A self-adaptive approach for assessing the criticality of security-related static analysis alerts

Miltiadis Siavvas[⋆], Ilias Kalouptsoglou, Dimitrios Tsoukalas, and Dionysios Kehagias

Centre for Research and Technology Hellas, Thessaloniki, Greece
{siavvasm,iliaskaloup,tsoukj,diok}@iti.gr

Abstract. Despite the acknowledged ability of automated static analysis to detect software vulnerabilities, its adoption in practice is limited, mainly due to the large number of false alerts (i.e., false positives) that it generates. Although several machine learning-based techniques for assessing the actionability of the produced alerts and for filtering out false positives have been proposed, none of them have demonstrated sufficient results, whereas limited attempts focus on assessing the criticality of the alerts from a security viewpoint. To this end, in the present paper we propose an approach for assessing the criticality of security-related static analysis alerts. In particular, we develop a machine learning-based technique for prioritizing and classifying security-related static analysis alerts based on their criticality, by considering information retrieved from the alerts themselves, vulnerability prediction models, and user feedback. The concept of retraining is also adopted to enable the model to correct itself and adapt to previously unknown software products. The technique has been evaluated through a case study, which revealed its capacity to effectively assess the criticality of alerts of previously unknown projects, as well as its ability to dynamically adapt to the characteristics of the new project and provide more accurate assessments through retraining.

Keywords: Software Quality · Software Security · Automated Static Analysis · Self-adaptive Systems · Vulnerability Prediction

1 Introduction

Security is an important aspect for modern software products, especially for those that are accessible through the Internet and handle sensitive information. The exploitation of a single vulnerability may lead to far reaching consequences both for the users and for the owning enterprises, ranging from information exposure to reputation damages and financial losses [1]. As a result, the software industry has recently shifted its focus towards building software that is highly secure from the ground up [2, 3]. For this purpose, several mechanisms have been proposed for adding security during the overall software development lifecycle (SDLC) [2, 3]. Among them, automated static analysis (ASA) has been proven

[⋆] Corresponding Author

effective in uncovering software vulnerabilities, and therefore adding security, during the coding phase of the SDLC [4–6], whereas several well-established secure SDLCs (e.g., Microsoft's SDL [7]), as well as leading technological firms like Google, Microsoft, and Intel (according to the BSIMM[1] initiative) highlight its importance in improving software security.

Despite its effectiveness in detecting vulnerabilities, static analysis has been observed to be underused in practice [8, 9]. The main reason for its limited adoption is that it tends to produce a large number of false alerts (i.e., false positives), that is, alerts that do not correspond to actual issues. This leads to the generation of large reports of alerts that developers and reviewers have to manually inspect in order to detect those that are actionable (i.e., that correspond to actual issues, which require immediate fix). This process, called triaging [8], is very time-consuming and effort demanding, discouraging developers from using ASA in practice. Several attempts have been made in the literature to reduce the number of the produced false positives. Since the construction of an ASA tool able to detect all existing issues while maintaining satisfactory performance is an undecidable problem [10], the majority of the research endeavors focus on proposing techniques that post-process the produced alerts, to detect those that are actionable, known as actionable alerts identification techniques (AAITs).

Despite the large number of AAITs that have been proposed until today, none of them have demonstrated very sufficient results [11, 10, 12]. In fact, their accuracy is observed to drop significantly when applied to software products that have not been used during their training. In addition to this, almost no contributions have been made with respect to extending these techniques towards the security realm, i.e., for assessing the criticality of security-related static analysis alerts (i.e., potential vulnerabilities). This would be beneficial for the production of secure software, as it would allow developers better prioritize their refactoring activities by focusing on fixing issues that are more likely to correspond to actual vulnerabilities. An extension is necessary, as vulnerabilities are considered special types of bugs that exhibit unique characteristics [13].

To this end, in the present paper, we propose a technique for assessing the criticality of security-related static analysis alerts. In particular, we propose a technique for prioritizing (and classifying) security-related static analysis alerts based on their criticality, by taking into account information retrieved from (i) the alerts themselves, (ii) vulnerability prediction models, and (iii) user feedback. The proposed technique is based on machine learning models, with emphasis on neural networks, which were built based on data retrieved from static analysis reports of real-world software applications. It is grounded on the generation of a general model based on real world data and on the regular application of retraining (based on user feedback) so that the model could provide more accurate results for the project on which it is applied. The proposed technique is operationalized in the form of web services that can be used in practice, whereas an intuitive web-based interface is also provided. Finally, the approach is demonstrated through a case study on a real-world commercial software product.

[1] https://www.bsimm.com/

2 Related Work

A large number of techniques have been proposed with the purpose to report alerts that are actionable and eliminate false positives produced by ASA. These techniques, which are known as actionable alerts identification techniques (AAITs), typically utilize machine learning to discriminate between actionable and non-actionable alerts [11, 14, 10, 15]. They are normally classified into two categories [14, 10]: (i) classification AAITs, and (ii) prioritization AAITs.

