applied sciences

IMPACT FACTOR 2.5

CITESCORE 5.5

Article

A Method for Processing Static Analysis Alarms Based on Deep Learning

Yaodan Tan and Junfeng Tian

MDPI

https://doi.org/10.3390/app14135542

applied sciences

MDPI

Article

A Method for Processing Static Analysis Alarms Based on Deep Learning

Yaodan Tan and Junfeng Tian *

School of Cyber Security and Computer, Hebei University, Baoding 071002, China; tanyaodan2025@gmail.com
* Correspondence: tjf@hbu.edu.cn

Abstract: Automatic static analysis tools (ASATs), also known as static analyzers, have demonstrated their significance and practicability in detecting software defects. ASATs assist developers to identify potential vulnerabilities, errors, and security hazards in source code without executing the software. As software systems grow in scale and complexity, ASATs are replacing manual security audits and becoming crucial for detecting issues in code. However, ASATs often generate numerous warnings with high false positive rates, while developers typically only take measures on a small portion of actionable alarms. To cope with this problem, we propose an innovative method that combines the pre-trained CodeBERT model and neural networks to reduce false positives detected by ASATs. Our approach was evaluated on the Defects4J dataset, which comprises 835 real-world software defects extracted from 17 open-source Java projects. The experimental results explicitly manifest the effectiveness in processing static analysis alarms. By employing a bidirectional recurrent neural network for context embeddings, our approach achieved an accuracy of 95.77% and an AUC score of 98.3%. This research enables developers to minimize false positive alarms and ensure a reasonable number of actionable warnings while guaranteeing software quality and security.

Keywords: static analysis; false alarms; deep learning

check for updates

Citation: Tan, Y.; Tian, J. A Method for Processing Static Analysis Alarms Based on Deep Learning. *Appl. Sci.* **2024**, *14*, 5542. https://doi.org/10.3390/app14135542

Academic Editors: Giuseppe Lacidogna and Keun Ho Ryu

Received: 26 April 2024
Revised: 12 June 2024
Accepted: 20 June 2024
Published: 26 June 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

A myriad of automatic static analysis tools (ASATs) are available in the software development ecosystem. These ASATs are designed to analyze source code or compiled code without executing it, with the aim of identifying potential vulnerabilities, code defects, and security issues in the development process. Nevertheless, one major drawback of ASATs is that they generate numerous alarms with a high false positive rate, which often discourages most developers from adopting these tools [1]. To improve the usability of ASATs, researchers have proposed several approaches for the post-processing of static analysis alarms, and these approaches were divided into six main categories by Muske et al. [2]: clustering, ranking, pruning, the automated elimination of false positives, the combination of static and dynamic analyses, and the simplification of manual inspection. In this study, we propose a novel approach that utilizes warning details and code snippets as input sequences for the deep learning (DL) model to classify and filter the static analysis warnings.

DL has emerged as a prominent technique in the field of software engineering, significantly improving the efficiency and effectiveness of diverse tasks. Its widespread adoption has greatly enhanced the overall performance of software development processes. First and foremost, DL has achieved remarkable success in the field of natural language processing (NLP). The GPT-3 model [3] is pre-trained on a large-scale unsupervised corpus, and it attains impressive performance across various NLP tasks. Similarly, DL has exhibited notable success in the realm of programming language processing, such as in code generation tasks. By training DL models on large codebases, researchers developed the AlphaCode model [4], which can achieve approximately human-level performance on the Codeforces

platform. Furthermore, DL has been utilized in the domain of vulnerability detection to enhance the security of software systems. For instance, Li et al. [5] proposed a systematic DL framework called SySeVR, which was designed to capture program representations that incorporate both syntax and semantic information pertinent to vulnerabilities. In summary, DL provides innumerable merits; however, regrettably, few researchers have investigated its application in processing of static analysis alarms. One possible reason for this is that effectively handling static analysis alarms necessitates specialized domain knowledge and expertise that may not be readily available to many programmers or practitioners. Furthermore, acquiring high-quality ground-truth data for this task can be challenging, as the data often pertain specifically to the software system or ASATs under analysis. In light of the aforementioned factors, we propose a novel approach that leverages the pre-trained model in combination with neural networks to dependably mitigate false positives identified through software analysis.

As illustrated in Figure 1, our proposed approach follows a systematic processing flow to effectively present actionable warnings to developers. We employ supervised learning with expert assistance in this study where the expertise and guidance of domain experts are utilized to enhance the accuracy and effectiveness of the learning process. The process encompasses multiple stages, beginning with the production of static analysis alarms and the collection of developer modifications. Subsequently, we utilize Python script programs to locate java code fragments associated with warning information. Following that, our approach integrates the warning details with code snippets to construct individual input sequences, which are thereafter transformed into context embeddings using the pre-trained model. These generated context embeddings are subsequently employed as inputs for multiple neural network architectures, facilitating the effective identification of actionable warnings. Following extensive training and optimization, neural networks acquired the capability to comprehend and analyze both the warning details and code snippets information encoded within the context embeddings. Our model was trained by neural networks to classify static analysis warnings and provide actionable insights for developers, enabling them to focus on critical issues. The resulting outputs from these models are further processed through fully connected layers for classification, leading to the detection and identification of actionable warnings. Ultimately, the identified actionable warnings are presented to developers for prompt resolution while mitigating the volume of warnings that developers are required to handle. From what has been mentioned above, our approach significantly enhances the efficiency of handing static analysis alarms, thereby improving the software reliability.

This study makes the following principal contributions:

- We propose a novel approach for effectively handling static analysis alarms that combines the utilization of warning features, context embeddings generated by the pre-trained model, and diverse neural network architectures. A reproducible package containing our results can be found at https://github.com/Don2025/ASAT-DL (accessed on 1 June 2024);
- To comprehensively assess the approach's effectiveness, we conducted an evaluation on the Defects4J dataset, which consists of 835 real-world software defects extracted from 17 open-source projects. By utilizing SpotBugs, we generated a total of 152,518 warnings on the bug version of the Defects4J dataset and subsequently located the corresponding code snippets associated with these alarms. This evaluation offers an empirical substantiation of our approach to tackle static analysis alarms, thereby demonstrating its inherent potential to elevate software reliability within real-world development processes;
- Our approach offers practical implications for developers by providing actionable warnings that are more reliable, which facilitates the prompt resolution of critical issues and the improvement of software quality.

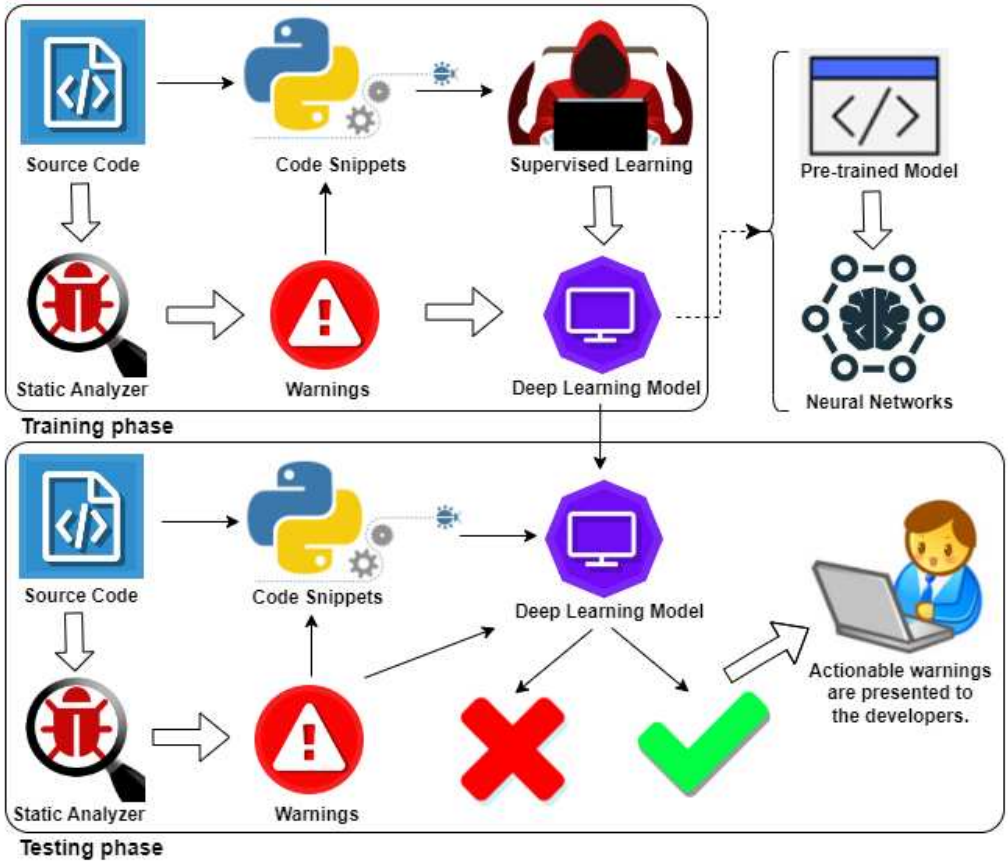

Figure 1. Overview of our proposed approach.

2. Background

In this section, we provide preliminary concepts that are prominent for understanding and implementing our approach.

