%Recall, the less effort is required for the code reviewer to find the $K\%$ of vulnerable lines. Moreover, Recall@K%LOC calculates the number of

true vulnerable lines found by making an effort equal to K LOC inspected. The higher the Recall@K%LOC the more vulnerable lines are found for a fixed amount of effort. Similarly to [36], we use the Effort@20%Recall and the Recall@1%LOC.

Finally, to ensure that our analysis is not affected by randomness in data shuffling, model training, and other computational processes, we repeat all experiments ten times using a different random seed each time. We thus collect ten different values for each evaluation metric and report their average.

4.5. Selection of baseline approach

To demonstrate the efficacy of LocVul in line-level VD, we compare it with the current state-of-the-art. Specifically, during the evaluation of our experimental analysis presented in Section 5, we conduct a one-to-one comparison with the most notable existing method used for VD, as identified through our literature review in Section 2.

It is well-known in the literature that XAI techniques have been widely adopted to address the challenge of VD at line-level of granularity [33],[35],[38]. By inspecting the related studies described in Section 2, we observed that graph-based methods, such as IVDetect [58], utilize the GNNExplainer [60] to interpret the predictions and, therefore, localize the vulnerabilities. However, although they achieve high coarse-grained accuracy, they are disadvantaged in line-level detection compared to text mining-based XAI approaches [36],[38].

Such a text mining-based methodology, which managed to outperform IVdetect leveraging XAI is the one proposed by Fu et al. namely LineVul [36]. They first fine-tuned the CodeBERT model in VP, and then used the Self-Attention mechanism of the Transformer-based model to explain its predictions. More specifically, LineVul summarizes the Attention scores of the tokens included in the vulnerable functions to calculate line-level scores and, subsequently, ranks the lines of the functions from the most likely to be a vulnerable line to the least likely.

In addition, they compared Self-Attention against various other XAI techniques applied on textual code representations. The results presented in LineVul study [36], demonstrated the superiority of Self-Attention among XAI-based techniques, since it clearly outperformed all of Layer Integrated Gradient (LIG) [80], Saliency [81], DeepLift [82],[83], DeepLiftSHAP [84], and GradientSHAP [84] approaches in identifying vulnerable lines. They also showed a great superiority of Self-Attention compared to a traditional (non-XAI based) approach, the static code analysis approach, through the CppCheck [62] analyzer. Hence, it can be argued that the Self-Attention mechanism has emerged as the cutting-edge approach for the detection of fine-grained vulnerabilities.

Accordingly, we choose the Self-Attention-based explainability method as our baseline for comparison. We do not just report the results presented in the study by Fu et al. [36], but we proceed with implementing it from our own (i.e., replicating their methodology) to avoid bias in the results due to the potential inconsistency in the implementation settings of the compared approaches (i.e., Self-Attention and our LocVul Detector). In this way, we can also evaluate Self-Attention using a more broad and representative set of evaluation metrics than those presented in LineVul study [36]. Moreover, it is not clear in [36] whether the localization results were computed using only the true positives or all the predicted as vulnerable samples, with the latter being what we consider to be the approach corresponding to a real-world scenario.

In particular, we implement the entire explainability-based methodology that leverages the Self-Attention mechanism of CodeBERT to localize the vulnerable lines in the functions based on the importance of the different tokens of the input in the function-level prediction. During the development of this implementation, we consulted the LineVul replication package [85]. From now on we will refer to this baseline as the Self-Attention approach.

5. Results

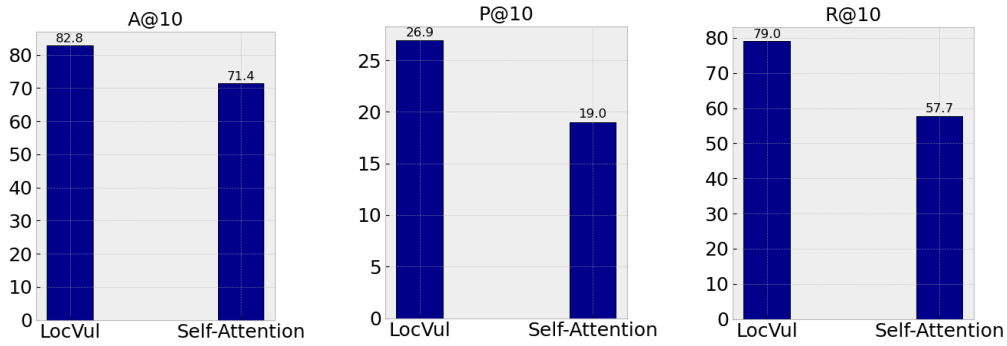
In this section, the results of our entire analysis are presented. Specifically, we present the evaluation results of the proposed Seq2Seq approach (i.e., LocVul) and we compare it against the Self-Attention approach that we selected as our baseline (see Section 4.5). Regarding the function-level VPM (i.e., LocVul Classifier), which is used as a fisrt step for both the Seq2Seq and Self-Attention approaches during our experimental study, the CodeBERT-based model achieves a high predictive performance in VP. Specifically, it identifies vulnerable functions with 99.04% Accuracy, 95.62% Precision, 86.88% Recall, and 91.04% F_1-score. Regarding line-level VD, we evaluate both the proposed Seq2Seq model (i.e., LocVul Detector) and the state-of-the-art Self-Attention approach in their capacity of localizing vulnerabilities. To this end, we provide a detailed analysis with respect to the two RQs that we defined in Section 4.1.