Classification AAITs (e.g., [16]) classify alerts into two groups, particularly alerts that are likely to be actionable (i.e. actual issues identified by static code analyzers that require fix) and alerts that are likely to be unactionable (i.e. false positives or less critical issues that are reported by static code analyzers and do no require immediate correction), and they prune those that are marked as unactionable. For instance, a static analysis alert that corresponds to a bug that can lead to a system crash (e.g., memory leak) is an actionable alert, whereas a wrong naming convention issue (e.g., a variable name that is very long) can be considered unactionable. The main advantage of the classification AAITs is that, due to alert pruning, they lead to a significant reduction in the number of the alerts that are reported to the developers. However, these techniques may lead to the introduction of false negatives, as they may mistakenly prune alerts that are in fact actionable due to the model's error [11, 14, 10]. Prioritization AAITs (e.g., [17, 18]) rank the alerts based on their "actionability", i.e., their likelihood to be actionable, without eliminating any of the reported alerts. This allows the developers to better prioritize their fortification activities, by starting their refactoring from alerts that are more likely to be actionable. Since no pruning is applied, no false negatives are introduced, but the volume of the alerts is not reduced as well, which may be still overwhelming.

None of these techniques has been widely-adopted in practice, mainly due to their inaccuracies, especially when applied to projects that were not used during their training [19, 14, 20]. In fact, the predictive performance of these models significantly drops, when applied to assess alerts from previously unknown projects, hindering, in that way, their practicality. For instance, Heckman et al. [17, 21] observed that updating the models adaptively may lead to more accurate rankings in future versions of a project. However, further work is needed [14].

AAITs focus on assessing the "actionability" of static analysis alerts in general. Security-related static analysis alerts are considered good indicators of software vulnerabilities [5, 6]. However, AAITs cannot be applied directly for assessing the criticality of security-related static analysis alerts, since software vulnerabilities are considered special types of software bugs, with unique characteristics [13]. In particular, the accessibility of a security bug from the attack surface determines its criticality, something that does not hold for common bugs [22, 4]. In fact, the criticality of a vulnerability, apart from its type, also depends on project-specific parameters, like its location in the source code, its reachability from the attack surface, the system configuration, etc. [23, 22, 4]. Therefore, these parameters (i.e., code semantics) need to be considered for assessing the criticality of a security issue, normally through code analysis and user feedback.

Hence, AAITs need to be extended towards the security realm. Very limited contributions exist toward this direction. The most representative contribution was made by Baca [4], who used context-sensitive data flow analysis in order to detect locations of code that are tainted, and marked all the alerts that belonged to these lines as potentially critical from a security viewpoint. However, no model was built, whereas emphasis was given only on the reachability, neglecting other information that may provide inference about the actual criticality of the alerts.

From the above analysis, it is clear that none of the existing AAITs has managed to sufficiently address the problem and therefore to be broadly adopted in practice. The main reason for this is that the accuracy drops when they are applied to previously unknown projects. Another open issue is that little work has been conducted with respect to the criticality of security-related static analysis alerts, which are alerts that are characterized by unique characteristics. To this end, in the present paper we focus on security-related static analysis alerts, and we propose an ML-based approach for assessing their criticality, leveraging the concept of retraining based on user feedback. The approach supports both the classification and prioritization concepts. Another novelty is that it considers information from vulnerability prediction, which has not been studied before.

3 Methodology

3.1 Overview of the Methodology

In Figure 1, a high-level overview of the proposed approach, as well as of the methodology that was adopted for constructing the required model is illustrated. The left part of Figure 1 (termed as "Model Construction") demonstrates the process that was followed for building the ML model that is used as the basis of our proposed approach based on real-world data. More specifically, as can be seen by Figure 1, we split the open-source projects that constitute our dataset into a list of software classes and then these classes were analysed both by a static code analyzer and by a vulnerability prediction model (VPM), to collect the required features of the dataset. Subsequently, a process for assigning labels in the dataset was followed, and, after the pre-processing, the final dataset was produced. The extracted features of the classes of the final dataset (i.e., static analysis alerts and vulnerability prediction results) along with their corresponding labels composed the input to our supervised learning model during its training. The right part of Figure 1 (termed as "Model Execution") illustrates how the produced model is used in practice for assessing the criticality of security-related static analysis alerts. In brief, as can be seen by the figure, the produced model receives as input a new static analysis alert and classifies it as critical or non-critical. As will be discussed later, it also reports the likelihood of the alert to be critical from a security viewpoint, allowing the developers to prioritize the reported alerts based on their criticality. Finally, the users are allowed to express their disagreement with respect to the model decisions by providing feedback. This feedback is stored in order to be used for the future retraining (see Section 3.4).

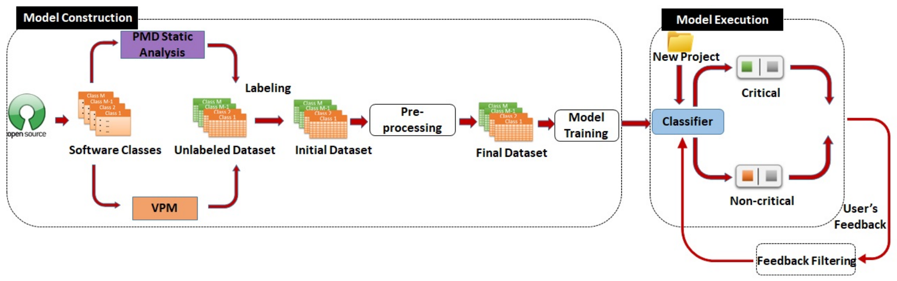


Fig. 1. High-level overview of the proposed approach

The overall approach that is depicted in Figure 1, can be summarized in the following seven steps:

