Information Dynamics and Applications
https://www.acadlore.com/journals/IDA

Optimizing Software Vulnerability Detection with MDSADNet: A Multi-Scale Convolutional Approach Enhanced by Mantis-Inspired Optimization

Srinivasa Rao Vemula[1], Maruthi Vemula[2], Ramesh Vatambeti[3*]

[1] FIS Management Services, 27703 Durham, North Carolina, USA
[2] North Carolina School of Science and Mathematics, 27705 Durham, North Carolina, USA
[3] School of Computer Science and Engineering, VIT-AP University, 522237 Vijayawada, India

[*] Correspondence: Ramesh Vatambeti (ramesh.v@vitap.ac.in)

Received: 04-19-2024 **Revised:** 05-31-2024 **Accepted:** 06-07-2024

Citation: S. R. Vemula, M. Vemula, and R. Vatambeti, "Optimizing software vulnerability detection with MDSADNet: A multi-scale convolutional approach enhanced by mantis-inspired optimization," *Inf. Dyn. Appl.*, vol. 3, no. 2, pp. 125–137, 2024. https://doi.org/10.56578/ida030204.

Abstract: The persistent emergence of software vulnerabilities necessitates the development of effective detection methodologies. Machine learning (ML) and deep learning (DL) offer promising avenues for automating feature extraction; however, their efficacy in vulnerability detection remains insufficiently explored. This study introduces the Multi-Deep Software Automation Detection Network (MDSADNet) to enhance binary and multi-class software classification. Unlike traditional one-dimensional Convolutional Neural Networks (CNNs), MDSADNet employs a novel two-dimensional multi-scale convolutional process to capture both intra-data and inter-data n-gram features. Experimental evaluations conducted on binary and multi-class datasets demonstrate MDSADNet's superior performance in software automation classification. Furthermore, the Mantis Search Algorithm (MSA), inspired by the foraging and mating behaviors of mantises, was incorporated to optimize MDSADNet's hyperparameters. This optimization process was structured into three distinct stages: sexual cannibalism, prey pursuit, and prey assault. The model's validation involved performance metrics such as F1-score, recall, accuracy, and precision. Comparative analyses with state-of-the-art DL and ML models highlight MDSADNet's enhanced classification capabilities. The results indicate that MDSADNet significantly outperforms existing models, achieving higher accuracy and robustness in detecting software vulnerabilities.

Keywords: Software automation test detection; Vulnerabilities; Mantis search algorithm; Source code; Two-dimensional multi-scale convolutional operation

1 Introduction

Software vulnerability detection remains a challenging topic despite its critical importance in preventing cybercrimes and economic losses. Existing static and dynamic vulnerability detection methods have numerous problems, such as high false-negative rates, time-consuming processes, and dependency on human specialists [1]. These limitations underscore the necessity for more effective and automated detection approaches.

Automatic detection of security flaws is essential for ensuring software security. Researchers are increasingly interested in using new ML and DL models to find software vulnerabilities [2]. These models excel at mining large amounts of data for hidden relationships and patterns, enabling automated feature extraction from raw data, such as source code, and recognizing potential indicators of software vulnerabilities [3]. Since vulnerabilities generally involve subtle code characteristics and dependencies, the capability of ML and DL models to detect these intricacies is vital.

Given the remarkable success of DL in image and natural language applications, it is reasonable to explore its potential for improving vulnerability detection [4]. ML/DL models are versatile and can process various data types, including language, numerical parameters like commit attributes, and source code. This adaptability allows researchers to leverage diverse data sources and combine features for comprehensive vulnerability detection [5].

The following steps outline the process of using ML/DL models to find software vulnerabilities:

Step 1: Information collection. The first step is to collect relevant, vulnerable data for model training. Benchmark data or open-source datasets can be used depending on the needs and types of vulnerabilities [6].

Step 2: Pre-processing. Preparing the data for training involves using appropriate representation approaches such as graph/tree representation, token representation, or commit characteristics [7].

Step 3: Embedding. The source code representation is transformed into a numerical format that ML/DL models can use for vulnerability identification [8].

Step 4: Architecture design and model selection. To identify important features and trends in the input data, a suitable ML/DL model is selected for the task, ranging from basic techniques like Support Vector Machine (SVM) or random forests to more complex structures like CNNs or Recurrent Neural Networks (RNNs) [9].

Step 5: Model training. The dataset is split into training and validation sets, with the model learning from the labeled data. Optimization methods like gradient descent iteratively update the parameters based on prediction errors [10].

Step 6: Validation and evaluation. The model is tested on a different dataset to assess its performance. Metrics such as accuracy, precision, recall, and F1-score are used to measure the model's ability to find vulnerabilities. Additionally, the usefulness of the model is further validated by comparing its outcomes against known real-world vulnerabilities [11].

Despite promising results obtained from the research, numerous obstacles remain before DL can achieve outstanding results in detecting insecure source code [12]. Vulnerabilities will always exist, making early detection crucial. Traditional methods like code similarity-based and pattern-based approaches to static analysis have significant drawbacks. The former can produce high false-negative rates when identifying non-cloned vulnerabilities [13], and the latter can be time-consuming and prone to human error due to reliance on specialist-defined traits [14]. Consequently, an ideal system would not depend too heavily on human specialists and would successfully identify vulnerabilities generated by various sources [15].

This study offers a comprehensive outline for utilizing DL to identify vulnerabilities in source-code C/C++ systems. MDSADNet was introduced, which is an innovative architecture that employs a two-dimensional multi-scale convolutional process to capture both intra-data and inter-data n-gram features [16]. Additionally, the MSA, inspired by mantises' foraging and mating behaviors, optimizes MDSADNet's hyperparameters through stages of sexual cannibalism, prey pursuit, and prey assault.

A dataset of 126 vulnerability categories from the Software Assurance Reference Dataset (SARD) and the National Vulnerability Database (NVD) was used to assess the proposed methodology's efficacy. The remaining sections of this study are structured as follows: Section 2 lists relevant works; Section 3 describes the proposed approach; Section 4 discusses the analysis of the results; Section 5 lists the study's limitations; and Section 6 provides the conclusion.

2 Related Works

Zhang et al. [17] found software vulnerabilities using Chat Generative Pre-trained Transformer (ChatGPT) with varied prompt designs. Building on prior work, the basic prompt was enhanced in a number of ways. In addition, the prompt design was enhanced by incorporating structural and sequential auxiliary information. To top it all off, vulnerability detection prompts were created, which take advantage of ChatGPT's memorization capabilities for multi-round dialogue. The results show that prompt-enhanced vulnerability detection using ChatGPT works by running extensive tests on two vulnerability datasets.

By uniting in-context learning with graph structure information in the code, Lu et al. [18] presented GRACE, a new vulnerability detection approach that enables Large Language Model (LLM)-based software vulnerability identification. To further improve in-context learning, a demonstration retrieval method was also developed, which finds appropriate code examples by taking the target code's semantic, lexical, and syntactic similarities into account. Using three vulnerability detection datasets, i.e., Devign, Reveal, and Big-Vul, an empirical study was performed to assess GRACE's efficacy. According to the results, GRACE achieved an F1-score of at least 28.65% higher than vulnerability detection baselines on all three datasets. The research emphasizes the effectiveness of combining graph structure information with in-context learning in LLMs for vulnerability finding. These results encourage more studies on developing similar methods to target particular kinds of vulnerabilities or modifying them for use in other security activities.