5.1. RQ_1 - Accuracy of LocVul for line-level vulnerability detection

An early sign that the Seq2Seq-based LocVul Detector has high detection accuracy is the fact that, for around 60% of vulnerable functions, the set of lines generated by the model exactly matches the ground truth vulnerable lines. Therefore, even if we ignore the recommended line ranking lists, LocVul manages to identify the exact vulnerable segment in 60% of the vulnerable functions. Nevertheless, to answer RQ_1, we evaluate the accuracy of LocVul and we compare it against Self-Attention using proper evaluation metrics (see Section 4.4). Initially, we calculate the Top-10 Accuracy of the LocVul Detector to measure how often it manages to detect one vulnerable line in the 10 first lines it suggests for inspection. We then evaluate LocVul in terms of Top-10 Precision and Top-10 Recall to also consider the number of lines detected in the 10 first lines. Figure 4 presents the values of these metrics for LocVul compared to Self-Attention.

As can be seen in Figure 4, the LocVul Detector achieves a Top-10 Accuracy equal to 82.8%, which is 11.4% higher than the 71.4% of Self-Attention, showing that the proposed model detects at least one vulnerable line in much more cases than the baseline. In addition, the Top-10 Precision of LocVul is 26.9% compared to 19.0% of Self-Attention, a result that indicates the superiority of LocVul in identifying multiple truly vulnerable lines in the top-10 of the lines it suggests as vulnerable. Specifically, LocVul's Top-10 Precision of 26.9% shows that the model identifies on average 2.69 actual vulnerable lines in the top-10 recommended lines. Moreover, a Top-10 Recall equal to 79.0%, which is much higher than Self-Attention's 57.7%, highlights the ability of LocVul to detect many of the function's vulnerable lines in the top-10 ranking.

At this point, we should mention that the results obtained with the developed Self-Attention approach for VD are slightly different from those reported in [36]. In particular, Top-10 Accuracy is higher by 6.4%. This can be attributed mainly to the fact that, in



(a) Comparison of Top-10 Accuracy (A@10). (↗) Higher A@10 - Better.

(b) Comparison of Top-10 Precision (P@10). (↗) Higher P@10 - Better.

(c) Comparison of Top-10 Recall (R@10). (↗) Higher R@10 - Better.

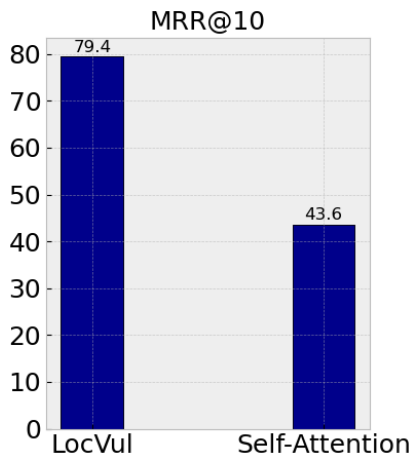
Figure 4: Top-10 Accuracy, Precision, and Recall of LocVul compared to the baseline Self-Attention method for line-level vulnerability detection.

our implementation, we remove the functions that are labeled as vulnerable but have incomplete line-level labels (i.e., they have no information on which are the vulnerable lines). Those samples act as noise in the line-level dataset and are also a barrier to training the Seq2Seq model. Therefore, they have to be removed from both training and testing sets.

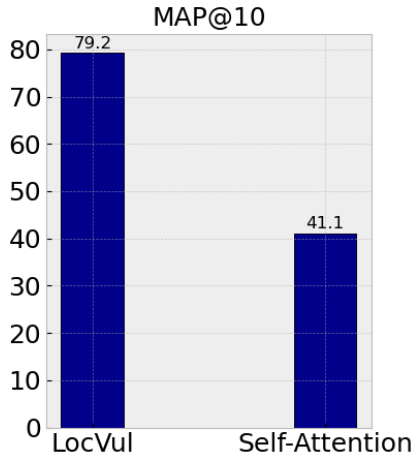
Furthermore, to evaluate the accuracy of our approach in a rank-aware manner by considering the order of the lines proposed as vulnerable by the model, we compute the Mean Reciprocal Rank (MRR) and the Mean Average Precision (MAP) at top-10 ranking list. Figure 5 shows the results for LocVul and Self-Attention approaches in terms of MRR@10 and MAP@10 in the left and right bar charts, respectively. Our Seq2Seq-based LocVul approach manages to outperform the XAI-based Self-Attention approach by almost double the score in terms of MRR@10 and MAP@10.

In particular, the LocVul Detector achieves MRR@10 equal to 79.4%, which is 35.8% higher than Self-Attention's 43.6% leading to the conclusion that it is able not only to detect a vulnerable line in more functions than XAI techniques, but also to detect it earlier in the ranking list of lines. In addition, MAP@10 of the LocVul Detector is equal to 79.2% in contrast to Self-Attention, which has MAP@10 equal to 41.1%. This observation demonstrates the enhanced capability of LocVul to detect more vulnerable lines higher up in the ranking list compared to the Self-Attention mechanism.