1. **Data Definition**. The first step of the study is to define the input variables that will be used for the construction of the ML models, as well as the class attribute of the model. Regarding the input variables, we decided to use information retrieved from the static analysis alerts themselves (e.g., vulnerability type, severity, etc.), along with information retrieved from vulnerability prediction models (see Section 4). The model's output is actually a binary prediction, indicating whether the analyzed alert is critical (from a security viewpoint). It also reports a continuous value, indicating the alert's "criticality", i.e., its likelihood to correspond to a critical security issue.
2. **Data Collection**. In this step, we applied a popular automated static analysis tool that was properly configured in order to report only security-related issues, on two real-world software projects, in order to retrieve the alerts that exist in source code. Text mining-based vulnerability prediction models were also used to get information about the vulnerability hot spots on source code (see Section 4). Regarding the labeling of these alerts as critical or non-critical, a manual code review method was followed (see Section 3.2).
3. **Data Pre-processing**. Pre-processing is responsible for bringing the dataset in a form suitable for ML model training. More specifically, the categorical features were encoded in 1-hot vector representation. No sampling techniques were performed since the dataset is quite balanced.
4. **Classification Techniques Selection**. In this step, the most suitable classification techniques are selected. A comparison of these techniques is also performed. Our selection is based on classification evaluation metrics.
5. **Model Training**. This step is responsible for building the model selected at the previous step, based on a carefully selected set of hyper-parameters.
6. **Model Execution**. This step corresponds to the actual execution of the produced model in practice. The pre-trained model is used in order to assess the criticality of previously unknown security-related static analysis alerts. To facilitate the adoption of the approach in practice, it has been operationalized in the form of a standalone tool (see Section 4).

7. **Retraining Process**. This step corresponds to the process of model retraining. The user is allowed to express their disagreement to specific decisions of the model, by providing their feedback. This feedback is then utilized (under specific circumstances described in Section 3.4) in order to retrain the model and provide more accurate results for the project to which it is applied.

3.2 Data Collection and Pre-processing

For the purposes of the present study, a dataset was created based on two real-world software products developed by colleagues in CERTH as part of ongoing and past EU Projects. The reasoning behind the decision to use in-house software products for the construction of our dataset is threefold. Firstly, we have complete access to the source code of these products, allowing us to verify the correctness of the retrieved data. Secondly, we have direct communication with their developers, allowing us to request their assistance with respect to the manual labeling of the produced alerts. Finally, it also allows us to request a sufficient number of instances, leading to the construction of reliable ML models.

The selected software projects are mid-size projects written in Java programming language with an approximate lifetime of three years. The first project is a cloud-based platform for analyzing software products written in Java and C/C++ with respect to their quality. The platform is implemented as a collection of RESTfull microservices that are interconnected. The second project is a crawler that crawls projects from online Git-based repositories (e.g., GitHub, Bitbucket, etc.) with the purpose to find software projects that have dependencies on third-party software that contain known vulnerabilities. The tool is accessible through command line and it is actually based on the OWASP Dependency Check[2] tool. For privacy reasons, the names of the projects are not disclosed, since they are not open-source projects.

For the construction of the dataset, a static code analyzer, which was properly configured for detecting security issues through the utilization of a novel Security Assessment Model [2, 24] (see Section 4), was applied on these software projects. The SAM is able to detect important security-related issues, including Null Pointer, Resource Handling, Exception Handling, Synchronization, and Logging issues [24]. The produced static analysis alerts were presented to the actual developers of the associated projects, who labelled them as critical and non critical. Since the developers were not security experts, in order to prevent instances of mislabelling, the labelling was inspected by the authors of the paper. Several instances of potential mislabelling were spotted and discussed with the developers. Based on the discussion outcomes the final labelling was decided.

Apart from alert-specific information, we decided to integrate security-related information retrieved from the software project itself. In particular, we applied text mining-based deep learning vulnerability prediction models [25–27] (see Section 4), in order to spot the security hot-spots of the software products, i.e., software classes that are likely to contain vulnerabilities. By highlighting the

[2] https://owasp.org/www-project-dependency-check/

security hot-spots, we expect our approach to consider as more critical those static analysis alerts that belong to the identified hot-spots. We included this information in the produced dataset, to empirically examine this assumption through the construction of ML models.

In brief, the features that we focused on and we used for model's training are the rule name, ruleset name, priority, and vulnerability score. It should be noted that the first three features are alert specific, whereas the last one is project specific. Since the approach is based on a binary classifier, the outcome of the model is a binary value (which is termed *crtiticality flag*), i.e., 0 or 1, where 0 denotes that the analyzed alert is not critical and 1 that the analyzed alert is critical. The model also reports a *criticality score*, i.e., a continuous value in the [0,1] interval that indicates the likelihood of the analyzed alert to be critical from a security viewpoint. A description of the selected features is provided below:

- The **rule name** is the name of the rule of the static code analyzer that is violated. It indicates the type of the issue that is reported by the alert.
- The **rule set** is a grouping of code analysis rules that can be detected by the selected static code analyzer. It actually indicates the broader category that the issue that is reported by the alert belongs to.
- **Priority** is the severity score provided by the vendor and it represents the importance of each alert based on static analysis. It usually gets discrete values between one (more severe) and five (less severe). It should be noted that the vendor-specific priority cannot be used solely as a reliable measure of the criticality of security related alerts, as it is highly subjective and it neglects the important code semantics that may affect the criticality of security issues [8, 13, 22]. It needs to be used in conjunction with other features.
- **Vulnerability score** is the score produced by a vulnerability prediction model. Actually, it is the probability of a software component to be vulnerable, and therefore it receives a continuous value in the [0,1] interval.

