Reducing False Positives of Static Bug Detectors through Code Representation Learning

Yixin Yang*§, Ming Wen‡†, Xiang Gao*, Yuting Zhang‡, Hailong Sun*
*Beihang University, ‡Huazhong University of Science and Technology
*{yixinyang, xiang_gao, sunhl}@buaa.edu.cn ‡{mwenaa, m202071389}@hust.edu.cn

Abstract—**With the increasing significance of software correctness and security, automatic static analysis tools (ASATs) play a more and more important role in software development due to their ability and scalability. However, compared to dynamic analysis methods, static tools often suffer from the severe problem of generating high false positive rates, due to their analysis mechanisms. To alleviate the false positive problem, many approaches have been proposed, which focus on manually extracted features from code snippets and then prioritize real warnings by means of statistics or machine learning techniques. However, manual encoded features are insufficient to achieve satisfactory performance across different datasets. In this study, we focus on exploring the effectiveness of various code representation learning (CRL) techniques in understanding the semantics of warnings generated by ASATs. In particular, our large-scale empirical study not only reveals that CRL models can effectively differentiate buggy code snippets (i.e., containing warnings detected by ASATs) from clean ones (the median of F1-score reaches 87.3% for binary classification, and reaches 77.4% for multi-class classification), they are also promising in identifying false positive warnings (the F1-score of best performer is 75.6%). Such findings drive us to further design a novel approach named `PRISM`, to PRIoritize Static warnings based on aggregating multiple CRL Models to reduce the false positives generated by existing ASATs. Extensive evaluations demonstrate that our designed approach can outperform existing baselines significantly.**

Index Terms—**Static Bug Detector, False Positive Warnings, Code Representation Learning**

I. Introduction

Identifying software bugs is a significant while challenging task, consuming about 50% of developers' efforts [1]. Nevertheless, critical bugs can still remain unveiled even after years of deployment. ***Static bug detectors*** like SpotBugs [2], Facebook's Infer [3], and Google's ErrorProne [4]. are therefore motivated, they have gained popularity and are extensively used in industry to help detect potential issues early in development. Static bug detectors are often designed based on static analysis (e.g., data-flow and control-flow analysis or AST-based pattern matching) , and can detect various types of defects such as *bad programming practices*, *performance issues* and *vulnerability issues*.

Despite the popularity and wide adoption of such tools, they are still suffering from the limitation of achieving ***high false positive rates***, in which case the detected issues might not be real bugs that are concerned by developers. Developers may regard some of the issues as merely "violations" or "warnings" instead of "bugs", so would many detectors define themselves. For instance, as revealed by recent studies, statically detected warnings might be overlooked by developers for a long period in practice [5], [6]. To address such limitations, various approaches have been proposed with the aim to prioritize real buggy issues over false positive cases. For instance, Kim *et al.* [7] proposed to prioritize warnings generated by static bug detectors via mining software evolution histories. Hanam *et al.* [8] utilized machine learning techniques to learn and prioritize true positive warnings based on labeled training datasets. Despite the fact that huge efforts have been made towards this problem, reducing the false positives of static bug detectors still remains to be a challenging task.

The past years have witnessed the rapid development of representation learning techniques in natural language processing [9], [10]. Thanks to the availability of massive codebases, it opens up new opportunities to utilize such advanced techniques (e.g., Word2Vec [9], BERT [10]) to effectively represent programs as *distributed vectors* in the domain of software engineering. Such distributed vectors, encoding the syntax or semantics of programs, are also known as *code embeddings*. Code embeddings are often in well-structured forms with a fixed length of vectors containing real numbers, thus enabling the utilization of deep learning techniques to further learn implicit while deep code features automatically. Currently, code embedding has been successfully applied to various software engineering applications, such as program understanding [11], bug detection [12], [13], automated program repair [14], [15], and so on. Compared to conventional learning models, which extracts engineered features from programs and then utilizes machine learning models, the new paradigm utilizing code representation learning (CRL) and deep learning techniques (i.e., denoted as ***CRL-based*** techniques in this study) has shown to be capable of achieving superior results in many applications [11], [12], [13], [14], [15], [16].

However, CRL-based techniques have not been widely used to reduce the false positive rate of ASATs, thus their effectiveness have not yet been fully studied in this area. Hence, this paper is therefore motivated to answer the following questions:

- *RQ1:* Can CRL-based techniques be utilized to detect bugs and capture the buggy types?
- *RQ2:* Which CRL-based techniques can effectively differentiate true positive warnings from false positive ones?

§This work was primarily undertaken by the author when she studied at the Huazhong University of Science and Technology.

†Corresponding author.

To answer these questions, this paper first performs a comprehensive study to evaluate and compare the effectiveness of different CRL-based techniques in classifying good and buggy code. In particular, we evaluate and compare seven different CRL models in combination with six different neural network models to recognize potential issues in large-scale open-source projects. Moreover, we further investigate the feasibility of utilizing CRL models to differentiate true positive warnings from false positive ones generated by static bug detectors. Our major findings are as follows:

- Existing CRL models can efficiently classify good and buggy code, with FastText + BGRU achieving the optimum results in binary bug detection (87.3% F1-score). CodeBERT combined with a SoftMax layer outperforms other models in multi-class bug classification, the optimal median of F1-score can reach 77.4%.
- CRL-based techniques show promising results in differentiating true positive warnings and false positive ones. The median of F1 Score of the best model (i.e., Word2Vec along with BLSTM) can reach 75.6%.

Inspired by our empirical study, we further designed a novel approach named **`PRISM`**, to **PRI**oritize **S**tatic warnings via aggregating CRL **M**odels to help reduce the false positives of ASATs. `PRISM` selects multiple top-performing CRL-based techniques and aggregating their results by majority voting to get final ranking and classification results. The results indicate that, after `PRISM` ranking, the proportion of the true positives among the top 500 reported warnings exceeds 90%. Comparing to two baselines HWP [7] and GP [17], the accuracy, precision and F1-score of `PRISM` outperform HWP by 37.2%, 36.5%, 24.9%, and outperform GF by 28.2%, 28.2%, 39.8%, respectively. Finally, the more models are aggregated, the better performance `PRISM` achieves, demonstrating its great extensibility to further reduce ASATs' false positives.

Our major contributions are summarized as follows:

- ***Originality:*** To the best of our knowledge, we present the first systematic and comprehensive study to explore and exploit code representation learning techniques in the application of static bug detectors. More importantly, we focus on leveraging CRL models to reduce the false positive cases generated by existing static bug detectors.
- ***Empirical Study:*** We perform a large-scale empirical study, including seven CRL models and six neural networks, to evaluate and compare their effectiveness in detecting static bugs and identifying false positives. These results can serve as a foundation for future advancements in this field.
- ***Approach:*** Based on our empirical study, a novel approach named `PRISM` is designed to prioritize true positive warnings among all warnings generated by static bug detectors.
- ***Dataset:*** We open-sourced our datasets and experimental details to facilitate future research. [18]

II. Background and Related Works

A. *Pruning False Positive Warnings*

ASATs usually produce high false positive rate, making users to suffer from long time on identifying true warnings. False positive warnings of ASATs not only arise from analytical errors or overestimations but also encompass warnings that developers will not act on, as developers think these warnings are irrelevant to actual bugs or are too risky to fix [19], [20]. Previous study have shown that 35%-91% of warnings reported by ASATs are false positives that ignored by developers [21]. Thus, it is hard for developers to perceive valuable and actionable warnings in all detected warnings (i.e., positives cases). Driven by such reasons, several methods are proposed to identify true positive warnings and prioritize them in the reported warning lists, which enable us to filter out warnings at the end of the list to reduce the false positive rate of ASATs.