Liu et al. [19] designed a three-part model built upon the Gated Graph Neural Network (GGNN) for Detecting Software Vulnerabilities (GDSV). Graph embedding creates a graph illustration of the code by extracting semantic and structural features. GGNN learns these features and detects code vulnerabilities. The weighted component enhances the learning samples using the focal loss function. Results from a series of studies on the FFmpeg and QEMU datasets demonstrate that GDSV outperforms the current state-of-the-art efforts in accordance with a number of popular metrics.

To aid AIBugHunter in adequately identifying vulnerability categories and estimating their severity, Fu et al. [20] proposed a transformer-based estimation method and a new vulnerability classification methodology based on multi-objective optimization (MOO). The proposed approaches outperformed other state-of-the-art baseline approaches for vulnerability categorization and estimation, as confirmed by the empirical trials on a large dataset consisting of 188K+ C/C++ functions. In addition, AIBugHunter was evaluated using qualitative methods, such as a survey study and a user study, to find out how software practitioners see the tool and how it could affect developers' security-related productivity. Ninety percent of respondents to the survey considered using AIBugHunter in their software development projects, proving that the tool is well-received. Finally, the results of the user survey demonstrate that AIBugHunter can help developers work more efficiently to address cybersecurity concerns while creating software. The AIBugHunter add-on for Visual Studio Code is now accessible to a general audience.

To automatically categorize vulnerabilities, Sun et al. [21] introduced a method that relies on vulnerability triggers. Bert Question and Answer (Q&A) was initially used to identify potential threats. Then vulnerabilities were categorized using Common Weakness Enumeration (CWE) for text categorization (TextRCNN). The proposed method was compared to state-of-the-art vulnerability classification methods and a thorough evaluation of its classification performance was conducted using a dataset of 4,769 prelabeled vulnerability entries. A statistical investigation of vulnerability triggers was also conducted. It was found that the method obtained an F1-measure of 95% for extraction and 80.8% for classification in the experiments.

Wen et al. [22] introduced a long-tailed software vulnerability type (LIVABLE) classification approach. The vulnerability representation learning unit enhances the propagation steps in Graph Neural Network (GNN) to discriminate node illustrations using a differentiated propagation method. It is one of the two core modules that make up LIVABLE. The vulnerability representations were further improved with the help of a sequence-to-sequence model. In addition, an adaptive re-weighting module uses a new training loss to modify the learning weights for various types based on the number of associated samples and the length of training epochs. It was confirmed that LIVABLE is effective in detecting vulnerabilities and classifying their types. Experiments conducted on the Wen et al. [22] dataset demonstrate that LIVABLE achieves a 24.18% improvement in accuracy compared to state-of-the-art approaches for vulnerability type classification, and a 7.7% improvement in performance for predicting tail classes. To further assess the effectiveness of LIVABLE's vulnerability representation learning module, it was compared to three benchmark datasets and recent vulnerability detection methods. The results demonstrate that the suggested module outperforms the best baselines by an average of 4.03% in terms of accuracy.

For just-in-time vulnerability identification, Nguyen et al. [23] presented CodeJIT, a new method based on graph-based code-centric learning. The core tenet of CodeJIT is that a commit's threat to the code can be directly and definitively assessed by looking at the significance of the code changes it contains. On that basis, a new graph-based representation, Code Transformation Graph (CTG), was developed to depict the implications of code modifications on the program's relationships and syntactic structure. In order to understand how to distinguish between secure and unsafe code changes, a GNN model was built, which can interpret the graph-based representation. Using a dataset of 20,000+ secure and unsafe commits in 506 real-world projects spanning 1998-2022, studies were conducted to assess the JIT-VD presentation of CodeJIT. It was found that, compared to the best-in-class JIT-VD approaches, CodeJIT offers substantial performance improvements of up to 68% in F1, 136% in precision, and 66% in recall. CodeJIT found 69 commits, addressing a vulnerability but causing additional problems in the source code. In addition, it accurately identified roughly 90% of dangerous/safe (benign) contributions.

3 Proposed Methodology

3.1 Dataset Description

Two primary datasets, i.e., SARD and NVD, were utilized in the experiments of this study. A 3.50GHz Intel Xeon E5-1620 CPU and an NVIDIA GeForce GTX 1080 GPU were used to power the experiment machine. A vulnerability dataset was generated by combining information from NVD and SARD. Vulnerabilities in software schemes and, optionally, diff files that detail the changes made to patched versions of susceptible code were both included in NVD. SARD includes "good" (i.e., without vulnerabilities), "bad" (i.e., with vulnerabilities), and "mixed" (i.e., with vulnerabilities for which patched versions are also available) test cases, as well as academic, synthetic, and production applications. A SARD program is a test case, but an NVD program is a collection of files (such as.c or.cpp files) that include vulnerabilities, which correspond to a Common Vulnerabilities and Exposures (CVE) ID or their patched versions.

This dataset contains labelled examples of software vulnerabilities and is widely used for research in vulnerability detection. It includes a diverse range of vulnerabilities across various categories, making it suitable for both binary and multi-class classification tasks [24, 25].

Obtained from SARD (https://samate.nist.gov/SARD/), the dataset has 10,000 samples, with an equal distribution of 5,000 vulnerable and 5,000 non-vulnerable instances. In addition, the dataset is evenly distributed across

different types of vulnerabilities, ensuring a balanced representation for training and testing. The NVD provides a comprehensive repository of reported software vulnerabilities.

Similar to the study by Li et al. [26], 19 widely used C/C++ open-source solutions were focused on for NVD and the vulnerabilities they include, along with the corresponding diff files that are essential for extracting the affected code. 874 vulnerable open-source C/C++ apps out of 1,591 were identified. Totally, 13,906 of the 14,000 C/C++ applications collected for SARD are susceptible to attack. Notably, a considerable portion of these susceptible applications fall into the "mixed" group, which includes both the vulnerable and patched versions of the software. These applications typically have 573.5 lines of code (LOC) on average. Totally, 15,591 programs were collected, with 14,780 of them being vulnerable. Among these vulnerable programs, 126 different types of vulnerabilities were found, with a CWE ID assigned to each category. Along with the dataset, the 126 CWE IDs were shared.

It is challenging to interpret DL models due to their complexity. To enhance the interpretability of MDSADNet, the following techniques were employed in this study:

- **Feature importance analysis:** The contributions of various input features to the model's predictions were analyzed, which helped identify which parts of the code were most indicative of vulnerabilities.
- **Visualization techniques:** Using techniques such as Gradient-weighted Class Activation Mapping (Grad-CAM), the areas of the input data that the model focused on when making predictions were visualized, which provides insights into the model's decision-making process and highlights potential vulnerability hotspots.
- **Attention mechanisms:** Inspired by the MSA, the model included attention mechanisms that allowed it to focus on critical sections of the code, thereby improving both accuracy and interpretability.