The final dataset comprises 1200 alerts produced by the aforementioned process. From these 1200 alerts, 650 are marked as non-critical, whereas the rest 550 are defined as critical. Hence, the final dataset is rather balanced, which is important for the construction of the ML models that are described in the following sections, as it reduces the probability of overfitting. From a preprocessing viewpoint, since the "rule name" and "rule set" features are categorical variables, and considering that ML models understand numerical values, 1-hot vector representation was used.

3.3 Model Selection

After constructing the final dataset, the next step was to build a set of ML models and select the one that demonstrates the best predictive performance as the basis of our approach. For this purpose, different ML algorithms are trained in order to discriminate static analysis alerts, which point to code lines in the classes, between critical and non-critical. We investigate various ML algorithms, including Support Vector Machines (SVM), Random Forest, Decision

Tree and the Naïve Bayes method. We also examine the ability of deep learning, specifically the Multi-Layer Perceptron (MLP), to provide reliable predictions in our dataset. The evaluation is performed using 10-fold cross-validation. As performance indicators we use accuracy, precision and recall. The results of the evaluation process are presented in Table 1.

Table 1. Comparison of main machine learning classification algorithms

Classifier	**Accuracy**	**Precision**	**Recall**
Decision Tree	76.36	78.08	75.48
Naïve Bayes	56.06	53.50	88.33
Support Vector Machines	78.44	80.38	75.24
MLP	**86.33**	**90.52**	**83.33**
Random Forest	79.52	80.19	77.14

From Table 1, it is clear that the MLP is the best performing model. In fact, MLP is the only model that outperforms all the others with respect to all the three selected performance metrics, except the recall which is higher in the case of Naive Bayes. However, Naive Bayes accuracy and precision are observed to be the lowest (i.e., 53.5% and 56.06%) indicating that it generates an excessive number of false positives.

For the construction of these models, hyper-parameter tuning was employed to find the parameters that produce the optimal predictive performance for each model. More specifically, we performed the commonly used Grid-search method. The selected properties of the best performing model (i.e., the MLP model) are presented in Table 2. As can be seen in the Table 2, specific techniques were employed in order to avoid overfitting (e.g., dropout layer, regularizer, etc.) and make the produced model as generic as possible. As explained later, even if minor bias exists in the parameters of the original model, they are expected to be corrected through the applied retraining.

Table 2. The selected hyperparameters of the Multi-layer Perceptrons (MLPs)

Hyperparameter Name	**Value**
Number of Layers	4
Number of Hidden Layers	3
Number of Hidden Units (per Hidden Layer)	1000/500/50
Weight Initialization Technique	Glorot Xavier
Learning Rate	0.01
Gradient Descent Optimizer	Adagrad
Batch Size	128
Activation Function	relu
Regularizer	Max Norm (3)
Output Activation Function	sigmoid
Loss Function	Binnary Cross-entropy
Over-fitting Prevention	Dropout = 0.15 (after last hidden layer)

3.4 Retraining Process

The best performing model that was selected in Section 3.3, can be then used in practice in order to assess the criticality of previously unknown static analysis alerts. In brief, after executing the model, the predictions (i.e., *criticality flags* and *criticality scores*) of the analyzed alerts are returned to the user and he/she can agree or disagree with the proposed criticalities. The users are equipped with the capacity to correct the model and improve its predictive performance, by sending their feedback to the model and retraining it. Developers can change the criticality of any alert they believe the model has classified wrongfully. They are able to submit to the system a non-critical alert as critical and vice versa, depending on which alerts they consider to be critical or not, for their own code.

As already stated, this is an important feature as the accuracy of AAITs normally drop when they are applied to previously unknown software projects [14, 10, 11]. This behavior is expected due to the fact that (i) there are types of alerts that were not part of the dataset used for the training of the models, and (ii) some types of alerts may be more (or less) important for specific types of software. For instance, an SQL Injection issue may not be critical for an offline application, but highly important for a cloud-based software. Hence, retraining, enables the developers to start with an original model and frequently update it in order to adapt to the specific needs of the project to which it is applied.

As can be seen in Figure 1, after the initial predictions, a user can give his/her feedback to the classifier and choose whether to retrain the model (i.e., retraining is an on-demand feature). The retraining process that is adopted by our approach is relatively simple, and it is summarized in the following steps:

1. Initially, the system collects the user feedback that has been retrieved through the several iterations of alerts inspection. This feedback consists of alerts with which the user disagreed with the criticality assigned by the model.
2. Subsequently, the system checks the user's changes, which are the user's corrections on the initial decisions of the model, in order to determine the subset of the alerts that can be used for retraining. In fact, the user feedback (i.e., user's changes) are passed through two filters to verify that (i) sufficient information has been provided by the user, and (ii) the user feedback is not contradictory. Those filters are described in detail later in this section.
3. The alerts that pass these filters are included in the training set that will be used for the retraining process, whereas those that do not pass, remain in the memory in order to be used in future retraining attempts.
4. Finally, the original dataset is updated by including the user-labeled alerts that passed the previous filtering process, and then the model is retrained.