Early work rank warnings by referring to warning fix history from source code repositories, such as the information extracted from commit histories. For example, Kim *et al.* proposed two history-based methods to differentiate the importance of warnings [22], [7]. One is to use the lifetime of warnings to measure their severities [22], that is, warnings having shorter average fixing time will be ranked higher. Another tool, named the HWP [7] algorithm, is to weight the warning patterns according to the historical statistics for warnings eliminated in different ways (i.e., fixed changes and non-fixed changes). The weight of each warning pattern is proportional to the number of warning instances from that category that were eliminated by software changes, with fixed changes contribute more than non-fixed changes. This kind of techniques are based on the historical statistics and can easily be utilized to weight warnings without other sophisticated calculation. However, they are limited to scenarios with multiple warning patterns. Besides, they are based on the assumption that all warnings are homogeneous, which means that all warnings belonging to one pattern are either false positives or true positives without considering the contextual information.

Recent work benefits from the development of machine learning [23], [24], [8], [21], [25], [26]. Specifically, they usually design certain important program features and identify whether the warnings are true positive or false positive utilizing machine learning technique . For example, Hanam *et al.* differentiated true positives and false positives by creating feature vectors based on code characteristics and identifying warnings with similar code patterns [8]. Heckman *et al.* performed a systematic study and drawn a conclusion that important characteristics of warnings should be explored to improve the performance of this task [21]. Subsequently, they explored the effectiveness of 51 warning features obtained by ASATs and 15 machine learning algorithms [27]. Furthermore, many work proposed methods combining different code features and machine learning algorithms [28], [29], [30], [31], [32]. For example, Wang *et al.* performed a systematic study on these works [17]. After studying 116 features from 10 related work, they identified 23 most important features, which is so-called the *Golden Features*. *Golden Features*, such as comment-code ratio [29](i.e., The ratio of the number of comment lines to the total number of code lines in a program), are examined to contribute most against other features. Furthermore, Yang *et al.* [32] analyzed Golden features using data from Wang *et al.*. They found that various machine learning techniques performed similarly, thus, simple machine learners like SVM were recommended due to low cost and comparable effectiveness compared to more complex methods. However, Kang *et al.* conducted experiments and proved that due to data duplication and data leakage, the effectiveness of *Golden Features* is seriously overestimated[33]. They conducted experiments and found that *Golden Features* was only slightly better than random predictions.

B. Code Representation Learning

CRL models often accept a sequence as input, and map each token or the entire sentence into a numeric vector through unsupervised learning. The earlier CRL models are often static, that is, the same token has only one semantic meaning with the same vector representation, regardless of their contexts. Common static CRL models are as follows:

- **Word2Vec** [9] is a classic unsupervised learning technique to learn static embeddings of tokens, chosen for its extensive use in software engineering(SE) tasks. [12], [16].
- **FastText** [34] is known for efficiency and handling Out-of-Vocabulary issues [35], we choose it as its popularity in academia and industry [36].
- **GloVe** [37] combines global matrix factorization with local context, offering faster convergence and superior performance compared to local-only models.

Later, contextual CRL models are proposed to solve the problem of polysemy. These models assign different vectors to tokens with varying semantics in different contexts.

- **ELMo** [38] utilizes bi-directional LSTM to generate token-level embeddings, chosen as it is the first significant work addressing polysemy issue.
- **BERT** [10] produces deep bidirectional, contextual, unsupervised word representations through pre-training. It is groundbreaking work significantly improves performance of multiple tasks, including SE tasks [39].
- **RoBERTa** [40] is proposed to solve undertrained problem of BERT and is well-performed in related work [41].
- **CodeBERT** [42] is dealt with natural language and programming language applications, such as code search. It has been extensively used in recent work [26], [43].

The above models also covers scratch models (i.e., Word2Vec, FastText, GloVe, ELMo) and pre-trained models (i.e., BERT, RoBERTa, CodeBERT). Besides, CRL models are often utilized together with various deep neural network models collectively to fulfill different software engineering tasks [16], [44], [45]. Li *et al.* [16] comprehensively compares the performance of NN models in vulnerability detection experiments, including Logistic Regression [46], one basic structure network(i.e., Multilayer Perception (MLP) [47]), two classic RNN models(i.e., Long Short-Term Memory (LSTM) [48], Gated Recurrent Unit (GRU) [49]), two bidirectional RNN models (i.e., BLSTM, BGRU [50]), a CNN model designed for sequences (i.e., TextCNN [51]).

III. Empirical Study Design

In this Section, we first introduce the study overview and details of the designed research questions, then we present the datasets that we collected and utilized in our empirical study. Finally, we discuss how we perform the experiments.

A. Study Overview

Our study mainly contains three parts. First, we aim to understand to what extent can existing CRL-based techniques be utilized to recognize static bugs as well as their types. The answers can lay the foundations for our subsequent investigations, in particular, exploring the feasibility of utilizing CRL models, especially those well behaved ones, in differentiating true warnings from false ones. If the results are positive, we can be further guided to reduce the false positive rate of existing ASATs by CRL-based techniques. Based on the above design, we devise research questions in Section III-B.

B. Research Questions

[RQ-1] *Can CRL-based techniques be utilized to detect bugs and capture the buggy types?* We select seven CRL models and six neural network models, and combine them in pairs to evaluate their performance on our manually-collected dataset by SpotBugs [2]. The first task is to differentiate whether the code snippets are buggy or not, and the second is to classify different types of bugs.

[RQ-2] *Which CRL-based techniques can effectively differentiate true positive warnings from false positive ones?* We choose the well-performed CRL-based techniques based on the results of the first research question, and evaluate their practicality and effectiveness of differentiating true positive and false positive warnings on a real-world dataset [5]. Besides, we verify the effectiveness of the trained CRL-based techniques on a ground truth dataset [52] collected from Defects4J [53].

C. Datasets Collection

We utilize three datasets to answer the above research questions. The first one is manually collected by us and serves for the evaluation of the first research question. The second and third datasets are used in the second research question.

TABLE I: Statistics of Buggy and Bug-Free Instances of the Manually Collected Datasets ❶.

Bug Type⋆	**STY**	**PER**	**BAD**	**I18N**	**COR**	**MT**	**SEC**	**EXP**	**TOTAL**
# Buggy instances	15,848	15,274	3,178	2,859	2,344	1,375	775	738	42,391
# Bug-free instances	74,859								74,859

⋆ denotes abbreviations of eight bug types in SpotBugs and each type contains many bug patterns. STY:'Dodgy style', PER:'Performance', BAD:'Bad practice', I18N:'Internationalization', COR:'Correctness', MT:'Multithreaded correctness', SEC:'Security', EXP:'Experimental'.[2]

1) Dataset ❶: We aim to cover more bug types with the latest versions of projects when explore the effectiveness of CRL-based techniques, so we manually collect a dataset for the first research question. In particular, we choose to leverage SpotBugs [2], a well-known static bug detector that has already been widely adopted to analyze 51 large-scale projects. These projects are all from the Apache Software Foundation collected from GitHub, and each of them has a relatively large number

TABLE II: Statistics of the Experimental Subjects in Dataset ❷.

Project Name	# Stars	# Forks	# Commits	# FW	# UFW	Project Name	# Stars	# Forks	# Commits	# FW	# UFW
Guava [54]	42889	9520	5665	102	9859	Junit4 [55]	8213	3122	2480	95	245
Netty [56]	27961	13789	10636	5	172	Activiti [57]	8166	6553	10768	2219	26233
Druid [58]	24779	7934	6502	200	9105	Metrics [59]	7448	1779	3076	189	1584
Redisson [60]	17772	4321	6908	108	1970	Dagger [61]	7282	3037	703	22	87
Mybatis-3 [62]	16471	11098	3869	16	727	Swagger-Core [63]	6875	2053	3985	166	3426
CodeGen [64]	13739	5677	11463	9	289	CheckStyle [65]	6447	8052	10837	1	79
WebMagic [66]	10138	4019	1134	14	489	Error-Prone [4]	5751	645	5123	19	4194
Jedis [67]	10131	3597	1854	2543	8152	Netty-SocketIO [68]	5526	1458	844	85	412
Auto [69]	9505	1113	1365	8	296	TiTan [70]	5155	1040	4434	69	597
Clojure [71]	9202	1361	3364	42	1622	Netflix [72]	4483	2296	2898	21	72

# denotes the total number; $\#FW$ denotes the number of "Fixed warnings"; $\#UFW$ denotes the number of "Unfixed warnings" after data deduplication and processing.

of stars and forks, which means that the detected defects may be more common or more widely used.