2.1. Automatic Static Analysis Tools

The initial generation of ASATs for Java, e.g., FindBugs [6,7], PMD (found at https://pmd.github.io (accessed on 3 January 2024)), and Checkstyle (found at https://checkstyle.org (accessed on 3 January 2024)), were developed to detect potential issues or violations of coding standards. These ASATs made significant contributions when they were initially presented but also had serious limitations, particularly in terms of generating numerous false alarms [8,9]. FindBugs is a widely recognized open-source static analysis tool that specializes in identifying potential defects for Java code, such as null pointer dereferences, resource leaks, and violations of coding conventions. Although FindBugs is now an abandoned project, its successors are still adopted by major software companies, e.g., Google [10–12], Facebook [13,14]. These ASATs were typically designed as an analysis framework based on various forms of static analysis, which allows them to effectively analyze complex programs and scale to handle large codebases. According to the study conducted by Vassallo et al. [15], 71% of developers pay attention to different warning categories depending on the development context. However, the abundance of false alarms generated by ASATs has led to the unwillingness of developers to configure and adopt them [16]. Habib et al. [17] have compared three popular ASATs to detect bugs, including Google's Error Prone [11], Facebook's Infer [13], and SpotBugs (accessed at https://spotbugs.github.io (accessed on 3 January 2024)). We chose SpotBugs as the static analysis tool after comprehensive consideration; this is a fork of FindBugs that carries on from the point at which it left off.

2.2. Transformer Architecture

The Transformer architecture [18] was introduced to convert collected data into a suitable format for further analysis, which was first proposed in 2017 to efficiently solve sequence-to-sequence tasks, even when dealing with long-range dependencies. It consists of an encoder–decoder structure and shares common internal components, such as attention mechanisms, feedforward layers, and normalization layers. Wan et al. [19] discovered that through attention analysis, the attention weights learned by the Transformer architecture exhibit a strong alignment with the structural patterns found in an abstract syntax tree (AST). Moreover, the attention mechanisms in different heads and layers of the Transformer architecture focus on the positions and content of source code tokens in distinct ways. Therefore, the Transformer architecture possesses the capability to capture and learn the syntax information of source code. In this study, we selected the bimodal pre-trained CodeBERT model [20] for our approach, as it leverages both source code and natural language descriptions during training and adopts a multilayer bidirectional Transformer as its architecture. The utilization of CodeBERT to generate context embeddings serves as a foundational step in our approach, facilitating the subsequent stages of alarm processing and classification.

2.3. Neural Network Architecture

As mentioned earlier, we take warning details and code snippets as inputs to utilize CodeBERT to generate context embeddings, which are then employed for the classification of alarms. We present a comparative empirical study of the multilayer perceptron (MLP) [21], the attention mechanism [22], recurrent neural networks (RNNs) [23], and convolutional neural networks (CNNs) [24] for the classification of false alarms and actionable warnings.

The MLP is a feedforward neural network that comprises multiple layers of nodes, with each layer being connected to the previous one through a set of weights. The MLP is a relatively simple algorithm that uses multiple layers of nodes to make predictions based on input features, but it may not perform as well as other more complex models. This is due to the limited capabilities of the MLP in capturing the entirety of information contained within the input features, making it challenging for the model to learn and comprehend more intricate patterns present in the data. The MLP was accordingly used as a baseline model to establish a performance benchmark for comparing performance with the following models.

The attention mechanism is a neural network architecture that can selectively focus on different parts of the input sequence to enhance classification performance. In contrast to the baseline model, the attention mechanism facilitates the model to emphasize notable features or regions of input data while suppressing less important ones. This can result in improved accuracy when distinguishing false positives from actionable warnings. Our findings indicate that the attention mechanism significantly enhances the performance of the model, particularly in the identification of actionable alarms.

RNNs are particularly well suited to modeling sequential data, as they can maintain a hidden state that is updated at each time step. Long short-term memory (LSTM) and the gated recurrent unit (GRU) [25] are two variants of RNNs that have been specifically designed to address the limitations of traditional RNNs and efficiently capture long-term dependencies in input sequences. The GRU has a simpler structure with fewer parameters, which simplifies the LSTM by combining the input and forget gates into a single update gate, resulting in faster training and improved computational efficiency. In this study, we define a bidirectional recurrent neural network (BiRNN) [26] architecture that includes a bidirectional GRU layer and a fully connected layer for classification. The bidirectional GRU layer comprises two sets of GRU layers: one processes the input sequence from left to right, while the other processes it from right to left. This configuration allows the model to take contextual information into account and effectively model the input sequence bidirectionally. The BiRNN model accepts context embedding sequences as the inputs and forwards them through the bidirectional GRU layer, which fully encompasses an encoded representation of

the context information in its output. The output of the GRU layer is ultimately fed into the fully connected layer for classification to recognize actionable warnings.

While the CNNs are often associated with computer vision tasks like image recognition, they have also been successfully applied to NLP tasks such as sentence classification [27,28] by treating text data as a one-dimensional sequence of word embeddings, where each word is represented by a vector in a high-dimensional space. In our work, the CNNs are used to apply convolutional operations to the sequence of context embeddings in order to identify local patterns or features within each input sentence. These features are then passed through a pooling layer to reduce their dimensionality before being fed into a fully connected layer for classification. By applying convolutional operations to context embeddings, the CNNs can learn to recognize local patterns or features in alarm data, which is valuable for distinguishing actionable alarms.

3. Methodology

In this section, we describe the DL models that were studied, along with how we encoded the warning details and code snippets as inputs to the neural networks.

3.1. CodeBERT Model and Data Representations

In this study, we selected a representative pre-trained CodeBERT model, which is a bimodal model pre-trained on source code and natural-language descriptions. CodeBERT follows the BERT [29] model and adopts a multilayer bidirectional Transformer as the model architecture, which is composed of 12 layers of Transformer with 12 attention heads, and the size of the representation in each Transformer layer was set to 768. It was pre-trained on CodeSearchNet [30], a large-scale of code corpora collected from GitHub across six programming languages. To generate each training input sequence, we took the concatenation of warning details and code snippets from the corpus, defined as $warning_1 = \{w_1, w_2, \cdots, w_n\}$ and $code_1 = \{c_1, c_2, \cdots, c_m\}$, where n and m denote the lengths of two segments, respectively. The two segments were always connected by a special separator token [SEP]. The initial and final tokens of each sequence were consistently padded with a special classification token [CLS] and a concluding token [EOS], respectively. There is an example in Figure 2 to explain how the warning details and code snippets are normalized and tokenized in vector form.

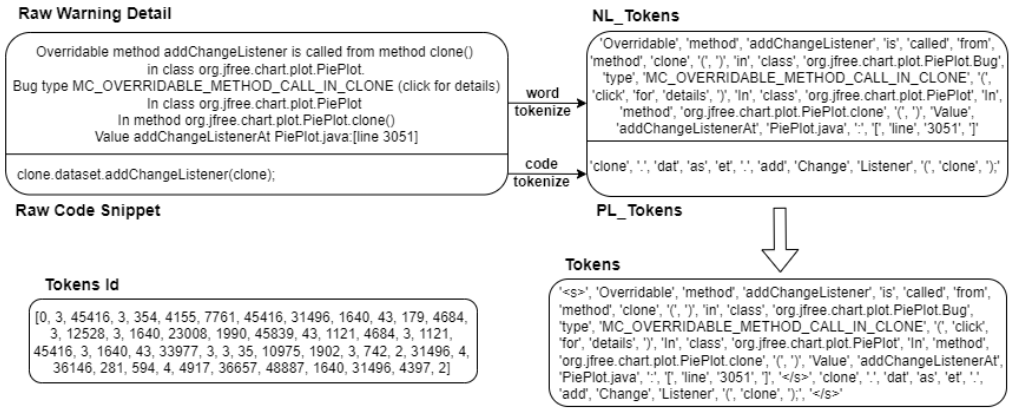

Figure 2. The warning details and code snippets are normalized and tokenized in vector form.

In conclusion, the input sequence of each pre-training sample is represented as follows:

$$token = [\text{CLS}], \underbrace{w_1, w_2, \ldots, w_n}_{warning_1}, [\text{SEP}], \underbrace{c_1, c_2, \ldots, c_m}_{code_1}, [\text{EOS}].$$

As shown in Figure 3, we propose an innovative method that takes warning details and code snippets as input sequences and then utilize the CodeBERT model to convert

them into a suitable format for further analysis. The tokenizer parses the input sequences into individual tokens, which allows the model to process the input in a more efficient and structured way. Once the input has been tokenized, each token is subsequently transformed into a vector through word embedding. Word embedding is a way of representing words as high-dimensional vectors, where each dimension represents a different semantic feature of the word. Then, we perform positional encoding on the input embeddings, which enables the model to more effectively capture the relative position information of each token within the original input sequence. After position encoding, the input sequence can be passed through multiple Transformer layers. Each Transformer layer typically contains a multi-head attention mechanism and a feedforward neural network, both of which help the model capture the semantics of the input sequence. In addition, the Transformer introduces layer normalization operations on the basis of residual connections to improve the performance and stability of the model. After passing through multiple Transformer layers, we obtain context embeddings, which are high-dimensional vector representations of the input sequence. The context embeddings are obtained by passing the output of the final Transformer layer through a pooling operation. The pooling operation aggregates the output vectors of the final layer into a single fixed-length vector, which captures the most relevant information about the input sequence. Once the context embeddings are obtained from the CodeBERT model, we can use them as input features for neural networks to predict whether alarms are actionable warnings.

