Prioritizing Alerts from Multiple Static Analysis Tools, using Classification Models

Lori Flynn, William Snavely, David Svoboda, Nathan VanHoudnos, Richard Qin
Jennifer Burns, David Zubrow, Robert Stoddard, Guillermo Marce-Santurio
Software Engineering Institute, Carnegie Mellon University
Pittsburgh, Pennsylvania

ABSTRACT

Static analysis (SA) tools examine code for flaws without executing the code, and produce warnings ("alerts") about possible flaws. A human auditor then evaluates the validity of the purported code flaws. The effort required to manually audit all alerts and repair all confirmed code flaws is often too much for a project's budget and schedule. An alert triaging tool enables strategically prioritizing alerts for examination, and could use classifier confidence. We developed and tested classification models that predict if static analysis alerts are true or false positives, using a novel combination of multiple static analysis tools, features from the alerts, alert fusion, code base metrics, and archived audit determinations. We developed classifiers using a partition of the data, then evaluated the performance of the classifier using standard measurements, including specificity, sensitivity, and accuracy. Test results and overall data analysis show accurate classifiers were developed, and specifically using multiple SA tools increased classifier accuracy, but labeled data for many types of flaws were inadequately represented (if at all) in the archive data, resulting in poor predictive accuracy for many of those flaws.

CCS CONCEPTS

• **Software and its engineering → Software verification**; **Automated static analysis**; **Software libraries and repositories**; **Formal software verification**; *Software notations and tools*; Software testing and debugging;

KEYWORDS

static analysis, alert, classification, rapid, accurate

ACM Reference Format:
Lori Flynn, William Snavely, David Svoboda, Nathan VanHoudnos, Richard Qin and Jennifer Burns, David Zubrow, Robert Stoddard, Guillermo Marce-Santurio. 2018. Prioritizing Alerts from Multiple Static Analysis Tools, using Classification Models. In *SQUADE'18:: SQUADE'18:IEEE/ACM 1st International Workshop on Software Qualities and their Dependencies, May 28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3194095.3194100

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SQUADE'18:, May 28, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5737-1/18/05...$15.00
https://doi.org/10.1145/3194095.3194100

1 INTRODUCTION

Static analysis tools examine code for flaws, including those that could lead to software security vulnerabilities, and produce diagnostic messages ("alerts") indicating the location of the purported flaw in the source code, the nature of the flaw, and often additional contextual information. A human auditor then evaluates the validity of the purported code flaws. The effort required to manually audit all alerts and repair all confirmed code flaws often exceeds the project's budget and schedule. Auditors need tools that allow them to triage alerts, strategically prioritizing alerts for examination. This paper describes research we conducted that developed classification models to predict if static analysis alerts are true or false positives. We created alert classification models using features derived from multiple static analysis tools, code base metrics, and archived audit determinations, and tested those models. The long-term goal of this work is to develop an automated and accurate statistical classifier, intended to efficiently use analyst effort and to remove code flaws.

"Alert fusion" refers to the practise of unifying alert information from different tools which map to the same condition (e.g., SEI CERT Coding Rule [6] (a.k.a. CERT rule) or Common Weakness Enumeration (CWE [14])) in the same part of the code (e.g., same line of same file). Fusion may be more or less precise, e.g., a particular CWE may occur in two different parts of the same line of code. By fusing alerts from different tools and then creating classifiers, this work attempts to gain an understanding of correlations between static analysis tools and alert accuracy.

The four types of classification techniques we compare are Lasso Logistic Regression, Classification and Regression Trees (CART), Random Forest (RF), and Extreme Gradient Boosting (XGBoost). These techniques assign membership to classes based on probabilities, but they differ in implementation ease and their tendency to over-fit to the training data set.

Our intended use of these classification techniques is to create models to automatically classify alerts as expected-true-positive (e-TP), expected-false-positive (e-FP), or indeterminate (I) based on user-specified confidence levels. Using the results of a binary classification model to classify alerts into three states based on user specified criteria is novel.

The data used in this work consists of archives for 19 CERT-audited codebases, plus data from 3 collaborating organizations (anonymous by request) that audited their own codebases.

1.1 Related Work

Bessey et al. [2] found many issues with using static analysis in practice, including tools ignoring constructs and thus producing many false positive alerts, users wrongly labeling alerts they find confusing as false, and user difficulty dealing with many alerts

resulting in tools producing less (possibly-true) alerts about possible defects than they could. Our work attempts to (eventually) help users handle many alerts effectively.

Delaitre et al. found that static analysis tools on average find about 20% of weaknesses in basic test cases without complexity. They found that complex control flow or data flow constructs significantly reduced the tools' success rates, and identify flaws with highest and fewest findings across a set of anonymized static analysis tools [3].