(i) Bugs Extraction. For each project, we first collect the latest version of JAR and corresponding source code of themselves and those of their dependencies via manually or automatically download with Maven [73]. After analyzing all projects, we extract the information provided by SpotBugs for each detected bug (i.e., relative source code path, package name, method name and number of lines where the bug is located, bug pattern and bug type.), and remove duplicate bugs based on these information. We further extract the surrounding contextual code lines, i.e., before and after five lines of the buggy lines. Utilizing a continuous 10 lines as code fragments is a common practice as adopted by many studies in code similarity comparing [74], [75], clone detection [76] and code comprehension [77]. After collecting the bug instances, we ruled out all buggy code snippets, and randomly selected ten lines of code without any reported bugs as bug-free instances. Finally, we de-duplicate data at the sequence level.

(ii) Code Normalization. As adopted by existing studies [78], [79], the normalization process is to assign a representative token for code elements belonging to the same category, plus a unique ID of the element. Specifically, we change the customized identifiers class name, method name, variable name, constants and fields into unified representations separately. For example, the first customized class '`A.B.C.D`' will be normalized to '`class0`', while the second class is normalized to '`class1`' and so on. Following the above rules, we normalize all instances utilizing JavaParser Library [80].

(iii) Pre-processing and Labelling. After data collection, we flatten each normalized code snippet into a single line and split the code into multiple tokens using JavaLang [81] to obtain the raw input data. Then buggy instances are regarded as positive instances with a label of '1', those without bugs are regarded as negative instances with a label of '0' for binary classification. For multi-class classification, the label of bug-free is '0' and the labels of instances corresponding to the eight different bug categories are denoted as '1'-'8' respectively. Table I summarizes dataset ❶ that we collect.

2) Dataset ❷: The second dataset is generated from dataset collected by Liu *et al.* [5], which contains 16,918,530 warnings, of which 88,927 are confirmed as fixed warnings (i.e., resolved by modifying specific lines), while the rest are unfixed (i.e., warnings still present in the latest version of the project). After checking the original dataset, we discovered a high number of duplicate samples. Besides, the number of unfixed warnings is an order of magnitude more than fixed warnings. Thus, we further process the original dataset to build dataset ❷ and make it suitable for our evaluation.

(i) Data Screening. For our experiment, we selected a subset of the original data. We choose instances of the top 35 most starred Github project repositories to ensure high data quality, and the 50 most frequently fixed warning patterns from the original dataset, since Liu *et al.* mainly performed their empirical study on these patterns and also only published unfixed instances of these patterns in the original dataset [82]. As warnings of the selected 50 patterns in the top 35 well-maintained projects tend to be fixed in time, the potential bias, such as true positive warnings being fixed after the dataset collection, can be reduced.

(ii) Data Pre-processing. Fixed and unfixed warnings are extracted, deduplicated, normalized and tokenized in the same way as how dataset ❶ was collected. Then fixed warnings are regarded as true positive samples with label '1', and unfixed warnings are regarded as false positive samples with label '0'. Consequently, dataset ❷ contains 5,933 true positive samples and 69,610 false positives, statistical information of the selected projects is shown in Table II.

3) Dataset ❸: We use a ground truth dataset collected from Defects4J [52]. The author uses SpotBugs to analyze projects contained in Defects4J and confirms the true bugs that SpotBugs can detect through three steps: diff-based method, fixed warning-based method and manual inspection. After reproduction, we obtained 418 false positive bugs and 10 true bugs reported by spotbugs.

D. Experimental Setup

Our experiments are mainly conducted on two servers, each of which is equipped with a Ubuntu18.04 OS, 2 Intel Xeon Gold 5117@2.00GHz CPU supporting 14 cores, 4 Nividia Tesla-V100-SXM2-32GB graphics cards and 252GB memory.

Model Training. Our selected CRL models not only include four scratch models (i.e., Word2Vec, FastText , Glove and ELMo) and three pre-trained models (i.e., BERT, RoBERTa and CodeBERT). The selected NN models MLP, LSTM, GRU, BLSTM, BGRU and TextCNN. We do not choose graph-based code representation learning models as they usually represent data at the function level, we find that the results

We use 'warning' in dataset ❷ following the original paper.

of graph models are often significantly worse than sequence-based models on our data.

For scratch CRL models, we train CRL models on each dataset with default hyperparameters, and perform token-level experiment by combining CRL with NN models. For pre-trained CRL models, we choose the default pre-trained models (i.e., 'BERT-BASE-UNCASED', 'CODEBERT-BASE' and 'ROBERTA-BASE') from HuggingFace [83] and fine-tune them on our data respectively. During fine-tuning, we not only conduct token-level experiments, but conduct sentence-level experiments following existing studies [84], [85], [42]. Due to configurations of the models, we exclude code snippets whose length are larger than 512 for BERT and RoBERTa, and exclude instances larger than 200 for CodeBERT.

Hyperparameter Tuning. We utilize *grid search* [86] for hyperparameter adjustment of NN models, with a large number of combinations of common values. These common values are derived from empirical values from the experimental setup of the original paper of NN models, the kaggle community [87] and some related works [44], [16]. To facilitate the replication of our experiments, we have made the tuned hyperparameters and their specific values publicly available in the PRISM project's GitHub repository [18].

IV. Experimental Results

In this Section, we present the experiments that we designed to answer the research questions. For each experiment, we state the objective and overview the execution details before presenting the results.

A. Effectiveness of Different CRL-based Techniques on Static Bug Detection

[Objective]: We investigate the capability of different CRL-based techniques to detect bugs and differentiate bug types. Our purpose is to study whether CRL-based techniques can understand and simulate the decision-making logic of SpotBugs. If the results are positive, they can hopefully further discover errors in decision-making of SpotBugs.

[Experiment Design]: For both binary and multi-class classification, there are 45 sets of experiments and each experiment is repeated for 10 times. During each experiment, we sample and balance dataset ❶ by random sampling following an existing study [88]. The training, validation and test set are constructed and NN models are trained through *Sierarchical 10-fold Cross Validation* [89] to avoid the potential bias caused by data sampling, model training and inference. *Sierarchical 10-fold Cross Validation* means that in each fold, the proportion of the various categories in the original data is maintained. It is effective to small datasets with multiple categories and is suitable for our experiments.

[Results]: Fig. 1 and Fig. 2 present the medians of F1 scores of different CRL-based techniques in binary bug detection and multi-class bug detection. We use 'CRL-NN' to denote the experiment of a CRL model combined with a NN model. We observe that, for both binary and multi-class classification, except for MLP, other NN models have demonstrated promising performance when combined with various CRL models. For binary bug detection, we find that the median of the F1 Scores of FastText-BGRU, GLoVe-GRU, Word2Vec-GRU can achieve 87.3%, 87.0%, 87.0%, These models work well, and can outperform almost all other models. For multi-class bug detection, the median of the WaF scores of CodeBERT-SOFTMAX is 77.4%, and that of FastText-GRU, FastText-BGRU can reach 77.1%, 76.0%.

> *Code representations learning models demonstrate promising performance on detecting bugs and identifying various types of static bugs. In particular, the achieved optimal median F1-score can reach* 87.3% *for binary classification and* 77.4% *for multi-class classification in recognizing specific bug types.*

We then analyze and compare the results in terms of CRL models and NN models. For scratch CRL models, we find that FastText outperforms other scratch CRL models, ELMo and Word2Vec enjoy good learning abilities, while GLoVe performs less stable, indicating that model with low training cost is easier to converge and suitable for static bug detection. For pre-trained CRL models, CodeBERT and RoBERTa perform significantly better than BERT, indicating that pre-training models on similar datasets can help improve the effectiveness on downstream tasks. With respect to different NN models, we observe that the effectiveness of scratch CRL models can be improved when combined with RNN models, especially with GRU and BGRU, some shallow-layer CRL models(i.e., FastText) even achieve similar performance with complex deep models(i.e., CodeBERT). Besides, Pre-trained CRL models will outperform their counterparts when combined with the bidirectional NN models or TextCNN. Furthermore, we observe that MLP performs significantly poorer than the others, indicating that feature extraction layers of NN is necessary when learning code sematics.