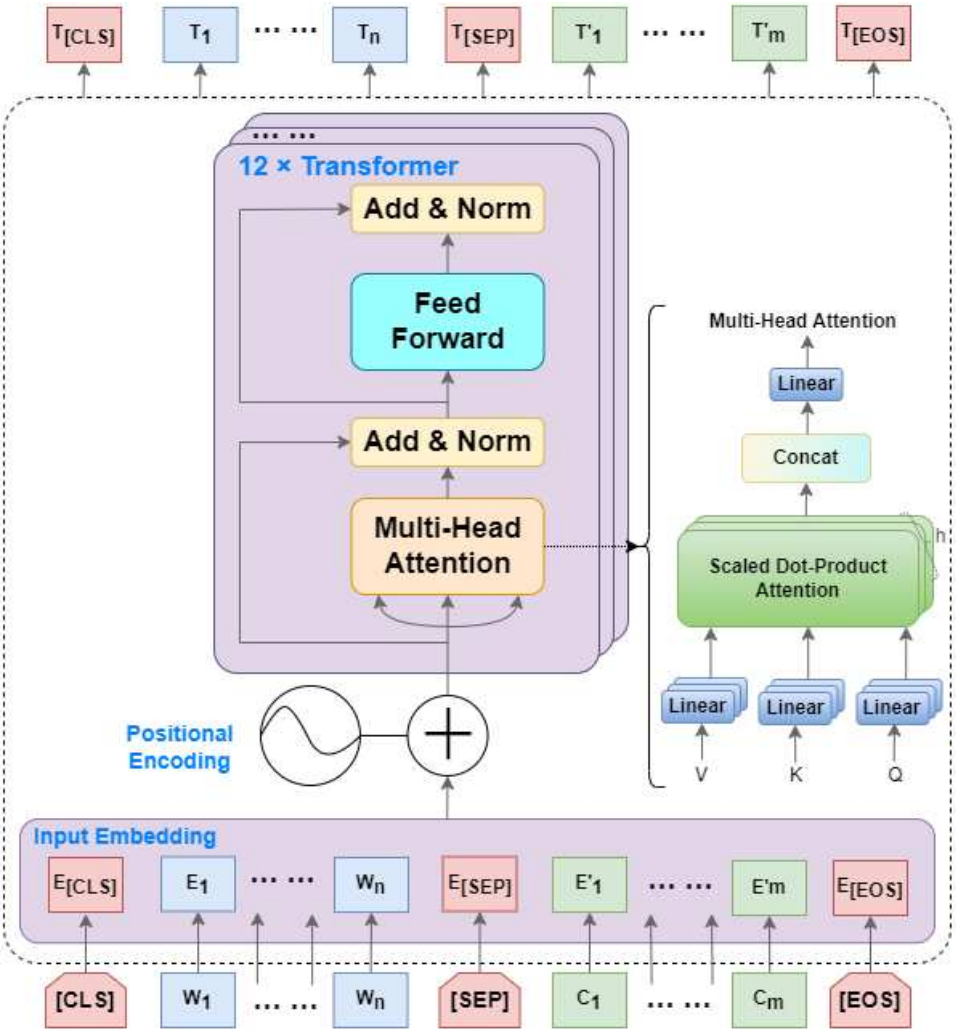

Figure 3. The process of generating context embedding for code snippets and warning details based on pre-trained CodeBERT model.

3.2. Classification Algorithms

As mentioned in Section 2.3, our study compares the performance of four different DL algorithms for alarm classification. We conducted an empirical evaluation to assess the effectiveness of each algorithm and compared the results to determine which algorithm performed best in this work.

3.2.1. Multilayer Perceptron

In this study, we employed the MLP as our baseline model to classify a set of input features, as it uses multiple layers of nodes to make predictions based on input features. The *MLPClassifier* is a feedforward neural network model designed to determine the classification of a specific static analysis alarm. It consists of two fully connected layers, namely, *fc1* and *fc2*, which perform linear transformations on the input data. The *fc1* layer maps the input features from the *input_size* dimension to the *hidden_size* dimension, followed by a rectified linear unit (ReLU) activation function to introduce nonlinearity, while the *fc2* layer further maps the output of *fc1* from the *hidden_size* dimension to a two-dimensional space, producing the final classification output. As is shown in Figure 4, the model architecture consists of an input layer with 768 features, a hidden layer with 128 neurons, and an output layer with 2 neurons. The input features consist of context embeddings generated by CodeBERT, which capture contextual and semantic information related to the analyzed warning details and code snippets. The input layer receives the input data with its 768 features, and the hidden layer performs computations and feature extraction. Finally, the output layer produces the predicted outputs, representing the two classes for classification. To train the MLP model, we utilize a ground-truth dataset of warning data, where each alarm is annotated as either true positive or false positive. In summary, the *MLPClassifier* is a two-layer MLP architecture that leverages linear transformations and nonlinear activation to extract relevant features and perform binary classification tasks. It serves as a baseline model for comparing performance with the following well-established neural network architectures.

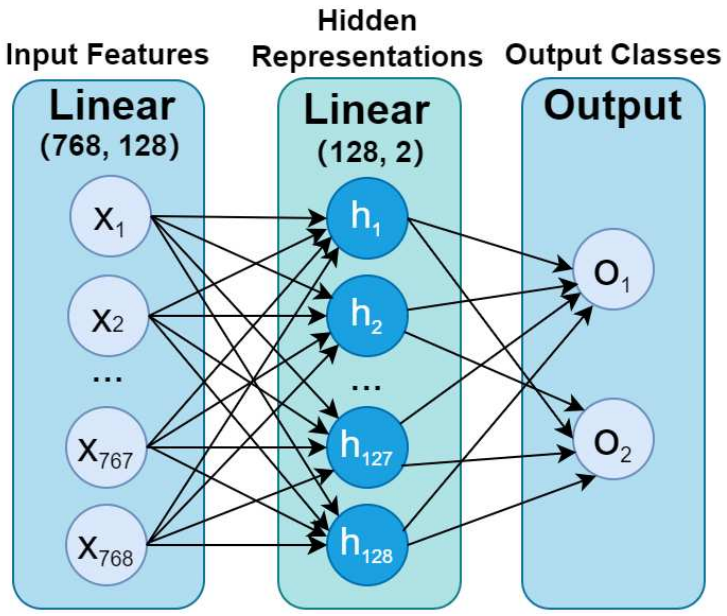

Figure 4. Architecture diagram of MLP.

3.2.2. Attention Mechanism

We evaluated the performance of an attention-based method in comparison with the baseline model MLP. In this study, we customized a class called *AttentionClassifier*, which is a classifier with an attention mechanism that was inherited from *nn.Module*, consisting of two linear layers called *attention* and *classifier*. The *attention* layer calculates the weights

of each input feature, while the *classifier* layer generates predicted categories based on the weighted sum of all input features. During the forward function, the *input_ids* are propagated through the *attention* layer, resulting in a tensor that represents the attention weights assigned to each input feature in the sequence. Subsequently, the *squeeze* function is applied to compress the last dimension of the tensor, and the *softmax* function is used to normalize the weights, yielding the weights of each input feature in the original sequence. Once the *attention_weights* are acquired, we utilize the broadcast mechanism to multiply the *input_ids* and *attention_weights* to calculate the importance or relevance of each feature within the context embeddings. By multiplying the input features with their associated attention weights, this model effectively highlights significant features while attenuating less important ones, enabling it to focus on the most relevant information within the input sequence for improving prediction accuracy. The *attention_output* is a weighted representation of the input features and is obtained by multiplying each feature with its corresponding attention weight. The weighted sum of *attention_output* integrates the importance assigned to each feature by the attention mechanism, efficiently capturing the most salient features and enhancing prediction accuracy. The resulting feature representation emphasizes crucial information of the original sequence, which is subsequently fed into the *classifier* layer to make the final predictions. In summary, this model utilizes an attention mechanism to assign weights to the input features, enabling the classifier to effectively comprehend the essential information contained within the *input_ids*, thereby enhancing classification performance.

3.2.3. Bidirectional Recurrent Neural Networks

BiRNNs consist of two separate RNNs: one processes the input sequence in forward order while the other processes it in reverse order. At each time step, the outputs of the forward and backward RNNs are concatenated, resulting in a final output that represents the combined information from both directions. By processing the sequence in both directions, the model is able to capture dependencies between input features that may be missed by a unidirectional RNNs. In this study, we implemented a customized class *BiRNN* that is the classifier inherited from *nn.Module*, and it comprises a bidirectional GRU layer and a fully connected layer for predicting alert categories. The bidirectional GRU layer contains two directions of GRU layers that process the input sequence in both forward and backward directions simultaneously, enabling the model to consider contextual information. The GRU layer utilizes gating mechanisms to regulate the flow of information within the neural network, enabling selective updates and disregarding of specific aspects of the hidden state depending on the input at each time step. Mathematically, for a given time step t, suppose that the input is a minibatch $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ (number of examples: n; number of inputs: d) and the hidden state of the previous time step is $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$ (number of hidden units: h). As illustrated in Figure 5, GRU has only two gates: the reset gate $\mathbf{R}_t \in \mathbb{R}^{n\times h}$, which controls whether to update using the previous hidden state H_{t-1} or the current input X_t, and the update gate $\mathbf{Z}_t \in \mathbb{R}^{n\times h}$, which determines whether to update the hidden state H_t (set to 1) or not (set to 0).

These calculations can be represented by the following formula:

$$\mathbf{R}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r),$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z),$$

where $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h\times h}$ are weight parameters, $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1\times h}$ are biases, and σ is the fully connected layer with activation function. The reset gate $\mathbf{R}_t$ leads to the following *candidate hidden state* $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n\times h}$ at time step t:

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\mathbf{W}_{hh} + \mathbf{b}_h),$$

where $\mathbf{W}_{xh} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$ are weight parameters, $\mathbf{b}_h \in \mathbb{R}^{1\times h}$ is the bias, and the symbol $\odot$ is the Hadamard product operator. The update grate $\mathbf{Z}_t$ determines the extent to

which the new hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ is just the old state $\mathbf{H}_{t-1}$ and by how much the new candidate state $\tilde{\mathbf{H}}_t$ is used. This leads to the final update equation for the GRU:

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t.$$

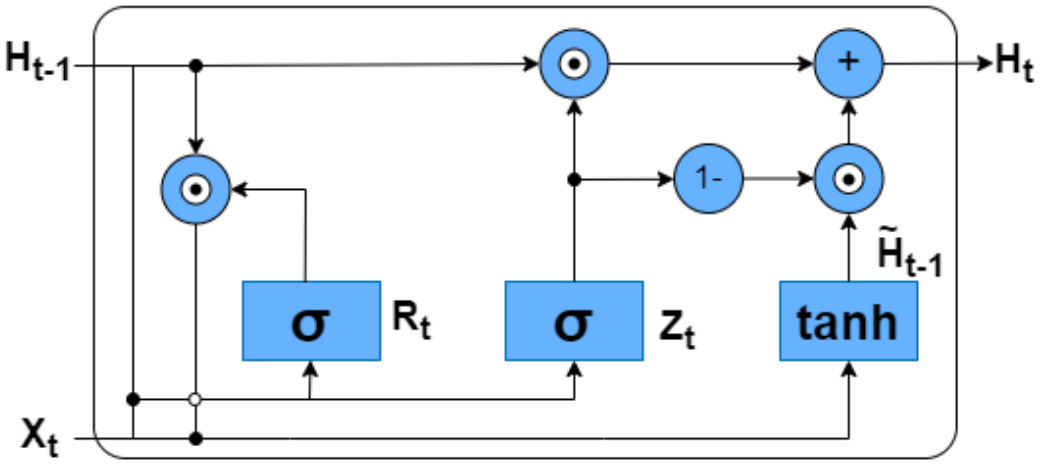

Figure 5. The illustration of a gated recurrent unit (GRU).