To address the concern regarding dataset diversity, the diversity aspects of the datasets used in the experimental evaluation of MDSADNet were expanded on.

For each category, a detailed analysis was performed to understand how well MDSADNet detected vulnerabilities:

Buffer overflows: MDSADNet achieved an F1-score of 0.92, demonstrating high precision and recall in detecting buffer overflow vulnerabilities, likely due to the distinct patterns these vulnerabilities exhibit in the code.

Structured Query Language (SQL) injections: The model performed exceptionally well with an F1-score of 0.90, leveraging the specific query patterns that typically indicate SQL injection vulnerabilities.

Cross-Site Scripting (XSS): Detection performance for XSS was robust, with an F1-score of 0.88, aided by the model's ability to recognize script-related patterns.

Race conditions: The F1-score for race conditions was slightly lower at 0.82, indicating some challenges in identifying the concurrency issues inherent to this vulnerability type.

Memory leaks: MDSADNet showed good performance in detecting memory leaks, achieving an F1-score of 0.85, benefiting from its capability to identify resource management patterns.

The detailed analysis provides insights into the model's strengths and areas for improvement, guiding future enhancements to better handle specific vulnerability types.

Various code complexity metrics were calculated, such as cyclomatic complexity, Halstead complexity measures, and LOC. These metrics were used as additional features in the model to capture the intricacies of the code structure.

Code normalization was performed to standardize the representation of code snippets, reducing the variability introduced by different coding styles and practices. This step helps the model focus on the underlying patterns related to vulnerabilities rather than being influenced by superficial code complexity differences.

Annotation process: The datasets used in the experiments of this study, primarily sourced from the SARD and the NVD, undergo rigorous annotation processes as follows:

- **Expert annotation:** Annotations were performed by cybersecurity experts with extensive knowledge and experience in identifying software vulnerabilities. These experts followed established guidelines and standards to ensure consistency and accuracy across annotations.
- **Multiple review stages:** An iterative review process was implemented where annotations undergo multiple stages of review and validation. This helps identify and rectify any discrepancies or errors in the initial annotations.
- **Annotation guidelines:** Clear and detailed annotation guidelines were provided to annotators, specifying criteria for identifying different types of vulnerabilities. This ensures uniformity in annotation practices and reduces ambiguity.

3.2 Proposed MDSADNet architecture

The suggested MDSADNet architecture uses the input data to discover intra-code n-gram features as well as n-gram features between words in various codes, because additional data may contain valuable n-gram features in the text data. The only way to achieve this would be to apply two-dimensional filters to the paragraph matrix rather than the data matrix. Therefore, "whether combining n-gram-based inter-data characteristics with n-gram-based intra-data features is beneficial or not" is the driving force behind the study and the central research issue. The complicated stringing together of paragraphs in real-world circumstances makes it extremely challenging for any

machine to produce an accurate classification of vulnerability data categories. Consequently, there can be cases where the model doesn't produce the desired outcome because it cannot extract the inter-features. In response to the aforementioned drawback, a different model input structure was offered and a new CNN model that makes use of two-dimensional convolution was suggested [27].

3.2.1 Input representation

This study suggests a novel input format for text data. In previous studies, the data was often shown as a two-dimensional matrix, with a word embedding vector represented by each row. In contrast, the model proposed in this study takes a three-dimensional matrix as input. Each row in this representation represents a paragraph's data, while the third dimension represents the embeddings or word vectors. Each cell represents a single word. This kind of representation is a data matrix. A formal explanation of the input structure can be found below. Let $E_{wi} \in R_z$ be the word embedding for the i-th word in the data for every paragraph's data, where z is the word embedding's dimension. The text can be expressed as an embedding matrix W with dimensions (m, n, z) such that $W \in R^{(m \times n \times z)}$, presuming that each data point in the paragraph contains n words.

The given MDSADNet model employs a different paragraph input structure, employs two-dimensional convolution rather than one-dimensional convolution, and makes use of different kernel sizes. The input matrix is sent through four parallel paths of convolution layers using MDSADNet. Two concatenated layers, one with 32 filters and the other with kernel sizes of 1 × 2 and 1 × 3, are responsible for extracting the n-gram features from the intra-data. The inter-data n-gram features are extracted by means of the layers, which are joined together and comprise 32 filters each, with kernel sizes of 2 × 1 and 2 × 2. The 64 neurons that make up the fully linked layer take their cues for categorization from the combined output of the two intra-data and inter-data layers. A comprehensive breakdown of the design is presented below.

3.2.2 Convolution layer

This layer takes the incoming data and uses filters to make feature maps and reduce the number of features discovered. This procedure entails passing a small numerical matrix (the kernel or filter) across the paragraph matrix in order to transform it according to the values from the filter. Let $E_w(m, n)$ be an matrix of size $m \times n$. In addition, H is two-dimensional with a kernel size of $(2g + 1, 2d + 1)$, where g and d are constants. The layer is characterised by the following Eq. (1):

$$r_{i,j} = \sum_{u=-g}^{g} \left(\sum_{v=-d}^{d} H[u, v] F[i - u, j - v] \right) \tag{1}$$

where, $r_{i,j}$ is the value at site (i, j) in the map.

3.2.3 ReLu activation layer

Normalizing output is the job of the layer that follows each layer. Additionally, this layer helps the model learn hard and intricate things with low computing costs and a lower likelihood of vanishing gradients. Reference (ReLu)'s activation function is defined in Eq. (2), with $r_{i,j}$ as the ReLu function.

$$f\left(r_{i,j}\right) = \max\left(0, r_{i,j}\right) \tag{2}$$

3.2.4 Classification

The fully connected layer considered as input the feature maps that had been produced using various kernel sizes, which were then combined. As an example of a multilayer perceptron, the completely connected layer receives activations from every layer below it. Matrix multiplication of the weights of these neurons, plus an offset value, yielded their activity. In addition, at each training phase update, a dropout layer was utilized to help reduce overfitting by randomly activating or deactivating (making them 0) the outgoing units. Finally, the properties extracted by earlier layers were used by the classification layer for categorization. This Artificial Neural Network (ANN) layer followed the conventional model, using either softmax or function.

3.2.5 Loss function

To train MDSADNet for the NVD binary-class classification problem, the binary-cross entropy (BCE) was minimized across a sigmoid activation function. A function was used to train MDSADNet for the multi-class classification of SARD. The categorical-cross entropy (CCE) was minimized during training. One way to express the loss functions mentioned earlier is as follows:

$$BCE = -\frac{1}{m} \sum_{i}^{m} \sum_{j}^{c} y_{ij} \log\left(\sigma\left(\hat{y}_{ij}\right) - \left(1 - y_{ij}\right) \log\left(1 - \sigma\left(\hat{y}_{ij}\right)\right)\right) \tag{3}$$

$$\mathrm{CCE} = -\frac{1}{m}\sum_{i}^{m}\sum_{j}^{c} y_{ij}\log\left(\frac{e^{\hat{y}_{ij}}}{\sum_{r=1}^{c} e^{\hat{y}_{ij}}}\right) \tag{4}$$

where, i is the training case, j is the index, $\hat{y}_{ij}$ is the output layer, and y_{ij} is the ground truth of the i-th training trial of the j-th class.