> ***[RQ1]**. For CRL models, FastText outperforms other scratch models in terms of both the effectiveness and stability. It performs better when combined with RNN. CodeBERT and RoBERTa outperform BERT, and perform better when combined with BRNN and TextCNN. For NN models, RNN models perform better, MLP gains the poorest performance and might not be suitable for the task of static bug detection.*

B. Differentiating True Positive and False Positive Warnings

[Objective]: Our aim is to explore which CRL-based techniques is more effective in distinguishing between true positive and false positive warnings. Based on the results, we can use well-performed CRL-based techniques to reduce the false positive rate of static detectors to a certain extent.

[Experiment Design]: Due to the poor performance of MLP as shown in Section IV-A, we exclude it and perform the remaining 38 sets of experiments. Each set of experiments is also performed for 10 times. During each experiment, we

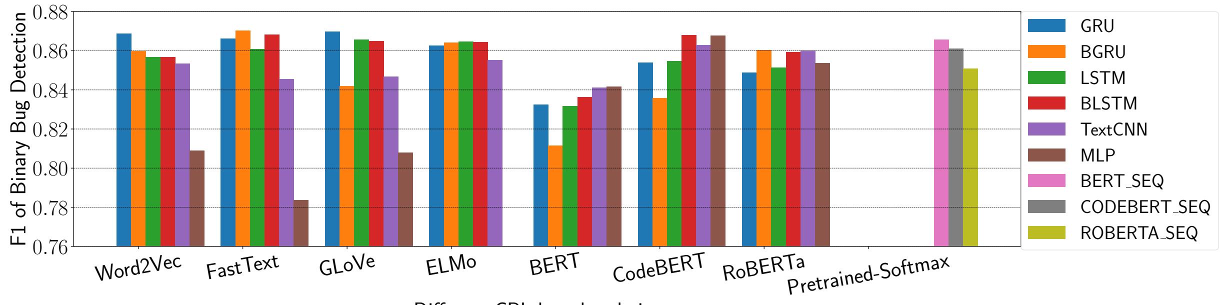

Fig. 1: F1 Score of Binary Bug Detection with CRL-based Techniques. 'Pretrained-Softmax' denotes sequence-level experiments

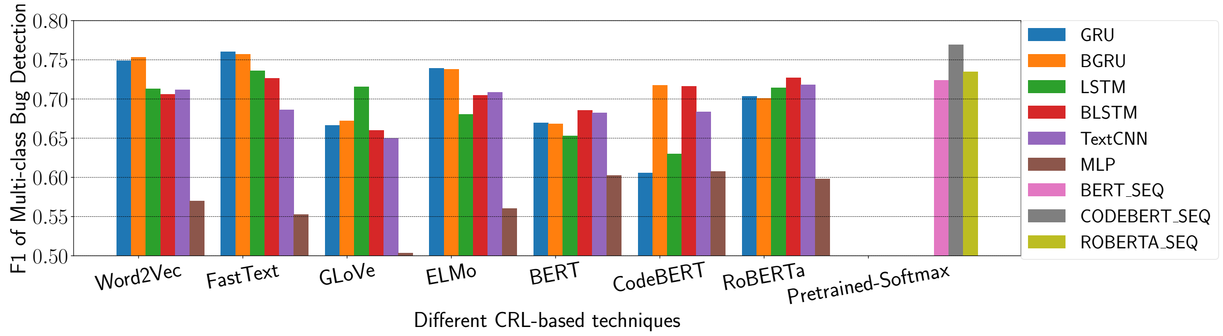

Fig. 2: Weighted-Average F1(WaF) score of Multi-class Detection with CRL-based Techniques.

balance dataset ❷ and randomly divide the balanced data into a ratio of 6:2:2 for training, validating and testing respectively.
[Results]: Fig. 3 shows the capabilities of different learning models on the task of distinguishing true positive warnings from false positive ones. For scratch CRL models, we can observe that Word2Vec, FastText is able to generate relatively stable and good results comparing to others. Almost all medians of F1 Score of them can reach or exceed 75%. However, the results of the remaining CRL models are less stable, especially when we utilize unidirectional RNNs (i.e., GRU, LSTM) with them, which may be caused by the limited learning abilities of unidirectional RNNs. For pre-trained CRL models, sequence-level experiments demonstrate poor performance. Taking BERT combined with a SoftMax layer(i.e., BT-S) as an example, BT-S is hard to converge during fine-tuning. However, when combining feature extraction layers (i.e., BGRU) with BERT in token-level experiments, its learning ability has been improved a lot, indicating that sequence embeddings are not suitable for the task. For NN models, the results of BGRU, BLSTM and TextCNN outperform that of GRU and LSTM, and medians of the F1 Score of the formers all reach 75%.

Among all CRL-based techniques, we find that static CRL models combined with bidirectional RNNs or CNN can obtain the best and most stable results. For one thing, applying a static CRL model with low training overhead is beneficial to represent the unified structural features of the same kind of warnings, and thus may reduce the difficulty of training NN models. For another, applying a bidirectional RNN or CNN model can extract and capture the data dependencies and control dependencies between tokens in each warning snippet, thereby improving the recognition ability of the model. Besides, the contextual CRL models combined with the bidirectional RNNs or CNN (e.g., BERT-BGRU) also perform well but require a high training cost, and sometimes the optimal hyperparameters are hard to be found in a short period. In addition, other combinations, such as static CRL models combined with unidirectional RNNs, pre-trained models combined with SoftMax achieve poor or unstable performance.

> ***[RQ2]**. Static CRL models, including Word2Vec, FastText combined with BGRU, BLSTM, TextCNN can achieve the highest and most stable F1 Score in distinguishing true positive and false positive warnings. Contextualized CRL models, including ELMo, BERT, CodeBERT, RoBERTa combined with TextCNN perform well but requires high training costs. In general, static CRL models along with bidirectional RNN models perform best.*

Further, we test CRL-based techniques trained with true positive and false positive warnings on dataset ❸ to see if they work in real scenarios. Table III, Table IV, Table V shows the efforts to discover 60%, 70%, 80% of true positive bugs. Effort refers to the number of bugs that need to be checked as a proportion of all reported bugs. We find that BERT-GRU and CodeBERT-GRU marked all warnings as true, so we excluded these outliers. From the results we can observe that after sorting by CRL-based techniques, we can find 60%, 70%, 80% of the true bugs at the earliest positions (i.e., 22.66%, 26.40%, and 29.67%) of the bug list. The best results are obtained by Word2Vec-TextCNN, GLoVe-BLSTM, and BERT-BGRU respectively, which are consistent with our previous conclusions.

