IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 12, 2022, accepted May 10, 2022, date of publication May 23, 2022, date of current version May 27, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3176865

Static Code Analysis Alarms Filtering Reloaded: A New Real-World Dataset and Its ML-Based Utilization

PÉTER HEGEDŰS AND RUDOLF FERENC
Software Engineering Department, University of Szeged, 6720 Szeged, Hungary

Corresponding author: Péter Hegedűs (hpeter@inf.u-szeged.hu)

This work was supported in part by the Security Enhancing Technologies for the Internet of Things (SETIT) Project under Grant 2018-1.2.1-NKP-2018-00004, and in part by the Ministry of Innovation and Technology National Research, Development and Innovation Office within the framework of the Artificial Intelligence National Laboratory Program under Grant RRF-2.3.1-21-2022-00004. The work of Péter Hegedűs was supported by the Bolyai János Scholarship of the Hungarian Academy of Sciences.

ABSTRACT Even though Static Code Analysis (SCA) tools are integrated into many modern software building and testing pipelines, their practical impact is still seriously hindered by the excessive number of false positive warnings they usually produce. To cope with this problem, researchers have proposed several post-processing methods that aim to filter out false hits (or equivalently identify ''actionable'' warnings) after the SCA tool produced its results. However, we found that most of these approaches are targeted (i.e., deal with only a few SCA warning types) and evaluated on synthetic benchmarks or small-scale manually collected data sets (i.e., with typical sample sizes of several hundred). In this paper, we present a dataset containing 224,484 real-world warning samples fixed (true positives) or explicitly ignored (false positives) by the developers, which we collected from 9,958 different open-source Java projects from GitHub using a data mining approach. Additionally, we utilize this rich dataset to train a code embedding-based machine learning model for filtering false positive warnings produced by 160 different SonarQube rule checks, one of the most widely adopted SCA tools today. This is the most extensive real-world public dataset and study we know of in this area. Our method works with an accuracy of 91% (best F1-score of 81.3% and AUC of 95.3%) for the classification of SonarQube warnings.

INDEX TERMS Static code analysis, filtering false positives, real-world dataset, code embedding, machine learning.

I. INTRODUCTION

Static Code Analysis (SCA) tools became first-class citizens of modern software development life cycles (SDLC). SCA tools are relatively fast, cost-efficient, and easy to integrate with continuous integration (CI) systems. They analyze the source code of the software and can efficiently detect various types of programming problems, like simple coding errors, vulnerabilities, performance issues, or design mistakes.

Despite their many favorable properties, the efficient usage of SCA tools in software development practice is still hindered by the excessive number of false positive warnings they usually produce. According to past studies, the ratio of false positive reports can reach 30-60% [1], [2]. This amount of false alarms (i.e., not actionable error reports) has a serious negative effect on the application of SCA tools. They can overwhelm the developers, thus shadowing real issues that may become undiscovered at the ''bottom of the pile''. Due to this, developers often start to neglect most of the warnings produced by the SCA tools and abandon them entirely in the worst case [3], [4].

The core reasons for SCA tools producing such a high number of false positives include the imperfect and shallow static code analysis (mostly to keep a reasonable analysis performance); applying an over-approximation in the detection strategy; the fact that some code patterns are considered to be an issue in some contexts but not in the others, and inherent limitations of static code analysis (i.e., handling polymorphism, reflection, pointer analysis).

Several approaches are proposed in the literature to handle this unwanted scenario by ranking SCA tool results, identifying ''actionable warnings'' or filtering false positive reports. Early methods started to appear in the late 2000s and mainly used statistical approaches [2], [5], [6] to reduce the number

The associate editor coordinating the review of this manuscript and approving it for publication was Chien-Ming Chen.

of false reports. Later on, machine learning (ML) methods became dominant [7]–[9], while most recent studies [10], [11] focus on applying natural language processing (NLP) techniques to represent the code context of warnings and classify the SCA reports using ML models. This wide variety of existing approaches to reduce false positive SCA reports demonstrate the theoretical and practical importance of the problem. Nonetheless, there are several common shortcomings of the past studies, which leaves the problem of false positive SCA warning reduction an open problem in general.

Koc *et al.* [10] already realized that even though the initial results of applying ML techniques to classify and filter false positive analysis reports have been promising, the long-term potential and best practices for this line of research are unclear due to the lack of detailed, large-scale empirical evaluation. They also emphasize the need for larger, real-world program datasets to validate the findings of prior works, which were mostly conducted on synthetic benchmarks. We made the same observations, which we can complement as follows: we found that all the previous works on false positive filtering are suffering from at least one of the following limitations: i) the filtering algorithm is specific to a limited subset of warnings (typically for 1-10 specific types) produced by one or more SCA tools; ii) the filtering approach is evaluated on synthetic benchmarks (like the OWASP benchmark [12] or Juliet [13]) or a small manually evaluated dataset (typically in the range of several hundreds of warning samples); iii) the presented evaluation dataset is company-specific and closed; therefore, the replication and validation of the presented method are not possible.

In this paper, we try to bridge the identified gaps in previous studies by presenting a novel, open dataset of 224,484 real-world labeled SCA warnings (true and false positives combined) in Java source code, produced by 160 different rule checks of SonarQube [14], one of the most popular SCA tools [15]–[17] today. We demonstrate the potential of this rich dataset with an NLP and ML-based false positive warning filtering approach. For collecting real-world data of that magnitude, we took advantage of the popularity of GitHub and the availability of millions of projects stored there. We employed data mining techniques to create a training dataset that is one to three orders of magnitude larger than any other existing dataset of real-world SCA reports we know of (and typically 100x larger than publicly available open-source datasets). We trained efficient ML models on the dataset for identifying false reports using an NLP-based code context representation (i.e., source code embedding with word2vec [18]). We chose embedding to represent source code as it eliminates the need for manual feature engineering and NLP-based code representation proved to be very effective in other SE tasks as well [19], [20]. Additionally, the method is computationally inexpensive compared to those relying on symbolic execution [21], [22] or the calculation of backward slices [9] for all the warnings to be classified.

Our method works with an accuracy of 91% (best F1-score of 81.3% and AUC of 95.3%) for the classification of SonarQube warnings. Recent works relying on NLP techniques similar to ours [10], [11] are reporting promising results as well; however, they were validated only for several types of issues either on a small number of manually labeled warnings (i.e., few hundreds) or a closed, proprietary dataset, while our models are validated on a real-world dataset, consisting of 224,484 labeled warning entries.

The main contributions of this work can be summarized as follows:

- A real-world dataset[1] consisting of 47,015 true positive and 177,469 false positive SonarQube warnings of 160 distinct types, collected with data mining techniques from 9,958 different open-source Java projects from GitHub;
- A light-weight and general NLP based source code embedding technique to represent the local context of SCA warnings;
- ML models trained on the dataset to filter the false positive warnings with high accuracy produced by one of the most popular SCA tools today, SonarQube.

The paper is organized as follows. In Section II we list the works related to ours. We present our data collection and false positive filtering approach in Section III. The results of the empirical evaluation of the methods are described in Section IV. We list the possible threats to the validity of our work in Section V and conclude the paper in Section VI.

II. RELATED WORK

The realization that SCA tools tend to produce an excessive amount of false positive alarms came soon after their integration into the daily development practices. It has triggered a new line of research [23] starting from the early 2000s that aimed to handle the situation by developing various post-processing methods to reduce the amount of false positive warnings produced by the SCA tools.

