scientific reports

OPEN

Vulnerability detection in Java source code using a quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction

Shumaila Hussain[1,2]✉, Muhammad Nadeem[3], Junaid Baber[2,4], Mohammed Hamdi[5], Adel Rajab[5], Mana Saleh Al Reshan[6] & Asadullah Shaikh[6]

Software vulnerabilities pose a significant threat to system security, necessitating effective automatic detection methods. Current techniques face challenges such as dependency issues, language bias, and coarse detection granularity. This study presents a novel deep learning-based vulnerability detection system for Java code. Leveraging hybrid feature extraction through graph and sequence-based techniques enhances semantic and syntactic understanding. The system utilizes control flow graphs (CFG), abstract syntax trees (AST), program dependencies (PD), and greedy longest-match first vectorization for graph representation. A hybrid neural network (GCN-RFEMLP) and the pre-trained CodeBERT model extract features, feeding them into a quantum convolutional neural network with self-attentive pooling. The system addresses issues like long-term information dependency and coarse detection granularity, employing intermediate code representation and inter-procedural slice code. To mitigate language bias, a benchmark software assurance reference dataset is employed. Evaluations demonstrate the system's superiority, achieving 99.2% accuracy in detecting vulnerabilities, outperforming benchmark methods. The proposed approach comprehensively addresses vulnerabilities, including improper input validation, missing authorizations, buffer overflow, cross-site scripting, and SQL injection attacks listed by common weakness enumeration (CWE).

Keywords Vulnerability detection, Self-attentive QCNN, Feature extraction, Hybrid GCN, Software security, CodeBERT

The COVID-19 pandemic has dramatically intensified the use of computer applications, leading to an unprecedented increase in software vulnerabilities. According to the national vulnerability database (NVD), there were 20,158 reported vulnerabilities in 2021[1]. This exponential growth in security vulnerability is causing significant economic impacts and substantial financial losses[2–7].

Therefore, software vulnerability detection has become more crucial and challenging than ever. The need for generalized, scalable, accurate, fine-grained, and high-speed automatic vulnerability detection approaches is evident. Vulnerability typically stems from programming oversights, which the current detection tools, using either static or dynamic code analysis, often fail to address adequately. The analysis of code for security vulnerability without execution is the static code analysis technique, while in dynamic code analysis, the running application is tested for security vulnerability. Static code analysis techniques can be resource-intensive, while dynamic

[1]Department of Computer Science, Sardar Bahadur Khan Women's University, Quetta, Pakistan. [2]Department of Computer Science and IT, University of Balochistan, Quetta, Pakistan. [3]Higher Colleges of Technology, Abu Dhabi, United Arab Emirates. [4]GIPSA-Lab, University Grenoble Alpes, 38000 Grenoble, France. [5]Department of Computer Science, College of Computer Science and Information Systems, Najran University, 61441 Najran, Saudi Arabia. [6]Department of Information Systems, College of Computer Science and Information Systems, Najran University, 61441 Najran, Saudi Arabia. ✉email: shumaila.hussain@sbkwu.edu.pk

code analysis can increase execution time and negatively impact performance[8]. Both of these approaches are language-specific, rule-based, and dependent on the knowledge of the developers, making them prone to errors, biased, coarse-grained, and leading to unacceptably high false-negative rates.

Machine learning (ML) techniques have proven promising in vulnerability assessment[9–14]. The deep neural networks (DNNs) have demonstrated capabilities in learning source code patterns, excelling in syntax-level bug detection and pattern recognition[15–17]. However, existing deep learning (DL) solutions for vulnerability assessments have certain limitations; they primarily concentrate on the syntactic structure of code, neglecting its semantic information[18–20]. They target either a single file of source code or a small dataset or rely on application processing interface APIs to address the selected vulnerability. Furthermore, DL techniques often struggle to understand the value transfers within source codes due to a lack of semantic information, resulting in a high false-positive rate and less scalable approach[21–23].

The employed self-attentive quantum convolutional neural network along with deep learning techniques significantly improves the memory bottleneck issue, semantics understanding of code pattern and accelerated the performance. The proposed vulnerability detection system can detect a range of vulnerabilities, including improper input validation, SQL injection attacks, missing authorization, cross-site scripting, and buffer overflow attacks listed among the top 25 most impactful security vulnerabilities by common weaknesses enumeration (CWE). The CWE is a project of Mitre and is responsible for listing the software and hardware weakness types according to their impact to help prevent the vulnerability. This research paper contributes to the field of automatic vulnerability detection in several significant ways:

1. It develops a novel vulnerability detection system that implies efficient and accurate vulnerability detection using hybrid feature extraction by concatenating graph-based and sequence-based approaches coping with complex vulnerability patterns, enhancing vulnerability detection granularity, and reducing false-positive rates.
2. It proposes a hybrid graph neural network based on GCN-RFEMLP to overcome the absence of order information of nodes in the graph. Our fused wrapper method has reduced the dimension of features and removed irrelevant features to improve efficiency.
3. It introduces bimodal pre-trained CodeBERT model to implement fine-tuned feature extraction, reducing thereby, the semantic gap to improve vulnerability detection.
4. It analyzes the vulnerability detection dataset and balances the dataset to avoid overfitting, thereby improving the performance.
5. It employs the benchmark comprehensive software assurance reference dataset (SARD) for model training and testing, preprocessing the datasets to achieve optimized results. The proposed system is tested with five different datasets to ensure its performance, robustness, and validity.
6. It employs novel quantum convolutional neural network using self-attentive pooling to improve the computation, long-term dependencies, and memory bottleneck issues to classify the vulnerable code and type of vulnerability. To the best of our knowledge, QCNN-Self Attentive pooling is used for the first time to classify the vulnerabilities.
7. It proposes a novel framework for effective feature selection, contributing to a broader understanding of this field and suggesting a more balanced and effective approach to vulnerability detection across diverse types.

The remainder of this paper is structured as follows:

"Related work" section delves into a review of relevant literature. "Methodology" section outlines the methodology employed in this research. "Experiments and results" section details the experiment and results, including the experimental setup and derived results from the proposed method. "Conclusion" section offers the conclusions drawn from this study.

Related work

Manual source code auditing, involving a team of security experts, scrutinizing source code for vulnerability, is the most traditional approach to finding software vulnerability[24]. However, conventional software vulnerability analysis techniques often struggle to cope with real-time and ever-increasing software security vulnerability.

Vulnerability detection based on code analysis is trending and is classified into three main approaches: static, hybrid, and dynamic vulnerability detection[25]. Static analysis scrutinizes source code without execution, whereas dynamic analysis examines it through execution. The hybrid analysis combines the two. Many tools and techniques, such as code comparison, symbolic execution, and inference techniques, have been developed for static analysis. However, these techniques do not cover all existing vulnerabilities and are ill-equipped to analyze emerging security threats. Dynamic analysis techniques, including fuzzing and taint analysis, require substantial computational time and resources[26–31]. Furthermore, the performance and reliability of these methods are insufficient to meet current security challenges.

The surge in software vulnerability has driven researchers to devise better detection strategies. Software security researchers have begun leveraging machine learning's predictive power to address these security challenges. Machine learning techniques, whether supervised, unsupervised, or semi-supervised, are increasingly used for vulnerability detection. Among various machine learning approaches, supervised machine learning is widely adopted for software vulnerability detection. Figure 1 illustrates the supervised machine learning approaches for vulnerability assessment.

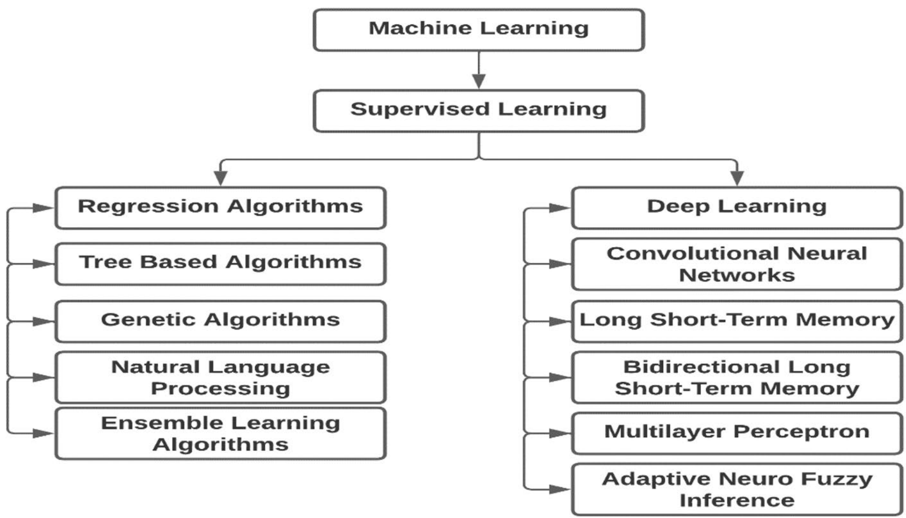

Figure 1. Machine learning techniques used for vulnerability assessments.

Code representation learning

The code must follow a specific format to implement machine learning techniques, categorized into three primary representation methods:

Sequence-based In this approach, data is divided into chunks, such as characters, tokens, or APIs, utilizing techniques like bag-of-words, n-gram, word2vec, etc. These techniques involve data preprocessing, tokenization, and the adoption of neural networks. However, they may lack long-term contextual code abstraction.