Although a clear difference in the values of the accuracy-related evaluation metrics between the two approaches is observed, we apply a statistical test to support further our findings. In particular, we conduct the Wilcoxon Signed-Rank Test [86] to check whether the paired accuracy scores, obtained from ten different random seed values, differ significantly between LocVul and Self-Attention. We repeat this analysis for all of the Top-10 Accuracy, Precision, Recall, MRR@10, and MAP@10 metrics. Considering as the null hypothesis that LocVul is not more accurate than Self-Attention for each metric, the Wilcoxon test returns p-values of 0.00098 across all metrics, which are well below the 0.05 significance threshold. Therefore, we reject the null hypothesis and state



(a) Comparison of Top-10 Mean Reciprocal Rank (MRR@10). (↗) Higher MRR@10 - Better.



(b) Comparison of Top-10 Mean Average Precision (MAP@10). (↗) Higher MAP@10 - Better.

Figure 5: Top-10 Mean Reciprocal Rank and Mean Average Precision of LocVul compared to the baseline Self-Attention method for line-level vulnerability detection.

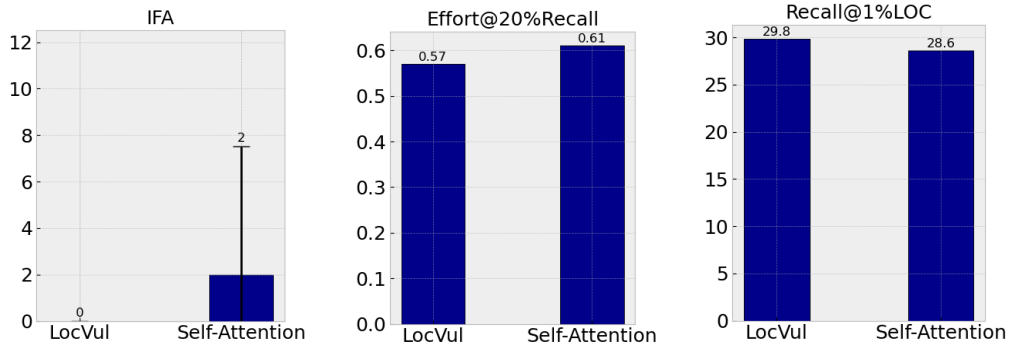
that LocVul is significantly more accurate than Self-Attention. Hence, we can argue that:

The LocVul approach, which extracts vulnerable lines from the source code of vulnerable functions using a code-aware Seq2Seq LLM, specifically CodeT5, achieves high accuracy in vulnerability detection, outperforming the baseline Self-Attention explainability method.

5.2. RQ_2 - Cost-effectiveness of LocVul for line-level vulnerability detection

In RQ_2, we investigate the cost-effectiveness of the proposed VD approach. In other words, we evaluate the predictive performance of LocVul with respect to the effort required to achieve this performance. To this end, we compute the cost-effectiveness evaluation metrics, which are described in Section 4.4. On the one hand, we evaluate LocVul by calculating the IFA metric that evaluates cost-effectiveness by counting the false alarms that one has to inspect until finding one truly vulnerable line in a function. On the other hand, we evaluate LocVul using the Effort@20%Recall to measure the effort required to find the 20% of the vulnerable lines in the entire testing set, while Recall@1%LOC is used to find the truly vulnerable lines detected by inspecting a fix amount of LOC (i.e., the 1% of the total LOC) in the testing set. Figure 6 presents the values of these metrics for the proposed LocVul approach compared to the baseline Self-Attention approach.

In particular, Figure 6a presents the IFA values of the compared approaches. Similarly to previous studies [36],[61],[63], our evaluation is based on median IFA values. As illustrated in Figure 6a, LocVul achieves a median IFA equal to 0, suggesting that the Seq2Seq model manages to include a vulnerable line in the generated lines in most of the functions (i.e., for at least half of the functions in the dataset). In contrast, the Self-Attention mechanism achieves a higher median IFA equal to 2. This observation



(a) Comparison of Initial False Alarm (IFA). (↘) Lower IFA - Better.

(b) Comparison of Effort@20%Recall. (↘) Lower Effort@20%Recall - Better.

(c) Comparison of Recall@1%LOC. (↗) Higher Recall@1%LOC - Better.

Figure 6: Cost-effectiveness evaluation of LocVul compared to the baseline Self-Attention method for line-level vulnerability detection.

indicates that LocVul improves the baseline approach in terms of the number of false alarms to inspect before finding a vulnerable line. In addition, as illustrated by the error bars representing half the Interquartile Range (IQR/2), LocVul demonstrates minimal or no variability, consistently achieving very low IFA values across all samples as opposed to Self-Attention, which has a larger spread.

Furthermore, Figure 6b demonstrates that LocVul achieves Effort@20%Recall equal to 0.57% against Self-Attention's 0.61%. The smaller the Effort@20%Recall, the less effort is required to detect vulnerabilities. Therefore, a code reviewer that uses LocVul can put less effort into identifying 20% of the truly vulnerable lines. In addition, Figure 6c shows that LocVul's Recall@1%LOC is 29.8% in contrast to Self-Attention's 28.6%. In other words, a code reviewer is able to detect 1.2% more vulnerable lines by inspecting the top-1% recommended lines when using LocVul instead of the baseline Self-Attention approach.