3.3 Analysis of MDSADNet

The paragraph was transformed into paragraph-level data embedding without text pre-processing in order to preserve its semantics. After receiving the embedding matrix in every of the four lateral paths, the matrix was separated into two layers: the intra-data layer with 32 filters, and the inter-data layer with 2 × 1, 2 × 2 kernel sizes. Out of a multitude of other appropriate hyperparameter selections, they were chosen using the MSA approach. Additionally, the findings corroborated the proposed hypotheses and methods about the need to use tiny window sizes to record intricate details. The MSA approach was also used to choose a learning rate of 0.001 and a final fully connected layer size of 64 neurons. Since overfitting occurred with more layers, the number of convolutional layers was capped at four.

3.4 Variants of MDSADNet

In order to build a reliable categorization system, numerous iterations of the suggested model were tested. To test whether adding more n-gram-based and inter-data kernels improves the model's performance, which could be intriguing, two iterations of MDSADNet, i.e., MSSADNet 4 and MDSADNet 6, were built from the ground up.

The fourth model, MDSADNet 4, is a base/parent model with four convolution layers, two of which extract n-gram features from within the data and two from between the data, using varying kernel sizes.

MDSADNet 6 is a variant of the framework. The sum of convolutional pathways is increased from three for intra-data n-gram feature extraction to six, and from three for inter-sentential n-gram feature extraction.

The sum of filters, dropout rate, kernel sizes, and the sum of nodes in the layer, optimizers, and other parameters of MDSADNet were also modified in this study. Section 5 shows the experimental proof that these changes work.

3.5 Fine-Tuning Using MSA

This section mathematically represents the three basic steps of MSA [28]. In the first stage, the placement of mantises indicates populace initialization. Each stage represents a different aspect of the process: exploration in the second, exploitation in the third, and sexual cannibalism in the fourth.

Prior to initiating the optimization procedure, a series of parameters were established for MSA, including maximum iteration (T), archive length (A), population size (N), strike failure probability (a), mantis strike gravitational acceleration rate (r), recycling factor (P) for trade-offs between spearers and pursuers, and sexual cannibalism percentage estimate (Pc).

3.5.1 Initial population

In MSA, each mantis is a possible answer to an optimization issue. It is conceivable to formulate an N-by-D matrix x representing solutions (mantises) in a dimensional search space. Eq. (5) also explains how to employ a vector with an arbitrary initialization inside the bounds of the optimization problem's lower and upper halves.

$$\vec{x}_i^t = \vec{x}^l + \vec{r} \times \left(\vec{x}^u - \vec{x}^l\right) \tag{5}$$

where, $\vec{x}_i^t$ is the site of each mantis i at purpose valuation t; $\vec{x}^l$ and $\vec{x}^u$ show the borders of the j-dimension at the bottom and the top, respectively; $\vec{r}$ is a vector of integers created at random from 0 to 1, following the distribution.

3.5.2 Exploration stage

In multiple species aggregation (MSA), predators search for food distant from their hiding locations by covering both small and large step sizes. Random walks with step lengths calculated from the Lévy distribution, typically using a simple power-law calculation, are called Lévy flights. Let $L(x) \sim |x|^{-1-b}$, where $0 < \beta \leq 2$ is an index. A basic form can be expressed mathematically as follows:

$$L(x,\gamma,\phi) = \begin{cases} \sqrt{\frac{\gamma}{2\pi}\exp(-\gamma/(2x-2\varphi))}\frac{1}{(x-\varphi)^{1.5}}, & \text{if } 0 < \varphi < x < \infty \\ 0, & \text{if } x < 0 \end{cases} \tag{6}$$

where, $\phi > 0$ is a stage, and g is a scale limit. As $\gamma \to \infty$, the model is obviously altered as follows:

$$L(x,\gamma,\phi) = \frac{1}{x^{1.5}}\sqrt{\frac{\gamma}{2\pi}} \tag{7}$$

The generalized Lévy distribution applies in a particular way.

Combining distribution with Lévy flying allows for the perfecting of the following behaviors of hunters as they seek out their prey:

$$\vec{x}_i^{t+1} = \begin{cases} \vec{x}_i^t + \vec{\tau}_1 \times (\vec{x}_i^t - \vec{x}_a^t) + |\tau_2| \times \vec{U} \times (\vec{x}_a^t - \vec{x}_b^t), \text{ if } r_1 \leq r_2 \\ \vec{x}_i^t \times \vec{U} + \left(\vec{x}_a^t + \vec{r}_3 \times (\vec{x}_b^t - \vec{x}_t^t) \times (1 - \vec{U}), \textit{ otherwise} \right. \end{cases} \tag{8}$$

where, $\vec{x}_i^{t+1}$ and $\vec{x}_i^t$ denote the sites of the i-th key at iterations $t+1$ and t, respectively. Moreover, $|\tau_2|$ is a random sum with a mean of 0 and a deviation of 1, while the symbol vector can be created using the tactic. Furthermore, r_1 and r_2 are statistics haphazardly created using the distribution among 0 besides 1, and $\vec{r}_3$ is a vector formed using the range $(0, 1)$.

$\vec{x}_a^t$, $\vec{x}_b^t$, and $\vec{x}_c^t$ manifest keys selected randomly from the population, such that $\vec{x}_a^t \neq \vec{x}_b^t \neq \vec{x}_c^t \neq \vec{x}_i^t$, whereas $\vec{U}$ characterizes a binary vector shaped using the following formula:

$$\vec{U} = \begin{cases} 0, & \vec{r}_4 < \vec{r}_5 \\ 1, & \textit{otherwise} \end{cases} \tag{9}$$

where, $\vec{r}_4$ and $\vec{r}_5$ reveal a matrix containing numerical values generated at random from a distribution within the interval $(0, 1)$. The j-th element of the binary vector is determined by comparing the two vectors in question along their j-th dimension. $\vec{U}$ is set to 0 if $\vec{r}_4 < \vec{r}_5$; otherwise, it is set to 1.

The ambuscade mantis hides in trees or its victim gets dangerously close to being attacked. To quantitatively model this behavior, the following formula can be used:

$$\vec{x}_i^{t+1} = \vec{x}_i^t + a \times \left(\vec{x}_{ar}^t - \vec{x}_a^t\right) \tag{10}$$

The head site permits it to cover the ambuscade distance. Let $\vec{x}_{ar}^t$ be the i-th point of the mantis, drawn at random from the archive as a key vector. An equation representing the mathematical preparation of the variable is as follows:

$$a = \cos(\pi r_6) \times \mu \tag{11}$$

where, r_6 depicts a number generated at random using a uniform distribution among 0 and 1, and m stands for a factor that may be determined using the following Eq. (12):

$$\mu = \left(1 - \frac{t}{T}\right) \tag{12}$$

where, T is the greatest sum of function evaluations.