Ayewah *et al.* [6] manually classified the warnings produced by FindBugs[2] on several open-source projects into three classes: false positives, trivial bugs, and serious bugs. Their goal was to evaluate the accuracy and value of the warnings the tool reports. They argue that although developers are willing to fix the reported issues, the false positives and trivial bugs impact the practical applicability of the tool. As opposed to this paper, we not just evaluate the accuracy of an SCA tool but also propose a concrete ML-based technique to post-process the results for removing false positive alarms.

Kremenek and Engler [2] proposed the so-called Z-ranking technique, a statistical model (i.e., counts successful and unsuccessful checks) to rank those error messages most likely to be valid errors over those that are least likely. The authors evaluated their method with three different warning types produced by the MC C static checker tool (lock errors, free errors, and format string errors) on the source code of Linux and a proprietary system. They found that within the

[1] https://doi.org/10.5281/zenodo.5885653
[2] http://findbugs.sourceforge.net/

first 10% of error report inspections, Z-ranking found 3-7 times more bugs than the average number of bugs found by random ranking. In contrast to this work, we aim to identify a wide range of false positive warnings, namely 160 different SonarQube alarm types in Java systems. Our method is based on the structural context of the error reports, which we capture by source code embedding.

Kim and Ernest [5] showed that the built-in prioritization of warnings in SCA tools tends to be ineffective. They observed the warnings from three bug-finding tools, FindBugs, JLint, and PMD, for three subject programs, Columba, Lucene, and Scarab. Only 6%, 9%, and 9% of warnings are removed by bug fix changes during 1 to 4 years of software development; the remaining approximately 90% of warnings are likely to be false positives. They proposed an automated history-based prioritization algorithm: if a warning instance from a warning category is eliminated by a fixing change, they assume that this warning category is important. Otherwise, the warning is assumed to be false positive. Our data collection strategy is similar to this; however, we mark a warning to be false positive only if the developers explicitly ignore it. Additionally, our filtering algorithm does not work at the level of warning types but individual alarm reports, which means that a warning might be valid in particular code contexts but false in others. Finally, our data samples for training and evaluation come from thousands of systems and not just three.

Hackman and Smith [24] proposed to rank alerts generated from automated static analysis tools via an adaptive model that predicts the probability of an alert being a true fault in a system. The model is based upon a history of the actions the software engineer has taken to either filter false positive alerts or fix true faults. As a continuation of this research, Heckman and Williams [7] proposed a process for building false positive mitigation models using machine learning techniques based on 51 code characteristics (i.e., software metrics, code history, code churn). They evaluated the models on the FAULTBENCH benchmark [25] and obtained 88-97% average accuracy. We have a similar goal in this paper; however, we did not apply manual feature engineering but employed source code embedding to represent source code for ML models. Furthermore, we did not use the FAULTBENCH benchmark for evaluation as it is outdated (requires Java 1.5) and contains only small subject systems. Instead, we created and published the largest real-world dataset we currently know.

Tripp *et al.* [26] presented an interactive false positive filtering method (Aletheia) that requires the user to manually classify a subset of the warnings. Then, based on this small sample, an automatically constructed statistical filter is run over the remaining warnings, removing the ones classified as false positives. Although the tool proved effective, it targets JavaScript programs embedded into websites. Contrary to this, we propose a fully automated method for filtering false positive SonarQube warnings in Java programs.

None of the research efforts mentioned so far incorporate detailed information about the structure of the actual code that is analyzed, which we think is essential to enhance current false positive classifiers. Koc *et al.* [9], [10] were the first to develop false positive filtering techniques that capture the actual code context of the warning to be classified. In [9], the authors train a Bayesian classifier and a Long Short-Term Memories (LSTM) neural network to filter false positive warnings in Java code produced by the FindSecBugs SCA tool. The authors trained their models on bytecode instructions of the reduced code context (either the method body containing the warning or the backward slice starting from the warning line). They evaluated their models on the OWASP benchmark [12], which contains synthetic security bugs (2,371 data points altogether). The best result was achieved by LSTM using the method body context (89.5% accuracy). In [10] the authors extended the scope of their study and evaluated further code representation and ML methods on 14 real-world programs with manually validated FindSecBug reports (194 true positives and 206 false positives). The authors pinpointed that the types of false positive security issues are inherently different in the synthetic OWASP benchmark compared to the real-world dataset. Overall, their results suggest that using ML models on source code embedding of the warning context outperforms other methods. Despite the similarities between these works and ours, there are several significant differences. Koc *et al.* evaluated their method on only six specific security-related FindSecBugs warnings, while we focus on 160 different rule checks of SonarQube. Our ML methods are trained on a dataset containing hundreds of thousands of warnings from thousands of projects as opposed to the 400 warnings from 14 programs used by Koc *et al.*, which is a clear improvement in the real-world empirical evaluation aspect. Furthermore, we take a fixed set of lines before and after the warning line as an embedding code context, which is a refinement of their method considering the whole method body. It reduces the time required for prediction while maintaining high accuracy (method body context worked best for Koc *et al.* with LSTM). It also has the advantage of omitting the need for costly code analysis, like slicing or PDG construction (both used by Koc *et al.* [10]).

Ruthruff *et al.* [27] observed code, churn, and history metrics to build a logistic regression model to filter out false positive FindBugs reports. They applied a quick screening test to eliminate lots of these metrics and kept the ones that could be calculated quickly and had appropriate prediction power. In a study at Google involving 1,652 triaged warnings, the authors found that the prediction accuracy of false positives was 85%.

Seongmin *et al.* [11] presented an automated classifier based on Convolutional Neural Networks (CNNs) for filtering false positive reports. They trained the CNN model using a total of about 10K historical static analysis alarms generated by six C/C++ static analysis checkers for over 27 million LOC and their labels assigned by actual developers. The authors applied a word2vec based token representation to embed the source code. They achieved an average of 79.72% precision across all six checkers.

Yuksel and Sozer [8] built and compared various binary classifiers using ML to distinguish between actionable and unactionable SCA warnings. They used ten characteristics (including the developers' idea) of the alerts to determine if they were actionable. The evaluation dataset consisted of 1,147 manually inspected alerts.

All of the above works are closely related to ours and evaluate the methods on real-world datasets; however, they all rely on proprietary and closed source subject systems and datasets. Nonetheless, even the largest dataset these studies use contains 10K samples, whereas our public dataset consists of more than 200K warnings from almost 10K open-source projects.

There are other, more formal methods for reducing false positives of some specific warnings. For example, Kim *et al.* [21], and Arzt *et al.* [22] apply symbolic execution and SMT solvers for reducing the false positive BufferOverflow warnings and Android alerts, respectively.

Nguyen [28] combines static analysis and deductive verification to reduce false positive warnings reported by SEI CERT C [29] compatibility checkers.

III. APPROACH

We focus on filtering false positive warnings produced by SonarQube in Java code. As outlined in Section I, lots of issue reports might be false positives that degrade the practical usefulness of an SCA tool. The problem of detecting and eliminating false reports is complicated as the same type of warnings might be valid in certain situations and code contexts, while false hits in others. In this section, we overview SonarQube and its use in practice that we leverage for identifying true positive and false positive reports. We describe how we mined real-world samples for such true positive and false positive warnings on GitHub. Finally, we outline our method for representing source code contexts of SCA warnings and an ML-based approach that demonstrates the potential lying in the collected data for identifying false positive reports based on these context embeddings.