Whenever the update gate $\mathbf{Z}_t$ is close to 1, we simply retain the old state. In this case, the information from $\mathbf{X}_t$ is essentially ignored, effectively skipping time step t in the dependency chain. In contrast, whenever $\mathbf{Z}_t$ is close to 0, the new latent state $\mathbf{H}_t$ approaches the candidate latent state $\tilde{\mathbf{H}}_t$. In summary, the BiRNN model receives sequences of context embeddings as input and forwards them through the bidirectional GRU layer. The output of the GRU layer contains an encoded representation of the contextual embeddings in both the forward and backward directions, and it is then fed into a fully connected layer for classification to identify actionable warnings based on the encoded contextual information.

3.2.4. Convolutional Neural Networks

In our work, the CNNs are employed to apply convolutional operations to the sequence of context embeddings in order to identify local patterns or features within each input sentence. The CNNs employ convolutional layers to capture local patterns or features in the input sequence of context embeddings. The input sequence is structured as a matrix, where each row corresponds to a word embedding and each column represents a dimension of the embedding vector. The convolutional layers utilize one-dimensional convolutions to process these matrices, where each filter performs a dot product with the input at each position, generating a feature map. These feature maps are then passed through a pooling layer to reduce their dimensionality before being fed into a fully connected layer for alarm classification. By applying convolutional operations to context embeddings, CNNs can learn to recognize local patterns or features in alarm data, which are valuable for distinguishing between false positives and actionable alarms. Figure 6 illustrates the architecture of the CNN model, as well as the overall classification procedures.

To elucidate this, we implement a customized class named *CNN*, which is a classifier inherited from the *nn.Module* class in which context embeddings undergo convolution along the second dimension. Following the application of multiple convolutional filters, the resulting outputs are then subjected to max-pooling, which serves to reduce the dimensionality of the features while preserving the most important ones. Finally, the resulting features are passed through the fully connected layers for classification, enabling the prediction of whether the alarms are actionable.

In general, we utilized the pre-trained CodeBERT model to generate context embeddings for the warning details and corresponding code snippets, which served as inputs to the classification algorithms mentioned above. The neural network architectures were

employed for classification to determine whether an alarm was an actionable warning or a false alarm. A detailed description of the performance of these four classifiers will be provided in the next section.

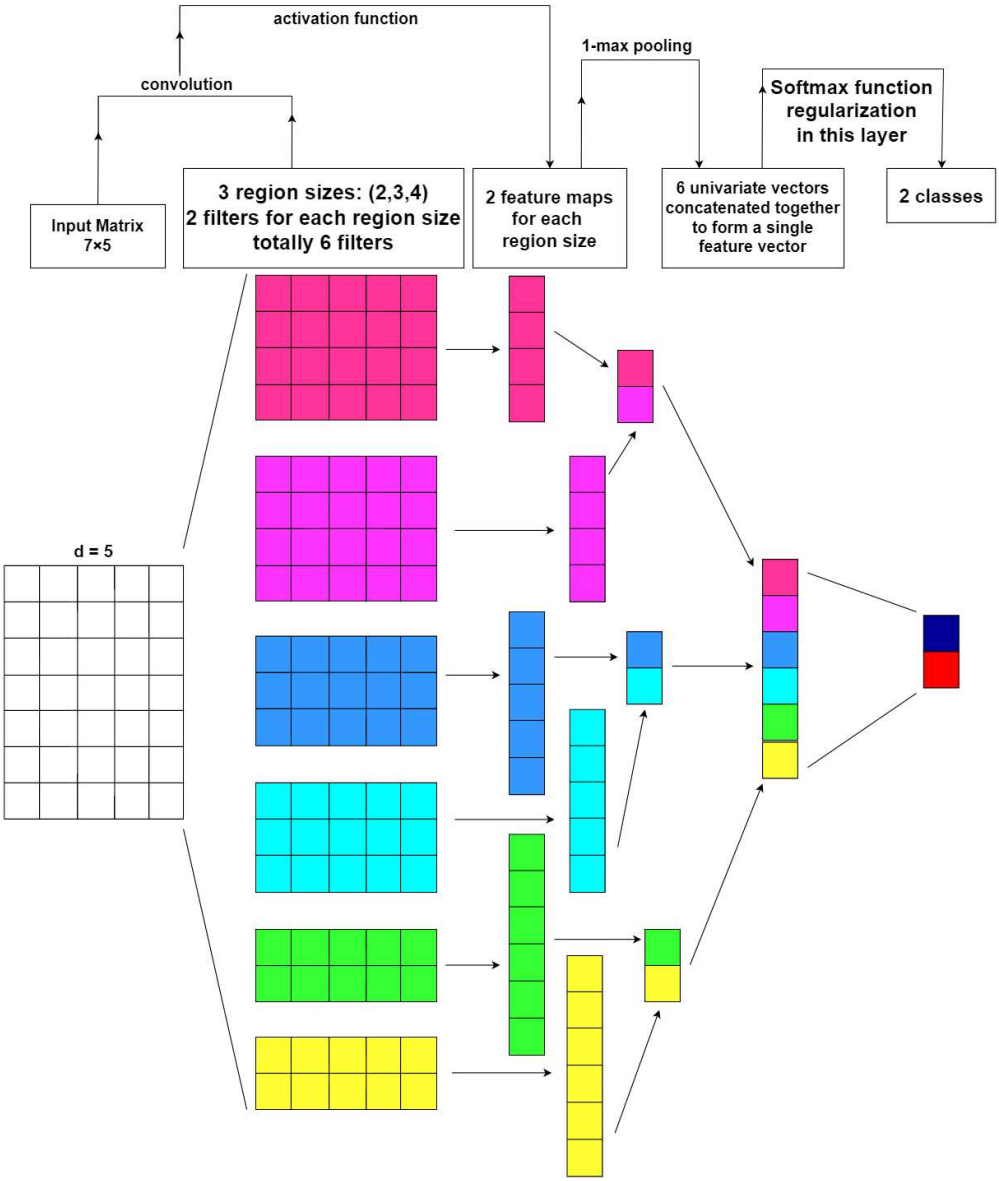

Figure 6. The overall classification process for the CNN model.

4. Dataset

4.1. Data Selection

Defects4J [31], a widely employed Java benchmark for testing and evaluating program repair tools, can also serve as a reliable benchmark for validating the effectiveness of processing methods for static analysis alarms. The Defects4J dataset contains 835 real-world software defects extracted from diverse open-source projects. Table 1 presents in detail the open-source projects we used for training.

There are several reasons why the Defects4J dataset is suitable for validating alarm processing methods:

- Real-world samples: By encompassing genuine defects from real-world software projects, Defects4J significantly ensures that the validation of alarm processing methods is both reliable and practically meaningful, reflecting the challenges encountered

in software development and maintenance. This helps in assessing the applicability and usefulness of these methods in real-world software development scenarios;
- Diversity and complexity: Defects4J covers multiple open-source projects and domains, encompassing a wide range of software defect types, e.g., logic errors, boundary condition errors, and data processing errors. This diversity facilitates a comprehensive evaluation of processing methods, ensuring their effectiveness across different defect types and software contexts;
- Annotation and verification: Defects4J not only includes the bugs themselves but also the corresponding bug fixes implemented by actual developers to facilitate the evaluation of processing methods;
- Open and shared: Defects4J is a publicly available and shared dataset, allowing researchers and practitioners to freely access and utilize it for validation purposes.

In conclusion, the utilization of the Defects4J dataset offers these advantages to enhance the reliability and practicality of the processing method for static analysis alarms.

Table 1. Detailed open-source projects that we used for training.

Identifier	Project Name	Number of Bugs	Used Bug Ids
Chart	jfreechart	26	1–26
Cli	commons-cli	39	1–5, 7–40
Closure	closure-compiler	174	1–62, 64–92, 94–176
Codec	commons-codec	18	1–18
Collections	commons-collections	4	25–28
Compress	commons-compress	47	1–47
Csv	commons-csv	16	1–16
Gson	gson	18	1–18
JacksonCore	jackson-core	26	1–26
JacksonDatabind	jackson-databind	112	1–112
JacksonXml	jackson-dataformat-xml	6	1–6
Jsoup	jsoup	93	1–93
JxPath	commons-jxpath	22	1–22
Lang	commons-lang	64	1, 3–65
Math	commons-math	106	1–106
Mockito	mockito	38	1–38
Time	joda-time	26	1–20, 22–27

4.2. Data Preprocessing

We wrote several Python scripts for data preprocessing. Initially, we checked out all of the buggy source code versions from the Defects4J dataset, and then compiled the sources and ran tests. It is noteworthy that Defects4J does not provide built-in support for generating static analysis alarms. Therefore, we relied on SpotBugs to generate the static analysis alarms. Once the compilation and testing processes of the Defects4J dataset were completed, we executed SpotBugs on each working directory to generate HTML reports. By employing SpotBugs, we successfully generated a total of 152,518 warnings on the bug version of the Defects4J dataset. Subsequently, we proceeded to extract alarm features from the HTML reports and store them in several CSV files. This involved extracting alarms and categories and obtaining their associated details, along with the corresponding code lines. Moreover, we collected the corresponding code snippets related to these alarms for further analysis and processing.