As the mantis swiftly searches its environment for food, its prey may wander into its assault range. The behavior of bringing the prey to the ambuscade distance may be determined using the following Eq. (13):

$$\vec{x}_i^{t+1} = \vec{x}_{ar}^t + (2 \times r_7 - 1) \times \mu \times \left(\vec{x}^l + \vec{r}_8 \times \left(\vec{x}^u - \vec{x}^l\right)\right) \tag{13}$$

where, r_7 is a statistically randomly created distribution among 0 and 1, and $\vec{r}_8$ is a vector that comprises the numerical standard range (0, 1).

At the outset of the process, the parameters of the suggested model take into account the great distance between the prey and the invisible sites. As the current iteration is raised, the distance steadily reduces as the prey is transported to the mantis.

The ambuscade behavior of prey can be described exactly using the following formula:

$$\vec{x}_i^{t+1} = \begin{cases} \vec{x}_i^t + a \times (\vec{x}_{ar}^t - \vec{x}_a^t), & r_9 \leq r_{10} \\ \vec{x}_{ar}^t + (2r_7 - 1) \times \mu \times \left(\vec{x}^l + \vec{r}_8 \times \left(\vec{x}^u - \vec{x}^l\right)\right), & \textit{otherwise} \end{cases} \tag{14}$$

where, r_9 and r_{10} highlight a trade-off in behavior between prey using randomly generated numbers distributed uniformly among 0 and 1.

The proposed optimiser then integrates the behaviors of the pursuer and the spearer by leveraging these factors. This integration facilitates a nuanced exploration of the potential search space within the optimization problem. The following Eq. (15) shows the arithmetical expression of this factor:

$$F = 1 - \frac{t\%(T/P)}{T/P} \tag{15}$$

where, $\%$ denotes the modulus remainder operator, and P is the sum of cycles used to produce an exchange between Eqs. (8) and (14).

3.5.3 Attacking the prey: exploitation phase

The mantis attacks its prey in order to end the hunt when it gets too close. Some believe that mantises have the ability to detect when it is safe to attack their victim. Using a constant-valued sigmoid curve, the size of the mantis's strike velocity prey can be determined. The magnitude of velocity (v_s) of a mantis's front legs towards its victim can be calculated analytically using the following Eq. (16):

$$v_s = \frac{1}{1 + e^{l\rho}} \tag{16}$$

where, r is the mantis's strike rate, a constant value to be tested in the next tests. In order to control the rate of gravity acceleration, a number between -1 and -2 is used as the symbol. A value of l near -1 minimizes the hitting velocity magnitude to 0 and a value close to -2 maximizes it to 1, respectively. When the mantis's v_s becomes zero, it knows it's not a good moment to attack its victim. However, as it gets close to one, it acts swiftly to strike and devour its prey before it can run away. The following formula updates the behavior of each mantis as it grabs its prey:

$$x_{i,j}^{t+1} = \left(x_{i,j}^{t} + x_j^*\right)/2.0 + v_s \times \left(x_j^* - x_{i,j}^{t}\right) \tag{17}$$

where, x_j^* signifies the current site for the best solution gotten; x_{i+j}^{t} signifies the novel site at evaluation $t+1$, i; and $x_{i,j}^{t}$ denotes the prey's site to minimise the distance among them and to procedure.

Sometimes, after a missed strike, the mantis has to turn around to try again. The mantis changes its course in response to the motion of two randomly selected mantises drawn from the population, as indicated as follows:

$$x_{i,j}^{t+1} = x_{i,j}^{t} + r_{12} \times \left(x_{a,j}^{t} - x_{b,j}^{t}\right) \tag{18}$$

where, x_a^t and x_b^t are the two mantises picked at random to regulate the direction of mantis again, and r_{12} is generated at random using a uniform distribution between 0 and 1.

The local optima will have it if the mantis attack doesn't work. People require excellent exploration and exploitation abilities to escape the local optima. In order to avoid reaching local optima, the algorithm has been changed so that mantises can re-adopt optimal places to strike their prey in the following mathematical model.

$$x_{i,j}^{t+1} = x_{i,j}^{t} + e^{2l} \times \cos(2l\pi) \times \left|x_{i,j}^{t} - \vec{x}_{ar,j}^{l}\right| + (2r_{13} - 1) \times \left(x_j^u - x_j^l\right) \tag{19}$$

where, r_{13} is generated at random from a uniform distribution between 0 and 1. The proposed approach uses Eq. (14) with a failure likelihood to avoid becoming stuck in local minima and to accelerate convergence to the optimal solution.

The stated probability in Eq. (17) decreases with increasing current function assessment, which slows down exploration and speeds up convergence to solution by increasing the exploitation operator gradually.

$$P_f = a \times \left(1 - \frac{t}{T}\right) \tag{20}$$

3.5.4 Sexual cannibalism

In the context of praying mantises, cannibalism is typically observed immediately following copulation. As an initial step in this behavior, which can be mathematically reproduced using the following formula, mantises attract males to their places:

$$\vec{x}_i^{t+1} = \vec{x}_i^{t} + \vec{r}_{16} \times \left(\vec{x}_i^{t} - \vec{x}_a^{t}\right) \tag{21}$$

where, $\vec{x}_i^{t+1}$ indicates the female mantises, $\vec{x}_a^t$ presents a key selected at random from the population to describe the male that is used for reproduction and consumption, and $\vec{r}_{16}$ reveals a vector containing numerical values generated at random from a uniform distribution within the range of 0 to 1, representing the attraction variable. While matched females infrequently attract males, unmatched females do so more frequently. This likelihood, denoted as P_t, is usually mathematically represented in the following way:

$$P_t = r_{17} \times \mu \tag{22}$$

where, P_t means the likelihood of breeding among the females and males, and r_{17} exemplifies a sum arbitrarily shaped using the unchanging distribution among 0 besides 1.

$$\vec{x}_i^{t+1} = \vec{x}_i^{t} \times \vec{U} + \left(x_{11}^{t} + \vec{r}_{18} \times \left(-\vec{x}_{11}^{t} + \vec{x}_i^{t}\right)\right) \times (1 - \vec{U}) \tag{23}$$

where, $\vec{r}_{18}$ is a vector counting numerical standards arbitrarily shaped using the uniform delivery in the variety (0, 1), and $\vec{x}_{11}^{t}$ indicates the charge dimension.

In mathematics, the following Eq. (24) can stand in for the female that consumes the male, either mating or not:

$$\vec{x}_{i}^{t+1} = \vec{x}_{a}^{t} \times \cos(2\pi l) \times \mu \tag{24}$$

where, $\vec{x}_{a}^{t}$ depicts the male. In addition, $\cos(2\pi l)$ is used to give the female some leeway to spin the male around while she eats, with m standing for the male's eaten portion.

To professionally search the key space, the search algorithms in the MSA were designed to balance exploration and exploitation. To strike a compromise between these two extremes, MSA integrates specific search algorithms. While the mechanism emphasizes exploitation through the refinement of potential solutions, the search mechanisms encourage exploration by diversifying the population and investigating new areas. The adaptive search apparatus ensures a process that efficiently explores and exploits the solution space by constantly adjusting the balance between these features. This balance is illustrated by the yellow diamond in the flow diagram, indicating a 50% probability (p) when transitioning from one stage of investigation to another.