A. SONARQUBE OVERVIEW

SonarQube[3] is an open-source platform for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities in multiple programming languages. SonarQube is widely adopted by practitioners [30] and attracts research focus [16], [31] as well.

Let us consider the code snippet shown in Listing 1. SonarQube will issue a warning report of type `squid:S1643` to line 3 of this snippet, which corresponds to the rule "Strings should not be concatenated using '+' in a loop". This rule is meant to detect performance issues both in terms of running time and memory consumption.

The developers might react to this information in three different ways:

[3]https://www.sonarqube.org/

```
1       String str = "";
2       for(int i = 0; i < arrayOfStrings.length; i++) {
3         str = str + arrayOfStrings[i]; // <<< squid:
              S1643
4       }
```

Listing 1. Sample Java code with a squid:S1643 warning.

```
1         StringBuilder bld = new StringBuilder();
2         for(int i = 0; i < arrayOfStrings.length; i++)
              {
3           bld.append(arrayOfStrings[i]);
4         }
5         String str = bld.toString();
```

(a) Fix the SonarQube warning (developer action A2)

```
1         String str = "";
2         for(int i = 0; i < arrayOfStrings.length; i++)
              {
3           str = str + arrayOfStrings[i]; //NOSONAR
4         }
```

(b) Explicitly ignore SonarQube warning (developer action A3)

Listing 2. Possible developer actions to a SonarQube warning.

A1. They do nothing. Either because they do not use SonarQube and the issue remains undiscovered, or they do use it but simply ignore this warning.

A2. They explicitly change the implementation so that SonarQube will not issue the warning in question to the fixed version anymore (see the snippet on Listing 2a).

A3. They explicitly ignore this warning by placing a "`//NOSONAR`" comment into line 3, which instructs SonarQube not to issue a warning to that location anymore (see the snippet on Listing 2b).

We exploit the developer actions A2 and A3 to identify true and false positive code samples, respectively. The technical details of the data collection process are described in Section III-B. A1 might be the sign of false positive reports as well but it is not conclusive (we do not know for sure if the warning stays in the code because developers consider it false positive or if it simply remains undiscovered); therefore, we do not deal with these cases. In contrast to action A1, actions A2 and A3 are both explicit; the developers express their opinions about the issue by deliberately fixing or ignoring it, thus admitting it was a true or false positive warning.

B. DATASET

We focused on the warning reports issued by SonarQube in this work. To build a real-world dataset suitable for building false positive SCA warning filtering methods, we exploited the observations of possible developer actions to an issued SonarQube warning. To create the dataset, we mined the history of Java projects on GitHub. To get as many samples as possible, we did not apply any restrictions on the projects we included. We started to explore all the Java projects by time intervals and kept those having any indications of SonarQube usage (see details below). We mined data back until the year 2010, the time when SonarQube entries first appear in code histories. We ended up having warning samples from 9,958 GitHub different Java projects.

True positive samples come from SonarQube issue reports that have been fixed by the project development team (there is a commit that removes the warning from the code base). It is straightforward that if the developers explicitly fix a SonarQube warning, they consider this report to be a valid (i.e., true positive) hit (this corresponds to action A2 described in Section III-A). For collecting false positive samples, we take advantage of the ignore mechanism provided by SonarQube (i.e., placing the particular comment "`//NOSONAR`" into the code base). We exploit the fact that if developers explicitly put these ignoring comments into the source code, they consider the warnings on that location to be false hits (this corresponds to action A3 described in Section III-A). Since detecting developer actions A2 and A3 require different approaches, we took separate data mining steps for collecting true positives and false positives. The technical details on the mining process for true positive and false positive SCA warning samples are detailed below (see Sections III-B1 and III-B2, respectively).

1) MINING TRUE POSITIVE SAMPLES

To find true positive samples, i.e., SonarQube warnings that have been fixed by the developers, we analyzed the code history of Java project repositories back until 2010 to find explicit SonarQube warning fix commits. We employed both the REST [32] and GraphQL [33] APIs provided by GitHub to perform the data analysis steps depicted in Figure 1.

The process starts with finding all the issues with attached pull requests that contain the term "sonar" in their title, description, or comments. This step is a very rough pre-filtering of all the possible issues related to SonarQube analysis. Then we collected all the related pull requests attached to these issues and all the commits belonging to the pull requests. For these first couple of mining steps, we used the GraphQL API of GitHub.

Next, we processed the pull requests and the collected commits with a Python script to select those that contain an exact reference to one of the SonarQube rules. Specifically, we kept only the commits that either had a commit message or belonged to a pull request that matched with one of the following regular expressions:

```
r'(squid:s\d+)'
r'(squid:\w+)'
r'(squid
r'(squid
```

Once we identified the warning fixing candidate commits, we searched for their parent commits to get the previous version of the source files modified by the commit. These previous file versions were likely to contain the SonarQube warnings that were removed by the fixing commit. We stored these previous file versions and the commit diffs (i.e., fixing patches). For these steps, we used the REST API of GitHub.

Finally, to verify that the heuristically identified warning removal commits do fix the SonarQube issues, we ran SonarQube both on the previous and current file versions

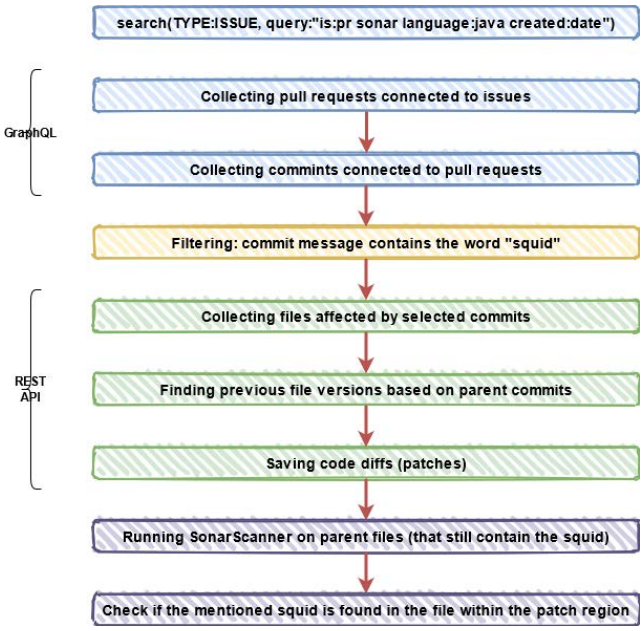


FIGURE 1. The data mining steps to identify true positive SonarQube warnings.

and kept only those where the previous version contained the matched "squid" (SonarQube warning) in the code region of the fix patch, while the current version did not. These previous file versions containing the later fixed SonarQube warnings formed our set of true positive samples, 47,015 warnings in total from 19,653 different Java source files.

To deal with the excessive amount of data returned by the queries and handle GitHub API limitations, we divided the queries into time ranges. Therefore we applied a date parameter in each query and summarized the results afterward.

2) MINING FALSE POSITIVE SAMPLES

To locate false positive SonarQube reports (i.e., to identify action A3 from developers), we had to perform a code search in the Java project repositories on GitHub. Since code search is unsupported in the GraphQL API, we performed all the mining steps with the help of the classic REST API. Figure 2 depicts the whole process.