As stated above, an important step of the retraining process is the filtering mechanism that is applied in order to choose the subset of the user-labeled alerts that could lead to the reliable retraining of the model. This process is based on two filters (i.e., criteria) that are applied to the user-labeled alerts:

– The user changes that are related to a specific alert type (i.e., rule name) have to be more than N in number.

– If there are contradictory samples (i.e., alerts), the proportion of the alerts that belong to the minority group, has to be less than $M\%$ of the overall feedback alerts of the same rule name (alert type). Two samples are contradictory if they have exactly the same values to their features, but the user assigned different criticalities to each other.

Hence, the model is retrained based on the user-labeled alerts that satisfied the above criteria. As can be seen the filters (i.e., criteria) are highly configurable. The values of N and M can be defined by the user, depending on how loose or strict they would like the retraining process to be. For instance, by assigning a large value in N and a small value in M, the retraining process is enforced to be feasible only when a large number of non-contradictory feedback is collected. In that way, the retraining process is less frequent, but the retrained model is expected to be more reliable, as it was built based on a lot of user feedback. The effectiveness of the retraining mechanism to improve the predictive performance of the original model when applied to a previously unknown software is examined in Section 5 through a case study.

4 Implementation

As a proof of concept, the proposed approach has been implemented in the form of a tool. This would enable its adoption by developers in practice, and, in turn, its further quantitative and qualitative evaluation by the community. The high-level overview of the tool is presented in Figure 2.

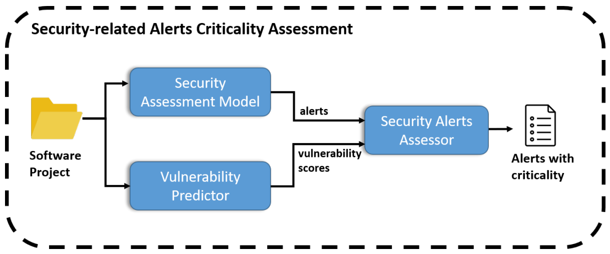


Fig. 2. High-level overview of the Security-related Alerts Criticality Assessment tool

As can be seen by Figure 2, the proposed tool, i.e., the Security Alert Assessor (SAA), depends on the outputs of two other components, namely the Security Assessment Model (SAM) and the Vulnerability Predictor (VP). The former is responsible for providing the security-related static analysis alerts that need to be assessed by the SAA, whereas the VP provides the vulnerability scores of the classes of the analyzed software, which is a feature that is considered by SAA during the assessment of the alerts' criticality.

The Security Assessment Model (SAM) [24] is a novel hierarchical model that quantifies the internal security level of software products that are written in Java based exclusively on static analysis. In brief, it employs static analysis in order to detect security-related static analysis alerts that reside in the source code and aggregates these alerts in a sophisticated way [28] in order to produce a single score (i.e., the Security Index) that reflects the internal security of the analyzed software. SAM is based on the PMD[3] static code analyzer, which is a popular open-source ASA tool. SAM is able to detect seven vulnerability categories based on their relevance, namely *Null Pointer, Logging, Exception Handling, Resource Handling, Misused Functionality, Assignment* and *Synchronization.*

The Vulnerability Predictor (VP) is based on machine learning models that are able to detect security hot spots that reside in the source code of a software product. More specifically, these models, which are based on deep neural networks, utilize text mining and software metrics in order to assess the likelihood of a software class to contain a vulnerability. The selected models are the outcome of previous research endeavors [29, 25, 26].

Hence, as can be seen by Figure 2, initially the given software product is analyzed using SAM and VP, to produce the security-related static analysis alerts that need to be assessed, and the vulnerability scores of its classes. This information is passed to the SAA, which combines them in order to assess the criticality of the reported alerts, by assigning a *criticality flag* and computing a *criticality score* for each alert (see Section 3.2). The tool also equips the users with the ability to express their disagreement with specific choices of the model, and retrain the model based on this feedback (see Section 3.4).

From a technical viewpoint, the back-end of the tool has been implemented in the form of microservices using the Docker technology. The tool is available for download on DockerHub[4]. Apart from the back-end we have also implemented an intuitive front-end (i.e., user interface), which communicates with the back-end, in order to facilitate its adoption in practice. As can be seen by Figure 3, the results of the tool are presented to the user in the form of a table, which provides information for each alert that was detected, displaying also the alerts *criticality flag* and *criticality score*. The user can also mark those alerts for which they disagree with the criticality assigned by the model through dedicated checkboxes and retrain the model based on their feedback. Useful guidelines on how to install and use the tool, can be found on the tool's wiki page[5].