4 Results and Discussion

In experiments, two primary datasets were utilized in this study: SARD and NVD. Dataset diversity plays a crucial role in assessing the generalization ability of vulnerability detection models like MDSADNet. This section provides an overview of the diversity aspects covered by the datasets used in the experimental evaluation. Inspired by the predatory and mating behaviors of mantises, MSA provides an efficient mechanism for optimizing hyperparameters in MDSADNet. The detailed steps of MSA are as follows:

Step 1: Initialization. A population of mantises, each representing a candidate solution (i.e., a set of hyperparameters), was initialized. The population size and initial hyperparameters were selected randomly within predefined ranges.

Step 2: Sexual cannibalism. In this stage, poorly performing mantises (based on a fitness function, such as model accuracy) were eliminated. This mimics the natural selection process, ensuring only the strongest candidates survive.

Step 3: Prey pursuit. The surviving mantises adjust their hyperparameters by exploring the search space more thoroughly by combining exploitation (refining existing good solutions) and exploration (searching new areas of the hyperparameter space). The adjustment strategy is guided by gradient-based methods and random perturbations.

Step 4: Prey assault. The best-performing mantises from the prey pursuit stage were further refined to achieve optimal hyperparameters. This involves fine-tuning hyperparameters using techniques like simulated annealing or genetic algorithms to avoid local optima and achieve global optimization.

Step 5: Iteration. The above steps are iterated until convergence, i.e., no significant improvement in the fitness function, is observed or a maximum number of iterations is reached.

4.1 Implementation Particulars

The model performance tests for MDSADNet 4 and MDSADNet 6 were conducted on a Macintosh machine with a 64-bit CPU, 8 GB of RAM, and a Dual-Core Intel Core i5 processor. The operating system was Mac OS X. Python 3.0 was used for all of the experiments. Using MSA as an optimizer, the representations were trained over a 32-mini-batch size. To prevent overfitting, the models were trained over 20 epochs with an early termination and a learning rate of 0.001. The MSA hyperparameter optimization method was used to choose all of these hyperparameters.

4.2 Validation Analysis of the Proposed Model on the SARD Dataset

The SARD dataset, comprising three classes of good, bad, and mixed, was employed to evaluate the efficacy of the model delineated in Figure 1.

Figure 1 shows the visual representation of the proposed model on the SARD dataset. For the good class type, the analysis yields an accuracy of 87, a precision of 90, a recall of 76, and an F1-score of 88. For the bad class type, the results are an accuracy of 85, a precision of 87, a recall of 68, and an F1-score of 87. For the mixed class type, the analysis shows an accuracy of 87, a precision of 90, a recall of 54, and an F1-score of 91.

4.3 Validation Analysis of the Proposed Model on the NVD Dataset

Figure 2 represents the validation study of the proposed model on two different classes of the NVD dataset.

Figure 2 shows the graphical description of the proposed model. For the normal class type, the analysis yields an accuracy of 89, a precision of 91, a recall of 93, and an F1-score of 98. For the vulnerable class type, the results are an accuracy of 90, a precision of 93, a recall of 95, and an F1-score of 97.

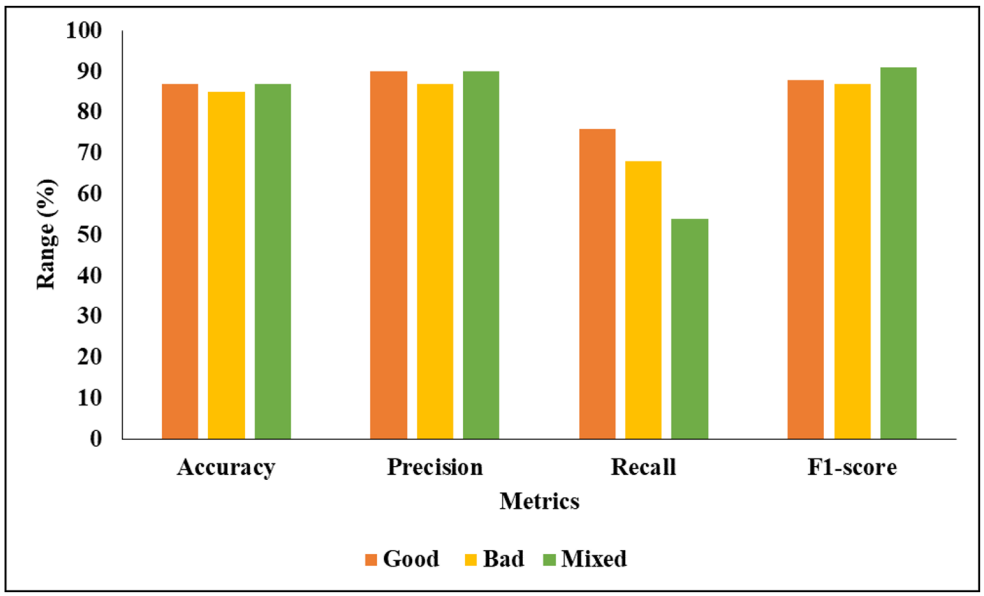

Figure 1. Visual representation of the proposed model on the SARD dataset

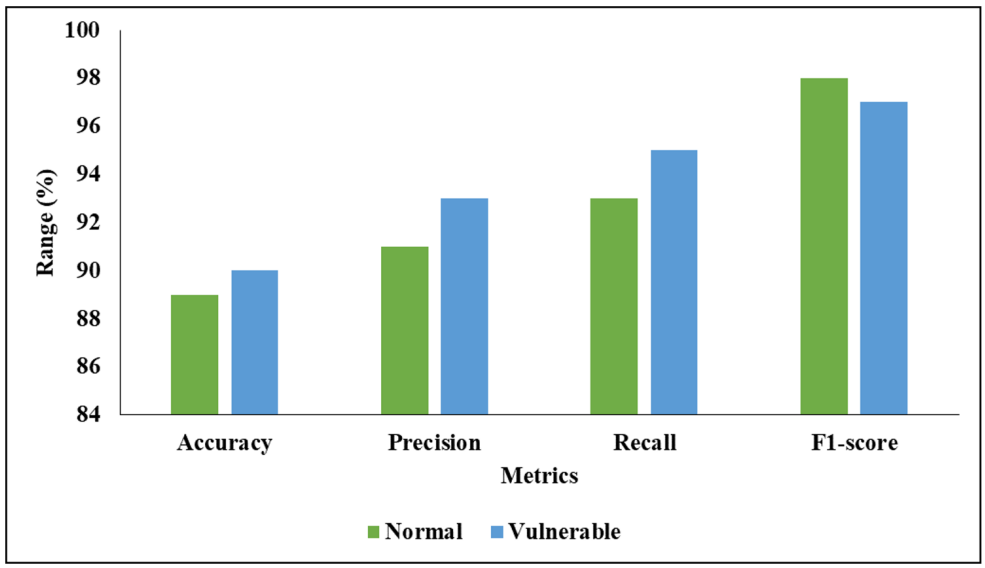

Figure 2. Graphical description of the proposed model

4.4 Learning Rate Analysis

Table 1 presents the validation analysis of the proposed MSA in terms of different metrics. For the 0.1 learning rate, the analysis shows an accuracy of 95.23, a precision of 95.37, a recall of 96.34, and an F1-score of 94.26. For the 0.01 learning rate, the results are an accuracy of 96.48, a precision of 96.50, a recall of 98.14, and an F1-score of 95.20. For the 0.001 learning rate, the analysis shows an accuracy of 98.53, a precision of 97.36, a recall of 99.05, and an F1-score of 97.06.