We started by performing a code search with the search term `NOSONAR`. We then downloaded the sources of all the matched source files, removed the "`//NOSONAR`" comments from them, and ran SonarQube on these modified files. The reported SonarQube warnings in the `NOSONAR` lines formed our set of false positive samples, 177,469 warnings in total from 18,333 different Java source files. It was not uncommon to get multiple SonarQube warnings on the same `NOSONAR` lines, in which cases we considered all these warnings to be false positives.

3) POST-PROCESSING AND STATISTICS OF THE DATASET

During the data collection, we identified 337,438 warning instances from 492 different SonarQube issue types across 11,269 Java projects (47,220 true positive warnings from 175, while 290,218 false positive warnings from

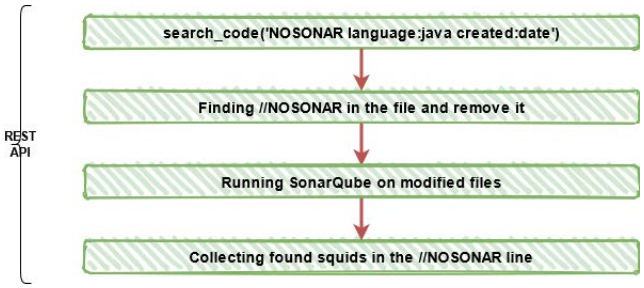


FIGURE 2. The data mining steps to identify false positive SonarQube warnings.

317 different SonarQube issue types). However, for building proper ML models, we needed only the issue types having samples both from the true positive and the false positive category. Therefore, we performed a filtering step and removed those SonarQube issue types for which we had only true positive or false positive samples. After this filtering step, we ended up with 160 warning types and 224,484 warning samples in total coming from 9,958 different Java projects. Figure 3 shows the distribution of true positive and false positive samples within the top 10 SonarQube warning types having the most instances in the dataset. The unique ids of the SonarQube issue types (i.e., squid) can be seen on the left side, while the total number of warning samples collected from that issue type is displayed on the right side. For the complete distribution of samples in all the 160 issue types, see the replication data package.[4]

4) EXAMPLES OF THE MINED WARNINGS

To motivate our work and demonstrate that warnings are indeed valid in one context but false positives in others, we show an example from the mined dataset. Listings 3a and 3b both show code snippets from two different Java projects that contain a "Null should not be returned from a "Boolean" method (squid:S2447)" SonarQube warning (at line 5 in both cases). However, the warning in Listing 3a is explicitly ignored using the "`//NOSONAR`" comment, thus considered to be a false positive. The developers of the code even added an explanation why this warning should be omitted: "null is used for further comparison". In contrast, the code shown in Listing 3b is a fix patch, where the developers changed the `null` returning line to an explicit `return Boolean.FALSE`. Therefore, in this case, the developers agreed with the warning and fixed the issue, which is an example of a true positive report for the same squid:S2447 warning.

This example nicely demonstrates that turning on and off entire SonarQube rules will not solve the issue of false positives. Moreover, turning off an entire rule that sometimes produces a false report will hide all the true warnings it might have produced as well. Our idea (similar to [10], [11]) is to use the code context of the issued warning (e.g., some lines before and after the reported line) to capture differences

[4]https://doi.org/10.5281/zenodo.5885653

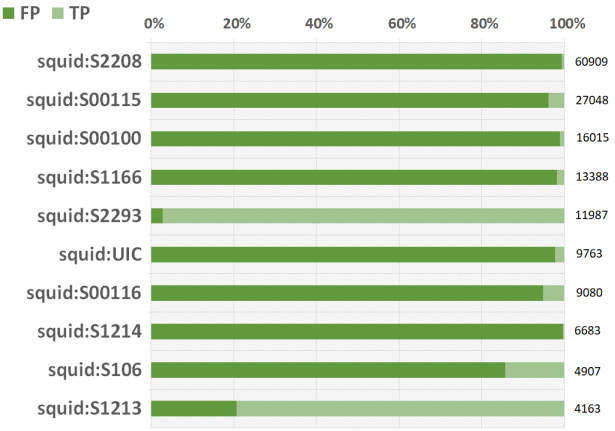


FIGURE 3. The distribution of true positive (TP) and false positive (FP) samples for the 10 most common SonarQube warning types.

```
1       private Boolean getBooleanValue(Row row) {
2         if (aggregation == Aggregation.MIN ||
                aggregation == Aggregation.MAX) {
3           return row.getBool(BOOL_POS);
4         } else {
5           return null; //NOSONAR, null is used for
                further comparison
6         }
7       }
```

(a) Ignored (i.e., false positive) squid:S2447

```
1       @@ -152,7 +152,7 @@ public
            JanusMediaConstraints getMedia() {
2
3         @Override
4         public Boolean getTrickle() {
5           -                          return null;
6           +                          return Boolean.
                FALSE;
7         }
```

(b) Fixed (i.e., true positive) squid:S2447

Listing 3. Null should not be returned from a "Boolean" method (squid:S2447) false and true positive samples.

in code structures where particular rules are more likely to be true or false. For instance, in this particular case, one difference might be the presence of the "else" keyword in the context of the false positive report and its lack in the true positive case. Since most of the SonarQube warning types detect pretty localized issues, we hypothesize that a local context of the warning might contain enough information to distinguish between true and false reports. The details on how we embed the source code context to extract feature vectors for machine learning and how we train the models are described in Section III-C.

C. METHOD FOR IDENTIFYING FALSE POSITIVE WARNINGS

Built on the collected real-world warning samples, we want to perform a binary classification task using ML models on the code snippets containing a SonarQube warning report. The input of the models is the vector representation (i.e., source code embedding) of the code context containing the warning

report. The output of the models is a binary label whether the particular SonarQube warning is a true or false report in the given code context.

1) CODE CONTEXT REPRESENTATION

To represent the source code context of a SonarQube warning, we used an NLP approach, word2vec [18], on the tokenized form of the program. It means that we took the source code line of the SonarQube warning and its preceding and following `n` lines. The exact value of `n` can be adjusted. Once we have the code context snippet of a warning, we run the `javalang` open-source Java lexer and parser tool.[5]

We replaced each line in the context with the sequences of tokens (keywords like `package`, `for` or modifiers like `public` or types like `int`, `long`) produced by the lexer. We extended the base tokens of javalang with three others for single and multiline comments and javadoc comments. We used the set of different tokens as our vocabulary for word2vec and trained a language model for these words using Gensim [34]. We ended up with 66 different tokens as our final vocabulary size. For the corpus of Java projects needed to train the word2vec model, we used the public dataset used to create code2vec [35]. We employed the medium set with 0.5 million Java files.

Once we have the word2vec vectors for the individual Java language tokens, we are ready to assemble the final feature vectors for the code snippets that can be fed into the ML training/prediction pipeline directly. The complete process is depicted in Figure 4.

The size of the feature vector will be the following:

$$160 + (2 \times n + 1) \times w_{size} + 1,$$

where `n` is the number of lines we include in the context before and after the warning line and w_{size} is the length of word2vec vectors for the language tokens. The first 160 columns in the feature vector are the one-hot encoding of the SonarQube warning type we observe in the corresponding code snippet. Then, for each line in the source code context (the preceding, succeeding, and warning lines), we calculate the average of the word2vec vectors for the tokens representing this line and concatenate them. Lastly, the final column stands for the class label we want to learn, namely, if the SCA warning report is true or false positive in the code context. Since we want to evaluate the ML models from the aspect of false positive detection performance, we use class label `1` for false positive instances and class label `0` for true positive instances.