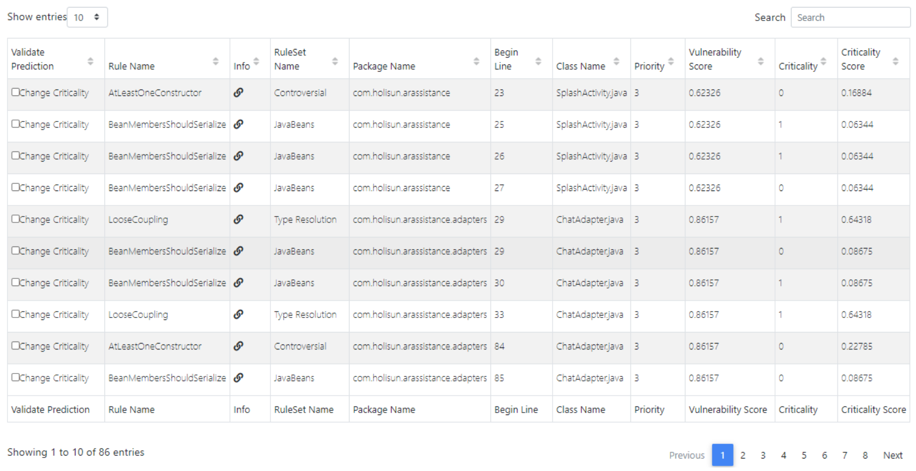
5 Case Study

In this section, a case study on a real-world commercial software product is presented in order to demonstrate the proposed approach and evaluate its correctness. For the purposes of the case study, we used a software application developed by a company that is working in the automotive industry. It is an Android application, written in Java programming language, consisting of 87

[3] https://pmd.github.io/

[4] https://hub.docker.com/repository/registry-1.docker.io/iliakalo/evit-image/

[5] https://gitlab.com/iliaskalou/aait/-/wikis/Exploitable-Vulnerability-Identification-Wiki


Show entries 10

Search

Validate Prediction	Rule Name	Info	RuleSet Name	Package Name	Begin Line	Class Name	Priority	Vulnerability Score	Criticality	Criticality Score
☐Change Criticality	AtLeastOneConstructor	🔗	Controversial	com.holisun.arassistance	23	SplashActivity.java	3	0.62326	0	0.16884
☐Change Criticality	BeanMembersShouldSerialize	🔗	JavaBeans	com.holisun.arassistance	25	SplashActivity.java	3	0.62326	1	0.06344
☐Change Criticality	BeanMembersShouldSerialize	🔗	JavaBeans	com.holisun.arassistance	26	SplashActivity.java	3	0.62326	1	0.06344
☐Change Criticality	BeanMembersShouldSerialize	🔗	JavaBeans	com.holisun.arassistance	27	SplashActivity.java	3	0.62326	0	0.06344
☐Change Criticality	LooseCoupling	🔗	Type Resolution	com.holisun.arassistance.adapters	29	ChatAdapter.java	3	0.86157	1	0.64318
☐Change Criticality	BeanMembersShouldSerialize	🔗	JavaBeans	com.holisun.arassistance.adapters	29	ChatAdapter.java	3	0.86157	0	0.08675
☐Change Criticality	BeanMembersShouldSerialize	🔗	JavaBeans	com.holisun.arassistance.adapters	30	ChatAdapter.java	3	0.86157	1	0.08675
☐Change Criticality	LooseCoupling	🔗	Type Resolution	com.holisun.arassistance.adapters	33	ChatAdapter.java	3	0.86157	1	0.64318
☐Change Criticality	AtLeastOneConstructor	🔗	Controversial	com.holisun.arassistance.adapters	84	ChatAdapter.java	3	0.86157	0	0.22785
☐Change Criticality	BeanMembersShouldSerialize	🔗	JavaBeans	com.holisun.arassistance.adapters	85	ChatAdapter.java	3	0.86157	0	0.08675
Validate Prediction	Rule Name	Info	RuleSet Name	Package Name	Begin Line	Class Name	Priority	Vulnerability Score	Criticality	Criticality Score

Showing 1 to 10 of 86 entries

Previous 1 2 3 4 5 6 7 8 Next

Fig. 3. A snapshot of the front-end data table of the Security Alerts Criticality Assessor

classes and 382 methods, comprising approximately 15,409 lines of code. It runs on Augmented Reality (AR) glasses and its main purpose is to connect a technician in the field with an engineer in a support center, in order to appropriately guide him/her to successfully complete a manual task.

Initially, the selected software application was analyzed with the Security Assessment Model (SAM) [24], in order to produce a report with the security-related static analysis alerts that it contains. Then its source code was parsed by our Vulnerability Prediction Models [29, 25, 26], in order to compute the vulnerability scores of the 87 classes of the software, which is a required input for the proposed approach for assessing the criticality of the observed alerts. Then the Security Alerts Assessor (SAA) analyzed the produced alerts and assigned a *criticality flag* and a *criticality score* to each one of them (see Figure 2).

The above mentioned process resulted in a list of 133 security alerts. These alerts were presented to developers with security expertise, who manually inspected them and expressed their agreement or disagreement with the criticalities assigned by the model. The criticalities assigned by the model were compared to the criticalities assigned by the manual inspection in order to evaluate the accuracy of the model when applied to a new (i.e., previously unseen) software. The results of the evaluation are presented in Table 3. As can be seen by Table 3, the overall accuracy of the model was approximately 82%, which is 4% lower than the model's descriptive accuracy, and the recall was found to be approximately 73%, which is 10% lower than the descriptive recall (see Table 1).

From this experiment, it is obvious that there is a drop in the predictive performance of the model. This drop was expected since it is applied to a previously unknown software product, which may exhibit alerts of specific types that were not present during the training of the model, whereas some of the already trained alert types may be more (or less) critical for the specific type

Table 3. Evaluation metrics showcasing the ability of the model to adapt to user feedback

Dataset/Model	Accuracy (%)	Precision (%)	Recall (%)
Initial Model	81.95	89	73.13
Retrained Model ($N = 2$,$M = 20$)	86.32	89.70	87.14
Retrained Model ($N = 0$,$M = 50$)	88.02	90.13	89.67
Retrained Model ($N = 5$,$M = 20$)	86.21	84.92	96.34

of software. However, at least for the given example, this drop is not observed to be significant. This provides confidence that sufficient prediction performance can be achieved when the model is applied to a project that was not used during its training, indicating that some general trends in the criticality of alerts are horizontal across different project types.