Tree-based This method employs a neural network structure on abstract syntax tree (AST)-based data representation. The tree is subdivided into small statements containing code snippets. Challenges include code fragment complexity and gradient vanishing.

Graph-based This approach represents code in a graph structure, primarily using a code property graph (CPG) composed of an abstract syntax tree (AST), control flow graph (CFG), context flow graph (XFG), and program dependency graph (PDG) for intermediate code representation[32]. While graph-based techniques can address long-term dependency issues, they require intensive computation.

In one of the related researches, a vulnerability analysis study used graph neural networks (GNN) and circle-gated graph neural networks to detect the vulnerable code[33,34]. In another study, the researchers used a flow graph for source code representation, performed vectorization through word2vec, and applied the graph neural network method to identify the vulnerability[35–37]. The software vulnerability detector named DeepVulSeeker used a pre-trained model to convert natural language descriptions to programming code. Another research study in context used intermediate code representation by applying AST, CFG, and DFG and deployed a pre-trained model, while CNN and FNN neural networks were used to classify the vulnerability[38,39]. The abstract syntax tree neural networks[40,41] and self-attentive deep neural network coupled with text mining were also tried[42]. Similarly, ChatGPT involves human interaction to identify vulnerabilities and recommend fixes[43].

Another study explored regression trees for vulnerability detection[44]. Similarly, a hybrid approach using deep learning-based lightweight-assisted vulnerability was used in a study pertaining to the same, while another research used minimum intermediate representation learning[45,46]. The researchers exploit program slicing and binary gated recurrent unit (BGRU) in a similar nature of study, while code slicing using code metrics as features is used to detect vulnerabilities related to pointer usage[47,48]. Other studies implemented deep learning techniques like CNN and others, along with feature selection, for detecting SQL and cross-site scripting vulnerability[48–51]. Yet another study proposed a model based on source feature learning and classification[52]. It has been observed that feature selection is frequently studied alongside machine learning approaches for vulnerability detection[53–55].

Two similar studies used word2vec and LSTM to identify code with cross-site scripting, SQL injection, cross-site forgery, and open redirect vulnerability[56,57]. The recurring neural network model called BiLSTM is used to focus on buffer errors and resource management vulnerability detection[58]. Similarly, BiLSTM and taint analysis performed well in one of the research pursuits conducted in the same context[59]. Techniques like CNN, long-short-term memory (LSTM), and directed graphs were used for vulnerability detection[60].

One of the related studies in this regard compared the Random forest, CNN, and RNN techniques to benchmark vulnerability detection[61]. Similarly, the GNN-based model outperformed for vulnerability detection[62,63]. Another study presented a comparative analysis using Naïve Bayes, decision trees, SVM, k-nearest neighbor, and RF to evaluate software vulnerability detection performance[64–67]. Yet another study focusing on SVM, multinomial Naïve Bayes classifiers, and bidirectional encoders based on BERT transfer learning concluded that BERT outperformed other methods in detecting vulnerability[68]. Notably, none of the studies reviewed considered the semantic similarity of code, prominent the gap in the deep learning techniques used for vulnerability detection. In contrast, our work extracts the semantic similarity of the code, enhancing system performance, as further detailed in the results section.

Improper input validation, a major cause of security vulnerability in computing applications, can trigger SQL injection attacks, missing authorization, cross-site scripting (XSS) attacks, and buffer overflows. The Common Weakness Enumeration (CWE) project of the Mitre organization, a comprehensive dictionary of software weaknesses, ranked input validation as the fourth most frequently occurring and dangerous security vulnerability in 2021[69,70]. Therefore, we selected improper input validation, cross-site scripting, buffer overflow, missing authorization, and SQL injection vulnerability ranked among the top 25 most impactfull and dangerous security vulnerabilities listed by CWE for evaluating our proposed system. Table 1 below shows some vulnerability detection techniques commonly used to analyze the selected vulnerability.

Methodology

This section describes our proposed system for vulnerability detection, which introduces fused feature extraction that leverages semantic and syntax understanding of code for a nuanced vulnerability assessment.

Framework of proposed vulnerability detection system

Code auditing is performed predominantly on C/C+ languages, while there is always a space for Java code auditing due to deficient code auditing techniques quantified for this language. Our system aims to automatically detect software vulnerability from Java code using DL, considering syntactic structure and code semantics, focusing on fine-grained vulnerability detection. Given that existing DL techniques often overlook the semantic relationships in code, our system is designed to fill this gap and improve the false-positive rate. The proposed system uses a novel mechanism based on hybrid feature extraction that concatenates sequence-based and graph-based feature extraction and detects the vulnerability using deep learning.

The proposed methodology is depicted in Fig. 2 given below. The proposed scheme is divided into three parts (1) Intermediate input representation (2) Hybrid feature extraction, and (3) Classification. The first step comprises a standard dataset converted into source code representation using code property graph and tokenization to get it presentable to leverage machine learning techniques. In the second step, the hybrid feature extraction is applied. The graph feature extraction used along with sequence-based feature extraction leverages the semantic and syntax structure of code. The extracted features are concatenated, and a quantum convolutional neural network with self-attentive pooling is employed to detect selected vulnerabilities.

The selected vulnerabilities are listed among the most impactful according to CWE and include improper input validation, SQL injection vulnerability, missing authorization, cross-site scripting, and buffer overflow. The system detects vulnerable functions and types of vulnerability.

Vulnerability	Approach
Cross-site scripting XSS attacks	1. Cross-site scripting attack XSS detection using a modified CNN model[71] 2. Automated server-side XSS attack detection using boundary injection[72] 3. Taint tracking-based analysis of DOM cross-site scripting named as TT-XSS[73] 4. Support Vector Machine is used to detect blind cross-site scripting vulnerability[74] 5. Reducing attack surfaces for cross-site scripting attacks using secure SDLC[75] 6. Detecting cross-site scripting vulnerability using LSTM and recurrent neural networks (RNN) named DeepXSS[76] 7. Using genetic algorithms and reinforcement learning for XSS attack detection[77] 8. Using ML with hybrid features for XSS attack detection[78] 9. Using Fuzzy inference for dynamic detection of XSS cross-site scripting attacks[79]
Buffer overflow attacks	1. Analyzing network intrusion for buffer overflow attacks[80] 2. Implementing string library function to detect integer overflow-to-buffer overflow attacks[81] 3. Performed static buffer overflow detection and suggested automatic detection[82] 4. Static buffer overflow detection and repair using the BovInspector tool[83]
SQL injection attacks	1. SQL injection attacks detection using a decision tree[84] 2. Using behavior and response analysis for SQL injection attacks[85] 3. SQL injection attack detection in web applications using heuristic-based analysis[86] 4. Applying neuro-fuzzy techniques to prevent and detect SQL injection attacks[87] 5. Algorithm designed for black box testing to mitigate SQL injection vulnerability[88] 6. A traffic-based technique called DIAVA to detect data leakages and SQL injection attacks[89] 7. A hybrid method consists of augmenting database tables with symbols, then using an algorithm for queries and another algorithm designed for string matching to prevent and detect SQL injection attacks[90] 8. Using intrusion set randomization to detect SQL injection attacks[91] 9. A tool is developed to detect SQL injection attacks and display suggestions to fix them[92]
Missing authorization	1. The tool is developed to detect missing authorization in distributed cloud systems using inferring variable definition, user-owned data, and critical system state[93] 2. The proposed role cast SE-based technique consists of the context of security-sensitive events that are control-dependent on roles[94] 3. The Vanguard is an approach consisting of static analysis for sensitive operations, analyzing sustainability using taint analysis, and the existence of risk degree of missing authorization[95] 4. VRust is proposed to analyze vulnerability, including missing authorization for Solana, by assigning validation rules for vulnerable input accounts[96] 5. The CRIX system consists of interprocedural, semantic, and context-aware systems[97] 6. MACE is based on checking the authorization state consistency[98]

Table 1. Commonly used techniques for vulnerability detection.

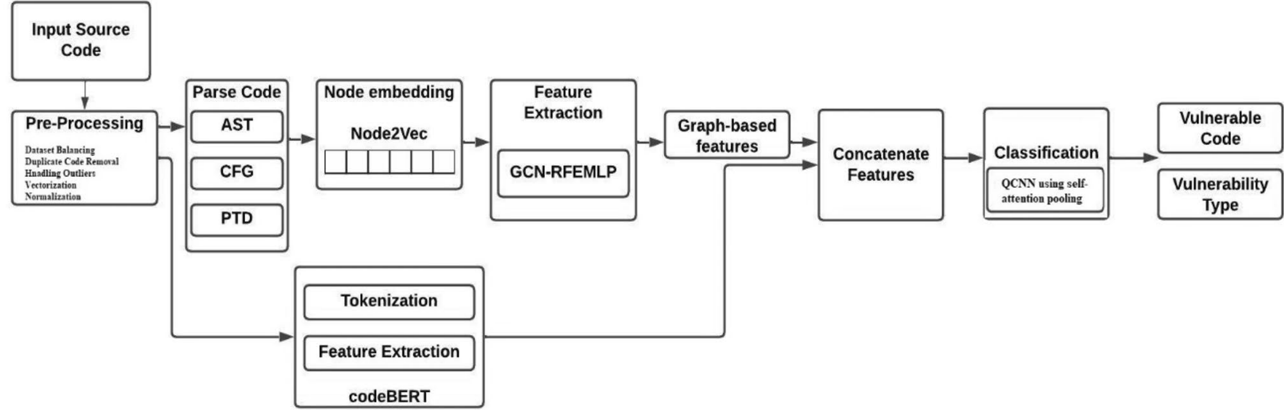

Figure 2. The framework of the proposed vulnerability detection system.