In addition, we perform the Wilcoxon Signed-Rank Test [86] to judge if the differences in the cost-effectiveness metrics between the LocVul and Self-Attention approaches are statistically significant. Using the metrics computed for 10 repetitions of the experiments with different random seeds, the p-values of the Wilcoxon test are equal to 0.00098, 0.02108, and 0.01368 for median IFA, Effort@20%Recall, and Recall@1%LOC metrics, respectively. Although the differences in the means of the cost-effectiveness metrics are not large, all the p-values are lower than the 0.05 threshold and, therefore, we can state that the differences in cost-effectiveness between the two approaches are statistically significant. Hence, considering IFA, Effort@20%Recall, and Recall@1%LOC scores, it is concluded that:

The Seq2Seq-based LocVul approach offers a cost-effective solution for line-level vulnerability detection, reducing the effort required to identify actual vulnerable lines compared to the baseline Self-Attention explainability method.

6. Discussion

In this section, we discuss the findings of our experimental analysis and provide useful insights. First, we explore the benefit gained from the line replacement mechanism that we implement to address LLM hallucinations in VD. Subsequently, implications to researchers and practitioners are provided, and then, the threats to validity of our study are disclosed.

6.1. Impact of the most similar line replacement mechanism

In Section 3.3, we described a similarity checking mechanism based on cosine similarity between the embeddings of tokenized lines. This mechanism is applied when running the LocVul Detector in order to check whether the line generated by the model is one of the lines of the function being analyzed and, if not, to replace it with the corresponding line by calculating which line is the most similar. In this way, the proposed methodology manages to handle cases where the Seq2Seq model returns as vulnerable lines that are not in the function (i.e., LLM hallucinations in this problem). Table 2 presents the evaluation of the LocVul approach with and without employing the similarity check mechanism.

Table 2: Comparison of results achieved by LocVul with and without the most similar line replacement mechanism.

	Accuracy Metrics					**Cost-Effectiveness Metrics**		
Approach	**A@10**	**P@10**	**R@10**	**MRR@10**	**MAP@10**	**Median IFA**	**Effort@20%Recall**	**Recall@1%LOC**
With Replacement	82.8%	26.9%	79.0%	79.4%	79.2%	0	0.57%	29.8%
Without Replacement	80.6%	25.3%	74.3%	76.9%	76.5%	0	0.70%	25.4%

As shown in Table 2, the mechanism applied to replace the most similar lines provides a benefit to the entire approach, which is reflected in all evaluation metrics used. In terms of accuracy-related metrics, we can see that the proposed mechanism achieves a Top-10 Accuracy, Precision, and Recall gain equal to 2.2%, 1.6%, and 4.7%, respectively. It also provides a gain of 2.5% and 2.7% in terms of MRR@10 and MAP@10, which means that LocVul with similar line replacement manages not only to detect more vulnerable lines, but also to place them higher in the list of the returned vulnerable lines compared to LocVul without similar line replacement. One can also observe an advantage of the hallucinations handling mechanism with regard to cost-effectiveness. Although median IFA is still zero, both Effort@20%Recall and Recall@1%LOC are deteriorated when removing this mechanism. Specifically, the former increases by 0.13% and the latter decreases by 4.4%, leading to the conclusion that without similar line replacement, more effort is required by the code reviewer to find actual vulnerabilities using LocVul without being confused by hallucinations.

In addition, to verify the benefit of the similar line replacement mechanism, we perform the Wilcoxon Signed-Rank Test [86] on the evaluation metrics obtained from ten repetitions of the experiments with different random seeds. Specifically, we conduct the test on both accuracy and cost-effectiveness metrics, namely Top-10 Accuracy, Top-10 Precision, Top-10 Recall, MRR@10, MAP@10, Effort@20%Recall, and Recall@1%LOC, excluding median IFA, as it is zero in both cases. The Wilcoxon test returns p-values equal to 0.00098 for all metrics. All the p-values are below the 0.05 threshold indicating that there is a statistically significant benefit gained from the similar line replacement mechanism in terms of both accuracy and cost-effectiveness in VD.

6.2. Implications

Having presented a thorough evaluation of the proposed method, the results demonstrate that the Seq2Seq approach for VD addresses important limitations of prior methods and outperforms traditional VD techniques that rely on XAI. Unlike the state-of-the-art Self-Attention-based approach, a Seq2Seq model, such as CodeT5, can directly identify vulnerable lines in source code without ranking tokens by importance or depending on error-prone function-level prediction models, which often capture spurious data correlations. The experiments carried out in the context of RQ_1 and RQ_2 demonstrate that fine-tuning a Seq2Seq model in detecting line-level vulnerabilities achieves both higher accuracy and better cost-effectiveness than approaches based on interpreting function-level predictions. These findings encourage stakeholders to focus on Seq2Seq-based solutions for VD rather than using XAI-based techniques by default.

Furthermore, we recommend that researchers further elaborate the Seq2Seq-based approach, examining specific Seq2Seq training paradigms, in order to develop even more accurate models. For instance, they could extend our analysis by comparing techniques such as question-answering, machine translation, text generation, etc. In addition, we suggest that researchers validate the accuracy of this approach on different datasets and programming languages. Research on constructing new datasets with fine-grained vulnerability labels is recommended, as well.

Moreover, we suggest researchers to enrich the VD-related literature by experimenting with various LLMs, either of the same or larger scale. They could repeat our analysis by fine-tuning models such as PLBART [87], GPT-4 [88], and Mistral [89] on the Seq2Seq approach to localize vulnerabilities within functions. An interesting research direction is also the exploration of innovative methods from the rapidly evolving field of artificial intelligence, such as Reinforcement Learning from Human Feedback (RLHF) [90] to increase the accuracy of the proposed VD approach, and Mixture of Experts (MoE) [91] to enhance the detection of vulnerabilities of different categories.