Subsequently, we retrained the model based on the feedback that was provided by the developers through the aforementioned manual inspection, and we evaluated its predictive performance. In fact, part of the user feedback was selected randomly and excluded from the model retraining to be used for the model evaluation. We considered the case of $N = 2$ and $M = 20$, as it was the most representative for the given dataset. The performance metrics of the retrained model are also presented in Table 3. As can be seen by Table 3, there is a significant increase in all the performance metrics, all of them being higher than 85%. This suggests that the retraining process leads to a significant increase in the predictive performance of the model, and, thus, that it is able to capture the user feedback.

For reasons of completeness, in Table 3, the evaluation results of the retrained model for two additional cases of N and M values are provided. In particular, we consider a more loose model ($N = 0$,$M = 50$), i.e., a model that accepts the user feedback without much filtering, and a more conservative model ($N = 5$,$M = 20$), i.e., a model that requires more data to be collected by the user in order to consider their input reliable to be used for retraining. As can be seen by Table 3, the predictive performance of the retrained models are also high and comparable to the model of the use case. This observation (which was expected) suggests that the model is able to learn the new data provided by the user and adapt well to the user feedback, regardless of the values of N and M. As mentioned in Section 3.4, the purpose of N and M are to allow the developer define how loose or strict the retraining process should be. In fact, the more conservative the model, the more non-contradictory data should be collected by the user, making the retraining process less frequent, but the produced model more reliable.

To sum up, the above analysis, although preliminary, led to some interesting observations. The results of the analysis highlight that when applying the original model to previously unknown alerts, the accuracy does not drop massively. This indicates that our approach can provide a relatively accurate assessment even on a dataset with alerts of different defined criticality. We can also notice that the model retraining process based on user feedback, leads to a substantial increase in the accuracy of the model. This fact suggests that the model, retrained based on feedback provided by the user, is capable of adapting to dynamically changing behaviors and it can actually improve its accuracy. Hence, frequent application of retraining would allow the model to adapt to the characteristics of the specific project to which it is applied.

At this point, it should be noted that a comparison with similar approaches and models was not feasible. Although a large number of AAITs have been proposed in the literature, very few contributions have been made with respect to security. In addition to this, the existing contributions (e.g., [4]), are not operationalized and have become obsolete, whereas no sufficient information and data are available that would enable their replication.

A note with respect to the validity threats of the present work is considered necessary. First of all, the model was based on information retrieved exclusively from Java projects, potentially affecting its generalizability to other programming languages. However, the adopted techniques are language agnostic enabling the developers to apply them in other programming languages. In addition, the selection of the model hyperparameters could be biased to the specific dataset that has been used for its training. However, techniques for avoiding over-fitting were employed (see Section 3.3), whereas the applied retraining process is expected to adjust the parameters to the project to which it is applied, neutralizing the potential bias of the parameters of the initial model.

6 Conclusions and Future Work

The purpose of the present paper was to develop a mechanism for assessing the criticality of security-related static analysis alerts. To this end, we developed a technique for prioritizing and classifying security-related static analysis alerts based on their criticality, by taking into account information retrieved from the alerts themselves, vulnerability prediction models, and user feedback.

To achieve this, a manually curated dataset of security-related static analysis alerts was constructed, by statically analyzing two real-world Java software products that were developed by CERTH as part of past and ongoing EU Projects. Based on this dataset, several machine learning models were built, for predicting alerts' criticality. Among the studied models, the Multi-layer Perceptron (MLP) demonstrated the best results, and thus it was chosen as the basis of our approach. The proposed approach was evaluated through a simple case study on a real-world commercial Java application provided by the automotive industry. The results of the case study revealed that the proposed model can be used as a good basis for assessing the criticality of the alerts of a new project, and that, through regular retraining, it can easily adapt to the characteristics of the model to which it is applied, providing more accurate assessments with time. The proposed technique has been operationalized in the form of web-services, providing also a web interface, to facilitate its adoption in practice. To the best of our knowledge, this is the first technique that focuses exclusively on security-related static analysis alerts, and adopts the concept of retraining. It is also the first approach that combines information retrieved from: (i) the alerts themselves, (ii) vulnerability prediction models, and (iii) user feedback.

Future work includes the investigation of the generalizability of the produced results by replicating our method using software products that are written in other programming languages (e.g., C/C++, Python, JavaScript, etc.), as well as by using different VPMs and static code analyzers. We are also planning to investigate how legacy systems could benefit from the proposed approach, such as those implemented in outdated languages like COBOL [30, 31].

Acknowledgements This work is partially funded by the European Union's Horizon 2020 Research and Innovation Programme through IoTAC project under Grant Agreement No. 952684.