Since single static analysis tools have different coverage of code flaws (warning about some but not others) [3] [2], multiple static analysis tools may be used to find more code flaws [16]. This approach compounds the problem of generating too many alerts to deal with, including too many false positives. Our work uses multiple static analysis tools and addresses handling the many alerts.

Beller et al. found that few open-source projects have static analysis tools integrated closely with their workflows, and most of those projects don't mandate that a codebase should be alert-free [1].

There are few previously-published peer-reviewed papers that classify or prioritize alerts from multiple static analysis tools together. Kong et al. [9] prioritize alerts using alert risk rating and confidence (if an alert is reported by multiple tools the score is raised), and provide experimental results from analyzing 3 codebases. They do not use features of the codebase to classify alerts, and their method cannot analyze confidence factors that are slightly complex, such as if a high likelihood of a true positive exists when only Tool A and Tool C produce an alert but Tool B does not produce an alert. Meng et al. [13] implement an approach to merge results from multiple tools, and propose two policies to prioritize results. Their prioritization simply orders alerts according to alert severity and then a count of tools that produced alerts of that type for that place in the code, and no analysis was done of correctness of the alerts. Kremenek ranks alerts by correlating data per static analysis tool, using features from the tools and from the codebases [11]. However, that method orders sets of alerts from different tools relatively (all alerts from one tool are prioritized below all alerts from another tool, not interspersed), and does not do alert fusion.

Our research builds on a technical report by some of our authors [16], which collected comparative statistics about the accuracy of static analysis tools and the violations they flag, and developed statistically significant binary logistic regression models to assess if an alert is a true or false positive for alerts that mapped to three CERT rules [18] [6] [12]. CERT has developed a tool called SCALe [19], which maps alerts to coding rules, and can order alerts according to a per-rule metric. That metric is based on three expert-determined values (static values for each coding rule), multiplied: $Severity * Likelihood * RemediationCost$. No measure of the probability of the alert being correct (a true positive) is currently used in that prioritization.

Heckman and Williams [7] did an extensive survey of methods that classify and prioritize actionable alerts, detailing 21 peer-reviewed studies. Our project uses 5 of the approaches (alert type selection, contextual information, data fusion, machine learning, and mathematical and statistical models) discussed in the paper, and doesn't use the other 3 (dynamic detection, graph theory, and model checking). Our project uses 2 of the artifact characteristics categories (alert characteristics, code characteristics), and doesn't use the other 3 (source code repository metrics, bug database metrics, and dynamic analysis metrics). See Section 1.2 for explanation of those choices. No previous work in their survey involves the combination of multiple static analysis tools, the set of features used, and competing classifiers as the work in our paper.

Heckman [8] adaptively ranked alerts using developer feedback supressing false positives and fixing true positives, along with application-specific data about the alert ranking factors (alert type accuracy and code locality). Her model found 81% of true positive alerts after investigating only 20% of the alerts.

Kremenek et al. [10] use a formula that adapts as human analyzers inspect static analyzers' outputs. This information could be dynamically used to re-order alerts, as the human analyst completes checking each alert from the (current) top of the ordered set of alerts. They observed a factor of 2-8 improvement over randomized ranking. Our work doesn't use dynamic auditor feedback but could incorporate that in future work.

Ruthruff et al. [17] built models to predict whether FindBugs alerts are false positives, and if true, whether the defects would be acted on by developers. Prediction was done using logistic regression analysis including metrics such as program size, file recent change history, file age since release, and recent fault history. This generated models more than 85% accurate at predicting false positives, and more than 70% accurate in identifying actionable alerts, in a case study performed at Google. Our work uses some features they did not, including use of multiple static analysis tools.

As part of this project, we developed and partially tested a common auditing lexicon and set of auditing rules [20]. A common lexicon and auditing rules are necessary to ensure that auditing determinations are made consistently, and also to ensure that audit archives contain enough precision for common development/system changes to efficiently and correctly use previous audit determinations. Although the auditing tool that we shared with our collaborators did not have the full audit determination lexicon described in the paper, our collaborators used (and helped us develop) the determination lexicon and full set of auditing rules.

1.2 Approach

We use classification features found helpful in related research, but with multiple static analysis tools. Our approach was further shaped by our existing trove of audit archive data, available tools, and wanting to minimize work required to integrate any more 1) static analysis and 2) code metrics tools into SCALe (1-2 weeks 1-engineer effort per tool, mostly to map tool alert types to CERT rules and to parse output). Also, we did not have access to code repository data or bug databases for our audit archives. Our alert archives only contained auditor determinations for CERT coding rules, so in this work we only developed CERT rule classifiers. (Conversely, with audit archives for CWEs, we could have developed CWE classifiers.) Budget constraints limited the set of SCALe-integrated commercial tools collaborators could use.