Dataset/data acquisition

To train our proposed system, we have used the Software Assurance Reference Dataset (SARD) benchmark dataset, which contains hundreds of thousands of source code programs with known vulnerabilities. This dataset includes 42,212 files comprising 29,258 safe samples and 12,954 unsafe samples of source code, covering 150 classes of bugs or weaknesses listed by CWE[99–104]. For our study, we have selected 46,447 Java programs from SARD, including vulnerabilities related to SQL injection attacks, missing authorization, cross-site scripting, improper input validation, and buffer overflow. The proposed system is validated using other benchmark datasets, including Juliet java 1.3[105–107], FUNDED, Vul4j, CVEfixes, and CodeXGLUE.

Dataset preprocessing

Data preprocessing involves several essential steps.

a. Dataset balancing.
   Addressing dataset imbalance is crucial for the optimal performance of machine learning algorithms. The benchmark dataset for vulnerability detection often exhibits a significant disparity between vulnerable and clean codes. Achieving a balanced dataset is vital for accurate and efficient algorithm performance, helping reduce false positive ratios. Additionally, missing values are appropriately handled.
b. Duplicate code removal
   Removing duplicate code enhances performance, reduces complexity, and minimizes execution time. Decision trees are employed for the efficient removal of duplicate code and code clones.
c. Handling outliers
   Organizing the dataset is essential for improved performance. Outliers are detected and effectively handled using log transformation, contributing to dataset normalization.
d. Vectorization
   Textual data is transformed into numerical form through vectorization, ensuring uniform scaling and enhancing algorithm performance.
e. Normalizing
   Further normalization of the dataset ensures consistent scaling without compromising range differences. Data normalization equalizes the impact of each feature, addressing potential accuracy issues arising from inherently large values. The Z-Scaling technique is employed for data normalization, converting text-based datasets into integers.

Graphical feature extraction

Intermediate code representation

We have applied the classical code property graph (CPG) for graphical code representation, which is a combination of abstract syntax tree (AST), control flow graph (CFG), and program dependency graph. It helps analyze the syntactic structure and code semantics. It is important to convert the code into intermediate code representation to remove the pointless points and reduce the dependencies.

a. Abstract Syntax Tree (AST)
   The AST is used to parse the syntactic structure of code effectively. The abstract syntax tree comprises a root node that holds functions, branches of statements, declarations, predictions, and expressions while the leaf nodes represent the operators, identifiers, and keywords.
b. Control Flow Graph (CFG)
   The CFG represents the order of code execution. It expounds statements and conditions that need to be met for the execution of code branches. The nodes in the CFG indicate the statements, while the edges denote the transfer of control.
c. Program Dependency Graph (PDG)

It describes the control and data dependencies in the function. The data dependency edge holds the declared variable to be used later, while the control dependency edges denote the impact of predicates on variables.

Node embedding
Node embedding aims to reduce the nodes' properties in smaller dimension vectors. The outcome of node embedding is fed as input to downstream machine learning-based processing techniques. Flexibility in exploring neighborhoods in node2vec has been observed to provide a richer representation. The rich structural information improves the ability of features to imply nonlinear information. Therefore, the node2vec is used for node embedding with random walk using skip-gram with negative sampling technique to maximize the probability of preserving the neighborhood of nodes. The node2vec is a second-order Markov chain. It implements random walk on graphs to extract the context pair using bootstrapping approach and use them for training the word2vec model. It transforms graphs to numerical representation while preserving the structure of the network in a way that the close nodes remain close in embedding. The structure of node2vec is given in Fig. 3.

Feature extraction
We have employed hybrid graph neural network GCN-RFEMLP based on graph convolutional neural network (GCN) and multilayer perceptron fused with recursive feature elimination wrapper. The GCN lacks feature similarity, which can create noise. We, therefore, have concatenated RFEMLP with GCN to overcome this issue. The graph convolutional neural network is designed to deal with graph structure data. It implements a message-passing technique where the embedding information of a node is updated based on the neighboring node. The node embedding is converted into graph embedding, serving as input to a fully connected classifier. We have added a bi-affine layer in GCN to achieve better dependency parsing and preserve code semantics. The structural composition of graph convolutional neural networks is illustrated in Fig. 4 given below.

We used an MLP neural network with a rectified linear activation function, ReLu, on the hidden layer and a Softmax activation function on the output layer. The generalized formula for ReLu is depicted in Eq. (1).

$$O = WA + B \tag{1}$$

where O is the output before applying the activation function, W represents the weights, A represents the input to the layer, and B represents the bias.

The Fig. 5 illustrates structural composition of MLP network. We have used adam, adadelta, momentum, and stochastic gradient descent (SDG) optimizer along with loss functions mean square error (MSE) and mean absolute error (MAE) to select the best fit. We have paired each optimizer with a loss function to get the results.

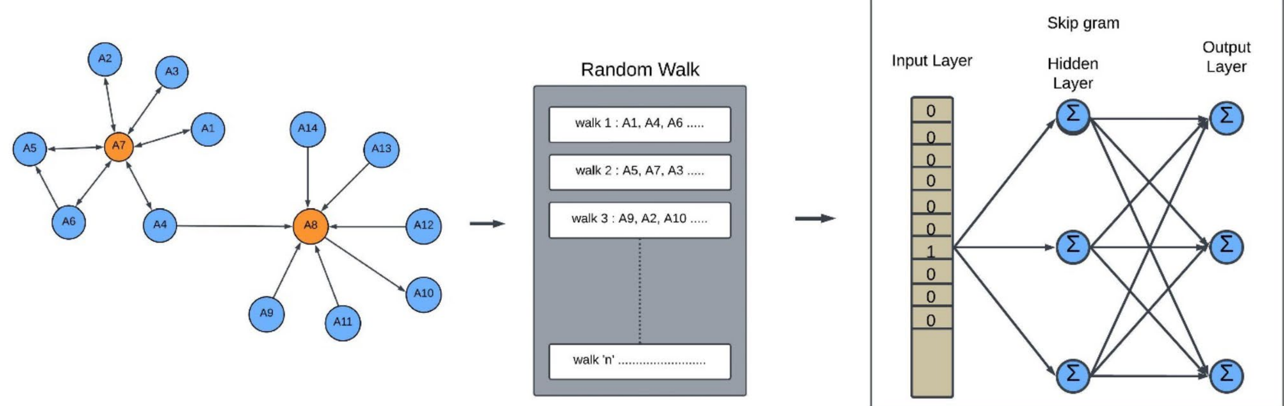

Figure 3. Structure of Node2Vec using random walk and skip-gram.

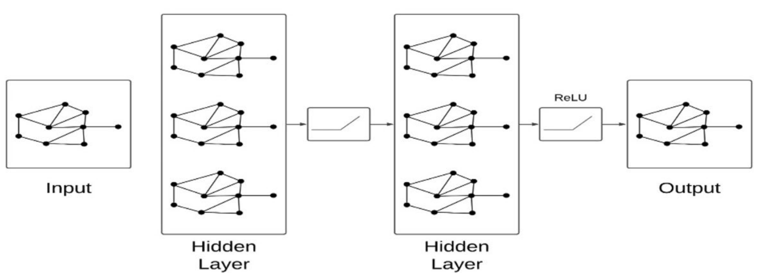

Figure 4. Structure of graph convolutional neural network.

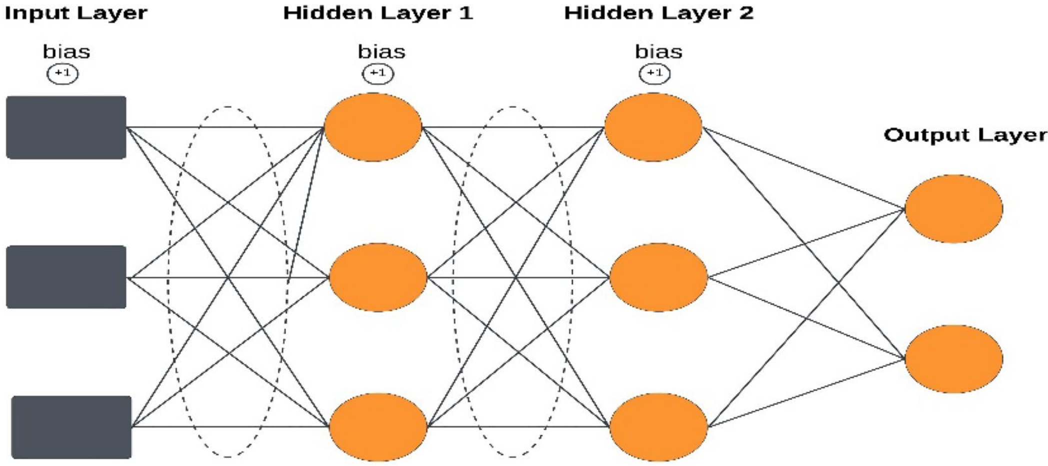

Figure 5. Structure of multilayer perceptron neural network model (MLP).

The selections given below show the combination of each optimizer and loss function. Selection 1 shows the combination of the adam optimizer with the MSE loss function similarly; Selection 2 shows the combination of the adam optimizer with the MAE loss function, and so on.