References

1. Luszcz, J.: Apache struts 2: how technical and development gaps caused the equifax breach. Network Security **2018**(1) (January 2018) 5–8
2. Siavvas, M., Gelenbe, E., Kehagias, D.: Static analysis-based approaches for secure software development. Security in Computer and Information Sciences (2018) 142
3. Mohammed, N.M., Niazi, M., Alshayeb, M., Mahmood, S.: Exploring Software Security Approaches in Software Development Lifecycle: A Systematic Mapping Study. Comp. Stand. & Interf. (2016)
4. Baca, D.: Identifying security relevant warnings from static code analysis tools through code tainting. In: 2010 International Conference on Availability, Reliability and Security, IEEE (2010) 386–390
5. Yang, J., Ryu, D., Baik, J.: Improving vulnerability prediction accuracy with secure coding standard violation measures. In: 2016 International Conference on Big Data and Smart Computing (BigComp), IEEE (2016) 115–122
6. McGraw, G.: Software security. Datenschutz und Datensicherheit - DuD (2012)
7. Howard, M., Lipner, S.: The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software. Microsoft Press (2006)
8. Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why don't software developers use static analysis tools to find bugs? In: 2013 35th International Conference on Software Engineering (ICSE), IEEE (2013) 672–681
9. Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Gall, H.C., Zaidman, A.: How developers engage with static analysis tools in different contexts. Empirical Software Engineering **25**(2) (2020) 1419–1457
10. Muske, T., Serebrenik, A.: Survey of approaches for handling static analysis alarms. In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE (2016) 157–166
11. Heckman, S., Williams, L.: A systematic literature review of actionable alert identification techniques for automated static code analysis. Inf. and Soft. Tech. (2011)
12. Yang, X., Chen, J., Yedida, R., Yu, Z., Menzies, T.: Learning to recognize actionable static code warnings. Empirical Software Engineering (2021)
13. Munaiah, N., Camilo, F., Wigham, W., Meneely, A., Nagappan, M.: Do bugs foreshadow vulnerabilities? an in-depth study of the chromium project. Empirical Software Engineering **22**(3) (2017) 1305–1347
14. Heckman, S., Williams, L.: A comparative evaluation of static analysis actionable alert identification techniques. In: Proceedings of the 9th International Conference on Predictive Models in Software Engineering. (2013) 1–10

15. Misra, S.: A step by step guide for choosing project topics and writing research papers in ict related disciplines. In: Information and Communication Technology and Applications: Third International Conference. (2021)
16. Heckman, S., Williams, L.: A model building process for identifying actionable static analysis alerts. In: 2009 International Conference on Software Testing Verification and Validation. (2009) 161–170
17. Heckman, S.S.: Adaptively ranking alerts generated from automated static analysis. XRDS: Crossroads, The ACM Magazine for Students **14**(1) (2007) 1–11
18. Ruthruff, J.R., Penix, J., Morgenthaler, J.D., Elbaum, S., Rothermel, G.: Predicting accurate and actionable static analysis warnings: An experimental approach. In: Proceedings of the 30th International Conference on Software Engineering. ICSE '08, New York, NY, USA, Association for Computing Machinery (2008) 341–350
19. Kremenek, T., Ashcraft, K., Yang, J., Engler, D.: Correlation exploitation in error ranking. In: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering. SIGSOFT '04/FSE-12, New York, NY, USA, Association for Computing Machinery (2004) 83–93
20. Tripp, O., Guarnieri, S., Pistoia, M., Aravkin, A.: Aletheia: Improving the usability of static security analysis. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. (2014)
21. Heckman, S., Williams, L.: On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In: 2nd International Symposium on Empirical Software Engineering and Measurement. (2008)
22. Younis, A.A., Malaiya, Y.K., Ray, I.: Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In: 15th International Symposium on High-Assurance Systems Engineering. (2014)
23. Younis, A.A., Malaiya, Y.K.: Using software structure to predict vulnerability exploitation potential. In: 8th International Conference on Software Security and Reliability-Companion. (2014) 13–18
24. Siavvas, M., Kehagias, D., Tzovaras, D., Gelenbe, E.: A hierarchical model for quantifying software security based on static analysis alerts and software metrics. Software Quality Journal (2021)
25. Kalouptsoglou, I., Siavvas, M., Tsoukalas, D., Kehagias, D.: Cross-project vulnerability prediction based on software metrics and deep learning. In: Computational Science and Its Applications – ICCSA 2020, Cham (2020)
26. Filus, K., Siavvas, M., Domańska, J., Gelenbe, E.: The random neural network as a bonding model for software vulnerability prediction. In: Modelling, Analysis, and Simulation of Computer and Telecommunication Systems. (2021)
27. Filus, K., Boryszko, P., Domańska, J., Siavvas, M., Gelenbe, E.: Efficient feature selection for static analysis vulnerability prediction. Sensors **21**(4) (2021) 1133
28. Siavvas, M.G., Chatzidimitriou, K.C., Symeonidis, A.L.: Qatch-an adaptive framework for software product quality assessment. Expert Sys. with Applications (2017)
29. Siavvas, M., Kehagias, D., Tzovaras, D.: A preliminary study on the relationship among software metrics and specific vulnerability types. In: 2017 International Conference on Computational Science and Computational Intelligence. (2017)
30. Mateos, C., Zunino, A., Misra, S., Anabalon, D., Flores, A.: Migration from cobol to soa: Measuring the impact on web services interfaces complexity. In: International Conference on Information and Software Technologies, Springer (2017) 266–279
31. Mateos, C., Zunino, A., Flores, A., Misra, S.: Cobol systems migration to soa: assessing antipatterns and complexity. Information Technology and Control (2019)