2 SYSTEM ARCHITECTURE

We developed static analysis alert prediction models using data collected from manual audits of software projects. The high level

workflow by which this data was collected and modelled for each codebase, involves code analysis with tools, then data ingestion and correlation in a database, followed by classifier development and testing. First, the source code was inspected by a suite of static analysis tools. Second, tool outputs (along with the original source) was passed to an enhanced version of the SCALe tool. Enhanced SCALe produces two artifacts: (1) static analysis alerts mapped to SEI CERT Coding Rule violations, and (2) various source code metrics. Next, mapped alerts were inspected by human auditors to determine whether they were true or false positives. Audit determinations, along with source code metrics, were processed into a training dataset. Training data from all software projects was combined together and used to construct prediction models.

2.1 Codebase Selection

The codebases used by this project come from a variety of sources. Some codebases are proprietary and belong to "auditor-collaborators" (henceforth called "collaborators") who audited their own codebases as part of their participation in this research. We asked collaborators to select C-language codebases they currently use that they wanted static analysis audits for, and that they could run at least two SCALe-integrated static analysis tools on. The three collaborators audited a total of 11 codebases.

We also used 19 CERT-audited codebases in C, C++, Java, and Perl, leveraging existing audit data archives created over a course of 8 years. Several open-source codebases are included, although the majority of these codebases are proprietary and belong to "previous non-auditor participating organizations" (distinct from the "collaborators" referenced above).

2.2 Static Analysis

Each codebase was inspected by a collection of static analysis tools, including commercial and open-source tools. Each static analysis tool produces a file containing a list of alerts for the codebase, typically in a structured format (e.g., `XML` or `JSON`).

The set of tools applied is not consistent across codebases. Fundamentally, different programming languages have different applicable static analysis tools. Additionally, individual codebases may have characteristics that exclude the use of certain tools. For example, a codebase may use a compiler that is incompatible with a particular static analysis tool. Next, tool availability varied between audits. This is in part due to the length of time over which the audits were conducted. CERT acquired access to and expertise with a growing set of tools over time. Moreover, our collaborators had access to a limited set of tools.

2.2.1 Legal Issues and Per-Tool Data. Commercial static analysis tools used in the project have terms of use that disallow publication of tool performance and tool output. Therefore, we anonymize tool names in this paper, as do other papers [3]. Similarly, we cannot publish the dataset our classifiers were developed from and tested on, because much of the data came from commercial tools and some of the features used are unique to one of the commercial tools.

2.3 Enhanced SCALe

The Source Code Analysis Laboratory (SCALe) is a tool developed by CERT for aggregating and evaluating static analysis alerts from multiple tools. SCALe accepts a collection of structured alert files from various static analysis tools, for a single codebase. The source code files are also uploaded to SCALe. The alerts are converted into a common data format. Moreover, each alert is mapped to a CERT Secure Coding Rule. The goal of this processing is to facilitate auditing of the alerts. An auditor does not have to inspect alerts from multiple files in different formats. They can instead inspect an aggregated list of alerts. The rule mapping serves to clarify the nature of the potential flaw detected by the alert, and additionally associates a rule-derived severity and priority with the flaw.

For this project, we made a number of enhancements to SCALe. We incorporated a source code metrics module based on the open-source Lizard tool [22]. This module measures a few common code metrics for projects processed by SCALe, including cyclomatic complexity and significant-lines-of-code (SLoC). This module currently works with code in C, C++, and Java, but not Perl. We also added a data sanitization module which is discussed in more detail below.

2.4 Auditing

The consolidated alerts were inspected by human auditors using the SCALe application. This application displays the alerts in a filterable, searchable table, and moreover allows an auditor to easily view the source code associated with a given alert. Other alert properties, such as the corresponding CERT rule, the severity, and the priority, are also displayed, and can be used as filter parameters.

For a given alert, the auditor determined whether the diagnosed code violated the CERT rule associated with the alert. The auditor chose a determination of `True`, `False`, `Suspicious`, `Ignore`, `Dead Non-Library`, or `Unknown` for the alert, using the SCALe application. (Although enhanced-SCALe didn't originally include a marking for `Dead Non-Library` unreachable code, in response to a collaborator's question we kludged a combination of a flag and the `Ignore` verdict to mean that.) For the purposes of this study, verdicts of `Suspicious`, `Ignore`, `Dead Non-Library`, and `Unknown` were interpreted as `Indeterminate`.

The auditors were drawn from experienced CERT personnel and from collaborators. Collaborators underwent specialized training on both the CERT rules and the auditing process. See Section 3 for more information about this training process.

2.5 Data Preparation & Classifier Development

The audited alerts required some additional processing before being used for classifier development.