2) ML MODELS FOR WARNING CLASSIFICATION

To solve the classification problem, we tested four different algorithms on the feature vectors produced as described in Section III-C1:

- Decision tree (DTree) – a classic decision tree algorithm implemented in `scikit-learn`[6]
- Naive Bayes (NBayes) – the Naive Bayes approach implemented in `scikit-learn`
- Random forest (RForest) – a random forest algorithm implemented in `scikit-learn`
- Neural Network (NeuralNet) – a Neural Network implemented in `tensorflow`[7]

Besides these common algorithms for classification, we experimented with SVM too, but the training was slow while it produced initial results that were not better than the others; therefore, we excluded it from the further evaluation. We performed a hyper-parameter optimization using HyperOpt,[8] a tool designed to automate the search for optimal hyper-parameter configuration based on a Bayesian Optimization and supported by the SMBO (Sequential Model-Based Global Optimization) methodology.

To evaluate the models, we applied 10-fold cross-validation by splitting the dataset in 80-10-10% ratio for training, hyper-parameter optimization, and testing. The method of separation into folds requires some previous considerations. We applied a stratified sampling per squid type, which means we created the folds so that each warning type (i.e., squid) is represented and the distribution of the true and false positive instances within the warning types is preserved.

Since the training data was imbalanced (we collected 3x times as many false positive samples as true positives), we applied upsampling during the training phase. To have the best possible effect, we did not perform the upsampling for the whole training dataset, but rather upsampled the instances within each SonarQube warning type. It means that we selected instances for upsampling from each SonarQube type until we achieved even distribution of true and false positives within that warning type. It made sure that we have balanced class labels not just at the global training set level, but at the level of individual SonarQube warning types as well. We achieved upsampling by adding some noise to randomly selected instances from the minority class [36]. More specifically, we created new, synthetic training samples so that for each randomly selected instance from a particular warning type the 1% of the vocabulary average (i.e., the mean of the word2vec vectors of each token in the vocabulary) has been added.

IV. RESULTS

In this section, we detail the obtained results for the four ML models trained for detecting false positive SonarQube warning reports. First, we describe the selected hyper-parameters for the word embeddings and ML models that we found by using the HyperOpt hyper-parameter search tool. Next, we outline the prediction performances of the ML models with these hyper-parameters. Finally, we demonstrate the working of these predictions through a code sample.

[5]https://github.com/c2nes/javalang
[6]https://scikit-learn.org/stable/
[7]https://www.tensorflow.org/
[8]http://hyperopt.github.io/hyperopt/

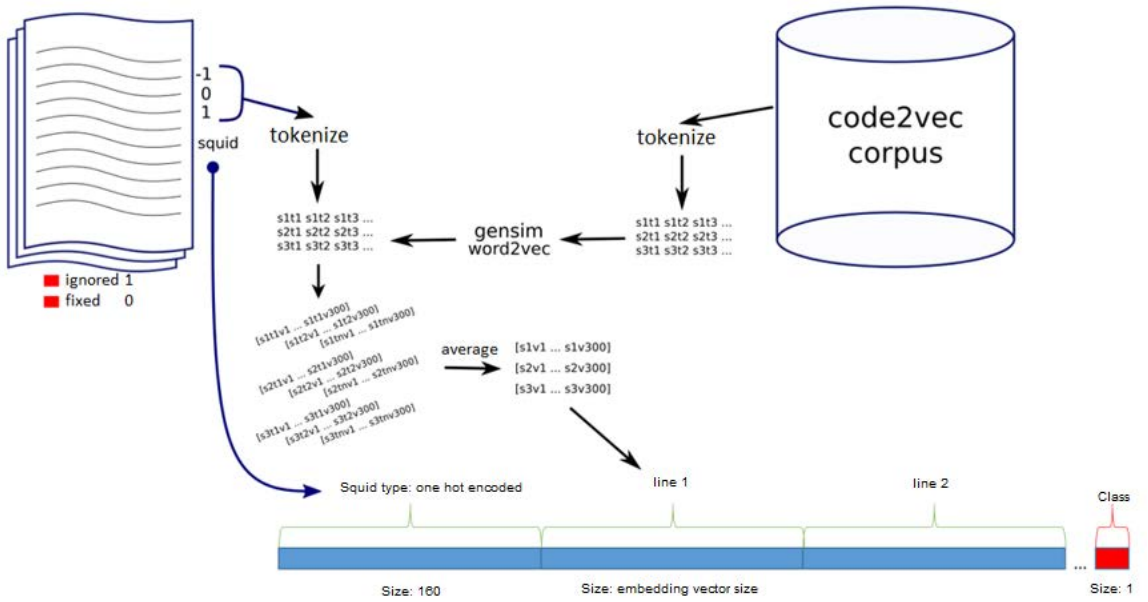


FIGURE 4. The final embedding process.

A. BEST HYPER-PARAMETERS

In the context of ML-based prediction, finding optimal hyper-parameters is a common task we need to solve. For this, we used an automated tool called HyperOpt. All the results presented later are obtained using the algorithms and the parameters described here.

1) Word2vec EMBEDDINGS AND CODE CONTEXT

We found that the word2vec vectors of size 64 work best, with a window size of 5. These values were obtained empirically by performing a grid search over the parameters 32, 64, 128, and 200 for vector size and 2, 5, 8, and 10 for window size. For `n`, the number of lines we consider before and after the warning line, we chose `2`. It means that the code context we work with is 5 lines long. It gave much better results than selecting a context of 3 (`n=1`, 1 line before and after the warning line). However, adding more lines to the context did not improve the overall prediction performance significantly. A context of 7 (`n=3`, 3 lines before and after the warning line), for example, produced practically the same results as the context of 5 in terms of performance measures (best F1 of 0.81 vs. 0.803 and best accuracy of 0.901 vs. 0.902 for NeuralNet for contexts of 7 and 5 lines, respectively). Furthermore, we observed that models based on the larger context had different confusion matrix characteristics. They produced about 20% more cases on average where we misclassified TP warnings to FP, which we consider the more dangerous direction of error (i.e., developers might miss real warnings). Therefore, we chose the smaller context.

2) NeuralNet

The input dimension of the NeuralNet is 480 ($160 + 5 \times 64$). We used three hidden dense layers with 800 neurons in each and two dropout layers with a dropout rate of 0.2. The activation function at the hidden layers was ReLU, while we used a sigmoid activation at the output layer. Further parameters were binary cross-entropy loss function, Adam optimizer, and a learning rate of 0.0001. The training ran for 100 epochs with a batch size of 512.

3) DTree

For the decision tree model, we used the Gini criterion, a max depth of 250, minimal samples in the leaves of 1, minimal samples split of 10, and the best splitter strategy.

4) RForest

For the random forest algorithm, we used the entropy criterion, a max depth of 100, maximal features of 480, minimal samples in the leaves of 1, minimal sample split of 10, and estimators of 250.

All the hyperparameters of the above ML algorithms have been identified by HyperOpt, while the Naive Bayes (NBayes) model does not require any hyper-parameters.

B. ML PREDICTION PERFORMANCE RESULTS

Note that we used a class label of '`1`' to denote false positive warnings and '`0`' for true positive reports. Therefore we present the standard performance measures for the false positive class labels. Table 1 summarizes the best achieved results for the four different ML models. These numbers are the averages over the 10-fold cross-validation.