4.3. Data Transformation

We performed a series of operations on the existing data to filter out repetitive alarms and assign ground-truth labels to actionable alarms that require attention from developers. These operations played a crucial role in refining the dataset and ensuring accurate and effective learning from the data. Initially, we merged data from different projects within the Defects4J dataset to create a consolidated dataset that contained all of the relevant

information required for the model. This stage allowed us to have a comprehensive and diverse dataset for training and evaluation. We made changes to the existing data to ensure that they were in the desired format for the DL model. These changes involved removing redundant information and correcting data errors or inconsistencies. In addition, we leveraged the version-specific properties known as *classes.modified*. These properties helped us determine whether an alarm required high attention from developers by indicating whether the code associated with a warning had undergone any modifications. If the code has been modified, it suggests that the alarm may require attention from developers. On the other hand, if the code has not been modified, the alarm may be considered less critical or to have already been addressed in previous development iterations. By utilizing the modification and repair records of developers, as well as the auditing of alarm data by professionals, we generated a portion of ground-truth for supervised learning to effectively distinguish actionable warnings. Data splitting is crucial to ensure that the model is trained and evaluated on separate, independent subsets of data to avoid overfitting and obtain a more accurate assessment of its generalization performance. Specifically, we partitioned the dataset in such a way that 80% of the data constituted the training set, while 10% was allocated to both the test and validation sets. The training set was used to train the model, while the validation set was used to fine-tune the model and select the best hyperparameters; in addition, the test set was used to evaluate the performance of the trained model. After performing the aforementioned operations, we ensured the effective preparation and utilization of the data for training and testing the DL model.

5. Experiments and Results

To evaluate the performance of the above algorithms, we present the empirical results in this section for the verification of the effectiveness and reliability of our method.

5.1. Experimental Setup

5.1.1. Hardware and Software

We used Pytorch 1.13 [32] with CUDA 11.6 on top of Python 3.10 for all the experiments, which were conducted on a Linux server with a single 10 GB RTX 3080 graphics card, 64 GB RAM, and 2 Intel Xeon W-2295 18-core CPUs @ 4.6GHz running Ubuntu 20.04 focal.

5.1.2. Performance Criteria

To quantitatively evaluate the performance of our approach, we employed several commonly used evaluation metrics, including accuracy, precision, recall, and F1-score. These metrics provided a holistic assessment of the model's performance in accurately classifying actionable warnings and false alarms. Accuracy is a commonly used metric for evaluating the performance of classification models, and it measures the proportion of correctly classified samples with respect to the total number of samples (all generated warnings). A higher accuracy score indicates better performance of the model in correctly classifying the samples. Precision and recall are two widely used measures for evaluating the completeness and exactness of a classifier, while the F1-score is a balanced metric that takes the harmonic mean of the precision and recall, providing a more comprehensive evaluation of how well the classifier performed. Furthermore, we utilized the receiver operating characteristic (ROC) curves, which graphically illustrate the performance of classification models at diverse threshold settings. While varying the threshold from 0 to the maximum based on the predicted results of the learner, the ROC curves visually depict the trade-off between the true positive rate (TPR) and the false positive rate (FPR) for different classification thresholds. The ROC curves help to compare the performance of different classification models, while the area under curve (AUC) summarizes the overall performance of the model across all possible threshold settings. The AUC score is a comprehensive metric for evaluating the overall performance of classification models across various thresholds, and it ranges from 0 to 1, with higher values indicating better performance. It provides a measure of the trade-off between model classification accuracy and the true positivity

rate. A model with a higher AUC generally indicates better performance in distinguishing between positive and negative instances. By analyzing these evaluation metrics, we can assess the effectiveness and reliability of our DL-based approach in accurately identifying actionable alarms from static analysis warnings.

5.1.3. Training Configuration

To extract context embeddings from the CodeBERT model, we passed the warning details and code snippets through the pre-trained CodeBERT model. The context embeddings were then obtained from the last hidden layer of the model. These context embeddings served as the input sequences for training the downstream classifiers. We experimented with different hyperparameters and training configurations to optimize the performance of our classifiers. Specifically, we employed the AdamW [33] optimizer and a linear learning rate scheduler to fine-tune the training process. All of the classifiers were trained for 20 epochs using a binary cross-entropy loss and the AdamW optimizer with a learning rate of 1×10^{-3}. Both the training and validation processes were performed on the same dataset with a batch size of 32. This allowed us to iteratively update the model parameters and evaluate the performance during the training phase. By tuning these hyperparameters and training configurations, we aimed to achieve the best possible performance for our classifiers in accurately classifying actionable alarms.

5.2. Experimental Design

The Defects4J dataset, in conjunction with the static analysis tool SpotBugs, was employed to generate alarms. This dataset comprised 835 bugs and associated fixes that were reproducible, resulting in a total of 152,518 warnings. The code snippets corresponding to these warnings were subsequently identified. The original data were stored in CSV files, facilitating direct loading and access. After removing redundant data, the dataset was divided into three sets: a training set, a test set, and a validation set. The training set contained 80% of the data, while the remaining 20% were allocated to both the test and validation sets, with 10% assigned to each. It is crucial to emphasize that the validation set was distinct from the test set, as it served as an independent dataset for the final evaluation of the model's performance after all adjustments had been made.

To process experimental data, we implemented a class named *dataset* that loads the data and performed tokenization. The dataset class applied distinct tokenization techniques for the warning details and code snippets. Word tokenization was utilized for the warning details, while code tokenization was employed for the code snippets. As a result, we obtained two sets of tokens: NL_tokens (tokens for natural language) and PL_tokens (tokens for programming language). To concatenate these tokens into a single token sequence, we added special tokens, such as [CLS] (for the start of the sequence), [SEP] (to separate the natural language and programming language tokens), and [EOS] (for the end of the sequence), which ensured that the token sequences were properly formatted and could be processed by the models effectively. The distribution of token sequence lengths is shown in Figure 7. To accommodate the majority of token sequences with a length less than 150, we defined the maximum sequence length (*max_seq_length*) as 150. This meant that any token sequence with more than 150 tokens would be truncated to fit the specified length, while those with less than 150 tokens would be filled with pads. By setting an appropriate *max_seq_length*, we could ensure that all input sequences were processed effectively by the models without exceeding their memory capacity or introducing unnecessary computational overheads.

The subsequent steps entailed generating context embeddings using the pre-trained CodeBERT model, followed by selecting neural networks with diverse architectures for the training process. These context embeddings captured both the syntactic and semantic information of code snippets, allowing the model to comprehend the context and relationships between diverse code elements. Following the generation of context embeddings, we proceeded with the selection process for neural networks with diverse architectures. This

step allowed us to leverage the unique advantages and capabilities of different architectures, maximizing the performance and accuracy of processing static analysis alarms. By carefully considering and comparing different architectures, we could effectively handle the complexities and challenges associated with processing static analysis alarms, ultimately leading to more accurate and reliable results.

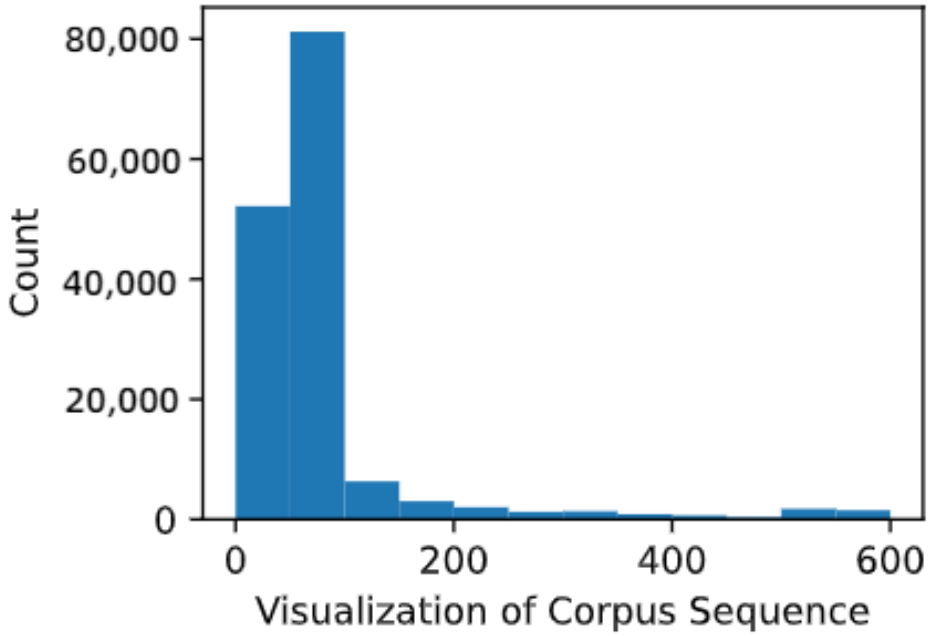

Figure 7. Distribution diagram of token sequence lengths.

5.3. Research Questions and Findings

With the above experimental setup and design, our study aimed to answer the following research questions:

- What are the current limitations of existing approaches for processing static analysis alarms?
- How effective is our proposed DL-based approach in accurately identifying actionable alarms from static analysis warnings?
- What are the benefits of using a DL-based approach to handle with static analysis alarms?

By tackling these research questions, our objective was to assess the effectiveness and feasibility of employing DL for processing static analysis alarms, providing insights into its potential advantages and limitations in this work.