Before receiving data from our collaborators, we were required to provide a mechanism for removing sensitive infomation from the audited alerts. For example, path and function names were considered to be sensitive by our collaborators. Therefore, we enhanced SCALe to obfuscate sensitive fields using a salted SHA-256 hash. We used hashing instead of outright omitting sensitive data, because we needed the ability to group alerts based on their location, for classifier development. Only collaborator data was hashed; data collected by CERT auditors was not hashed.

The alert data underwent some additional processing before being consumed by our statistical software. Alerts that occured on the same line of the same file, and that mapped to the same CERT Secure Coding Rule, were merged together. Suppose that

two different tools detected the same problem with a line of source code. In this case, the two alerts would be combined into a single observation. The merged alert has a ternary feature for each analysis tool, with a value of 1 if the tool detected the issue, 0 if the tool did not, and 2 if the tool was not run on the associated codebase.

A few other additional features were computed during this processing phase, for example, the number of alerts in each file, the number of alerts in each function, etc. These aggregated features were added to each alert, when appropriate (e.g. if alert A occured in a file F and function X, the alert counts for F and X would be added as features to A).

See Table 1 for a complete description of the features associated with each alert.

The prepared audit data was used to develop machine learning classifiers. We used the R environment to program and test classifiers.

2.6 Envisioned Use

The envisioned use of the classifiers is in a software development and/or software assurance system, automating use of the classifiers to classify alerts as e-TP, I, and e-FP. The e-TP alerts would be separately prioritized for code repair. The I alerts would be automatically prioritized for manual auditing using a heuristic taking into account classifier confidence, risk metric if the code flaw is true, and a cost metric to fix the code flaw. The e-FP alerts' data would be used for further classifier development, and in the unlikely case that all the I alerts get audit determinations then the e-FP alerts could also be audited. Ideally, as described in related research, new alert audit determinations would dynamically feed back into a classifier prediction system to re-prioritize remaining alerts.

3 DATA QUALITY: COLLABORATOR TRAINING

Auditing static analysis alerts is a highly technical task, requiring skilled appraisers. For a given alert, an auditor must first understand what behavior the alert detects. They must then be able to read and understand the diagnosed source code, and determine if the associated CERT Rule has been violated. This may require nontrivial intraprocedural and interprocedural analysis.

In response to the complexity of this task, we developed a variety of training materials, as well as auditor evaluation tools. In order to obtain high quality data, we needed to ensure that our auditor subjects made accurate, consistent determinations [20]. Our training materials covered several subjects: the CERT rules, rules for auditing, and usage of the SCALe application. Our principle evaluation tool was an auditing exam. Although the exact material developed for project collaborators is not publicly available, much of that content evolved into an auditing rules paper [20], plus slides [5] and a virtual machine (including test programs and static analysis output) distributed in a hands-on auditing tutorial [4].

3.1 Coding Rules

In this study, auditors evaluated an alert based on the SEI CERT Coding Rule associated with the alert. It was therefore critical that auditors had a clear understanding of these rules before evaluating an alert. We instructed our collaborators to focus on 8 SEI CERT C Coding Rules (summarized in Table 2), and provided focused training sessions on those 8 rules for two of our collaborators. Our third collaborator had enough experience to forgo training.

3.2 Auditing Rules

Early on in the project, we realized that we did not have a sufficiently clear definition of the auditing process. Therefore, it was difficult to give our collaborators strong guidance on how to audit alerts. To mitigate this, we developed a set of twelve auditing rules. These rules aimed to help auditors make consistent determinations, especially in complex or ambiguous auditing scenarios. Additionally, the rules establish some basic assumptions that auditors should make. For example, an auditor should assume, without strong evidence to the contrary, that program inputs may be malicious.

Two collaborators were given on-site training covering the auditing rules, and the third collaborator was provided with materials for self-training. See Table 3 for summary of the auditing rules we provided. (We have developed a later version of these rules in [20]. The later version expanded the auditing rules' scope to apply beyond CERT rules (e.g., applicable to CWEs) and to use more-precise determination labels, e.g., Complex and Dependent.)

3.3 Using Tools: Enhanced-SCALe

All of our collaborators used an enhanced version of the SCALe auditing tool for evaluating static analysis alerts. We provided various installation media, including source code distributions and virtual machines, and provided documentation and support for the setup and use of the application. Our on-site training sessions included some instruction on use of the enhanced SCALe application.

4 RESULTS

4.1 Audited Data Characterization

Fig. 1 shows that CERT-audited alerts that map to particular rules tend to be determined in only one way (always True or always False). The chart shows there are 58 SEI CERT Coding Rules with 20 or more audited alerts. The other 324 CERT Rules have little or no labeled data in CERT's audit archives. The chart shows the True:False spread, where the left side of the chart is for a close-to-even spread, and the right side is for nearly or exactly 100% one-way audits. 25 rules are in the rightmost division, nearly or completely determined one-way. $2,487$ of the CERT-audited alerts had True determinations and $4,980$ had False determinations. CERT-audited data contains audited alerts that map to 158 of the 382 CERT rules.