In addition, the accuracy of LocVul combined with its cost-effectiveness and the efficiency in its execution is an important practical advantage. Software development workflows that require real- or near-real-time applications can benefit from models such as CodeT5, which, when fine-tuned, can detect vulnerable lines in source code without any substantial delay. Specifically, although training LLMs is a time-consuming process, the perception time of LocVul (i.e., average time required to analyze one function during model execution) is only 213.83 milliseconds (ms). Therefore, LocVul can detect with sufficient accuracy where a vulnerability is located in the source code of a function in 213.83 ms, without necessarily requiring additional effort to inspect the line-ranking list, as in the case of XAI-based solutions. Thus, we recommend that practitioners use such solutions as copilots (e.g., through their Integrated Development Environments) in their daily development activities to improve the overall security and productivity of the SDLC with little or no disruptions to their regular workflows.

Furthermore, practitioners are suggested to use LocVul during software development and testing, and to compare it with existing static code analysis solutions, which are traditionally used for identifying potential vulnerabilities through source code scanning. In this way, practitioners could provide useful insights regarding the comparison of solutions like LocVul against static code analysis tools, which often demand substantial time to analyze large code bases [92],[93]. Although transfer learning approaches have been

shown to outperform static code analyzers in terms of accuracy [36], a holistic evaluation considering the computational footprint of each approach would be useful.

6.3. Threats to validity

A remark about threats to the validity of our approach is needed. Threats to construct validity concern the selection of the metrics used in our evaluation scheme. In our experimental analysis, we use A@10, P@10, and R@10 to measure the accuracy of LocVul and Self-Attention approaches. We also use MRR@10 and MAP@10 measurements to evaluate the accuracy of the compared approaches in a rank-aware manner. In addition, the use of the IFA, Effort@20%Recall and Recall@1%LOC metrics contributes to a comprehensive assessment that takes into account cost-effectiveness. These metrics are widely used in the related literature [10],[36],[61],[63], which effectively mitigates construct validity risks.

The primary threat to internal validity relates to hyperparameter selection when fine-tuning our LocVul model. Although we have performed extensive hyperparameter tuning, it is not certain that we have tested all possible combinations of hyperparameter values. It can be difficult to discern the effects of specific model hyperparameters due to the correlation between them, which could lead to a lower than ideal model performance. To reduce this risk, we use a quite thorough hyperparameter tuning procedure based on the Grid-search method [94].

Moreover, a threat to internal validity relates to the accuracy and correctness of our own approach. To mitigate this concern, we have not only meticulously reviewed our code, but also aim to make the code publicly available [43] to facilitate future replication and validation of our findings. In addition, a potential threat to internal validity also is the accurate implementation of the state-of-the-art Self-Attention approach that we use as a baseline mechanism. It is important to ensure that this approach is faithfully and correctly reproduced. To this end, we thoroughly inspected and consulted the code provided by the LineVul study [36] and, additionally, we include the code we developed for the Self-Attention approach in our replication package, along with the code for the LocVul approach.

Finally, external validity is related to the generalizability of the LocVul approach. Our study is limited to one dataset, which contains open-source projects written in C/C++ language, and thereby, the findings may not extend to projects in other languages or proprietary software. We selected Big-Vul [68], which is a widely-referenced dataset in the vulnerability-related literature, especially in studies interested to line-level detection [36],[57],[58]. It consists of 348 different projects containing code changes and vulnerabilities retrieved from GitHub and the CVE database, respectively. The size and the trustworthiness of Big-Vul mitigates the generalizability risk. Other line-level datasets including projects in different programming languages could be explored in the future.

7. Conclusions and future work

This study identified limitations in existing vulnerability detection (VD) techniques and proposed the LocVul methodology, which is based on a Sequence-to-Sequence approach. Specifically, we proposed a two-step mechanism, which first, fine-tunes a Large Language Model (LLM) in predicting the vulnerable functions (i.e., a classification task),

and second, fine-tunes another LLM in extracting the vulnerable lines of a given function. The analysis found that LocVul achieved beyond state-of-the-art results, managing to clearly surpass explainability-based solutions both in accuracy and cost-effectiveness. Useful insights were also gained about the effective handling of the LLM hallucinations in VD and about the potential of tools such as LocVul to act as a co-pilot in software development.

Suggestions for future work include the construction of a complete AI-driven VD pipeline that will identify vulnerable software components, localize the specific lines of the vulnerabilities, classify them to vulnerability categories, and assign them severity scores. We aim also at examining the accuracy of LocVul to other programming languages and different application domains.

Declaration of competing interest

The authors declare no conflict of interest.

Acknowledgments

Work reported in this paper has received funding from the European Union's Horizon Europe Research and Innovation Program through the DOSS project, Grant Number 101120270.

CRediT authors' contribution statement

Ilias Kalouptsoglou: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing - Original Draft, Visualization. **Miltiadis Siavvas:** Methodology, Validation, Investigation, Writing - Review & Editing, Supervision. **Apostolos Ampatzoglou:** Methodology, Writing - Review & Editing, Supervision, Project administration. **Dionysios Kehagias:** Writing - Review & Editing, Resources, Supervision, Project administration, Funding acquisition. **Alexander Chatzigeorgiou:** Methodology, Writing - Review & Editing, Supervision, Project administration.