NeuralNet achieved the highest recall. Overall, RForest performed the best; it made predictions with an accuracy of 0.91. Its precision of 0.874 is also the best (however, all the models performed well). It means that more than 87% of the warnings that the RForest model classified as false positive

TABLE 1. ML performances using the best hyper-parameters.

Algorithm	Accuracy	F1-score	Precision	Recall	MCC
DTree	0.872	0.750	0.755	0.745	0.664
RForest	**0.910**	**0.813**	**0.874**	0.760	**0.757**
NBayes	0.870	0.727	0.789	0.675	0.646
NeuralNet	0.902	0.803	0.831	**0.778**	0.739

TABLE 2. Confusion matrix for the RForest algorithm.

	Predicted	
Actual	False alarm	True alarm
False alarm	846 (TP)	88 (FN)
True alarm	270 (FP)	3118 (TN)

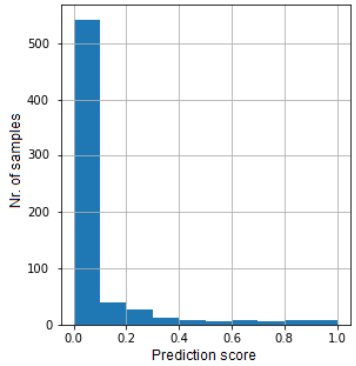
reports were indeed false positives. It is favorable as we want to filter out as few as possible true warnings. The recall of 0.76 means that the model can detect (and filter out) 76% of all the false positives produced by SonarQube (in this aspect, NeuralNet performed even better with 0.778). The best F1-score is 0.813; the model performances are quite balanced in terms of precision/recall.

Table 2 displays the confusion matrix of the RForest algorithm on the test set. As can be seen, the number of false positives compared to the true negatives (i.e., the true-negative rate) is low (8%), meaning that the model filters out only a few true SonarQube alarms, which is an essential property since when our model falsely identifies a true positive warning as a false hit, developers might not fix a real issue. Nonetheless, our method can be converted to rank warnings instead of filtering them, so we never lose any true positive warnings but can still help developers focus their effort.

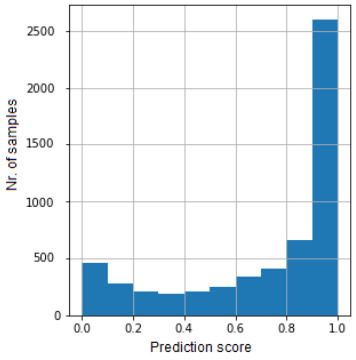
Similarly, the number of false negatives is low compared to the true positives (i.e., recall is high). This property (i.e., what portion of the false warnings can be filtered out with our method) is important for the practical relevance of the approach. Failing to identify a false positive warning will result in developers having to deal with a false alarm. Even though it causes no harm, it can be inconvenient; luckily, the model detects 76% of such false alarms. In line with the high accuracy measures, both the number of true positive and true negative instances are high, meaning that the model is capable of filtering out false alarms, but at the same time, it precisely identifies the true (i.e., actionable) warnings.

Looking at the type-wise performance measures, we can see that the models do not perform equally well for the different warning types. The accuracy varies from 0% (for *squid:S1210, squid:S00114*, and *squid:S3864*) to 100% (for *squid:S1067, squid:S2225, squid:S4201*, and *squid:S1215*) by SonarQube warning types with an average of 82.1%. Average precision, recall, and F1-scores for the 160 warning types are 70.54%, 55.34%, and 70.73%, respectively. The complete breakdown of accuracy and other performance measures to the level of individual SonarQube warning types can be found in the replication data package.

We did not perform a systematic evaluation of failed cases (i.e., when a model misclassifies a warning); however, we investigated a small random sample manually. We observed that the model fails mostly for warnings with a low number of instances in our training/test sets. These warnings have different severity ranging from minor to blocker according to SonarQube. Nonetheless, a deeper analysis is required to formally investigate the phenomena.



(a) Prediction scores for true positive warnings



(b) Prediction scores for false positive warnings

FIGURE 5. Distribution of NeuralNet prediction scores.

To get a better picture of the class labels assigned to the warnings, we performed a deeper analysis of the NeuralNet results. We chose to evaluate NeuralNet over the best performing RForest because the analysis required us to have a continuous prediction value that comes naturally for NeuralNet but not for the other algorithms. Additionally, NeuralNet performs close to RForest in every aspect, and it produces the highest recall. Therefore, we plotted the histograms of the NeuralNet model output scores (a continuous value from the [0,1] interval). Figure 5 shows the distribution of assigned scores for the true and false positive samples in our test dataset.

The histogram of the prediction output of the NeuralNet model for the true positive instances in our dataset is displayed in Figure 5a. We use a threshold of 0.5 for the classification, which means that each sample having a score lower than 0.5 will get the class label `0` (i.e., true positive). The vast majority of true positive SonarQube warning reports get a prediction score between 0 and 0.1, thus will be classified correctly with very high confidence. There are very few samples that get prediction scores higher than 0.5; therefore, get misclassified by the predictor. It is a very favorable property of the model because it will not filter out/reject true positive warnings due to falsely labeling them to be false positives.

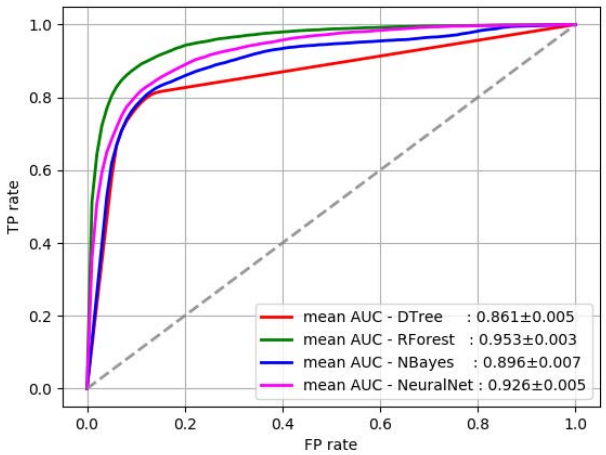


FIGURE 6. 10-fold cross-validation average ROC curves and AUC values.

On the other hand, as displayed in Figure 5b, most of the false positive instances get a prediction score above 0.8. It means that the model will assign a class label of `1` (i.e., false positive) to most of the false SonarQube reports with high confidence. However, there is a non-negligible amount of false positive instances that get a score close to 0. Not just that the model misclassify some of the false positive instances, but it does it with high confidence. However, this only means that it will not filter out some false positives, which is still more tolerable than filtering out true positives. Moreover, as we summarized in Table 1, the model will filter out 77.8% of all the false positives, which is already an improvement from a practical perspective compared to the raw SonarQube results.

To evaluate how our models work concerning the true positive/false positive rate trade-off, we calculated the average ROC curves and the corresponding area under the curve (AUC) measures. Figure 6 shows the ROC curves and AUC values calculated as the average of the 10-folds we performed for the evaluation.

In general, all the ML models have a good performance considering the AUC measure. RForest performs the best in this respect too, with an AUC of 0.953 but NeuralNet is very close with an AUC of 0.926. All the other models have an AUC close to 0.9, too. Based on this, we conclude that the models are efficient in distinguishing between true and false SonarQube warnings.