Approximately 30% of the CERT-audited archive data consists of alerts mapped to rule INT31-C.

Our collaborators audited 354 total alerts, mapped to the 8 targeted rules and more. They labeled 144 alerts False and 210 alerts True.

Table 4: Collaborator-audited data

Collaborator	Alerts Audited	CERT Rules Mapped To
Collaborator 1	93	4
Collaborator 2	195	15
Collaborator 3	66	3

Table 1: Set of alert features, with descriptions.

Feature	Description
Codebase	The name of the codebase where the alert was detected.
Determination	The audit determination (`TRUE`, `FALSE`, `SUSPICIOUS`, `IGNORED`, `DEAD NON-LIBRARY`, `UNKNOWN`).
Path	The full path to the file where the alert occurs.
Line	The line number in the file where the alert occurs.
Rule	The name of the CERT rule associated with the alert.
Title	The title of the CERT rule associated with the alert.
Severity	The severity field of the CERT rule.[a]
Likelihood	The likelihood field of the CERT rule.[a]
Remediation	The remediation field of the CERT rule.[a]
Priority	The priority field of the CERT rule.[a]
Level	The level field of the CERT rule.[a]
Tool1 ... ToolN (N Features)	For each static analysis tool, there is a feature indicating whether the static analysis tool detected this alert. This is a ternary value: 0 indicates the tool did not detect the alert; 1 indicates the tool did detect the alert; 2 indicates that the tool was not run for this codebase.
Function	The name of the function where the alert occurs.[b]
Function Length	The number of lines of code in the function.[b]
Function SLoC	The number of significant lines of code in the function.[b]
Function Cyclomatic Complexity	The cyclomatic complexity of the function.[b]
Function Parameter Count	The number of parameters to the function.[b]
Function Token Count	The number of lexical tokens in the function.[b]
Function Start Line	The line number where the function definition starts.[b]
Function End Line	The line number where the function definition ends.[b]
Function Alert Count	The number of alert that occur in this function.[b]
Filename	The base name of the file where the alert occurs, without parent directories.
File SLoC	The number of significant lines of code in the file.
File Function Count	The number of functions/methods in the file.
Average Function SLoC	The average significant lines of code in functions in the file.
Average Function Token Count	The average number of tokens in functions in the file.
File Alert Count	The number of alerts that occur in the file
Depth	The depth of the file where the alert occurs, in the directory structure

[a] See [6] for more information.
[b] These features are not applicable for alerts that occur outside of a function.

Table 2: CERT rules audited by our collaborators.

Rule	Title
ARR36-C	Do not subtract or compare two pointers that do not refer to the same array
DCL31-C	Declare identifiers before using them
EXP33-C	Do not read uninitialized memory
EXP34-C	Do not dereference null pointers
EXP36-C	Do not cast pointers into more strictly aligned pointer types
INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data
INT33-C	Ensure that division and remainder operations do not result in divide-by-zero errors
STR31-C	Guarantee that storage for strings has sufficient space for character data and the null terminator

4.2 Types of Classifiers

We developed 15 “featureless classifiers”, or classifiers for SEI CERT Coding Rules with 20 or more audited alerts, with 100% of those determinations being one-way (always true or always false). Featureless classifiers are not a standard classifier type.

We used two dataset types for classifier development and testing, as shown in Fig. 2: 1) all alerts with a True or False determination; and 2) a subset of audited alerts that map to a particular CERT rule.

We used two methods to develop and test classifiers: 1) We developed classifiers with 70% of the data, then tested on the remaining

Table 3: Auditing rules for collaborators.

Rule 1	Understand the language and the secure coding rule in question.
Rule 2	Some alerts are too complex to judge; they should be marked Suspicious
Rule 3	It is OK to mark an alert true even if you think the code maintainers will protest.
Rule 4	Assume that external inputs to the program are malicious.
Rule 5	Unless instructed otherwise, assume that code must be portable.
Rule 6	When auditing an alert, if you discover a second true violation, mark its alert as true.
Rule 7	Do not arbitrarily extend the scope of a CERT rule.
Rule 8	Code that behaves as expected might still violate a CERT rule.
Rule 9	An alert might indicate a true violation of the CERT rule, even if its message text is useless or incorrect.
Rule 10	Multiple messages help in understanding an alert.
Rule 11	Assume no violations occur before the line in question.
Rule 12	Handle an alert in unreachable code depending on if it is in library or non-library code.