Table 1. Learning rate analysis

Learning Rate	Accuracy	Precision	Recall	F1-score
0.1	95.23	95.37	96.34	94.26
0.01	96.48	96.50	98.14	95.20
0.001	98.53	97.36	99.05	97.06

4.5 Comparative Analysis

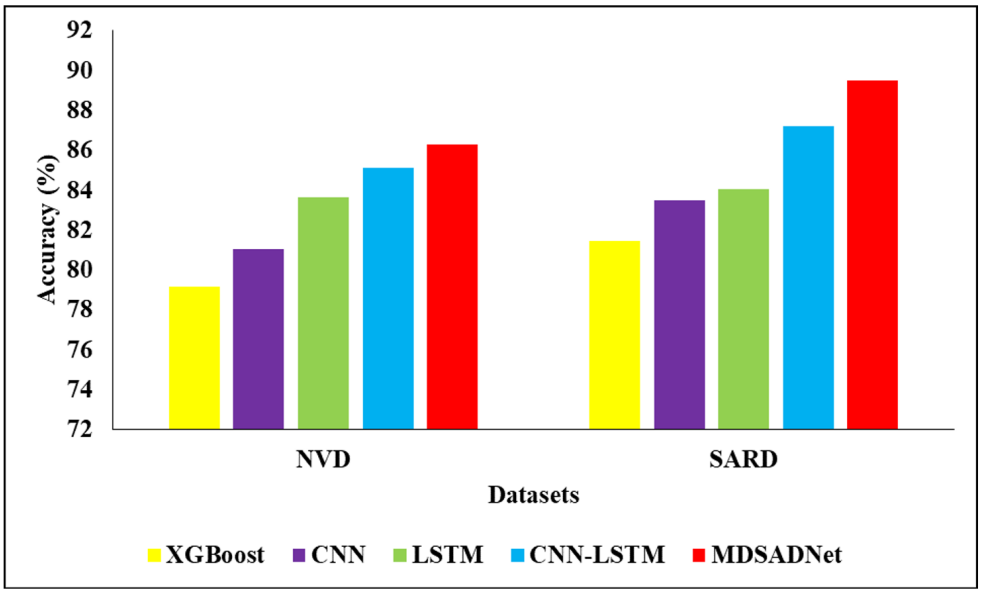

Figure 3. Visual representation of proposed model with existing techniques

Figure 3 shows the visual representation of the proposed model compared with existing techniques, such as eXtreme Gradient Boosting (XGBoost), CNN, Long Short-Term Memory (LSTM) and the hybrid model, on two datasets in terms of accuracy. In the NVD dataset, the XGBoost technique attained an accuracy of 79.16, the CNN model reached 81.04, the LSTM model achieved 83.64, the CNN-LSTM technique reached 85.12, and the MDSADNet model achieved 86.3. In the SARD dataset, the XGBoost technique attained an accuracy of 81.47, the CNN model reached 83.48, the LSTM model achieved 84.06, the CNN-LSTM technique reached 87.19, and the MDSADNet model achieved 89.5.

5 Limitations

This study has several limitations. First, with an exclusive focus on C/C++ program source code vulnerability detection, the system could be adjusted to handle other programming languages and executable formats. Second, the trials cover 93.6% of the vulnerable programs found in SARD, centered on four types of vulnerability syntactic characteristics. This coverage is not comprehensive. It should be noted that SARD data might not be representative of real-world software applications. Third, only one model was used to identify vulnerabilities in the trials, which may limit the generalizability of the findings of this study. Lastly, the proposed method could be improved by pinpointing the specific LOC that contains vulnerabilities more precisely, potentially by detecting them at the slice level.

6 Conclusion

To classify software automation, a DL architecture based on CNNs was introduced in this study. The most notable feature of MDSADNet is its ability to extract both intra-data and inter-data n-gram features. An alternative data input representation, the paragraph matrix, with a two-dimensional CNN model was used in this study. An exhaustive evaluation of the MDSADNet architecture's performance on various datasets was conducted. The parameters of the suggested MDSADNet were optimized using the MSA in a three-stage process. The results demonstrated that MDSADNet outperformed benchmark ML models and state-of-the-art DL representations. The classification performance of the CNN model was enhanced when both intra-data and inter-data characteristics were extracted.

A novel DL architecture, MDSADNet, was introduced for software automation classification in this study. The ability of MDSADNet to extract both intra-data and inter-data n-gram features significantly enhances its classification performance. The proposed model demonstrated superior results compared to existing ML and DL models. This study confirms the potential of using advanced CNN architectures combined with effective optimization techniques like MSA for improved software vulnerability detection.

7 Future Work

Future research should focus on identifying more comprehensive vulnerability syntactic characteristics. Instead of utilizing a single model to identify various types of vulnerabilities, it would be beneficial to develop specialized

models tailored to detect specific types of vulnerabilities. Additionally, the approach could be extended to handle other programming languages and executable formats to improve its applicability. Further studies should also aim to refine the detection process to pinpoint the exact LOC-containing vulnerabilities, potentially through slice-level detection.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] L. Oleshchenko, "Machine learning algorithms comparison for software testing errors classification automation," in *Advances in Computer Science for Engineering and Education VI*, Z. Hu, I. Dychka, and M. He, Eds. Cham: Springer Nature Switzerland, 2023, pp. 615–625. https://doi.org/10.1007/978-3-031-36118-0_55

[2] A. Thirumalraj, R. J. Anandhi, V. Revathi, and S. Stephe, "Supply chain management using fermatean fuzzy-based decision making with ISSOA," in *Convergence of Industry 4.0 and Supply Chain Sustainability*, M. Khan, N. Khan, and N. Jhanjhi, Eds. IGI Global, 2024, pp. 296–318. https://doi.org/10.4018/979-8-3693-1363-3.ch011

[3] A. Abo-eleneen, A. Palliyali, and C. Catal, "The role of Reinforcement Learning in software testing," *Inf. Softw. Technol.*, vol. 164, p. 107325, 2023. https://doi.org/10.1016/j.infsof.2023.107325

[4] A. Fontes and G. Gay, "The integration of machine learning into automated test generation: A systematic mapping study," *Softw. Test. Verif. Reliab.*, vol. 33, no. 4, p. e1845, 2023. https://doi.org/10.1002/stvr.1845