5.3.1. RQ1: Motivation

There are myriad limitations in existing approaches for processing static analysis warnings; for example, they often rely on manually crafted rules or patterns that may not capture all possible issues or may generate false positives due to the complexity of modern software systems. On the other hand, DL techniques could be used to automatically learn complex patterns and relationships in software code and can be trained to recognize a wide range of issues with high accuracy. By leveraging the power of DL to learn from large amounts of data, one can overcome the limitations of existing methods and improve the accuracy and efficiency of alarm processing. For example, DL models can be trained on large datasets of ground-truth to automatically identify and classify alarms with high accuracy. Additionally, DL-based approaches could be used to generate more precise and actionable alarms, reducing the time and effort required for manual analysis and reducing false positive rates.

5.3.2. RQ2: Effectiveness

To demonstrate the effectiveness of our approach, we conducted a comprehensive evaluation of four neural network algorithms for classifying context embeddings generated from warning details and code snippets.

Table 2 accurately reflects the performance of our approach; it provides a comparison of the accuracy, precision, recall, and F1-score metrics between the four different algorithms.

Table 2. Comparison of performance between different models in our experiment.

Model	Accuracy	Precision	Recall	F1-Score
MLP	76.38%	88.15%	72.21%	79.39%
Attention	86.81%	93.75%	84.60%	88.94%
BiRNNs	93.72%	95.77%	98.07%	96.90%
CNNs	95.10%	96.49%	93.50%	94.97%

As observed in the table, the CNN model exhibited the highest accuracy and precision compared to the other models, achieving an accuracy of 95.10% and a precision of 96.49%, which indicated its superior performance in correctly identifying actionable alarms. However, the BiRNN model demonstrated superior performance in terms of the recall and F1-score. It achieved a remarkable recall of 98.07% and an impressive F1-score of 96.90%. These metrics highlight the model's ability to effectively recognize actionable alarms while maintaining a favorable balance between precision and recall. Both the BiRNN and CNN models demonstrated promising potential for practical deployment in accurately identifying actionable warnings. The training and testing processes of these two classifiers can be observed in Figure 8.

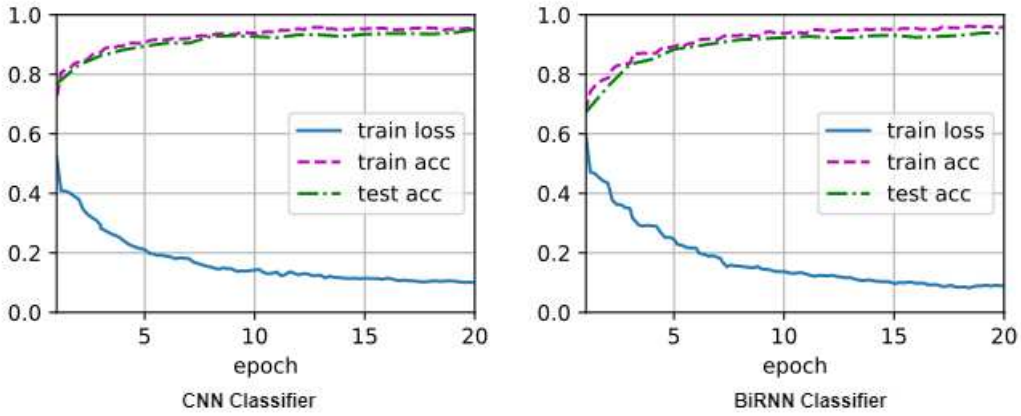

Figure 8. The training and testing processes for BiRNN and CNN classifiers.

Furthermore, we implemented a function called *predict* to determine whether the warning detail and code snippet pair inputted by the user is an actionable alert or a false alarm. The *predict* function accepts the warning details and corresponding code snippets as the input from the user. It leverages the pre-trained CodeBERT model to generate context embeddings for the input and then utilizes a neural network classifier to predict the labels based on these context embeddings.

By visualizing the ROC curves and calculating the AUC score, we assessed the performance of our proposed approach in accurately identifying actionable warnings. The results, as depicted in Figure 9, demonstrate that the BiRNN model attained the highest AUC score of 0.983, closely followed by the CNN model with an AUC score of 0.976. The attention model exhibited an AUC score of 0.94, while the MLP model yielded a comparatively lower AUC score of 0.847, indicating its inferior performance compared to the other models.

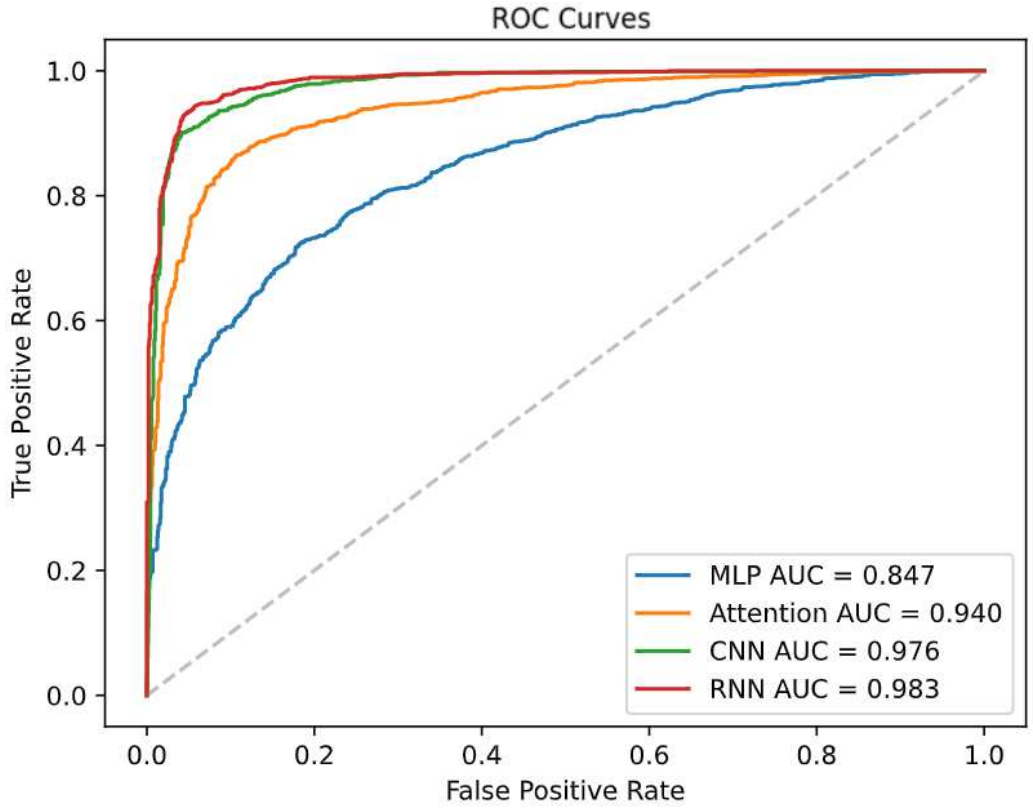

Figure 9. ROC curves and AUC scores.

Table 3 shows the benchmark evaluation results. The comparative experiments in the table were all replicated and validated by the author using the same dataset on different models in the same environment. Our designed BiRNN model, with its ability to capture contextual information from both the forward and backward directions, achieved the highest AUC score in our evaluation, closely followed by our CNN model. Lee at al. [34] presented a classifier based on CNNs for identifying false positive alarms, and it achieved an average precision of 81.72% and an average recall of 73.09%. The convolutional neural network model designed in this study achieved an accuracy of 96.49% and a recall rate of 93.5% in the static analysis alarm classification task after preprocessing. The AUC score was 97.6%, which shows it can effectively identify false positive alarms. Using the support vector machine (SVM) proposed by Yang et al. [35] for the experiments, the recall rate exceeded 87% with AUC scores of over 97% when predicting actionable warnings. While Yang et al. would not recommend DL models, Yerramreddy et al. [36] demonstrated that neural networks that learn via source code outperform traditional models. This sparked our interest in attempting the application of DL models for processing static analysis alarms. The bidirectional recurrent neural network designed in our study achieved a recall rate of 98.07% and an AUC score of 98.3% in predicting actionable alarms.

Table 3. Comparison of performance with relevant research methods.

Method	Average Precision	Average Recall	AUC
Lee et al. [34]	81.72%	73.09%	77.16%
Yang et al. [35]	92.14%	87.6%	97.3%
Our CNNs	96.49%	93.50%	97.6%
Our BiRNNs	95.77%	98.07%	98.3%

5.3.3. RQ3: Benefits

There are several benefits of utilizing DL for processing static analysis alarms. DL models have the ability to directly learn from raw code and original warning data, taking advantage of the hierarchical structure inherent in the data. This enables the models to capture intricate relationships and patterns within the alarms. Additionally, DL models excel in automatically learning relevant features from static analysis alarms, eliminating the need for manual feature engineering. This characteristic of DL significantly reduces the dependence on domain expertise and minimizes the effort required to extract meaningful information from static analysis alarms. Furthermore, DL models demonstrate adaptability by accommodating various types of static analysis alarms. This flexibility allows the models to scale well to large datasets and efficiently process a substantial number of alarms, making them suitable for real-world software development scenarios. Once DL models are trained on a diverse dataset, they exhibit robust generalization capabilities across various programming languages and types of static analysis alarms. This scalability means that they can effectively handle new alarms without extensive manual adjustments, further contributing to their scalability and applicability in practice. Finally, DL has the ability to reduce false positives by distinguishing between genuine actionable alerts and benign alarms. These advantages collectively contribute to the effectiveness of DL in handling static analysis alarms. However, the effectiveness of DL models depends on various factors, including the quality and representativeness of the training data, the chosen model architecture, and the tuning of hyperparameters. Proper consideration of these factors is crucial in order to attain optimal performance.

6. Related Work

The challenge of minimizing false positives from the reports generated by static analysis while ensuring a reasonable number of actionable warnings has been the focus of extensive research studies. Researchers have proposed the pruning of alarms to identify actionable warnings that developers would like to fix.