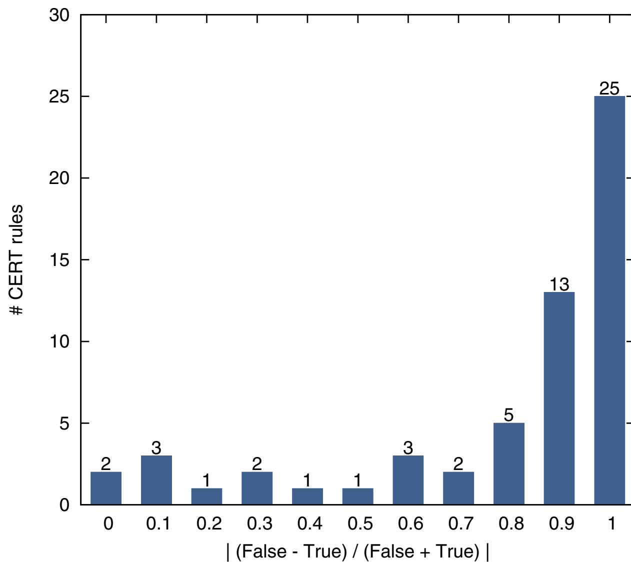

Figure 1: Frequency of True:False ratio, for CERT rules with a minimum of 20 audits

30% of the data.; and 2) We developed classifiers using all of the CERT-audited data, then tested on pooled collaborator data.

Classifier accuracy test results, classifier from pooled all-alerts data (CERT plus collaborators):

- Lasso Logistic Regression: 88%
- Random Forest: 91%
- CART: 89%
- XGBoost: 91%

Table 6 shows test results for classifiers developed on single rule-mapped alerts. Most of the rule names are followed by an asterisk ("*") to indicate there is very little labeled data for this rule, so results are suspect. However, three rules had large quantities of labeled data in the audit archives, and one of them (INT31-C) produced a very high-accuracy classifier (98% accurate).

In addition to using the four types of classifier (RF, CART, LLR, and XGBoost) named in the Introduction, we developed variants that used only a subset of the data to develop classifiers. Variants include per-rule, per-coding-language, features removed that didn't have a value for one or more alerts, alerts removed if they were missing a value for any feature, toolnames removed, and features removed for Lizard code metrics outside of functions.

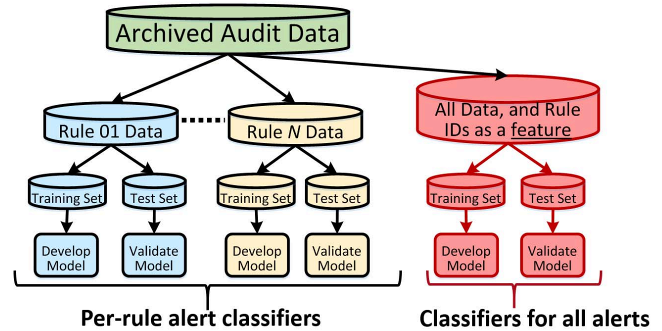

Figure 2: Classifiers developed from 2 dataset types: all-alerts and per-rule alerts

Rule ID	Lasso LR	Random Forest	CART	XGBoost
INT31-C	95%	96%	96%	95%
EXP01-J	68%	83%	89%	87%
OBJ03-J	73%	86%	86%	83%
FIO04-J*	75%	75%	83%	71%
EXP33-C*	90%	100%	90%	90%
EXP34-C*	74%	84%	87%	81%
DCL36-C*	100%	100%	100%	100%
ERR08-J*	99%	99%	97%	97%
IDS00-J*	97%	94%	94%	88%
ERR01-J*	100%	100%	100%	100%

Table 5: Accuracy: per-rule classifiers, CERT-audited data only

We developed 201 classifiers for 11 CERT rules that have a mix of True and False determinations. We also developed 72 all-rules classifiers where the rule name was used as a feature), including 44 per-language classifiers.

Results for CERT-audited data alone are very similar to the previous all-data results, because most of the data is CERT-audited:

- Lasso Logistic Regression: 87%
- Random Forest: 90%
- CART: 89%
- XGBoost: 92%

Rule ID	Lasso LR	Random Forest	CART	XGBoost
INT31-C	98%	97%	98%	97%
EXP01-J	74%	74%	81%	74%
OBJ03-J	73%	86%	86%	83%
FIO04-J*	80%	80%	90%	80%
EXP33-C*	83%	87%	83%	83%
EXP34-C*	67%	72%	79%	72%
DCL36-C*	100%	100%	100%	100%
ERR08-J*	99%	100%	100%	100%
IDS00-J*	96%	96%	96%	96%
ERR01-J*	100%	100%	100%	100%
ERR09-J*	100%	88%	88%	88%

Table 6: Accuracy: per-rule classifiers, data from all auditors

Fig. 5 shows accuracy for classifiers made from per-rule data, using only CERT-audited data. Due to less data, notice that rule ERR09-J no longer has a classifier.