C. ML PREDICTION SAMPLE

To demonstrate the working mechanism of the prediction models, we show the prediction scores of the NeuralNet model on a concrete example. We then remove some of the tokens from the context and re-run the prediction to observe their impact on the overall prediction. Consider the true positive "Null pointers should not be dereferenced (squid:S2259)" SonarQube warning instance from our test dataset shown in Listing 4. The source code snippet containing the SonarQube warning and its context is displayed in Listing 4a. The variable `bos` is a

```
1       } // end catch: java.io.IOException
2       finally {
3         try{ bos.close();} catch( Exception e){}
4       } // end finally
```

(a) A sample true positive squid:S2259 warning

```
1       Separator SingleLineComment
2       Keyword_finally Separator
3       Keyword_try Separator Identifier Separator
            Identifier Separator Separator Separator
            Separator Keyword_catch Separator
            Identifier Identifier Separator Separator
            Separator
4       Separator SingleLineComment
```

(b) Tokenized sample code with a ML prediction output of 0.03 (0 means valid warning)

```
1       Separator SingleLineComment
2       Keyword_finally Separator
3       Keyword_try Separator Identifier Separator
            Identifier Separator Separator Separator
            Separator Keyword_catch Separator
            Identifier Identifier Separator Separator
            Separator
4       Separator SingleLineComment
```

(c) Tokenized sample code leaving out the tokens "SingleLineComment" with a ML prediction output of 0.24

```
1       Separator SingleLineComment
2       Keyword_finally Separator
3       Keyword_try Separator Identifier Separator
            Identifier Separator Separator Separator
            Separator Keyword_catch Separator
            Identifier Identifier Separator Separator
            Separator
4       Separator SingleLineComment
```

(d) Tokenized sample code leaving out the keyword "finally" with a ML prediction output of 0.54

Listing 4. Null pointers should not be dereferenced (squid:S2259) predictions.

`Base64.OutputStream` type object that is initialized to `null` earlier; therefore, the possible null pointer dereference warning is valid (the issue has been fixed by the developers).

Based on the tokenized form of this original code snippet (see Listing 4b) and the feature vector created by the word2vec embedding of these lines, the NeuralNet model outputs a prediction score of 0.03, which corresponds to a confident `0` class label (i.e., true positive instance). Listings 4c and 4d show the modified token sequences and the model prediction scores after artificially removing some of the tokens. For instance, if we remove the comments from the code context (Listing 4c), the prediction score increases to 0.24. It still corresponds to a class label of `0`, but the model's confidence is much lower. In the case of removing the `finally` keyword (see Listing 4d), however, the model becomes entirely clueless. It predicts a score of 0.54 that corresponds to total uncertainty and even yields a wrong class label of `1`. This example demonstrates how the model looks for a particular set of tokens to identify patterns with which the model can connect the code snippet to be classified to the training samples it saw during the training phase to decide if the warning is true or false positive (i.e., if the structure

resembles more to the code snippets with the class label of `0` or `1`).

V. THREATS TO VALIDITY

Our data mining approach implies some threats to the validity of the collected data. The presence of the "`//NOSONAR`" comment in the source code might be temporal, applied as a quick fix before eventually removing the issue. Code generator created files might contain "`//NOSONAR`" in big bulks, which does not necessarily mean the warnings are false positives, just that they are irrelevant in generated code. Nonetheless, if we extend the scope of our prediction model so that we aim to identify and filter "unactionable" warnings instead of strictly false alarms (similarly to related work), both of the above cases yield correct data samples.

Regarding the source code embedding, we did not consider other techniques than word2vec. Our goal was to keep source code embedding as simple as possible and provide an extensive evaluation of a real-world dataset on a scale never done before in this area. As the context we embed is relatively small, we do not expect the results to be much affected by the chosen embedding method. Furthermore, we kept the separator and identifier tokens during the word2vec tokenization, which have excessive numbers. It is common in the literature to remove these frequent tokens from the vocabulary, but we have performed experiments with and without these tokens and found that including them yields better results.

When mining false positive data samples, we omitted the cases when developers do nothing with a SonarQube alarm (i.e., the warning stays in the code for a long time). Even though it might be the sign of developers ignoring the alarm (i.e., a false positive alert), we cannot be sure as the warning might have been simply undiscovered. Since we wanted to minimize the number of incorrect samples in our training dataset, we skipped these cases.

We applied stratified random sampling to split the data for training and evaluating prediction models, which does not take the chronology of code changes into account (i.e., we might train models on warnings that appear in code created later than the code on which we test the model). This property might be crucial in particular domains, for example, within-project defect prediction or malware detection. However, we treat warnings locally in isolation independent of the containing project. Since there is no chronological dependence between code contexts in which particular warnings get fixed or omitted, our results are valid.

We evaluated our method only on SonarQube warnings; therefore, its generalizability is hard to assess. However, SonarQube includes many warning checks that exist in other SCA tools as well (e.g. "squid:S2384 Mutable members should not be stored or returned directly" in SonarQube is very similar to "EI: May expose internal representation by returning a reference to a mutable object (EI_EXPOSE_REP)" in SpotBugs). Moreover, considering the similar, local nature of SCA warnings, we expect the method to generalize well across various SCA tools.

VI. CONCLUSION

The excessive amount of false positive warnings produced by SCA tools is still one of the obstacles to their adoption in practice. Even though researchers have proposed several techniques for reducing the number of false alarms or equivalently identifying "actionable" alarms, the problem is still unsolved in general. Most of these techniques focus on a specific subset of SCA warnings (e.g., security-related alarms or memory handling issues), and more importantly, they are evaluated on small synthetic benchmarks (e.g., OWASP benchmark [12] or Juliet [13]), manually validated small datasets (i.e., few hundred samples) or closed corporate data.

In this work, we presented a dataset containing 224,484 SonarQube true positive and false positive warnings combined that we mined from 9,958 Java GitHub projects. To the best of our knowledge, this is by far the most complete public real-world dataset in this area. To showcase its practical benefit, we trained ML models built on top of the word2vec representation of the code context using the dataset, providing a lightweight method for identifying and filtering false positive SonarQube alarms. SonarQube is one of the most widely used SCA tools today; our method with the best model works with an accuracy of 91% (best F1-score of 81.3% and AUC of 95.3%) for the classification of 160 different types of SonarQube warnings and identifies 77.8% of all the false positive alarms, while filters out (i.e., misclassifies) only 8% of the true positive alarms on average. The technique relies on an NLP-based source code embedding of the warning and its context that Koc *et al.* [10] already demonstrated to be efficient for this task. Based on the performance measures reported in the literature, our method is among the best ones, while most of the existing approaches are evaluated only on small, synthetic benchmarks (like OWASP Benchmark or Juliet). Nonetheless, we focused more on producing a real-world warning dataset, underpinning its practical relevance, and opening up new possibilities for evaluating, comparing, and assessing the practical feasibility of false alarm filtering methods, rather than improving the state-of-the-art algorithms per se.

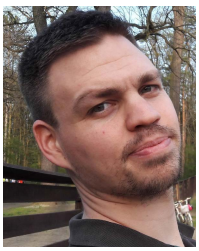
Many modern CI/CD pipelines already include the SonarQube analysis into which our ML-based post-processing solution could be easily integrated. The prediction model does not need time-consuming deep analysis to get its input, only the word2vec representation of the warning line and its small context (i.e., two lines before and after) to be classified. These properties make the method ideal for practical applications.