Table 2 depicts different compositions of optimizers and loss functions. The results obtained from each selection are compared to implement the best combination of optimizer and loss function to improve the system's accuracy. We have conducted experiments to acquire the optimal combination with minimal loss to improve the algorithm's performance. The loss function enumerates the difference between the actual value and the predicted value. The selection 3 and selection 7 showed improved results. We, therefore, have selected selection 7 to use with MLP to boost the performance. Moreover, the model training contains regulating the parameters, hyper-parameter tuning, CommitCount functions, setting bias, optimizers, loss functions, and weights to reduce false positive rate. The fine-tuned model detects the vulnerability. The specified learning rate set in the proposed model is 0.0005 on 300 epochs, neurons = 128, early stopping = 30, and batch size = 64. The RFEMLP imposes a machine learning-based wrapper technique called recursive feature elimination (RFE) on a multilayer perceptron neural network. The RFE keeps on eliminating the irrelevant feature on each iteration until it reaches the most impactful features. The RFE reduces the redundant features to improve efficiency. We have implemented a decision tree classifier for RFE. Based on the aggregate difference between the features space, we have set the ranking of features from the most important to the least important.

No	Combination
Selection 1	Adam + MSE
Selection 2	Adam + MAE
Selection 3	AdaDelta + MSE
Selection 4	AdaDelta + MAE
Selection 5	Momentum + MSE
Selection 6	Momentum + MAE
Selection 7	SDG + MSE
Selection 8	SDG + MAE

Table 2. Different combinations of optimizers and loss functions.

```
Input: Total selected features by GCN
Process:
Function FeatureSelection(MLP_features):
  total_selected_features = 0
  For each stf in MLP_features:    // stf is the total selected features by MLP
    If stf > 0:
      For each f1 in stf:
        TempM.append(f1.[a])    // Assign to a random temporary memory location
        Score = eliminate(train, validate, TempM)    // Evaluate feature's score for elimination
        FX[f1[a]] = Score    // Store evaluated feature performance
        FMax = getFMax(FX)    // Identify feature with maximum performance
        a.remove(FMax)    // Remove feature with maximum performance from consideration
        If FX[FMax] > SetMax:    // Compare evaluated feature performance with maximum performance value, SetMax is
        the maximum performance value
          SetMax = FX[FMax]    // Update maximum performance value
          A = f1    // Store the selected feature
          Increment counter
        Else
          Decrement counter
      End For
    Else:
      Return f1
  End For
  Return Total selected features by RFEMLP
```

Algorithm 1 Feature selection using RFEMLP

Sequence-based feature extraction

CodeBERT

The pre-trained models are effective in vulnerability prediction[108,109]. The CodeBERT combines bidirectional encoder representation from transformers and optimized BERT called RoBERTa[110]. The BERT is a self-supervised model that utilizes the characteristics of mask-based goals and a transformer-based architecture. The CodeBERT is the only large bimodal pre-trained model using natural and programming languages[111]. It effectively analyzes the semantic connections between programming language and eliminates the long-range dependency in code. Moreover, the multi-head attention mechanism of transformers effectively analyzes multiple key variables of data flow.

The Fig. 6 illustrates the architecture of the CodeBERT model. In the first step, the CodeBERT takes code input and tokenizes the code. We have implemented the greedy longest match first algorithm for tokenizing. In the second step, the tokens are used to extract the features. To perform feature extraction, we have fine-tuned

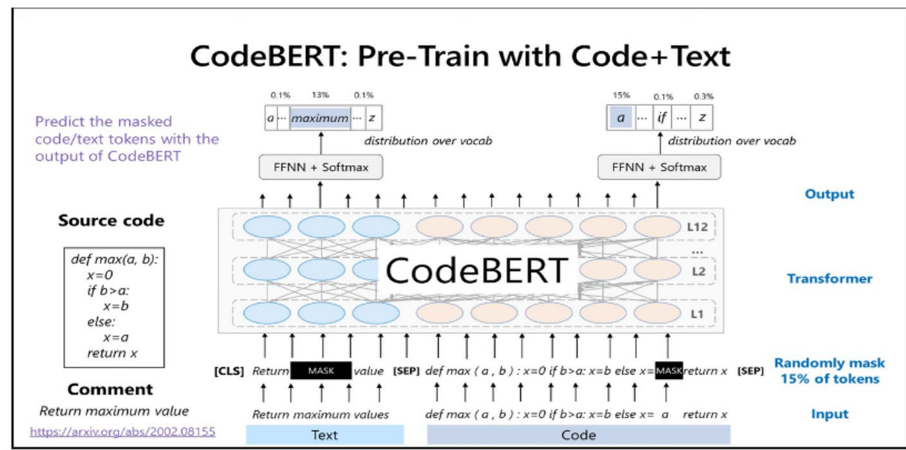

Figure 6. Structure of CodeBERT model.

the CodeBERT by setting the batch size to 32, the learning rate of 10^{-3}, 50 epoch size, and used early stopping to avoid overfitting.

Classification

Quantum convolutional neural network with self-attentive pooling

The software Java source code has a complex lexical structure, and intricate syntactic and semantic features with longer length which is difficult to tackle. Moreover, the large and complex software can create computational and memory bottleneck issues while dealing with vulnerability detection. We have, therefore, employed a quantum neural network to overcome these issues with quantum mechanisms. The quantum mechanism is based on quantum entanglement and quantum superposition states. Quantum neural networks are embedding entanglement and quantum superposition states to improve the accuracy of neural networks. It utilizes the quantum bit, interference, superposion, and entanglement mechanism for information processing. The q-bit is a state vector depicted in the equation below

$$|\Psi\rangle = \theta|0\rangle + \delta|1\rangle \tag{2}$$

where θ and δ are the probability amplitudes that are represented by complex numbers and $|\theta^2| + |\delta^2| = 1$. The quantum mechanism implies that any unitary matrix is a quantum gate U given below in Eq. (3).

$$\mathrm{UU\dagger} = \mathrm{U\dagger U} = \mathrm{I} \tag{3}$$

where U† is the conjugate transpose of a matrix U, and I is an identity matrix. There are three qubit gates 1. one qubit gate, which is a square root of NOT gate, also known as Pauli gates 2. two qubit gate which work on 4×4 unitary matrices; and 3. multiple-qubit gates which work on multiple qubits as $2n \times 2n$ unitary matrices. The quantum mechanism resolved memory issues in huge computations and structural bottleneck issues and attained higher computing capabilities than classical computing.

The quantum convolutional neural network provides a promising machine learning paradigm. We have used a quantum pennyLane device to mimic the four-qubit device. The RY gate is responsible for converting the code into quantum bits. The quantum convolutional layer works as the conventional convolutional layer in the CNN model using a quantum computing mechanism. Quantum convolution works as small random quantum circuits (RQCs) to calculate convolution operation. It consists of three phases: encoding, RQC, and decoding. The RQC is applied to the convolutional layer and pooling layer. The encoding layer is responsible for converting the extracted features in classical form into a high-dimensional quantum bit state. We have applied basis encoding to convert the data into qubits. The concatenated features are converted into binary features and then into a quantum state. The embedded quantum state is the bit-wise conversion of binary string into a quantum subsystem; thus, the source code is transformed into the quantum bit. The paddle library in Python is used for basis encoding.

In the second layer, RQC is applied at a convolutional layer that uses multiple qubit gates among the adjacent qubit. Similarly, the qubit gates applied on pooling reduce the size of the quantum system. We have applied a self-attention mechanism on the pooling layer to improve the system's performance. The fully connected circuit is responsible for decoding and classifying the vulnerable code and the type of vulnerability identified. The QCNN uses multiscale entanglement MERA in the reverse direction and repeats until sufficiently reduces the size of the quantum system.

We have applied a novel pooling technique using a multi-head self-attention mechanism to improve the computation and memory footprints, thus improving the model's performance. The proposed self-attention mechanism comprised tokenization, multihead self-attention, spatial channel restoration, and sigmoid and soft max activation functions applied on the pooling layer to make it self-attentive. The input features are tokenized, and multi-head self-attention manages the long-term dependencies in the tokens, while the spatial channel restoration helps in decoding and restoring the tokens to self-attention maps. The activation function softmax rectifies the self-attention maps. Adding a self-attention mechanism in QCNN further improves the memory footprints and computation. The quantum convolutional neural network classifies the vulnerable code and identifies the vulnerability type.

The Fig. 7 above illustrates the overall structure of self-attentive QCNN model proposed to identify the security vulnerability and type of vulnerability.