Figure 4 shows the proportion of true bugs found in the

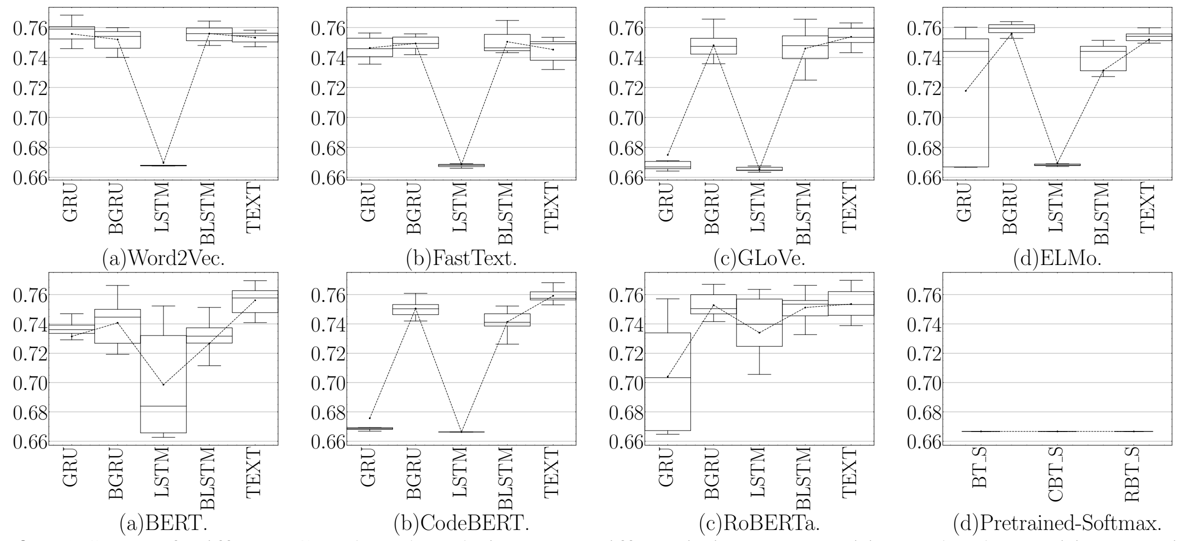

Fig. 3: F1 Score of Different CRL-based Techniques on Differentiating True Positive and False Positive Warnings.

TABLE III: Effort Required to Detect 60% True Bugs Across Different CRL-based Techniques. (* marks outliers)

	Word2Vec	**FastText**	**GLoVe**	**ELMo**	**BERT**	**CodeBERT**	**RoBERTa**
GRU	62.85	52.10	50.93	46.26	17.99*	4.44*	**26.64**
BGRU	49.53	48.36	68.22	46.50	**25.23**	60.51	33.64
LSTM	46.73	51.87	45.56	57.24	79.21	43.94	62.85
BLSTM	46.03	**41.59**	**23.13**	**39.95**	34.58	**43.93**	62.15
TextCNN	**22.66**	45.09	51.64	46.50	65.65	67.29	77.57

TABLE IV: Effort Required to Detect 70% True Bugs Across Different CRL-based Techniques.

	Word2Vec	**FastText**	**GLoVe**	**ELMo**	**BERT**	**CodeBERT**	**RoBERTa**
GRU	70.56	60.05	61.92	56.54	18.22*	4.67*	**26.87**
BGRU	56.07	57.94	75.93	48.60	**26.87**	**73.13**	38.79
LSTM	51.64	68.46	45.79	63.08	82.24	75.23	65.89
BLSTM	55.14	57.71	**26.40**	**45.09**	41.59	75.23	68.69
TextCNN	**38.08**	**45.33**	61.68	67.76	67.06	76.17	79.21

TABLE V: Effort Required to Detect 80% True Bugs Across Different CRL-based Techniques (%).

	Word2Vec	**FastText**	**GLoVe**	**ELMo**	**BERT**	**CodeBERT**	**RoBERTa**
GRU	72.90	64.02	64.49	60.75	18.46*	4.91*	77.57
BGRU	74.30	64.72	83.18	60.51	**29.67**	78.27	**48.13**
LSTM	65.89	70.09	46.03	71.96	85.05	77.58	72.66
BLSTM	60.05	66.36	**36.45**	**45.79**	70.56	**77.57**	70.33
TextCNN	**50.93**	**61.68**	67.76	73.36	84.58	80.37	82.24

top N predicted warnings to all true positives when we use the best CRL-based technique(i.e., GLoVe-BLSTM) to sort warnings. From the figure, we find that when predicting the top 60 warnings out of all 428 warnings, the optimal CRL-based technique has found almost 50% of the true positives. When predicting the top 100 warnings, it has found almost 60% of the true positives. When predicting the first 150 warnings, it has found almost 80% of the true positives. Such results confirm that CRL-based techniques can assist static detection tools to report true positives preferentially, thereby reducing their false positive rates.

However, although the model quickly find 80% of true positives, 90% and 100% of true positives are not found until top 325 warnings and top 335 warnings are predicted. This may be because some types of true positives are not easily recognized and prioritized by the model. Therefore, we hope to further improve the CRL-based techniques.

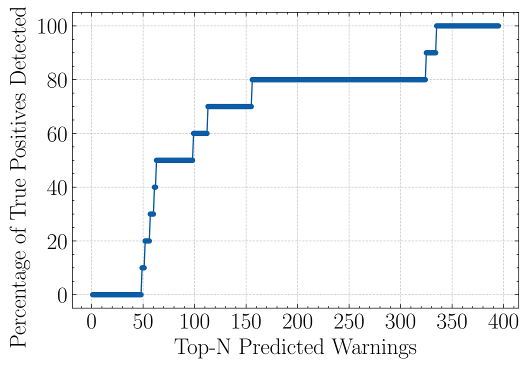

Fig. 4: The Number of True Positive Bugs in Top-N Predictions of the Best Model

V. PRISM: Prioritizing Static Warnings based on multiple CRL Models

[Objective]: Based on the above empirical findings, we investigate whether we can further reduce the number of false positives generated by static bug detectors. Our goal is to sort as many true positives as possible first. Therefore, we design the following approach, and further compare it with existing state-of-the-art baselines.

[Approach]: To fully exploit the advantages of multiple CRL models and improve the effectiveness of single CRL-based technique, we devise a simple yet effective approach named `PRISM`, to **PRI**oritize **S**tatic warnings based on multiple CRL **M**odels via utilizing the Majority Voting strategy [90]. Specifically, we integrate M models that are reported to achieve the optimum results (in terms of F1 Score) as revealed in section IV-B. In particular, two strategies are adopted for integration. The first one is *hard voting*. For a specific warning, according to all labels predicted by the top M models, label with the most occurrences is taken as the final predicted label of this warning. The second one is *soft voting*. For a warning, the average value of the predicted probabilities for each category generated by the M models is calculated, label corresponding to the category with the largest average value is taken as its label. Here each model is given the same weight and contributes equally to the ensemble voting results. Finally, a warning sorted list as well as the classification results of the tested warnings can be obtained according to their probability predicted by `PRISM`.

[Baselines]: We choose a historical statistics-based method called HWP algorithm [7] (denoted as **HWP**), and four machine learning models trained on six *"Golden Features"* [17] (denoted as **GF**) as our baseline models.

For HWP, we consider warnings from the highest-weighted patterns as true positives, while treating the rest as false positives. For GF, we choose four classic machine learning algorithms including Support Vector Machine(SVM) [91], RandomForest(RF) [92], DecisionTree(DT) [93], KNeighbors(KN) [94] to conduct experiments. In particular, given the limited information from the original dataset [82], we could only gather partial 'Golden Features', including 'warning context in file', 'warning context for warning type', 'defect likelihood for warning pattern', 'warning pattern', 'warning type', and 'warning priority'. Extracted using information from Git commits and static bug detectors, these features were refined by us, particularly the first three, to address label leakage issues highlighted in [33]. Specifically, while calculating these three features, an instance is marked as fixed only if its fixed date is before the revision under analysis, regardless of whether it has been fixed by the day of data collection.

[Experiment Design]: To evaluate the effectiveness of our proposed approach, we use *Precision@N* and conventional machine learning metrics (i.e., *Accuracy, Precision, Recall and F1 Score*). *Precision@N* denotes the proportional of true positive warnings (i.e., real warnings) to the top N warnings in the ranking list (N=10, 50, 100...) [95].

In evaluating `PRISM`, we aggregate the top M models, and perform Hard voting and Soft voting experiments respectively. Each experimental setup is denoted as `PRISM` $_{H/S}^{TopM}$. For each setup we repeat experiment for 10 times and final

TABLE VI: Precision@N of `PRISM` and the Best CRL-based Technique.