Using toolname as a feature for developing classifier generally increased accuracy, with good accuracy as can be seen in (the lines on) Fig. 3. (Those classifiers were developed and tested on only CERT data, using aggregated data for all rules.) All the classifiers do well, with similar lines and all come close to the top-left edge of the chart (optimal cut-point accuracy is high). Dots on the figure show performance of each tool alone (anonymized as tool A, B, etc.). Tools A, C, and D exhibit low false positive rates, but high false negative rates. Tools B, E, and F perform relatively poorly in identifying violations of CERT rules. Overall, this figure demonstrates that our models are quite successful in predicting true issues, outperforming individual tools. We stress, however, that these classifiers were validated with the holdout method, using only CERT-internal data.

We further evaluated these classifiers developed with CERT data by testing them on pooled collaborator data. Results are worse than training on CERT-audited data alone, not suprising because the training and testing datasets differ.

- Lasso Logistic Regression: 82%
- Random Forest: 32%
- CART: 77%
- XGBoost: 78%

Results for fully-pooled data, Java code only:

- Lasso Logistic Regression: 83%
- Random Forest: 98%
- CART: 86%
- XGBoost: 90%

Results for fully-pooled data, C code only:

- Lasso Logistic Regression: 93%
- Random Forest: 95%
- CART: 94%
- XGBoost: 93%

Results for fully-pooled data, C++ code only:

- Lasso Logistic Regression: 92%
- Random Forest: 92%
- CART: 100% (very little data, suspect)
- XGBoost: 100% (very little data, suspect)

There was too little Perl data to create classifiers alone.

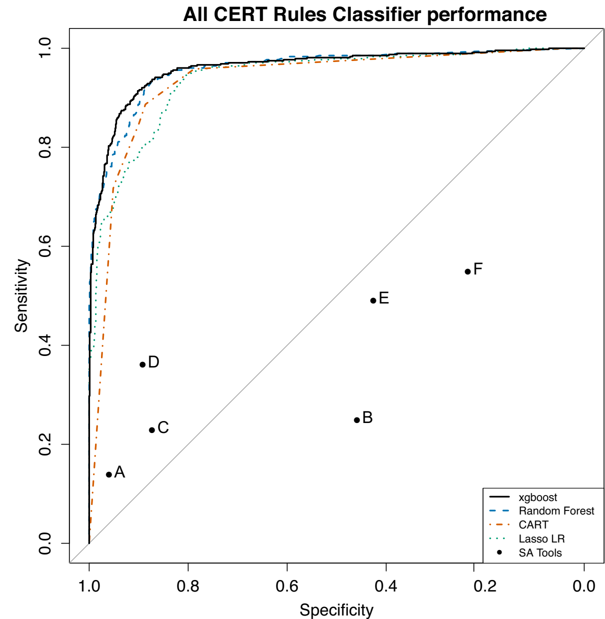

Figure 3: Tool as feature improves accuracy across classifiers

4.3 Issues

A major issue encountered in the project was lack of data. There was too little data per rule to create per-rule classifiers for all except 3 rules, and too little Perl data to create Perl-only classifiers. Some data that we wanted didn't exist in the archives, such as auditor ID, date of the alert audit, and date of the coding rule version.

One issue we encountered is licenses of the proprietary static analysis tools that forbid publication of tool performance data. We have anonymized these tool names in this paper. See Section 2.2.1 for more information.

One tool alone has has 1,396 violations of INT31-C, which represents about 30% of the trimmed data.

The enhanced-SCALe tool did not include all of the auditing determination options that we think it needed [20], which limited utility and precision of the resultant classifiers. Also, enhanced-SCALe didn't provide the utility of a comprehensive framework that provides insights from historical alert determinations on a codebase and it wasn't integrated with a code repository. Those limits precluded our collaborators from being able to use enhanced-SCALe as part of their standard auditing and development, in turn affecting the audit archives for building classifiers.

The type of complexity causing major differences in static analysis tool flaw-finding success mentioned in [3] is not directly addressed with our code complexity metrics. Those may be better matched using semantic features [21].

The issue of how to make a classifier resilient to archives with different versions of static analysis tools, coding rules, and even coding language international standards is one that needs to be addressed. The precision of a tool's alerts may change over time. The SEI CERT Coding standards are constantly changing, by design (they exist on a publicly-viewable wiki, and receive comments from

around the world that result in edits that improve the rules). CWEs and related coding taxonomies also change over time. For this work, we treated identically all archived data that mapped to the same CERT rule, although some archived alerts might have had a different audit determination using a later version of the CERT rule.