References

[1] ISO/IEC, ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models, ISO/IEC, 2011.

[2] B. Boehm, V. R. Basili, Software defect reduction top 10 list, Software engineering: Barry W. Boehm's lifetime contributions to software development, management, and research 34 (2007) 75.

[3] G. McGraw, Software security, Building security in (2006).

[4] Published CVE Records, `https://www.cve.org/about/Metrics`.

[5] ISO/IEC 27000:2018, `https://www.iso.org/obp/ui/\#iso:std:iso-iec:27000:ed-5:v1:en` (2023).

[6] Cve-2021-44228 detail, https://nvd.nist.gov/vuln/detail/cve-2021-44228 (2021).

[7] M. Siavvas, E. Gelenbe, D. Kehagias, D. Tzovaras, Static analysis-based approaches for secure software development, in: Security in Computer and Information Sciences, Springer International Publishing, 2018, pp. 142–157.

[8] J. Walden, J. Stuckman, R. Scandariato, Predicting vulnerable components: Software metrics vs text mining, in: 2014 IEEE 25th international symposium on software reliability engineering, 2014.

[9] J. Witts, The top 8 static code analysis solutions, https://expertinsights.com/insights/the-top-static-code-analysis-solutions/ (2024).

[10] T.-T. Nguyen, H. D. Vo, Context-based statement-level vulnerability localization, Information and Software Technology 169 (2024) 107406.

[11] B. Aloraini, M. Nagappan, D. M. German, S. Hayashi, Y. Higo, An empirical study of security warnings from static application security testing tools, Journal of Systems and Software 158 (2019) 110427.

[12] M. Siavvas, I. Kalouptsoglou, D. Tsoukalas, D. Kehagias, A self-adaptive approach for assessing the criticality of security-related static analysis alerts, in: Computational Science and Its Applications–ICCSA:, Cagliari, Italy, September 13–16, 2021, Proceedings, Part VII 21, Springer, 2021, pp. 289–305.

[13] A. Sadeghi, H. Bagheri, J. Garcia, S. Malek, A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software, IEEE Transactions on Software Engineering 43 (6) (2016) 492–530.

[14] Y. Shin, L. Williams, Is complexity really the enemy of software security?, in: Proceedings of the 4th ACM workshop on Quality of protection, 2008, pp. 47–50.

[15] I. Kalouptsoglou, M. Siavvas, D. Tsoukalas, D. Kehagias, Cross-project vulnerability prediction based on software metrics and deep learning, in: International Conference on Computational Science and Its Applications, Springer, 2020, pp. 877–893.

[16] R. Ferenc, P. Hegedűs, P. Gyimesi, G. Antal, D. Bán, T. Gyimóthy, Challenging machine learning algorithms in predicting vulnerable javascript functions, in: 7th International RAISE workshop, IEEE, 2019, pp. 8–14.

[17] K. Filus, P. Boryszko, J. Domańska, M. Siavvas, E. Gelenbe, Efficient feature selection for static analysis vulnerability prediction, Sensors 21 (4) (2021) 1133.

[18] R. Scandariato, J. Walden, A. Hovsepyan, W. Joosen, Predicting vulnerable software components via text mining, IEEE Transactions on Software Engineering 40 (10) (2014).

[19] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, A. Ghose, Automatic feature learning for predicting vulnerable software components, IEEE Transactions on Software Engineering 47 (1) (2018) 67–85.

[20] Y. Pang, X. Xue, H. Wang, Predicting vulnerable software components through deep neural network, in: Proceedings of the 2017 International Conference on Deep Learning Technologies, 2017.

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

[22] I. Kalouptsoglou, M. Siavvas, D. Kehagias, A. Chatzigeorgiou, A. Ampatzoglou, An empirical evaluation of the usefulness of word embedding techniques in deep learning-based vulnerability prediction, in: International ISCIS Security Workshop, Springer International Publishing Cham, 2021, pp. 23–37.

[23] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and discovering vulnerabilities with code property graphs, in: 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 590–604.

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

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

[26] S. Chakraborty, R. Krishna, Y. Ding, B. Ray, Deep learning based vulnerability detection: Are we there yet?, IEEE Transactions on Software Engineering 48 (9) (2022) 3280–3296. doi:10.1109/TSE.2021.3087402.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in neural information processing systems, 2017.

[28] D. Coimbra, S. Reis, R. Abreu, C. Păsăreanu, H. Erdogmus, On using distributed representations of source code for the detection of c security vulnerabilities, arXiv:2106.01367 (2021).

[29] I. Kalouptsoglou, M. Siavvas, A. Ampatzoglou, D. Kehagias, A. Chatzigeorgiou, Vulnerability prediction using pre-trained models: An empirical evaluation, in: 2024 32nd International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2024, pp. 1–6. doi:10.1109/MASCOTS64422.2024.10786510.

[30] A. Bagheri, P. Hegedűs, A comparison of different source code representation methods for vulnerability prediction in python, in: Quality of Information and Communications Technology, Springer, 2021.

[31] X. Zhou, T. Zhang, D. Lo, Large language model for vulnerability detection: Emerging results and future directions, in: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, 2024, pp. 47–51.

[32] I. Kalouptsoglou, M. Siavvas, A. Ampatzoglou, D. Kehagias, A. Chatzigeorgiou, Software vulnerability prediction: A systematic mapping study, Information and Software Technology (2023) 107303.