	PRISM $_{S}^{Top3}$	**PRISM** $_{S}^{Top5}$	**PRISM** $_{S}^{Top7}$	**PRISM** $_{S}^{Top9}$	**W2V-BLSTM**
N=10	88.00%	94.00%	100.00%	97.00%	84.00%
N=50	93.80%	96.00%	96.60%	96.40%	86.60%
N=100	95.50%	95.80%	96.80%	96.30%	87.20%
N=500	91.60%	91.70%	91.80%	91.60%	86.78%
N=1000	79.30%	80.60%	80.90%	81.00%	78.52%
N=2000	57.30%	57.50%	57.50%	57.60%	57.33%

results are the averaged values. Training, validating and test data are consistent with those in Section IV-B. In evaluating GF, for experiments using SVM, RF, and DT, we employ SKLearn's default hyperparameters [96]. For KN, we test four hyperparameter sets, varying the number of neighbors (1, 3, 5, 10) while keeping other parameters at their defaults. We also perform each experiment for 10 times to reduce the potential bias. In evaluating HWP [7], we set the variable K to represent the number of true warning patterns and evaluate the performance of HWP under different values of K.

[Results]: (1) Differentiating True Positive and False Positive Warnings

As shown in Table VI, when N is less than 500, all results almost approach or exceed 90%. Besides, the presicion@N drops as N grows, which shows that we have a high probability to obtain the most real and important warnings that need to be fixed at the top of warning prioritizing list.

The comparison results of `PRISM` and GF baseline are presented in Table VII. We can find that `PRISM` can obtain the highest F1 Score, Accuracy, Precision when the top-9 models are combined together and soft voting is performed. The top-7 combined models with soft voting and the top-9 combined models with hard voting also can obtain good prediction results. F1 Scores of the top three best performers exceed those of W2V-BLSTM by 9.3%, 9.3%, 9.3%, and exceed those of GF-SVM by 39.8%, 39.8% and 39.8%. The results show that the effectiveness of `PRISM` is promising, and will get better as the number of integrated models (i.e., M) grows. However, the recall of `PRISM` drops as M grows. When M reaches 9, the recall values of `PRISM` do not significantly exceed that of W2V-BLSTM. Besides, the precision of `PRISM` increases as M grows. Such results show that our approach is very sensitive to false positives, once the number of the aggregated models increases, the number of misidentified false warnings decreases faster than the number of correctly identified true warnings.

Fig. 5 shows the comparison between the results of `PRISM`, the HWP baseline and W2V-BLSTM. The performance of `PRISM` and W2V-BLSTM is invariant to the changes of K, and thus the results of them are displayed as straight lines. From the results, we can observe that both `PRISM` and W2V-BLSTM perform significantly better than HWP. F1 Score of `PRISM` $_{S}^{Top9}$, `PRISM` $_{H}^{Top9}$, `PRISM` $_{S}^{Top7}$ and W2V-BLSTM outperform HWP by 24.9%, 24.8%, 24.8% and 14.2%. However, one metric, on which `PRISM` fails to outperform HWP when K grows, is recall. This can be intuitively understood, as most of the warnings are already included and considered as true positives when K is close to the number of all warning

TABLE VII: Performance of PRISM against GF Baselines and the Best CRL-based Technique.

Approach	PRISM$_H^{Top3}$	PRISM$_H^{Top5}$	PRISM$_H^{Top7}$	PRISM$_H^{Top9}$	PRISM$_S^{Top3}$	PRISM$_S^{Top5}$	PRISM$_S^{Top7}$	PRISM$_S^{Top9}$
Recall	77.40%	75.86%	75.46%	**73.44%**	76.05%	75.38%	**74.70%**	**72.34%**
Precision	79.62%	84.83%	89.86%	**94.47%**	81.33%	88.51%	**92.48%**	**96.40%**
F1-score	78.50%	80.11%	82.03%	**82.64%**	78.61%	81.41%	**82.65%**	**82.66%**
Accuracy	78.79%	81.16%	83.47%	**84.57%**	79.30%	82.80%	**84.32%**	**84.82%**
Approach	**W2V-BLSTM**	**GF-SVM**	**GF-RF**	**GF-DT**	**GF-KN-1**	**GF-KN-3**	**GF-KN-5**	**GF-KN-10**
Recall	78.07%	47.01%	45.01%	41.30%	43.21%	44.12%	44.13%	41.34%
Precision	73.34%	75.20%	68.29%	64.12%	68.13%	68.51%	67.48%	70.13%
F1-score	75.60%	59.11%	54.03%	50.62%	53.61%	54.41%	53.05%	52.18%
Accuracy	74.80%	66.14%	62.71%	59.13%	61.20%	62.10%	61.12%	61.34%

⋆ H, S denote PRISM utilizes hard voting and soft voting. TopM denotes combing top M CRL-based techniques with the best performance. Results of the top 3 best performers of PRISM are shown in bold. W2V-BLSTM denotes Word2Vec combined with BLSTM, and is the best CRL-based technique. GF denotes the Golden Features and SVM, RF, DT, KN denotes Support Vector Machine, RandomForest, DecisionTree, KNeighbors. GF-KN-T denotes applying KNeighbors on the golden features with M neighbors.

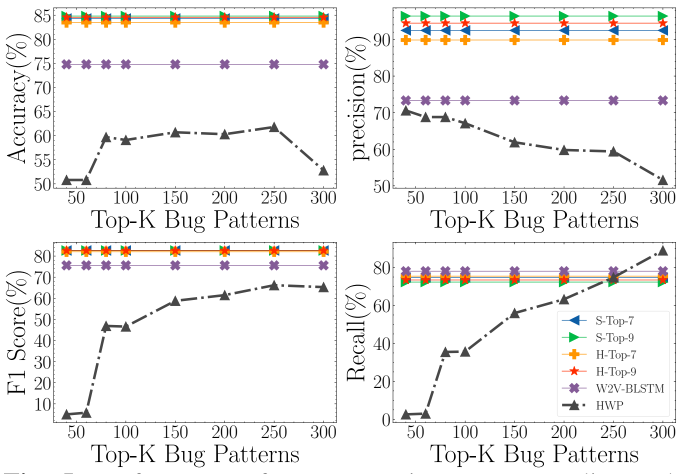

Fig. 5: Performance of PRISM against HWP Baseline and the best CRL-based Technique. S/H-Top-M denotes PRISM under the setting of soft/hard voting, combining top M best-performed CRL-based techniques.

patterns provided by FindBugs. We also observe an interesting finding that the accuracy and F1 Score of HWP increase when K is less than 250, and decrease when it exceeds 250, which means true positive warnings are most likely to fall into the 250 most important patterns evaluated by HWP. Besides, the precision of HWP always decreases, which means the number of false positive instances significantly increases when we consider more warning pattern as true positive ones.

PRISM can achieves good performance on differentiating true warnings and false ones. The scores of Precision@N of PRISM under all experimental setups exceed 90% when N is less than 500. Besides, accuracy, precision and F1 Score of PRISM outperform HWP by 37.2%, 36.5%, 24.9%, outperform the GF by 28.2%, 28.2%, 39.8%, outperform the best CRL-based technique by 13.4%, 31.4%, 9.3%.

(2) Prioritizing True Bugs.

We test PRISM on dataset ❸. Figure 6 shows that after different models sort the warnings detected by static detection tools, the proportion of true positives found in the top N predicted warnings to all true positives. Compare to single CRL-based technique, PRISM can indeed find more true bugs earlier. PRISM aggregated by the top 5 models can find almost 90% of true positives in the top 210 warnings and almost 100% of the top 220 warnings. Such results means that we can filter out nearly half of the false positive warnings. It is very meaningful as static detection tools often report tens of thousands of warnings in large-scale real-world projects.