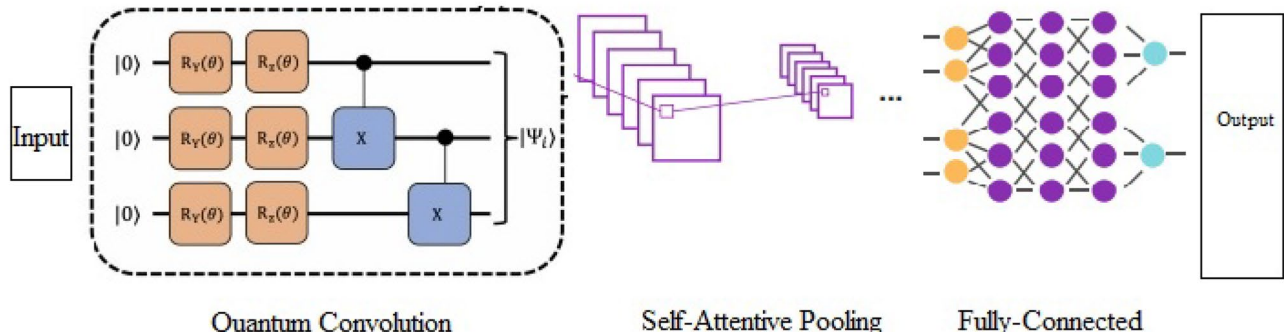

Figure 7. Structural composition of quantum neural network with self-attentive pooling.

```
1. Input_dataset = load_dataset()      // Input dataset with Java/JavaScript Code (i.e. SARD)
2. Balanced_dataset = balance_dataset(input_dataset)      // Preprocess the data (dataset balancing, duplicate removal, handling outliers, missing values)
deduplicated_dataset = remove_duplicates(balanced_dataset)
processed_dataset = handle_outliers_missing_values(deduplicated_dataset)
3. Vectorized_data = vectorize_data(processed_dataset)      // Vectorize the data by converting string data into integers
4. Normalized_features = z_score_normalization(vectorized_data)      // Normalize feature scaling using z-score
5. Intermediate_representation = generate_intermediate_representation(normalized_features)      // Perform graph-based feature extraction
        node_embeddings = apply_node_embedding(intermediate_representation)
        extracted_features = perform_feature_extraction(node_embeddings)
6. Sequence_features = extract_sequence_features(normalized_features, CodeBERT_model)      // Employ sequence-based feature extraction using a pre-trained model, CodeBERT
7. Concatenated_features = concatenate(sequence_features, extracted_features)      // Concatenate the sequence-based and graph-based features
8. Encoded_qubits = encode_code_to_qubits(concatenated_features)      // Apply quantum convolutional neural network with self-attentive pooling
        rqc_output = apply_random_quantum_circuit(encoded_qubits)
        self_attentive_output = self_attentive_pooling(rqc_output)
        decoded_output = decode_output(self_attentive_output)
9. Vulnerable_code, vulnerability_type = identify_vulnerability(decoded_output)
produce_output(vulnerable_code, vulnerability_type)      //Produce output as vulnerable code and type of vulnerability
```

Algorithm 2 Composition of the proposed vulnerability detection system

Experiments and results

Experimental setup

The proposed automatic vulnerability detection system was evaluated via numerous experiments on a Windows-based computer equipped with an Intel® Core™ i7-10700H processor and 128 GB of RAM. The model is implemented using Python and Tensorflow framework using library packages like Keras, NumPy, sci-kit-learn, and Pandas. The hyper-parameters are set as epoch = 50, learning rate = 0.005, momentum = 0.9, dropout rate = 0.3, loss = cross-entropy.

Performance metrics

We assessed the performance of the proposed system using various metrics, including recall, precision, and accuracy. Accuracy was calculated according to Eq. (4).

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (4)$$

In this equation, TN stands for true negative, TP for true positive, FP for false positive, and FN for false negative. Additional metrics employed for performance validation were precision (see Eq. 5), which represents the fraction of correct positive predictions, and recall (see Eq. 6), which indicates the ratio of correct positive predictions with all positive predictions.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (5)$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (6)$$

Comparative analysis

The proposed system is developed to effectively predict the software systems' security vulnerability. To analyze the performance of the proposed system, it underwent testing on source code to identify potential security vulnerabilities.

The Table 3 compares our technique with other deep learning techniques like CNN, SVM, GNN, LSTM, BiLSTM, ANN, MLP, DNN, and FFDNN. The proposed model displayed superior accuracy, precision, and recall, suggesting its enhanced effectiveness in detecting maximum security vulnerability.

The research focused on different types of vulnerability, each possessing unique semantic features. The proposed system underwent training with the balanced SARD dataset containing synthesized data, making it universally applicable to various vulnerability types. To effectively assess the validity and performance of our system, the system was trained using other datasets, including Juliet Java 1.3, FUNDED, Vul4J, and CVEfixes. The SARD and Juliet java 1.3 are benchmark datasets made public by NIST.

The Table 4 depicts that the proposed system performed well with the other datasets FUNDED, Vul4j, CVEfixes, CodeXGLUE, SARD, VUDDY, and Julia jave 1.3, which proves the proposed system's validity.

In Table 5 our proposed model is compared with the commercial vulnerability detection tools VulDeepecker, SQVDT, Exp-Gen, PreNNsem, ISVSF, VULDEFF, SedSVD, VulANalyZeR, FUNDED, GraphSPD, BiTCN_DRSN, and VERI. The proposed system outperformed in accuracy, precision, and recall rates.

The Fig. 8 shows the proposed system's training and test accuracy. Data underscores the superior performance of our system, achieved by integrating hybrid feature extraction with syntax and semantic information of the

Classifier	Accuracy	F1
CNN[112]	0.92	0.92
SVM[113]	0.96	0.95
GNN[114]	0.95	–
LSTM[115]	0.96	0.96
BiLSTM[116]	0.96	–
ANN[117]	0.98	0.98
MLP[118]	0.84	–
DNN[119]	0.82	–
FFDNN[120]	0.77	–
Proposed model	0.99	0.97

Table 3. Comparative analysis with machine learning techniques.

Dataset	Precision	Recall	F1 score
FUNDED	0.95	0.92	0.91
Vul4J	0.96	0.96	0.95
CVEfixes	0.98	0.95	0.93
CodeXGLUE	0.96	0.98	0.99
SARD	0.98	0.97	0.95
VUDDY	0.92	0.97	0.97
Juliet java 1.3	0.98	0.95	0.96

Table 4. Performance evaluation of the proposed vulnerability detector using well-known datasets.

	Accuracy	F1 Score	Precision	Recall
VulDeepecker[121]	0.95	0.93	0.92	0.94
SQVDT[122]	0.97	0.95	0.94	0.96
Exp-Gen[123]	–	–	0.95	–
PreNNsem[124]	0.96	0.97	0.96	0.98
ISVSF[125]	0.95	0.90	–	–
MFXSS[126]	0.98	0.98	0.97	0.96
VULDEFF[127]	–	0.88	0.91	0.85
SedSVD[128]	0.91	0.95	–	–
VulANalyZeR[129]	0.89	0.90	0.85	0.95
FUNDED[130]	0.92	–	–	0.94
GraphSPD[131]	0.80	–	–	–
BiTCN_DRSN[132]	0.95	0.95	0.92	0.98
VERI[133]	0.92	0.93	0.94	0.91
Proposed model	0.99	0.97	0.98	0.96

Table 5. Comparative analysis with existing vulnerability detector.

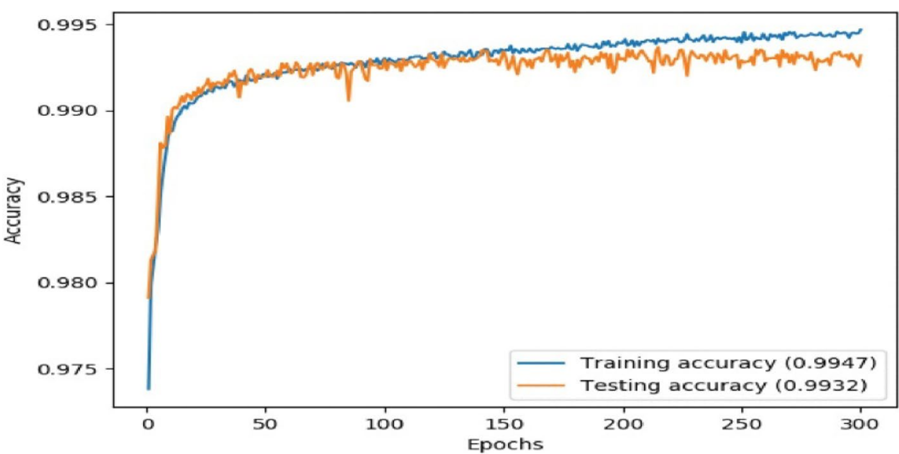

Figure 8. The training and test accuracy of the proposed system.

code. Notably, our system successfully reduced the false-positive rate while ensuring a minimum number of missing values.

Conclusion

This study proposes an innovative system designed to analyze vulnerability in software code, aiming to address limitations found in previous deep learning techniques. The vulnerability detection methods have fallen short in considering code semantics, leading to suboptimal performance. Our proposed system, combining graph-based feature extraction and sequence-based feature extraction with a proposed novel GCN-RFEMLP neural network, pre-trained model CodeBERT, and QCNN-self-attentive pooling, successfully audits source code for any potential security vulnerabilities. We leverage intermediate code representation, using a code property graph (CPG) for graphical code representation, consisting of an abstract syntax tree (AST), control flow graph (CFG), and program dependency graph.

The dataset is preprocessed considering the importance of data balancing, duplicate code removal, missing values, handling outliers, vectorization, and normalization for robustness, efficiency, and computational speed. Moreover, a quantum convolutional neural network with self-attentive pooling is used as a classifier. Our research concentrates on specific types of vulnerability: improper input validation, cross-site scripting (XSS), missing authorization, integer overflow, and SQL injection, which are listed among the top 25 most significant software security vulnerabilities in the common weakness enumeration (CWE). The Software Assurance Reference Dataset (SARD), a benchmark dataset, was employed to train our model. Furthermore, to prove the system's validity, the proposed system is used with other benchmark datasets, including FUNDED, Vul4j, CVEfixes, CodeXGLUE, SARD, VUDDY, and Juliet Java 1.3.