[33] M. Siavvas, I. Kalouptsoglou, E. Gelenbe, D. Kehagias, D. Tzovaras, Transforming the field of vulnerability prediction: Are large language models the key?, in: 2024 32nd International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2024, pp. 1–6. doi:10.1109/MASCOTS64422.2024.10786575.

[34] I. Kalouptsoglou, M. Siavvas, A. Ampatzoglou, D. Kehagias, A. Chatzigeorgiou, Vulnerability Classification on Source Code using Text Mining and Deep Learning Techniques, in: 2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C), IEEE, 2024, pp. 47–56.

[35] A. Marchetto, Can explainability and deep-learning be used for localizing vulnerabilities in source code?, in: Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024), 2024, pp. 110–119.

[36] M. Fu, C. Tantithamthavorn, Linevul: A transformer-based line-level vulnerability prediction, in: 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), 2022. doi:10.1145/3524842.3528452.

[37] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, arXiv:2002.08155 (2020).

[38] B. Cheng, S. Zhao, K. Wang, M. Wang, G. Bai, R. Feng, Y. Guo, L. Ma, H. Wang, Beyond fidelity: Explaining vulnerability localization of learning-based detectors, ACM transactions on software engineering and methodology 33 (5) (2024) 1–33.

[39] A. Sotgiu, M. Pintor, B. Biggio, Explainability-based debugging of machine learning for vulnerability discovery, in: 17th International Conference on Availability, Reliability and Security, 2022.

[40] C. Peng, X. Yang, A. Chen, Z. Yu, K. E. Smith, A. B. Costa, M. G. Flores, J. Bian, Y. Wu, Generative large language models are all-purpose text analytics engines: text-to-text learning is all your need, Journal of the American Medical Informatics Association (2024).

[41] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, arXiv preprint arXiv:2109.00859 (2021).

[42] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, et al., A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions, ACM Transactions on Information Systems (2023).

[43] I. Kalouptsoglou, Line-level Vulnerability Localization based on a Sequence-to-Sequence approach, https://sites.google.com/view/linelevellocvuln/ (2025).

[44] Y. Shin, L. Williams, An empirical model to predict security vulnerabilities using code complexity metrics, in: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, 2008, pp. 315–317. doi:10.1145/1414004.1414065.

[45] I. Chowdhury, M. Zulkernine, Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities, Journal of Systems Architecture 57 (3) (2011) 294–313.

[46] A. Hovsepyan, R. Scandariato, W. Joosen, J. Walden, Software vulnerability prediction using text analysis techniques, in: Proceedings of the 4th international workshop on Security measurements and metrics, 2012, pp. 7–10.

[47] K. Filus, M. Siavvas, J. Domańska, E. Gelenbe, The random neural network as a bonding model for software vulnerability prediction, in: Modelling, Analysis, and Simulation of Computer and Telecommunication Systems: 28th International Symposium, MASCOTS 2020, Nice, France, November 17–19, 2020, Springer, 2021, pp. 102–116.

[48] I. Kalouptsoglou, M. Siavvas, D. Kehagias, A. Chatzigeorgiou, A. Ampatzoglou, Examining the capacity of text mining and software metrics in vulnerability prediction, Entropy 24 (5) (2022) 651.

[49] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, arXiv:1301.3781 (2013).

[50] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov, Enriching word vectors with subword information, Transactions of the association for computational linguistics 5 (2017) 135–146.

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

[52] Y. Chen, Z. Ding, L. Alowain, X. Chen, D. Wagner, Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection, in: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, 2023, pp. 654–668.

[53] B. Steenhoek, M. M. Rahman, R. Jiles, W. Le, An empirical study of deep learning models for vulnerability detection, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 2237–2248.

[54] S. Kim, J. Choi, M. E. Ahmed, S. Nepal, H. Kim, Vuldebert: A vulnerability detection system using bert, in: IEEE International Symposium on Software Reliability Engineering Workshops, 2022. doi:10.1109/ISSREW55968.2022.00042.

[55] H. Hanif, S. Maffeis, Vulberta: Simplified source code pre-training for vulnerability detection, in: 2022 International Joint Conference on Neural Networks (IJCNN), IEEE, 2022, pp. 1–8.

[56] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, V. Stoyanov, Roberta: A robustly optimized bert pretraining approach, arXiv:1907.11692 (2019).

[57] D. Hin, A. Kan, H. Chen, M. A. Babar, Linevd: statement-level vulnerability detection using graph neural networks, in: Proceedings of the 19th international conference on mining software repositories, 2022, pp. 596–607.

[58] Y. Li, S. Wang, T. N. Nguyen, Vulnerability detection with fine-grained interpretations, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 292–303.

[59] J. Ferrante, K. J. Ottenstein, J. D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems (TOPLAS) 9 (3) (1987) 319–349.

[60] Z. Ying, D. Bourgeois, J. You, M. Zitnik, J. Leskovec, Gnnexplainer: Generating explanations for graph neural networks, Advances in neural information processing systems 32 (2019).

[61] F. Yang, F. Zhong, G. Zeng, P. Xiao, W. Zheng, Lineflowdp: A deep learning-based two-phase approach for line-level defect prediction, Empirical Software Engineering 29 (2) (2024) 50.

[62] A tool for static c/c++ code analysis, `https://cppcheck.sourceforge.io/`.

[63] C. Pornprasit, C. K. Tantithamthavorn, Deeplinedp: Towards a deep learning approach for line-level defect prediction, IEEE Transactions on Software Engineering 49 (1) (2022) 84–98.