[5] C. Birchler, S. Khatiri, B. Bosshard, A. Gambi, and S. Panichella, "Machine learning-based test selection for simulation-based testing of self-driving cars software," *Empir. Softw. Eng.*, vol. 28, no. 3, p. 71, 2023. https://doi.org/10.1007/s10664-023-10286-y

[6] I. Mehmood, S. Shahid, H. Hussain, I. Khan, S. Ahmad, S. Rahman, N. Ullah, and S. Huda, "A novel approach to improve software defect prediction accuracy using machine learning," *IEEE Access*, vol. 11, pp. 63 579–63 597, 2023. https://doi.org/10.1109/ACCESS.2023.3287326

[7] V. S. Moshkin, A. A. Dyrnochkin, and N. G. Yarushkina, "Automation of software code analysis using machine learning methods," *Pattern Recognit. Image Anal.*, vol. 33, no. 3, pp. 417–424, 2023. https://doi.org/10.1134/S1054661823030318

[8] M. Felderer, E. P. Enoiu, and S. Tahvili, "Artificial Intelligence Techniques in System Testing," in *Optimising the Software Development Process with Artificial Intelligence*, J. R. Romero, I. Medina-Bulo, and F. Chicano, Eds. Singapore: Springer Nature Singapore, 2023, pp. 221–240. https://doi.org/10.1007/978-981-19-9948-2_8

[9] J. P. Appadurai, T. Rajesh, R. Yugha, R. Sarkar, A. Thirumalraj, B. P. Kavin, and G. H. Seng, "Prediction of EV charging behavior using BOA-based deep residual attention network," *Rev. Int. Metod. Numer. Calc. Diseno Ing.*, vol. 40, no. 2, 2024. https://doi.org/10.23967/j.rimni.2024.02.002

[10] P. Talele, S. Apte, R. Phalnikar, and H. Talele, "Semi-automated software requirements categorisation using machine learning algorithms," *Int. J. Electr. Comput. Eng. Syst.*, vol. 14, no. 10, pp. 1107–1114, 2023. https://doi.org/10.32985/ijeces.14.10.3

[11] E. Borandag, "Software fault prediction using an RNN-based deep learning approach and ensemble machine learning techniques," *Appl. Sci.*, vol. 13, no. 3, p. 1639, 2023. https://doi.org/10.3390/app13031639

[12] L. P. G. Nascimento, R. B. C. Prudêncio, A. C. Mota, A. D. A. P. Filho, P. H. A. Cruz, D. C. C. A. D. Oliveira, and P. R. S. Moreira, "Machine learning techniques for escaped defect analysis in software testing," in *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*, 2023, pp. 47–53. https://doi.org/10.1145/3624032.3624039

[13] A. Ramírez, R. Feldt, and J. R. Romero, "A taxonomy of information attributes for test case prioritisation: Applicability, machine learning," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 1–42, 2023. https://doi.org/10.1145/3511805

[14] R. Khan, "Deep learning system and it's automatic testing: An approach," *Ann. Data Sci.*, vol. 10, no. 4, pp. 1019–1033, 2023. https://doi.org/10.1007/s40745-021-00361-w

[15] Y. Gurovich, Y. Hanani, O. Bar, G. Nadav, N. Fleischer, D. Gelbman, L. Basel-Salmon, P. M. Krawitz, S. B. Kamphausen, M. Zenker, L. M. Bird, and K. W. Gripp, "Identifying facial phenotypes of genetic disorders using deep learning," *Nat. Med.*, vol. 25, no. 1, pp. 60–64, 2019. https://doi.org/10.1038/s41591-018-0279-0

[16] I. H. Sarker, "Machine learning for intelligent data analysis and automation in cybersecurity: Current and future prospects," *Ann. Data Sci.*, vol. 10, no. 6, pp. 1473–1498, 2023. https://doi.org/10.1007/s40745-022-00444-2

[17] C. Zhang, H. Liu, J. Zeng, K. Yang, Y. Li, and H. Li, "Prompt-enhanced software vulnerability detection using ChatGPT," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 276–277. https://doi.org/10.1145/3639478.3643065

[18] G. Lu, X. Ju, X. Chen, W. Pei, and Z. Cai, "GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning," *J. Syst. Softw.*, vol. 212, p. 112031, 2024. https://doi.org/10.1016/j.jss.2024.112031

[19] H. Liu, S. Jiang, X. Qi, Y. Qu, H. Li, T. Li, C. Guo, and S. Guo, "Detect software vulnerabilities with weight biases via graph neural networks," *Expert Syst. Appl.*, vol. 238, p. 121764, 2024. https://doi.org/10.1016/j.eswa.2023.121764

[20] M. Fu, C. Tantithamthavorn, T. Le, Y. Kume, V. Nguyen, D. Phung, and J. Grundy, "AIBugHunter: A practical tool for predicting, classifying and repairing software vulnerabilities," *Empir. Softw. Eng.*, vol. 29, no. 1, p. 4, 2024. https://doi.org/10.1007/s10664-023-10346-3

[21] X. Sun, L. Li, L. Bo, X. Wu, Y. Wei, and B. Li, "Automatic software vulnerability classification by extracting vulnerability triggers," *J. Softw. Evol. Process*, vol. 36, no. 2, p. e2508, 2024. https://doi.org/10.1002/smr.2508

[22] X. Wen, C. Gao, F. Luo, H. Wang, G. Li, and Q. Liao, "LIVABLE: Exploring long-tailed classification of software vulnerability types," *IEEE Trans. Softw. Eng.*, vol. 50, no. 6, pp. 1325–1339, 2024. https://doi.org/10.1109/TSE.2024.3382361

[23] S. Nguyen, T. T. Nguyen, T. T. Vu, T. D. Do, K. T. Ngo, and H. D. Vo, "Code-centric learning-based just-in-time vulnerability detection," *J. Syst. Softw.*, vol. 214, p. 112014, 2024. https://doi.org/10.1016/j.jss.2024.112014

[24] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2244–2258, 2022. https://doi.org/10.1109/TDSC.2021.3051525

[25] W. L. Caldas, J. P. P. Gomes, and D. P. P. Mesquita, "Fast Co-MLM: An efficient semi-supervised co-training method based on the minimal learning machine," *New Gener. Comput.*, vol. 36, no. 1, pp. 41–58, 2018. https://doi.org/10.1007/s00354-017-0027-x

[26] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018. https://doi.org/10.48550/arXiv.1801.01681

[27] E. Merdivan, A. Vafeiadis, D. Kalatzis, S. Hanke, J. Kroph, K. Votis, D. Giakoumis, D. Tzovaras, L. Chen, R. Hamzaoui, and M. Geist, "Image-based text classification using 2D convolutional neural networks," in *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, 2019, pp. 144–149. https://doi.org/10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00066

[28] M. Abdel-Basset, R. Mohamed, M. Zidan, M. Jameel, and M. Abouhawwash, "Mantis Search Algorithm: A novel bio-inspired algorithm for global optimization and engineering design problems," *Comput. Methods Appl. Mech. Eng.*, vol. 415, p. 116200, 2023. https://doi.org/10.1016/j.cma.2023.116200