To validate the efficiency of our system, we compared its performance against not only prevalent deep learning approaches like CNN, SVM, GNN, LSTM, BiLSTM, ANN, MLP, DNN and FFDNN but also other available

systems such as VulDeepecker, SQVDT, Exp-Gen, PreNNsem, ISVSF, VULDEFF, SedSVD, VulANalyZeR, FUNDED, GraphSPD, BiTCN_DRSN, and VERI. The results from our experiments demonstrate the superior performance of our proposed system across various metrics, signifying a promising advancement in the field of automatic vulnerability detection.

Future directions

The proposed security vulnerability detection system, with its efficient feature extraction and quantum mechanism, including self-attentive pooling, successfully addresses existing issues in vulnerability detection in Java source code. While the system is tailored for the structural complexities of Java source code, extending the proposed mechanism to other programming languages is a crucial future direction to assess its effectiveness across diverse codebases. Additionally, exploring the applicability of the proposed system in resolving natural language processing (NLP) tasks holds promise for mitigating time, cost, and memory bottleneck issues in broader contexts.

Data availability

The datasets generated and/or analysed during the current study are available in the Github repository using the link https://github.com/Vul-Detect-Code/Vul-Detect.

Received: 19 October 2023; Accepted: 12 March 2024
Published online: 28 March 2024

References

1. CVSS Security Distribution Over Time. (2023) https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time.
2. Tassey, G. *The Economic Impact of Inadequate Infrastructure for Software Testing* (RTI Health, Social, and Economics Research, 2002).
3. Zhivich, M. & Cunningham, R. K. The real cost of software errors. *IEEE Secur. Priv.* **7**(2), 87–90 (2009).
4. Starsbug, J. & Bunge, J. Loss swamps trading firm. *Wall Street J* **8**(2). 1–15 (2012).
5. Geppert, L. Lost radio contact leaves pilots on their own. *IEEE Spectrum* **41**(11), 16–17 (2004).
6. Berr, J. wannacry-ransomware-attacks-wannacry-virus-losses. https://www.cbsnews.com/news/wannacry-ransomware-attacks-wannacry-virus-losses/ (Accessed 2022).
7. Chen, Y., Chen, J., Gao, Y., Chen, D. & Tang, Y. Research on software failure analysis and quality management model. In *IEEE International Conference on Software Quality, Reliability and Security Companion, Lisbon, Portugal* (2018).
8. Marjanov, T., Pashchenko, I. & Massacci, F. Machine learning for source code vulnerability detection: What works and what isn't there yet. *IEEE Secur. Priv.* **20**, 60–76 (2022).
9. Wang, X. *et al.* Federated deep learning for anomaly detection in the internet of things. *Comput. Electr. Eng.* **108**, 108651 (2023).
10. Srivastava, A. & Bharti, M. R. Hybrid machine learning model for anomaly detection in unlabelled data of wireless sensor networks. *Wirel. Pers. Commun.* **129**, 2693–2710 (2023).
11. Gao, Y., Yin, X., He, Z. & Wang, X. A deep learning process anomaly detection approach with representative latent features for low discriminative and insufficient abnormal data. *Comput. Ind. Eng.* **176**, 108936 (2023).
12. Tekerek, A. A novel architecture for web-based attack detection using convolutional neural network. *Comput. Secur.* **100**, 102096 (2021).
13. Gupta, R., Patel, M. M., Shukla, A. & Tanwar, S. Deep learning-based malicious smart contract detection scheme for internet of things environment. *Comput. Electr. Eng.* **97**, 107583 (2022).
14. Dairi, A., Harrou, F., Bouyeddou, B., Senouci, S.-M. & Sun, Y. Semi-supervised deep learning-driven anomaly detection schemes for cyber-attack detection in smart grids. In *Power System Cybersecurity*, 265–295 (2023).
15. Lam, A. N., Nguyen, A. T., Nguyen, H. A. & Nguyen, T. N. Combining deep learning with information retrieval to localize buggy files for bug reports (N). In *IEEE International Conference on Automated Software Engineering* (ASE), Lincoln, NE, USA (2015).
16. Pu, Y., Narasimhan, K., Solar-Lezama, A. & Barzilay, R. sk_p: a neural program corrector for MOOCs. In *SPLASH Companion 2016: Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, New York,* (2016).
17. White, M., Vendome, C., Linares-Vásquez, M. & Poshyvanyk, D. Toward deep learning software repositories. In *MSR '15: Proceedings of the 12th Working Conference on Mining Software Repositories* (2015).
18. Scandariato, R., Walden, J., Hovsepyan, A. & Joosen, W. Predicting vulnerable software components via text mining. *IEEE Trans. Softw. Eng.* **40**(10), 993–1006 (2014).
19. Morrison, P., Herzig, K., Murphy, B. & Williams, L. Challenges with applying vulnerability prediction models. In *HotSoS '15: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security* (2015).
20. Dam, H. K. *et al.* Automatic feature learning for predicting vulnerable software components. *IEEE Trans. Softw. Eng.* **47**(1), 67–85 (2018).
21. Choi, M.-J., Jeong, S., Oh, H. & Choo, J. End-to-end prediction of buffer overruns from raw source code via neural memory networks. In *IJCAI'17: Proceedings of the 26th International Joint Conference on Artificial Intelligence, Melbourne, Australia* (2017).
22. Pang, Y., Xue, X. & Namin, A. S. Predicting vulnerable software components through N-gram analysis and statistical feature selection. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)* (2015).
23. Hovsepyan, A., Scandariato, R., Joosen, W. & Walden, J. Software vulnerability prediction using text analysis techniques. In *MetriSec '12: Proceedings of the 4th International Workshop on Security Measurements and Metrics*, *New York* (2012).
24. Piantadosi, V., Scalabrino, S. & Oli, R. Fixing of security vulnerabilities in open source projects: A case study of Apache HTTP server and Apache tomcat. In *International Conference on Software Testing, Verification, and Validation, ICST*, (2019).
25. Hanif, H., Md Nasir, M. H. N., Ab Razak, M. F., Firdaus, A. & Anuar, N. B. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *J. Netw. Comput. Appl.* **179**, 103009 (2021).
26. Beaman, C., Redbourne, M., Mummery, J. D. & Hakak, S. Fuzzing vulnerability discovery techniques: Survey, challenges and future directions. *Comput. Secur.* **120**, 102813 (2022).
27. Kronjee, J., Hommersom, A. & Vranken, H. Discovering software vulnerabilities using data-flow analysis and machine learning. In *ARES '18: Proceedings of the 13th International Conference on Availability, Reliability and Security* (2018).
28. Kim, S., Woo, S., Lee, H. & Oh, H. VUDDY: A scalable approach for vulnerable code clone discovery. In *IEEE Symposium on Security and Privacy* (2017).