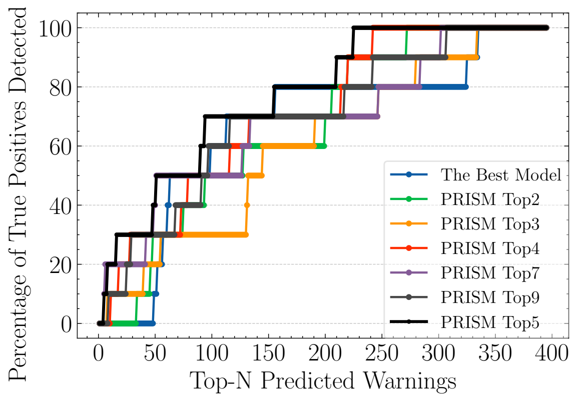

Fig. 6: The Percentage of True Positive Bugs in Top-N Predictions of the Best Model and PRISM

Table VIII indicates the effort we need to find the 60%-100% real bugs in the sorted list of different models. Please note that the baseline results reported in our table are all optimal parameter settings. We find that the performance of PRISM aggregated by 5 models and soft voting performs best, When identifying all real bugs, PRISM$_S^{Top5}$ required 48.9% less effort than the best CRL-based technique, 6.2% less than GF, and 87.9% less than HWP.

It is meaningful to aggregate models, which can detect all real bugs earlier than a single CRL-based technique and baselines. Besides, PRISM can also improve the precision of single CRL-based technique while maintaining the recall of it.

VI. Discussion

A. Flexibility and Application of PRISM

PRISM is designed with flexibility to adjust the number and complexity of its integrated models, allowing a balance between precision and computational overhead. In the scenarios requiring high precision, like safety-critical systems, improving precision is prioritized over saving computational overhead. However, in resource-limited environments, PRISM

TABLE VIII: Effort Required to Detect True Bugs, the Best CRL-based Technique and Baselines on Defects4J(%).

# True Bugs	The Best Model	PRISM $_S^{Top2}$	PRISM $_S^{Top3}$	PRISM $_S^{Top4}$	PRISM $_S^{Top5}$	PRISM $_S^{Top7}$
60%	23.13	30.14	33.87	27.33	**21.02**	29.67
70%	26.40	46.96	44.62	31.07	**21.96**	31.30
80%	36.44	48.36	57.71	50.23	**36.21**	57.71
90%	75.93	51.63	65.65	51.63	**49.06**	66.35
100%	78.27	63.78	78.27	56.77	**52.57**	70.56
# True Bugs	**PRISM $_S^{Top9}$**	**GF-SVM**	**GF-RF**	**GF-DT**	**GF-KN**	**HWP**
60%	22.66	83.17	74.53	54.90	54.90	16.35
70%	27.33	84.81	74.76	55.14	55.14	18.45
80%	50.70	85.51	75.00	55.37	55.37	50.00
90%	56.77	93.45	80.84	91.35	55.60	86.68
100%	71.72	94.15	95.32	92.52	55.84	98.83

can reduce the number of integrated models to reduce cost while still achieving precision required by the system.

While the integration of multiple models in PRISM initially leads to higher costs, it offers long-term savings of manpower and time. First, PRISM significantly lowers the cost of manual bug verification and provides reliable bug for decision-makers, reducing the risk of erroneous decisions. Second, as computing technology advances and hardware costs decrease, these expenses are expected to become more acceptable in the long term. In the future, we will optimize the efficiency and overhead of PRISM through more effective training strategies, learning algorithm, and hardware acceleration technologies.

B. Threats to Validity

Threats to external validity. First, model selection can affect how well our findings can be applied to other types of models. To mitigate this threats, when selecting models, we follow two criteria: 1.Comprehensiveness: The chosen models cover various types and differ in complexity and characteristics, ensuring a broad representation of approaches. 2.Effectiveness: The chosen models include SOTA transformer-based models like BERT, CodeBERT, and RoBERTa, as well as foundational models from the past. These models have been proven effective in numerous software engineering tasks. Second, our study relies on pre-processed and accurately labeled data, which may not always be readily available in real-world scenarios.This limitation could impact the generalizability of our findings to practical applications. Future research could explore methods less reliant on extensively pre-processed and labeled data, possibly incorporating automated preprocessing or advanced machine learning techniques that require less manual intervention.

Threats to internal validity. First, since the promising results of neural networks are often caused by the potential data duplication and data leakage, on one hand, we rigorously deduplicate three datasets we used. For dataset ❶, we remove all duplicated JAR packages and perform strict deduplication based on the location information of bugs. For dataset ❷, we consider warnings that in the same location and under the same bug pattern, but in different commits, as likely duplicates and perform strict deduplication. On the other hand, we also refined three of ”Golden Features” to eliminate data leakage problems proven by existing research [33]. Second, there may be a risk of overfitting of PRISM. We mitigate it by testing models with different data source covering different projects. We also use methods like early stopping, Dropout and Adam adaptive learning rate algorithm.

Threats to construct validity Method of labeling false positives and true positives for dataset ❷ may not be absolutely accurate. Disappearing warnings might not indicate true positives, possibly due to random code changes, and persistent warnings might not be false positives, as they could be true warnings that consistently unaddressed. To mitigate these errors, our study exclude warnings that disappear due to file or function deletions, since labels for these warnings cannot be determined. We also select the most starred 35 projects and the 50 most commonly fixed bug patterns to ensure that true positive warnings can be resolved as quickly as possible. Future efforts will focus on developing more effective data cleaning and labeling techniques to enhance data quality.

VII. Conclusion

We have conducted comprehensive experiments on CRL techniques on detecting bugs and differentiating static analysis warnings, and further conducted experiments to exploit code embedding techniques on warning prioritization. Based on our detailed experimental research, we observe that CRL models can effectively learn distributed representations for static warnings, and can also differentiate true positive and false positive warnings effectively. Motivated by such empirical results, we further designed a novel approach, which can prioritize true positive warnings via exploiting CRL models. Extensive experiments demonstrate that our proposed approach, `PRISM`, can outperform existing baselines significantly.

VIII. Acknowledgements

This work was supported by National Natural Science Foundation of China under Grant Nos (62141209, 62202026, 62372193) , and partly by Guangxi Collaborative Innovation Center of Multi-source Information Integration and Intelligent Processing and the Young Elite Scientists Sponsorship Program by CAST (No. 2021QNRC001).

References

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, “Reversible debugging software ”quantify the time and cost saved using reversible debuggers”,” *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

[2] “Spotbugs,” https://spotbugs.github.io, 2021-04-23.

[3] "Infer," https://fbinfer.com, 2021-04-23.

[4] "Errorprone," https://errorprone.info, 2021-04-23.

[5] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, 2018.

[6] N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," 01 2007, pp. 1–8.

[7] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, Sep. 2007, pp. 45–54.

[8] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: improving actionable alert ranking," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 152–161.

[9] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, E. P. Xing and T. Jebara, Eds., vol. 32, no. 2. Bejing, China: PMLR, 22–24 Jun 2014, pp. 1188–1196. [Online]. Available: http://proceedings.mlr.press/v32/le14.html

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jun 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[11] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 281–293.

[12] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019.

[13] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[14] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 981–992.

[15] Y. Li, S. Wang, and T. N. Nguyen, "DLFix: context-based code transformation learning for automated program repair," in *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[16] 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 Transactions on Dependable and Secure Computing*, 2021. [Online]. Available: https://arxiv.org/abs/1807.06756

[17] J. Wang, S. Wang, and Q. Wang, "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*, ser. ESEM '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3239235.3239523

[18] "Prism," https://github.com/ZoeResearch/PRISM, 2021-04-23.

[19] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, p. 92–106, dec 2004. [Online]. Available: https://doi.org/10.1145/1052883.1052895

[20] H. Shen, J. Fang, and J. Zhao, "Efindbugs: Effective error ranking for findbugs," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 299–308.

[21] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Inf. Softw. Technol.*, vol. 53, no. 4, p. 363–387, apr 2011. [Online]. Available: https://doi.org/10.1016/j.infsof.2010.12.007

[22] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, 2007, pp. 27–27.

[23] P. Avgustinov, A. I. Baars, A. S. Henriksen, G. Lavender, G. Menzel, O. De Moor, M. Schafer, and J. Tibble, "Tracking static analysis violations over time to capture developer characteristics," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 437–447.

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

[25] R. Yedida, H. Kang, H. Tu, X. Yang, D. Lo, and T. Menzies, "How to find actionable static analysis warnings: A case study with findbugs," *IEEE Transactions on Software Engineering*, vol. 49, no. 04, pp. 2856–2872, apr 2023.