REFERENCES

[1] L. M. R. Velicheti, D. C. Feiock, M. Peiris, R. Raje, and J. H. Hill, "Towards modeling the behavior of static code analysis tools," in *Proc. 9th Annu. Cyber Inf. Secur. Res. Conf. (CISR)*. New York, NY, USA: ACM, 2014, pp. 17–20, doi: 10.1145/2602087.2602101.

[2] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *Proc. 10th Annu. Int. Static Anal. Symp.* Berlin, Germany: Springer, 2003, pp. 295–315.

[3] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 672–681.
[4] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2016, pp. 470–481.
[5] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *Proc.6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 45–54.
[6] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proc. 7th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE)*, 2007, pp. 1–8.
[7] S. Heckman and L. Williams, "A model building process for identifying actionable static analysis alerts," in *Proc. Int. Conf. Softw. Test. Verification Validation*, Apr. 2009, pp. 161–170.
[8] U. Yuksel and H. Sozer, "Automated classification of static code analysis alerts: A case study," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2013, pp. 532–535.
[9] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter, "Learning a classifier for false positive error reports emitted by static code analysis tools," in *Proc. 1st ACM SIGPLAN Int. Workshop Mach. Learn. Program. Lang.*, Jun. 2017, pp. 35–42.
[10] 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 *Proc. 12th IEEE Conf. Softw. Test., Validation Verification (ICST)*, Apr. 2019, pp. 288–299.
[11] S. Lee, S. Hong, J. Yi, T. Kim, C.-J. Kim, and S. Yoo, "Classifying false positive static checker alarms in continuous integration using convolutional neural networks," in *Proc. 12th IEEE Conf. Softw. Test., Validation Verification (ICST)*, Apr. 2019, pp. 391–401.
[12] *OWASP Benchmark Project*. Accessed: Jul. 22, 2021. [Online]. Available: https://owasp.org/www-project-benchmark/
[13] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, Oct. 2012.
[14] *SonarQube Website*. Accessed: Jul. 23, 2021. [Online]. Available: https://www.sonarqube.org/
[15] J. García-Munoz, M. García-Valls, and J. Escribano-Barreno, "Improved metrics handling in SonarQube for software quality monitoring," in *Proc. 13t Int. Conf. Distrib. Comput. Artif. Intell.* Cham, Switzerland: Springer, 2016, pp. 463–470.
[16] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are SonarQube rules inducing bugs?" in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2020, pp. 501–511.
[17] M. T. Baldassarre, V. Lenarduzzi, S. Romano, and N. Saarimäki, "On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube," *Inf. Softw. Technol.*, vol. 128, Dec. 2020, Art. no. 106377.
[18] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*.
[19] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, May 2016, pp. 297–308.
[20] J. Wang, S. Wang, and Q. Wang, "Is there a 'golden' feature set for static warning identification? An experimental evaluation," in *Proc. 12th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Oct. 2018, pp. 1–10.
[21] Y. Kim, J. Lee, H. Han, and K.-M. Choe, "Filtering false alarms of buffer overflow analysis using SMT solvers," *Inf. Softw. Technol.*, vol. 52, no. 2, pp. 210–219, Feb. 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058490900175X
[22] S. Arzt, S. Rasthofer, R. Hahn, and E. Bodden, "Using targeted symbolic execution for reducing false-positives in dataflow analysis," in *Proc. 4th ACM SIGPLAN Int. Workshop State Art Program Anal.*, Jun. 2015, pp. 1–6.
[23] 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, pp. 363–387, Apr. 2011.
[24] S. S. Heckman, "Adaptive probabilistic model for ranking code-based static analysis alerts," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE Companion)*, May 2007, pp. 89–90.
[25] S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *Proc. 2nd ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Oct. 2008, pp. 41–50.
[26] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin, "ALETHEIA: Improving the usability of static security analysis," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 762–774.
[27] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings," in *Proc. 13th Int. Conf. Softw. Eng. (ICSE)*, 2008, pp. 341–350.
[28] T. T. Nguyen, P. Maleehuan, T. Aoki, T. Tomita, and I. Yamada, "Reducing false positives of static analysis for SEI CERT C coding standard," in *Proc. IEEE/ACM Joint 7th Int. Workshop Conducting Empirical Stud. Ind. (CESI) 6th Int. Workshop Softw. Eng. Res. Ind. Pract. (SER IP)*, May 2019, pp. 41–48.
[29] *SEI CERT*, CERT Coding Standards, Softw. Eng. Inst., Carnegie Mellon Univ., Pittsburgh, PA, USA, 2016.
[30] *SonarQube Customers*. Accessed: Aug. 30, 2021. [Online]. Available: https://discovery.hgdata.com/product/sonarqube
[31] N. Saarimaki, M. T. Baldassarre, V. Lenarduzzi, and S. Romano, "On the accuracy of SonarQube technical debt remediation time," in *Proc. 45th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Aug. 2019, pp. 317–324.
[32] *GitHub REST API*. Accessed: Aug. 30, 2021. [Online]. Available: https://docs.github.com/en/rest
[33] *GitHub GraphQL API*. Accessed: Aug. 30, 2021. [Online]. Available: https://docs.github.com/en/graphql
[34] R. Řehůřek and P. Sojka, "Software framework for topic modelling with large corpora," in *Proc. LREC Workshop New Challenges NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50. [Online]. Available: http://is.muni.cz/publication/884893/en
[35] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proc. Program. Lang. (ACM)*, vol. 3, Jan. 2019, pp. 1–29.
[36] R. Mohammed, J. Rawashdeh, and M. Abdullah, "Machine learning with oversampling and undersampling techniques: Overview study and experimental results," in *Proc. 11th Int. Conf. Inf. Commun. Syst. (ICICS)*, Apr. 2020, pp. 243–248.



PÉTER HEGEDŰS received the Ph.D. degree in computer science from the University of Szeged, in 2015.

He currently works as an Assistant Professor at the Software Engineering Department, University of Szeged, and as a Researcher at FrontEndART Ltd. Besides teaching and research involvement, he also takes part in various software development projects as a Project Manager and a Lead Developer. His publication record consists of over 50 papers that appeared in top conferences and high-impact journals. His research interests include software maintainability models, deep learning applications, source code analysis, and vulnerability detection and prediction. He was a PC Member of the CSMR, MSR, ICCSA, and SQM conferences, and also holds a Bolyai János Research Scholarship.

RUDOLF FERENC received the Ph.D. degree in computer science from the University of Szeged, in 2005, and the Habilitation degree, in 2015.

He is currently an Associate Professor and acting as the Head of the Department of Software Engineering, University of Szeged. He leads the Static Code Analysis Group, which develops tools for analyzing the source code of various languages. These tools calculate code metrics, and detect coding issues and duplications. He has more than 100 publications in these fields with over 2000 citations. He is leading several research and development projects, which are related to quality assessment, improvement and architecture reconstruction of software systems for major banks and software development companies in Hungary. His research interests include static code analysis, metrics, quality assurance, design pattern and antipattern mining, and bug detection. He has been serving as a Program Co-Chair and a Program Committee Member for the major conferences in this field, such as ICSE, ICSME, ESEC/FSE, SANER, CSMR, WCRE, ICPC, SCAM, and FASE, since 2005.