[64] D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, arXiv:1409.0473 (2014).

[65] C. Guide, What is Sequence-to-Sequence Models, `https://www.chatgptguide.ai/2024/03/01/what-is-sequence-to-sequence-model-llms-explained/` (2024).

[66] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu, Exploring the limits of transfer learning with a unified text-to-text transformer, The Journal of Machine Learning Research 21 (1) (2020) 5485–5551.

[67] P. Sitikhu, K. Pahi, P. Thapa, S. Shakya, A comparison of semantic similarity methods for maximum human interpretability, in: 2019 artificial intelligence for transforming business and society (AITB), Vol. 1, IEEE, 2019, pp. 1–4.

[68] J. Fan, Y. Li, S. Wang, T. N. Nguyen, Ac/c++ code vulnerability dataset with code changes and cve summaries, in: Proceedings of the 17th International MSR Conference, 2020, pp. 508–512.

[69] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, Y. Chen, Vulnerability detection with code language models: How far are we?, in: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), Los Alamitos, CA, USA, 2025, pp. 469–481.

[70] V. Carletti, P. Foggia, A. Saggese, M. Vento, et al., Predicting source code vulnerabilities using deep learning: A fair comparison on real data, in: CEUR WORKSHOP, Vol. 3731, 2024, pp. 1–1.

[71] R. Jain, N. Gervasoni, M. Ndhlovu, S. Rawat, A code centric evaluation of c/c++ vulnerability datasets for deep learning based vulnerability detection techniques, in: Proceedings of the 16th Innovations in Software Engineering Conference, 2023, pp. 1–10.

[72] T. Zimmermann, N. Nagappan, L. Williams, Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista, in: 2010 Third international conference on software testing, verification and validation, IEEE, 2010, pp. 421–428.

[73] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, et al., Transformers: State-of-the-art natural language processing, in: Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations, 2020, pp. 38–45.

[74] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., Pytorch: An imperative style, high-performance deep learning library, Advances in neural information processing systems 32 (2019).

[75] I. Loshchilov, F. Hutter, Decoupled weight decay regularization, arXiv preprint arXiv:1711.05101 (2017).

[76] A. Mao, M. Mohri, Y. Zhong, Cross-entropy loss functions: Theoretical analysis and applications, in: International conference on Machine learning, PMLR, 2023, pp. 23803–23828.

[77] C.-Y. Lin, Rouge: A package for automatic evaluation of summaries, in: Text summarization branches out, 2004, pp. 74–81.

[78] C.-Y. Lin, F. J. Och, Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics, in: Proceedings of the 42nd annual meeting of the association for computational linguistics (ACL-04), 2004, pp. 605–612.

[79] Y.-M. Tamm, R. Damdinov, A. Vasilev, Quality metrics in recommender systems: Do we calculate metrics consistently?, in: Proceedings of the 15th ACM Conference on Recommender Systems, 2021, pp. 708–713.

[80] M. Sundararajan, A. Taly, Q. Yan, Axiomatic attribution for deep networks, in: International conference on machine learning, PMLR, 2017, pp. 3319–3328.

[81] K. Simonyan, A. Vedaldi, A. Zisserman, Deep inside convolutional networks: Visualising image classification models and saliency maps, arXiv preprint arXiv:1312.6034 (2013).

[82] M. Ancona, E. Ceolini, C. Öztireli, M. Gross, Towards better understanding of gradient-based attribution methods for deep neural networks, arXiv preprint arXiv:1711.06104 (2017).

[83] A. Shrikumar, P. Greenside, A. Kundaje, Learning important features through propagating activation differences, in: International conference on machine learning, PMlR, 2017, pp. 3145–3153.

[84] S. Lundberg, A unified approach to interpreting model predictions, arXiv preprint arXiv:1705.07874 (2017).

[85] LineVul, LineVul Replication Package, `https://github.com/awsm-research/LineVul` (2022).

[86] F. Wilcoxon, Individual comparisons by ranking methods, in: Breakthroughs in statistics: Methodology and distribution, Springer, 1992, pp. 196–202.

[87] W. U. Ahmad, S. Chakraborty, B. Ray, K.-W. Chang, Unified pre-training for program understanding and generation, arXiv preprint arXiv:2103.06333 (2021).

[88] OpenAI, Gpt-4 technical report, `https://cdn.openai.com/papers/gpt-4.pdf` (2023).

[89] F. Jiang, Identifying and mitigating vulnerabilities in llm-integrated applications, Master's thesis, University of Washington (2024).

[90] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al., Training language models to follow instructions with human feedback, Advances in neural information processing systems 35 (2022) 27730–27744.

[91] Z. Chen, Y. Deng, Y. Wu, Q. Gu, Y. Li, Towards understanding the mixture-of-experts layer in deep learning, Advances in neural information processing systems 35 (2022) 23049–23062.

[92] M. Siavvas, E. Gelenbe, D. Kehagias, D. Tzovaras, Static analysis-based approaches for secure software development, in: Security in Computer and Information Sciences: First International ISCIS Security Workshop 2018, Euro-CYBERSEC 2018, London, UK, February 26-27, 2018, Revised Selected Papers 1, Springer International Publishing, 2018, pp. 142–157.

[93] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 672–681.

[94] J. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, Algorithms for hyper-parameter optimization, Advances in neural information processing systems 24 (2011).