Muske et al. [37] proposed version-aware static analysis techniques aimed at reducing the number of warnings by suppressing repeated alarms that are not impacted by the code changes between the two versions. Pruning is achieved by conducting an impact analysis, wherein the impacts of changes made between two consecutive versions are analyzed to identify and eliminate alarms that remain unaffected by these changes [38]. This suggests that leveraging developer modifications and fixing records as ground-truth can be beneficial in identifying veritably actionable warnings. Developer modifications offer practical and real-world perspectives into the code changes made to address specific issues, allowing us to accurately identify which warnings are truly actionable and require further investigation or resolution.

In recent years, multiple studies [39–42] have employed machine learning to distinguish between actionable warnings and false alarms. By framing the problem as a binary classification problem, some approaches [43–45] identify actionable warnings depending on the likelihood of developers acting upon the alarms. If developers do not think that warnings represent errors, they may not take action based on warnings. Dillig et al. [46] introduced a technique for assisting users with classifying error reports generated by static analyses. With user input to semi-automatically classify alarms into errors and false positive, alarm-specific queries are revealed to the user for the effective and efficient manual inspection of alarms. Hanam et al. [45] conducted a study where they achieved binary classification by identifying alarms with similar patterns that were determined based on the code surrounding the alarms. Pradel et al. [47] formulated bug detection as a binary classification problem and trained a classifier to distinguish correct from incorrect code, resulting in DeepBugs successfully identifying 102 programming mistakes with a true positive rate of 68% in real-world code. Liu et al. [48] proposed an approach to automatically learn features and fix patterns of static analysis violations by leveraging CNNs and

X-means. Lee at al. [34] presented a classifier based on CNNs for identifying false positive alarms, achieving an average precision of 79.72%.

Wang et al. [49] completed a systematic evaluation of the features for static analysis warning recognition in the literature and found that 23 features could be treated as a Golden Feature set for detecting actionable warnings. With these Golden Features, Yang et al. [35,50] proposed that the detection of false alarms is an intrinsically solvable problem and introduced machine learning algorithms to effectively recognize actionable static code warnings. They conducted experiments with a support vector machine (SVM), achieving recall rates over 87% and AUC scores over 97% when predicting actionable warnings. However in 2022, Kang et al. [51] addressed two flaws in the work of Yang et al. and discovered a decline in the performance of Golden Features. After addressing data leakage and data duplication issues, they indicated that the performance of the Golden Features SVM was not almost perfect, with only marginal improvements over a strawman baseline that always predicted that a warning is actionable. In response, Yedida et al. [52] proposed four localized treatments that were tested on the data generated by Kang et al., achieving a median AUC score of 92%. They also mentioned that their attempts failed, even after experimenting with various architectures including feedforward networks, CNNs, and CodeBERT. Moreover, Yerramreddy et al. [36] presented a comparative empirical study of three machine learning techniques for classifying correct and incorrect results generated by ASATs. Their observations suggest that neural networks, such as RNNs, trained on source code outperform traditional models.

7. Discussion

In this study, we presented a novel approach based on DL for processing static analysis alarms. Our results demonstrated promising performance in identifying false alarms and actionable alarms. However, several points should be discussed in interpreting the results.

Firstly, the effectiveness of our approach was primarily evaluated on a specific dataset limited to the Java programming language. Although the dataset covers a wide range of Java code, the generalizability of our model to other programming languages, such as C, C++, Python, JavaScript, and others, remains uncertain. The context and syntax of the code in different programming languages can vary significantly, which may impact the performance of our model when applied to code written in these languages. Further research is required to explore the transferability of our approach across different programming languages. Moreover, the domain of software applications also represents a potential limitation. Our study primarily focuses on alarms present in Java programming code, but the nature of alarms can differ depending on the software domain. For instance, alarms in AI applications, games, or scientific software may exhibit unique characteristics and require specific detection techniques. Therefore, the applicability of our approach to various software domains should be investigated in future studies to assess its effectiveness and adaptability. Regarding the scope of alarms covered by our training data, it is essential to acknowledge that our model's performance is contingent on the types of alarms it has been trained on. Although we trained the model using a diverse set of project samples, it is unlikely to cover all possible warning scenarios. Thus, there is a possibility of encountering alarm scenarios that our model may not effectively detect. Ongoing efforts should focus on expanding the training dataset to encompass a broader range of warning types.

8. Threats to Validity

In this section, we discuss the threats to validity associated with our study. These threats can be categorized into four main aspects: internal threats to validity, external threats to validity, conclusion threats to validity, and construct threats to validity.

8.1. Internal Threats to Validity

One potential internal threat to validity is related to the implementation of our DL model. The model's performance heavily relies on the hyperparameters chosen during

training, such as the number of filters, filter sizes, and hidden size. Suboptimal choices of these hyperparameters may impact the model's ability to detect bugs accurately. To mitigate this threat, we conducted extensive hyperparameter tuning using cross-validation techniques to find the optimal configuration. However, there is still a possibility that alternative hyperparameter settings could yield different results. Another internal threat to validity is the potential presence of bias in the training data. Our training dataset primarily consists of warning samples detected by SpotBugs, which may introduce bias towards the alarm patterns recognized by SpotBugs. This bias can affect the generalizability of our model to other ASATs. Future research could explore the integration of diverse ASATs to mitigate this bias.

8.2. External Threats to Validity

The external validity of our study is subject to potential threats due to the limited scope of the dataset used for evaluation. Although we utilized a comprehensive dataset of Java code containing a wide range of alarms, it may not fully represent the diversity of bugs present in real-world software projects. The characteristics of warnings in different contexts and domains may vary, which could impact the generalizability of our model's performance. Future studies should aim to incorporate more diverse datasets from various software projects to enhance the external validity of our findings.

8.3. Conclusion Threats to Validity

The conclusions drawn from our study are based on the evaluation results obtained from our specific experimental setup and dataset. While our findings demonstrate promising alarm detection performance, it is essential to recognize that the efficacy of our approach may vary in different settings and scenarios. Factors such as code complexity and project size can influence the practical applicability of our model. Therefore, caution should be exercised when generalizing the conclusions of our study to real-world alarm detection scenarios.

8.4. Construct Threats to Validity

Construct validity refers to the extent to which the operationalization of constructs aligns with the intended theoretical concepts. In our study, the construct of alarms is defined based on the bug samples present in the training dataset. However, the definition of alarms can be subjective, and different bug detection tools or researchers may have varying interpretations of what constitutes a bug. Therefore, the construct validity of our study is contingent on the accuracy and representativeness of the bug samples used in the training data. To address these threats to validity, future research should aim to incorporate larger and more diverse datasets, encompassing multiple programming languages and software domains.

9. Conclusions

This study delves into the application of deep learning technology in static analysis alert processing. By combining the context embedding generated by the pre-trained model and the generalization ability of deep learning models, a large number of data samples were trained to learn the patterns in warning information and related codes. This research achieves accurate identification of actionable alarms from a large number of static analysis alarms with high false positive rates. This achievement has practical significance for developers, as they will be able to quickly respond to alarms of genuine software defects, thereby optimizing the software development process and improving software reliability and development efficiency.

Firstly, this approach combines alarm information and its corresponding code fragments to construct an independent input sequence and then utilizes the powerful generalization ability of the pre-trained model to transform it into context embedding. These contextual embeddings enhance the ability of downstream classification models to extract

key features from input sequences and help capture valuable contextual information from code snippets and warning information. The research results indicate that the contextual embeddings generated by pre-trained model are crucial for the performance of subsequent classification tasks, helping downstream tasks to complete more accurate classification tasks and significantly improving the accuracy of identifying actionable alarms. In addition, the experimental results of this study also indicate that the choice of neural network architecture has a significant impact on the performance of alarm classification. The experimental plan designed in this study confirmed that deep learning models, such as bidirectional recurrent neural networks and convolutional neural networks, can effectively address the challenges in the field of static analysis alarm processing. This breakthrough is expected to significantly improve the software development process, significantly enhance software reliability, and enhance the work efficiency of developers.

In summary, this article proposes a static analysis alarm processing method based on pre-trained models and neural networks that was rigorously verified through empirical evaluation. We believe that this method can provide important insights for other researchers and have a profound impact on the development of static analysis alarm processing technology.

Author Contributions: Conceptualization, J.T. and Y.T.; methodology, J.T. and Y.T.; validation, J.T. and Y.T.; formal analysis, J.T. and Y.T.; investigation, J.T. and Y.T.; resources, J.T. and Y.T.; data curation, J.T.; writing—original draft preparation, Y.T.; writing—review and editing, J.T. and Y.T.; supervision, J.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the Natural Science Foundation of Hebei Province under grant F2021201049 and F2021201058. Additionally, we acknowledge the support from the Funds of the Central Government for Local Science and Technology Development under grant 236Z0701G. Any opinions, findings, or conclusions expressed in this work are those of the authors and do not reflect the views of the funding agencies in any sense.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The dataset used to support the findings was derived from GitHub available at https://github.com/Don2025/defects4j (accessed on 19 April 2023).