29. Shuai, B., Li, H., Zhang, L., Zhang, Q. & Tang, C. Software vulnerability detection based on code coverage and test cost. In *International Conference on Computational Intelligence and Security* (2015).
30. Yu, Z., Theisen, C., Williams, L. & Menzies, T. Improving vulnerability inspection efficiency using active learning. *IEEE Trans. Softw. Eng.* **47**, 2401–2420 (2015).
31. Liu, S. *et al.* DeepBalance: Deep-learning and fuzzy oversampling for vulnerabilities detection. *IEEE Trans. Fuzzy Syst.* **28**(7), 1329–1343 (2019).
32. Yamaguchi, F., Golde, N., Arp, D. & Riek, K. Modeling and discovering vulnerabilities with code property graphs. In *IEEE Symposium on Security and Privacy* (2014).
33. Hin, D., Kan, A., Chen, H. & Babar, M. A. LineVD: Statement-level vulnerability detection using graph neural networks. In *MSR '22: Proceedings of the 19th International Conference on Mining Software Repositories, New York* (2022).
34. Fan, Y., Wan, C., Han, C. F. L. & Xu, H. VDoTR: Vulnerability detection based on tensor representation of comprehensive code graphs. *Comput. Secur.* **130**, 103247 (2023).
35. Suneja, S., Zheng, Y., Zhuang, Y., Laredo, J. & Morari, A. Learning to map source code to software vulnerability using code-as-a-graph. In *International Conference on Science and Technology* (2020).
36. Mikolov, T., Chen, K., Corrado, G. & Dea, J. Efficient estimation of word representations in vector space. In *ICLR Workshop Track 2013*, *Scottsdale, AZ, USA* (2013).
37. Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M. & Monfardini, G. The graph neural network model. *IEEE Trans. Neural Netw.* **20**(1), 61–80 (2008).
38. Wang, J., Xiao, H., Zhong, S. & Xiao, Y. DeepVulSeeker: A novel vulnerability identification framework via code graph structure and pre-training mechanism. *Future Gener. Comput. Syst.* **148**, 15–26 (2023).
39. Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M. & Yin, J. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics* (2022).
40. Liang, H., Wang, L. S. M. & Yang, Y. Deep learning with customized abstract syntax tree for bug localization. *IEEE Access* **7**, 116309–116320 (2019).
41. Partenza, G., Amburgey, T., Deng, L., Dehlinger, J. & Chakraborty, S. Automatic identification of vulnerable code: Investigations with an AST-based neural network. In *IEEE Annual International Computer Software and Applications Conference (COMPSAC)* (2021).
42. Vishnu, P. R., Vinod, P. & Yerima, S. Y. A deep learning approach for classifying vulnerability descriptions using self attention based neural network. *J. Netw. Syst. Manag.* **30**, 9 (2021).
43. Sobania, D., Briesch, M., Hanna, C. & Petke, J. An analysis of the automatic bug fixing performance of ChatGPT. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR),* 23–30 (2023).
44. Ren, J., Zheng, Z., Liu, Q., Wei, Z. & Yan, H. A buffer overflow prediction approach based on software metrics and machine learning. *Secur. Commun. Netw.* **2019**, 1-13, 8391425 (2019).
45. Li, R., Feng, C., Zhang, X. & Tang, C. A lightweight assisted vulnerability discovery method using deep neural networks. *IEEE Access* **7**, 80079–80092 (2019).
46. Li, X. *et al.* Automated vulnerability detection in source code using minimum intermediate representation learning. *Appl. Sci.* **10**(5), 1692 (2020).
47. Tian, J., Xing, W. & Li, Z. BVDetector: A program slice-based binary code vulnerability intelligent detection system. *Inf. Softw. Technol.* **123**, 106289 (2020).
48. Zagane, M., Abdi, M. K. & Alenezi, M. Deep learning for software vulnerabilities detection using code metrics. *IEEE Access* **8**, 74562–74570 (2020).
49. Bashir, O. A. Detecting cross-site scripting attacks using deep neural networks. In *2023 3rd International Conference on Computing and Information Technology (ICCIT)* (2023).
50. Zhou, Y., Liu, S., Siow, J., Du, X. & Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *NeurIPS* **32**, 1-11 (2019).
51. Guo, N., Li, X., Yin, H. & Gao, Y. VulHunter: An automated vulnerability detection system based on deep learning and bytecode. In *International Conference of Information and Communication Security, China* (2019).
52. Xuan, C. D., Mai, D. H., Thanh, M. C. & Cong, B. V. A novel approach for software vulnerability detection based on intelligent cognitive computing. *J. Supercomputing* **79**(15), 17042–17078 (2023).
53. Russell, R. *et al.* Automated vulnerability detection in source code using deep representation learning. In *IEEE International Conference on Machine Learning and Applications (IEEE ICMLA 2018)*, *Orland, Florida, USA* (2018).
54. Hu, L., Chang, J., Chen, Z. & Hou, B. Web application vulnerability detection method based on machine learning. *J. Phys.* **1827**(1), 012061 (2021).
55. Alves, H., Fonseca, B. & Antunes, N. Experimenting machine learning techniques to predict vulnerabilities. In *Latin-American Symposium on Dependable Computing (LADC)* (2016).
56. Saccente, N., Dehlinger, J., Deng, L., Chakraborty, S. & Xiong, Y. Project Achilles: A prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network. In *34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)* (2019).
57. Pang, Y., Xue, X. & Wang, H. Predicting vulnerable software components through deep neural network. In *Proceedings of the 2017 International Conference on Deep Learning Technologies* (2017).
58. Wartschinski, L., Noller, Y., Vogel, T., Kehrer, T. & Grunske, L. VUDENC: Vulnerability detection with deep learning on a natural codebase for python. *Inf. Softw. Technol.* **14**, 106809 (2022).
59. Niu, W. *et al.* A deep learning based static taint analysis approach for IoT software vulnerability location. *Measurement* **152**, 107139 (2020).
60. An, J. H., Wang, Z. & Joe, I. A CNN-based automatic vulnerability detection. *EURASIP J. Wirel. Commun. Netw.* **2023**(1), 41 (2023).
61. Phan, A. V., Nguyen, M. L. & Bui, L. T. Convolutional neural networks over control flow graphs for software defect prediction. In *International Conference on Tools for Artificial Intelligence (ICTAI)*, *Boston, MA, USA* (2017).
62. Hanif, H., Nasir, M. H. N. M., Razak, M. F. A., Firdaus, A. & Anuar, N. B. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *J. Netw. Comput. Appl.* **179**, 103009 (2021).
63. Luo, Y., Xu, W. & Xu, D. Compact abstract graphs for detecting code vulnerability with GNN models. In *ACSAC'22:Proceeding of the 38th Annual Computer Security Applications Conference ACM, Texas* (2022).
64. Nguyen, V. A., Nguyen, D. Q., Nguyen, V., Le, T., Tran, Q. H. & Phung, D. ReGVD: Revisiting graph neural networks for vulnerability detection. In *ICSE'22:Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, New York* (2021).
65. Boghdady, A. A., Ramly, M. E. & Wassif, K. iDetect for vulnerability detection in internet of things operating systems using machine learning. *Sci. Rep.* **12**(1), 17086 (2022).
66. Perl, H. *et al.* VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *ACM* (2015).
67. Grieco, G., Grinblat, G. L., Uzal, L. C., Rawat, S., Feist, J. & Mounier, L. Toward large-scale vulnerability discovery using machine learning. In *CODASPY '16: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (2016).

68. Chernis, B. & Verma, R. M. Machine learning methods for software vulnerability detection. In *IWSPA '18: Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics, New York* (2018).
69. Iorga, D., Corlătescu, D., Grigorescu, O., Săndescu, C., Dascălu, M. & Rughiniş, R. Early detection of vulnerabilities from news websites using machine learning models. In *Roedunet International Conference (RoEduNet)* (2020).
70. CWE. https://cwe.mitre.org (2022) https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.
71. Yan, H. *et al.* Cross-site scripting attack detection based on a modified convolution neural network. *Front. Comput. Neuro Sci.* **16**, 981739 (2022).
72. Shahriar, H. & Zulkernine, M. S2XS2: A server side approach to automatically detect XSS attacks. In *IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC), Australia* (2011).
73. Wang, R., Xu, G., Zeng, X., Li, X. & Feng, Z. TT-XSS: A novel taint tracking based dynamic detection framework for DOM Cross-Site Scripting. *J. Parallel Distrib. Comput.* **118**, 100–106 (2018).
74. Kaur, G., Malik, Y., Samuel, H. & Jaafar, F. Detecting blind cross-site scripting attacks using machine learning. In *SPML '18: Proceedings of the 2018 International Conference on Signal Processing and Machine Learning, Shanghai China* (2018).
75. Fang, Y., Li, Y., Liu, L. & Huang, C. DeepXSS: Cross site scripting detection based on deep learning. In *ICCAI '18: Proceedings of the 2018 International Conference on Computing and Artificial Intelligence, Chengdu China* (2018).
76. Tariq, I. *et al.* Resolving cross-site scripting attacks through genetic algorithm and reinforcement learning. *Expert Syst. Appl.* **168**, 114386 (2015).
77. Prasetio, D. A., Kusrini, K. & Arief, M. R. Cross-site scripting attack detection using machine learning with hybrid features. *J. Infotel* **13**, 1–6 (2021).
78. Falana, O. J., Ebo, I. O., Tinubu, C. O., Adejimi, O. A. & Ntuk, A. Detection of cross-site scripting attacks using dynamic analysis and fuzzy inference system. In *International Conference in Mathematics, Computer Engineering and Computer Science (ICMCECS), Ayobo, Nigeria* (2020).
79. Tsai, D. R., Chang, A. Y., Liu, P. & Chen, H. C. Optimum tuning of defense settings for common attacks on the web applications. In *43rd Annual 2009 International Carnahan Conference on Security Technology* (2009).
80. Day, D. J., Zhao, Z. & Ma, M. Detecting return-to-libc buffer overflow attacks using network intrusion detection systems. In *The Fourth International Conference on Digital Society, ICDS, Netherland Antilles* (2010).
81. Sun, H., Zhang, X., Su, C. & Zeng, Q. Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability. In *ASIA CCS '15: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, Singapore Republic of Singapore* (2015).
82. Ye, T., Zhang, L., Wang, L. & Li, X. An empirical study on detecting and fixing buffer overflow bugs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)* (2016).
83. Gao, F., Wang, L. & Li, X. BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. In *The 31st IEEE/ACM International Conference* (2016).
84. Kasım, Ö. An ensemble classification-based approach to detect attack level of SQL injections. *J. Inf. Security Appl.* **59**, 102852 (2021).
85. Xiao, Z., Zhou, Z., Yang, W. & Deng, C. An approach for SQL injection detection based on behavior and response analysis. In *International Conference on Communication Software and Networks, ICCSN, Guangzhou, China* (2017).
86. Ciampa, A., Visaggio, C. A. & Penta, M. D. A heuristic-based approach for detecting SQL-injection vulnerabilities in web applications. In *SESS '10: Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, Cape Town, South Africa* (2010).
87. Nofal, D. E. & Amer, A. A. SQL injection attacks detection and prevention based on neuro—fuzzy technique. *Mach. Learn. Big Data Anal. Paradigms Anal. Appl. Challenges* **77**, 93-112 (2021).
88. Qureshi, K. N., Ghani, I. & Aliero, M. S. An algorithm for detecting SQL injection vulnerability using black-box testing. *J. Ambient Intell. Human. Comput.* **11**, 249–266 (2019).
89. Gu, H. *et al.* DIAVA: A traffic-based framework for detection of SQL injection attacks and vulnerability analysis of leaked data. *IEEE Trans. Reliab.* **69**(1), 188–202 (2019).
90. Ghafarian, A. A hybrid method for detection and prevention of SQL injection attacks. In *Science and Information Conference (SAI), London, UK* (2017).
91. Ping, C. A second-order SQL injection detection method. In *IEEE Information Technology, Networking, Electronic and Automation Control Conference, Chengdu, China* (2017).
92. Dysart, F. & Sherriff, M. Automated fix generator for SQL injection attacks. In *International Symposium on Software Reliability Engineering (ISSRE)* (2008).
93. Cui, S., Zhao, G., Gao, Y., Tavu, T. & Huang, J. VRust: Automated vulnerability detection for solana smart contracts. In *CCS '22: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, New York* (2022).
94. Lu, J., Li, H., Liu, C., Li, L. & Cheng, K. Detecting missing-permission-check vulnerabilities in distributed cloud systems. In *CCS '22: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, New York* (2022).
95. Lu, K., Pakki, A. & Wu, Q. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *USENIX Security Symposium* (2019).
96. Monshizadeh, M., Naldurg, P. & Venkatakrishnan, V. N. MACE: Detecting privilege escalation vulnerabilities in web applications. In *CCS '14: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale Arizona USA* (2014).
97. Situ, L., Wang, L., Liu, Y., Mao, B. & Li, X. Vanguard: detecting missing checks for prognosing potential vulnerabilities. In *Internetware '18: Proceedings of the 10th Asia-Pacific Symposium on Internetware, Beijing, China* (2018).
98. Son, S., McKinley, K. S. & Shmatikov, V. RoleCast: Finding missing security checks when you do not know what checks are. In *OOPSLA'11: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (2011).
99. Park, J., Shin, J. & Choi, B. Detection of vulnerabilities by incorrect use of variable using machine learning. *MDPI* **12**(5), 1197 (2023).
100. Al-Boghdady, A., El-Ramly, M. & Wassif, K. iDetect for vulnerability detection in internet of things operating systems using machine learning. *Sci. Rep.* **12**(1), 17086 (2022).
101. Ziems, N. & Wu, S. Security vulnerability detection using deep learning natural language processing. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (2021).
102. Li, X. *et al.* Automated software vulnerability detection based on hybrid neural network. *Appl. Sci.* **11**(7), 3201 (2021).
103. Jeon, S. & Kim, H. K. AutoVAS: An automated vulnerability analysis system with a deep learning approach. *Comput. Secur.* **106**, 102308 (2021).
104. Li, Z. *et al.* SySeVR: a framework for using deep learning to detect software vulnerabilities. *IEEE Trans. Dependable Secure Comput.* **19**, 2244–2258 (2022).
105. Haojie, Z., Yujun, L., Yiwei, L. & Nanxin, Z. Vulmg: A static detection solution for source code vulnerabilities based on code property graph and graph attention network. In *International Conference on Wavelet Active Media Technology and Information Processing (ICWAMTIP)* (2021).