[26] A. Kharkar, R. Z. Moghaddam, M. Jin, X. Liu, X. Shi, C. Clement, and N. Sundaresan, "Learning to reduce false positives in analytic bug detectors," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1307–1316. [Online]. Available: https://doi.org/10.1145/3510003.3510153

[27] S. Heckman and L. Williams, "A model building process for identifying actionable static analysis alerts," in *2009 International Conference on Software Testing Verification and Validation*, 2009, pp. 161–170.

[28] U. Koc, S. Wei, J. S. Foster, M. Carpuat, and A. A. Porter, "An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool," in *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 2019, pp. 288–299.

[29] G. Liang, L. Wu, Q. Wu, Q. Wang, T. Xie, and H. Mei, "Automatic construction of an effective training set for prioritizing static analysis warnings," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 93–102. [Online]. Available: https://doi.org/10.1145/1858996.1859013

[30] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1419–1457, 2020.

[31] C. C. Williams and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 466–480, 2005.

[32] X. Yang, J. Chen, R. Yedida, Z. Yu, and T. Menzies, "Learning to recognize actionable static code warnings (is intrinsically easy)," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–24, 2021.

[33] H. J. Kang, K. Loong Aw, and D. Lo, "Detecting False Alarms from Automatic Static Analysis Tools: How Far are We?" *arXiv e-prints*, p. arXiv:2202.05982, Feb. 2022.

[34] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[35] S. R. Young, "Detecting misrecognitions and out-of-vocabulary words," *Proceedings - ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 3, pp. 21–24, 1994.

[36] E. Bogomolov, Y. Golubev, A. Lobanov, V. Kovalenko, and T. Bryksin, "Sosed: a tool for finding similar software projects," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1316–1320.

[37] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[38] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *arXiv preprint arXiv:1802.05365*, 2018.

[39] S. S. Das, E. Serra, M. Halappanavar, A. Pothen, and E. Al-Shaer, "V2w-bert: A framework for effective hierarchical multiclass classification of software vulnerabilities," *arXiv preprint arXiv:2102.11498*, 2021.

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

[41] M. Sharma, I. Kandasamy, and W. Kandasamy, "Comparison of neutrosophic approach to various deep learning models for sentiment analysis," *Knowledge-Based Systems*, vol. 223, p. 107058, 07 2021.

[42] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[43] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 608–620.
[44] 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.
[45] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: A deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 04, pp. 2821–2837, jul 2022.
[46] D. G. Kleinbaum, K. Dietz, M. Gail, M. Klein, and M. Klein, *Logistic regression*. Springer, 2002.
[47] D. W. Ruck, S. K. Rogers, and M. Kabrisky, "Feature selection using a multilayer perceptron," *Journal of neural network computing*, vol. 2, no. 2, pp. 40–48, 1990.
[48] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, nov 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735
[49] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," 2014.
[50] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural networks*, vol. 18, no. 5-6, pp. 602–610, 2005.
[51] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach," *IEEE transactions on neural networks*, vol. 8, no. 1, pp. 98–113, 1997.
[52] A. Habib and M. Pradel, "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*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 317–328. [Online]. Available: https://doi.org/10.1145/3238147.3238213
[53] R. Just, D. Jalali, and M. D. Ernst, "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*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055
[54] "guava," https://github.com/google/guava.git, 2021-04-23.
[55] "junit," https://github.com/junit-team/junit.git, 2021-04-23.
[56] "netty," https://github.com/netty/netty.git, 2021-04-23.
[57] "activiti," https://github.com/Activiti/Activiti.git, 2021-04-23.
[58] "druid," https://github.com/alibaba/druid.git, 2021-04-23.
[59] "metrics," https://github.com/dropwizard/metrics.git, 2021-04-23.
[60] "redisson," https://github.com/mrniko/redisson.git, 2021-04-23.
[61] "dagger," https://github.com/square/dagger.git, 2021-04-23.
[62] "mybatis," https://github.com/mybatis/mybatis-3.git, 2021-04-23.
[63] "core," https://github.com/swagger-api/swagger-core.git, 2021-04-23.
[64] "codegen," https://github.com/swagger-api/swagger-codegen.git, 2021-04-23.
[65] "checkstyle," https://github.com/checkstyle/checkstyle.git, 2021-04-23.
[66] "webmagic," https://github.com/code4craft/webmagic.git, 2021-04-23.
[67] "jedis," https://github.com/xetorthio/jedis.git, 2021-04-23.
[68] "socketio," https://github.com/mrniko/netty-socketio.git, 2021-04-23.
[69] "auto," https://github.com/google/auto.git, 2021-04-23.
[70] "titan," https://github.com/thinkaurelius/titan.git, 2021-04-23.
[71] "clojure," https://github.com/clojure/clojure.git, 2021-04-23.
[72] "netflix," https://github.com/spring-cloud/spring-cloud-netflix.git, 2021-04-23.
[73] "Maven central repository," https://mvnrepository.com, 2022-08-19.
[74] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner, "Challenges of the dynamic detection of functionally similar code fragments," in *2012 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 299–308.
[75] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 542–553. [Online]. Available: https://doi.org/10.1145/3196398.3196431
[76] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 81–92. [Online]. Available: https://doi.org/10.1145/1572272.1572283
[77] L. Pollock, K. Vijay-Shanker, and G. Sridhara, "Automatically detecting and describing high level actions within methods," in *2011 33rd International Conference on Software Engineering (ICSE 2011)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2011, pp. 101–110. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/1985793.1985808
[78] 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," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.
[79] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. Srikant, "Ir2vec: Llvm ir based scalable program embeddings," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–27, 2020.
[80] "Javaparser," http://javaparser.org, 2022-08-19.
[81] "Javalang," https://github.com/c2nes/javalang, 2021-04-23.
[82] "the original data of the second dataset collection," https://github.com/TruX-DTF/findbugs-violations, 2022-08-19.
[83] "Huggingface," https://huggingface.co, 2021-04-23.
[84] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, Washington, USA: Association for Computational Linguistics, Oct. 2013, pp. 1631–1642. [Online]. Available: https://aclanthology.org/D13-1170
[85] A. Warstadt, A. Singh, and S. R. Bowman, "Neural network acceptability judgments," *Transactions of the Association for Computational Linguistics*, vol. 7, pp. 625–641, 2019. [Online]. Available: https://aclanthology.org/Q19-1040
[86] P. Lerman, "Fitting segmented regression models by grid search," *Journal of the Royal Statistical Society Series C: Applied Statistics*, vol. 29, no. 1, pp. 77–84, 1980.
[87] "kaggle," https://www.kaggle.com, 2021-04-23.
[88] T. Ganz, M. Härterich, A. Warnecke, and K. Rieck, "Explaining graph neural networks for vulnerability discovery," in *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 145–156. [Online]. Available: https://doi.org/10.1145/3474369.3486866
[89] J. Shao, "Linear model selection by cross-validation," *Journal of the American statistical Association*, vol. 88, no. 422, pp. 486–494, 1993.
[90] D. Ruta and B. Gabrys, "Classifier selection for majority voting," *Information fusion*, vol. 6, no. 1, pp. 63–81, 2005.
[91] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and their applications*, vol. 13, no. 4, pp. 18–28, 1998.
[92] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.
[93] Priyanka and D. Kumar, "Decision tree classifier: a detailed survey," *International Journal of Information and Decision Sciences*, vol. 12, no. 3, pp. 246–269, 2020.
[94] J. M. Keller, M. R. Gray, and J. A. Givens, "A fuzzy k-nearest neighbor algorithm," *IEEE transactions on systems, man, and cybernetics*, no. 4, pp. 580–585, 1985.
[95] L. Wei, Y. Liu, and S.-C. Cheung, "Oasis: Prioritizing static analysis warnings for android apps based on app user reviews," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 672–682. [Online]. Available: https://doi.org/10.1145/3106237.3106294
[96] "Sklearn," https://scikit-learn.org/stable/, 2022-08-19.