Acknowledgments: We would like to thank all the anonymous reviewers for their constructive comments on improving this paper.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ASATs	Automatic Static Analysis Tools
AST	Abstract Syntax Tree
AUC	Area Under Curve
BiRNNs	Bidirectional Recurrent Neural Networks
CNNs	Convolutional Neural Networks
DL	Deep Learning
FPR	False Positive Rate
GRU	Gated Recurrent Unit
LSTM	Long Short-Term Memory
MLP	Multilayer Perceptron
NLP	Natural Language Processing
RNNs	Recurrent Neural Networks
ROC	Receiver Operating Characteristic
SVM	Support Vector Machine
TPR	True Positive Rate

References

1. Christakis, M.; Bird, C. What developers want and need from program analysis: An empirical study. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3–7 September 2016; pp. 332–343.
2. Muske, T.; Serebrenik, A. Survey of approaches for postprocessing of static analysis alarms. *ACM Comput. Surv. (CSUR)* **2022**, *55*, 1–39. [CrossRef]
3. Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1877–1901.
4. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Lago, A.D.; et al. Competition-level code generation with AlphaCode. *Science* **2022**, *378*, 1092–1097. [CrossRef] [PubMed]
5. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 2244–2258. [CrossRef]
6. Hovemeyer, D.; Pugh, W. Finding more null pointer bugs, but not too many. In Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, San Diego, CA, USA, 13–14 June 2007; pp. 9–14.
7. Ayewah, N.; Pugh, W.; Morgenthaler, J.D.; Penix, J.; Zhou, Y. Evaluating static analysis defect warnings on production software. In Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, San Diego, CA, USA, 13–14 June 2007; pp. 1–8.
8. Thung, F.; Lo, D.; Jiang, L.; Rahman, F.; Devanbu, P.T. To what extent could we detect field defects? An empirical study of false negatives in static bug finding tools. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 3–7 September 2012; pp. 50–59.
9. Thung, F.; Lo, D.; Jiang, L.; Rahman, F.; Devanbu, P.T. To what extent could we detect field defects? An extended empirical study of false negatives in static bug-finding tools. *Autom. Softw. Eng.* **2015**, *22*, 561–602. [CrossRef]
10. Ayewah, N.; Pugh, W. The google findbugs fixit. In Proceedings of the 19th International Symposium on Software Testing and Analysis, Trento, Italy, 12–16 July 2010; pp. 241–252.
11. Aftandilian, E.; Sauciuc, R.; Priya, S.; Krishnan, S. Building useful program analysis tools using an extensible java compiler. In Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, Riva del Garda, Italy, 23–24 September 2012; pp. 14–23.
12. Sadowski, C.; Aftandilian, E.; Eagle, A.; Miller-Cushon, L.; Jaspan, C. Lessons from building static analysis tools at google. *Commun. ACM* **2018**, *61*, 58–66. [CrossRef]
13. Calcagno, C.; Distefano, D.; Dubreil, J.; Gabi, D.; Hooimeijer, P.; Luca, M.; O'Hearn, P.; Papakonstantinou, I.; Purbrick, J.; Rodriguez, D. Moving fast with software verification. In Proceedings of the NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, 27–29 April 2015; Proceedings 7; Springer: Berlin/Heidelberg, Germany, 2015; pp. 3–11.
14. Distefano, D.; Fähndrich, M.; Logozzo, F.; O'Hearn, P.W. Scaling static analyses at Facebook. *Commun. ACM* **2019**, *62*, 62–70. [CrossRef]
15. Vassallo, C.; Panichella, S.; Palomba, F.; Proksch, S.; Gall, H.C.; Zaidman, A. How developers engage with static analysis tools in different contexts. *Empir. Softw. Eng.* **2020**, *25*, 1419–1457. [CrossRef]
16. Johnson, B.; Song, Y.; Murphy-Hill, E.; Bowdidge, R. Why don't software developers use static analysis tools to find bugs? In Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013; pp. 672–681.
17. Habib, A.; Pradel, M. How many of all bugs do we find? A study of static bug detectors. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 317–328.
18. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. In Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA, 4–9 December 2017; Volume 30.
19. Wan, Y.; Zhao, W.; Zhang, H.; Sui, Y.; Xu, G.; Jin, H. What do they capture? A structural analysis of pre-trained language models for source code. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 2377–2388.
20. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), Online, 16–20 November 2020.
21. Taud, H.; Mas, J.F. Multilayer perceptron (MLP). In *Geomatic Approaches for Modeling Land Change Scenarios*; Springer: Cham, Switzerland, 2018; pp. 451–455.
22. Guo, M.H.; Xu, T.X.; Liu, J.J.; Liu, Z.N.; Jiang, P.T.; Mu, T.J.; Zhang, S.H.; Martin, R.R.; Cheng, M.M.; Hu, S.M. Attention mechanisms in computer vision: A survey. *Comput. Vis. Media* **2022**, *8*, 331–368. [CrossRef]
23. Schmidt, R.M. Recurrent neural networks (rnns): A gentle introduction and overview. *arXiv* **2019**, arXiv:1912.05911.
24. Li, Z.; Liu, F.; Yang, W.; Peng, S.; Zhou, J. A survey of convolutional neural networks: Analysis, applications, and prospects. *IEEE Trans. Neural Networks Learn. Syst.* **2021**, *33*, 6999–7019. [CrossRef] [PubMed]
25. Chung, J.; Gulcehre, C.; Cho, K.; Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. In Proceedings of the NIPS 2014 Deep Learning and Representation Learning Workshop, Montreal, QC, Canada, 8–13 December 2014.

26. Schuster, M.; Paliwal, K.K. Bidirectional recurrent neural networks. *IEEE Trans. Signal Process.* **1997**, *45*, 2673–2681. [CrossRef]
27. Kim, Y. Convolutional Neural Networks for Sentence Classification. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 25–29 October 2014; pp. 1746–1751.
28. Wu, Y.; Zou, D.; Dou, S.; Yang, W.; Xu, D.; Jin, H. Vulcnn: An image-inspired scalable vulnerability detection system. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 2365–2376.
29. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the NAACL-HLT 2019, Minneapolis, MI, USA, 2–7 June 2019; pp. 4171–4186.
30. Husain, H.; Wu, H.H.; Gazit, T.; Allamanis, M.; Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv* **2019**, arXiv:1909.09436.
31. Just, R.; Jalali, D.; Ernst, M.D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, San Jose, CA, USA, 21–25 July 2014; pp. 437–440.
32. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. Pytorch: An imperative style, high-performance deep learning library. In Proceedings of the NIPS'19: Proceedings of the 33rd International Conference on Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019; Volume 32.
33. Loshchilov, I.; Hutter, F. Decoupled weight decay regularization. In Proceedings of the International Conference on Learning Representations (2017), Toulon, France, 24–26 April 2017.
34. Lee, S.; Hong, S.; Yi, J.; Kim, T.; Kim, C.J.; Yoo, S. Classifying false positive static checker alarms in continuous integration using convolutional neural networks. In Proceedings of the 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), Xi'an, China, 22–27 April 2019; pp. 391–401.
35. Yang, X.; Chen, J.; Yedida, R.; Yu, Z.; Menzies, T. Learning to recognize actionable static code warnings (is intrinsically easy). *Empir. Softw. Eng.* **2021**, *26*, 1–24. [CrossRef]
36. Yerramreddy, S.; Mordahl, A.; Koc, U.; Wei, S.; Foster, J.S.; Carpuat, M.; Porter, A.A. An empirical assessment of machine learning approaches for triaging reports of static analysis tools. *Empir. Softw. Eng.* **2023**, *28*, 28. [CrossRef]
37. Muske, T.; Serebrenik, A. Classification and Ranking of Delta Static Analysis Alarms. In Proceedings of the 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), Limassol, Cyprus, 3–4 October 2022; pp. 197–207.
38. Venkatasubramanyam, R.D.; Gupta, S. An automated approach to detect violations with high confidence in incremental code using a learning system. In Proceedings of the Companion Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 472–475.
39. Heo, K.; Oh, H.; Yi, K. Machine-learning-guided selectively unsound static analysis. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; pp. 519–529.
40. Alikhashashneh, E.A.; Raje, R.R.; Hill, J.H. Using machine learning techniques to classify and predict static code analysis tool warnings. In Proceedings of the 2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA), Aqaba, Jordan, 28 October–1 November 2018; pp. 1–8.
41. Koc, U.; Saadatpanah, P.; Foster, J.S.; Porter, A.A. Learning a classifier for false positive error reports emitted by static code analysis tools. In Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, Barcelona, Spain, 18 June 2017; pp. 35–42.
42. Yüksel, U.; Sözer, H. Automated classification of static code analysis alerts: A case study. In Proceedings of the 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, 22–28 September 2013; pp. 532–535.
43. Tripp, O.; Guarnieri, S.; Pistoia, M.; Aravkin, A. Aletheia: Improving the usability of static security analysis. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014; pp. 762–774.
44. Yang, X.; Yu, Z.; Wang, J.; Menzies, T. Understanding static code warnings: An incremental AI approach. *Expert Syst. Appl.* **2021**, *167*, 114134. [CrossRef]
45. Hanam, Q.; Tan, L.; Holmes, R.; Lam, P. Finding patterns in static analysis alerts: Improving actionable alert ranking. In Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, India, 31 May–1 June 2014; pp. 152–161.
46. Dillig, I.; Dillig, T.; Aiken, A. Automated error diagnosis using abductive inference. *ACM SIGPLAN Not.* **2012**, *47*, 181–192. [CrossRef]
47. Pradel, M.; Sen, K. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.* **2018**, *2*, 1–25. [CrossRef]
48. Liu, K.; Kim, D.; Bissyandé, T.F.; Yoo, S.; Le Traon, Y. Mining fix patterns for findbugs violations. *IEEE Trans. Softw. Eng.* **2018**, *47*, 165–188. [CrossRef]
49. Wang, J.; Wang, S.; Wang, Q. Is there a "golden" feature set for static warning identification? An experimental evaluation. In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Oulu, Finland, 11–12 October 2018; pp. 1–10.
50. Yang, X.; Menzies, T. Documenting evidence of a reproduction of 'is there a "golden" feature set for static warning identification?—An experimental evaluation'. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021; p. 1603.

51. Kang, H.J.; Aw, K.L.; Lo, D. Detecting false alarms from automatic static analysis tools: How far are we? In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 698–709.
52. Yedida, R.; Kang, H.J.; Tu, H.; Yang, X.; Lo, D.; Menzies, T. How to Find Actionable Static Analysis Warnings: A Case Study With FindBugs. *IEEE Trans. Softw. Eng.* **2023**, *49*, 2856–2872. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.