106. Grahn, D. & Zhang, J. An analysis of C/C++ datasets for machine learning-assisted software. In *Proceedings of the Conference on Applied Machine Learning for Information Security, 2021* (2021).
107. Amankwah, R., Chen, J., Song, H. & Kudjo, P. K. Bug detection in Java code: An extensive evaluation of static analysis tools using Juliet Test Suites. *Softw. Pract. Exp.* **53**(5), 1125-1143 (2022).
108. Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A. & Devanbu, P. On the "naturalness" of buggy code. In *ICSE '16: Proceedings of the 38th International Conference on Software Engineering May 2016* (2016).
109. Allamanis, M., Devanbu, E. T. B. P. & Sutton, C. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.* **51**(4), 1–37 (2018).
110. Feng, Z. *et al.* CodeBERT: A pre-trained model for programming and natural languages. In *Association for Computational Linguistics EMNLP*, 1536–1547 (2020).
111. Liu, Y. *et al.* RoBERTa: A robustly optimized BERT pretraining approach. In *International Conference on Learning Representations, Adis Ababa* (2019).
112. Yang, K., Miller, P. & Martinez-Del-Rincon, J. Convolutional neural network for software vulnerability detection. In *Cyber Research Conference-Ireland (Cyber-RCI)* (2022).
113. Rawat, R. & Shrivastav, S. K. SQL injection attack detection using SVM. *Int. J. Comput. Appl.* **42**(13), 1–4 (2012).
114. Luo, Y., Xu, W. & Xu, D. Compact abstract graphs for detecting code vulnerability with GNN models. In *ACSAC '22: Proceedings of the 38th Annual Computer Security Applications Conference, New York* (2022).
115. Xu, A., Dai, T., Chen, H., Ming, Z. & Li, W. Vulnerability detection for source code using contextual LSTM. In *5th International Conference on Systems and Informatics (ICSAI)* (2018).
116. Wang, M., Xie, Z., Wen, X., Li, J. & Zhou, K. Ethereum smart contract vulnerability detection model based on triplet loss and BiLSTM. *Electronics* **12**(10), 2327 (2023).
117. Kasongo, S. M. & Sun, Y. Performance analysis of intrusion detection systems using a feature selection method on the UNSW-NB15 dataset. *J. Big Data* **7**(105), 1-20 (2020).
118. Nawir, M., Amir, A., Yaakob, N. & Lynn, O. B. Multi-classification of UNSW-NB15 dataset for network anomaly detection system. *J. Theor. Appl. Inf. Technol* **96**(15), 5094–5104 (2018).
119. Kasongo, S. M. & Sun, Y. A deep learning method with wrapper based feature extraction for wireless intrusion detection system. *Comput. Secur.* **92**, 101752 (2020).
120. Eunice, A. D., Gao, Q., Zhu, M.-Y., Chen, Z. & Lv, N. Network anomaly detection technology based on deep learning. In *2021 IEEE 3rd International Conference on Frontiers Technology of Information and Computer (ICFTIC)* (2021).
121. Li, Z. *et al.* VulDeePecker: A deep learning-based system for vulnerability detection. In *Network and Distributed System Security (NDSS) Symposium* (2018).
122. Akram, J. & Luo, P. SQVDT: A scalable quantitative vulnerability detection technique for source code security assessment. *Softw. Pract. Exp.* **51**(2), 294-318 (2020).
123. Huang, H. *et al.* ExpGen: A 2-step vulnerability exploitability evaluation solution for binary programs under ASLR environment. *Appl. Sci.* **12**(13), 6593 (2022).
124. Wang, L. *et al.* PreNNsem: A heterogeneous ensemble learning framework for vulnerability detection in software. *Appl. Sci.* **10**(22), 7954 (2023).
125. Zhang, H., Bi, Y., Guo, H., Sun, W. & Li, J. ISVSF: Intelligent vulnerability detection against Java via sentence-level pattern exploring. *IEEE Syst. J.* **16**(1), 1032–1043 (2021).
126. Liu, Z., Fang, Y., Huang, C. & Xu, Y. MFXSS: An effective XSS vulnerability detection method in JavaScript based on multi-feature model. *Comput. Secur.* **124**, 103015 (2023).
127. Zhao, Q., Huang, C. & Dai, L. VULDEFF: Vulnerability detection method based on function fingerprints and code differences. *Knowl.-Based Syst.* **260**, 110139 (2022).
128. Dong, Y., Tang, Y., Cheng, X., Yang, Y. & Wang, S. SedSVD: Statement-level software vulnerability detection based on Relational Graph Convolutional Network with subgraph embedding. *Inf. Softw. Technol.* **158**, 107168 (2023).
129. Li, L. *et al.* VulANalyzeR: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution. *ACM Trans. Privacy Secur.* **26**, 1–25 (2023).
130. Wang, H. *et al.* Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Trans. Inf. Forensics Secur.* **16**, 1943–1958 (2020).
131. Wang, S., Wang, X., Sun, K., Jajodia, S., Wang, H. & Li, Q. GraphSPD: Graph-based security patch detection with enriched code semantics. In *IEEE Symposium on Security and Privacy* (2023).
132. Chen, J. *et al.* BiTCN_DRSN: An effective software vulnerability detection model based on an improved temporal convolutional network. *J. Syst. Softw.* **204**, 111772 (2023).
133. Cheng, Y., Yang, S., Lang, Z., Shi, Z. & Sun, L. VERI: A large-scale open-source components vulnerability detection in IoT firmware. *Comput. Secur.* **126**, 103068 (2023).

Author contributions

Shumaila Hussain: conceived and designed the analysis; performed the formal statistical analysis, wrote the paper, original draft; writing review and editing. Muhammad Nadeem: performed formal the statistical analysis; contributed reagents, materials, analysis tools or data; wrote the paper; writing review and editing. Junaid Baber, Mohammed Hamdi: conceived and designed the analysis; performed the analysis; analyzed and interpreted the data; Contributed reagents, materials, analysis tools; Wrote the paper; writing review and editing. Adel Rajab, Mana Saleh Al Reshan, Asadullah Shaikh: analyzed and collected the review; contributed reagents, materials, analysis tools; Wrote the paper; writing review and editing.

Funding

The authors are thankful to the Deanship of Scientific Research at Najran University for funding this work under the Research Groups Funding Program grant code (NU/RG/SERC/12/34).

Competing interests

The authors declare no competing interests.

Additional information

Correspondence and requests for materials should be addressed to S.H.

Reprints and permissions information is available at www.nature.com/reprints.

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

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

© The Author(s) 2024