5 CONCLUSIONS, LIMITS, FUTURE WORK

We developed classifiers for static analysis alerts mapped to CERT rules, and testing showed they are accurate. However, analysis of our dataset showed that little or no ground-truth labeled data exists for many CERT rules, in the audit archives we had access to. In future work, to address that we will try to obtain larger audit archives, and also will develop a test suite with CERT rule code flaw injection for under-represented labeled rules. Also, auditor experience affects the data (and resultant classifier) quality.

In future work, we plan to modify our classifiers to provide accurate metrics for confidence, to enable us to implement our 3-partition goal (e-TP, I, and e-FP). We also plan to add features that have been used successfully to increase classifier accuracy, including semantic features [21], code repository data [7], dynamic analysis output [7], dynamic feedback during auditing [10], and developer feedback [8]. We plan to develop a labeled and sanitized dataset using only open-source static analysis and code metrics tools, and publish that dataset. Another area of future work includes developing adaptive heuristics for classifiers, particularly when the initial and subsequent datasets differ significantly. In the future, we also will develop methods to make the classifiers (and the audit archive data they are created from) resilient as changes happen to coding languages, coding rules, and static analysis tools. For future cross-project classification, we will use methods that have been shown to improve cross-project classification such as Wang using semantic features [21], Nam and Kim [15] determining a subset of labeled data and features to use, and Zhang et al. [23] using a connectivity-based unsupervised classifier.

Future work will also include integration with automated code repair. Our models could be used after automatic code repair of provably-correct repairs, to prioritize (for expert analysis) potential automated repairs that are not provably-correct. Also, analyses could be done on the costs and benefits associated with specifying different confidence thresholds for the classification models. This may make the models more amenable for adoption by software organizations. Our model supports analysis with any coding taxonomies (e.g., CWE). A key limitation of this work resides with the set of analyzers used and the willingness of potential adopters to invest in multiple analyzers. Future work could develop a trade-off curve between the number and specific analyzers used.

ACKNOWLEDGMENTS

Copyright 2018 ACM. All Rights Reserved. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. Carnegie Mellon® and CERT® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM18-0147

REFERENCES

[1] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 470–481.

[2] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.

[3] Aurelien Delaitre, Bertrand Stivalet, Elizabeth Fong, and Vadim Okun. 2015. Evaluating Bug Finders–Test and Measurement of Static Code Analyzers. In *Complex Faults and Failures in Large Software Systems (COUFLESS), 2015 IEEE/ACM 1st International Workshop on*. IEEE, 14–20.

[4] Lori Flynn, David Svoboda, and William Snavely. 2017. Hands-On Tutorial: Auditing Static Analysis Alerts Using a Lexicon & Rules. In *Cybersecurity Development (SecDev), 2017 IEEE*. IEEE, 1–2.

[5] Lori Flynn, David Svoboda, and William Snavely. 2017. Presentation for Hands-On Tutorial: Auditing Static Analysis Alerts Using a Lexicon & Rules. Software Engineering Institute, 1–108. https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=505451

[6] CERT Secure Coding group. [n. d.]. SEI CERT Coding Standards (wiki). https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards. ([n. d.]). Accessed October 26, 2016.

[7] Sarah Heckman and Laurie Williams. 2011. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology* 53, 4 (2011), 363–387.

[8] Sarah Smith Heckman. 2007. Adaptively ranking alerts generated from automated static analysis. *Crossroads* 14, 1 (2007), 7.

[9] Deguang Kong, Quan Zheng, Chao Chen, Jianmei Shuai, and Ming Zhu. 2007. ISA: a source code static vulnerability detection system based on data fusion. In *Proceedings of the 2nd international conference on Scalable information systems*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 55.

[10] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. 2004. Correlation exploitation in error ranking. In *ACM SIGSOFT Software Engineering Notes*, Vol. 29. ACM, 83–93.

[11] Ted Kremenek and Dawson Engler. 2003. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *International Static Analysis Symposium*. Springer, 295–315.

[12] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean Sutherland, and David Svoboda. 2012. *The CERT Oracle Secure Coding Standard for Java*. Pearson Education.

[13] Na Meng, Qianxiang Wang, Qian Wu, and Hong Mei. 2008. An approach to merge results of multiple static analysis tools (short paper). In *2008 The Eighth International Conference on Quality Software*. IEEE, 169–174.

[14] MITRE. [n. d.]. Common Weakness Enumeration: A Community-Developed Dictionary of Software Weakness Types. https://cwe.mitre.org. ([n. d.]). Accessed June 22, 2016.

[15] Jaechang Nam and Sunghun Kim. 2015. Clami: Defect prediction on unlabeled datasets (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 452–463.

[16] Daniel Plakosh, Robert Seacord, Robert W Stoddard, David Svoboda, and David Zubrow. 2014. Improving the Automated Detection and Analysis of Secure Coding Violations. (2014).

[17] Joseph R Ruthruff, John Penix, J David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. 2008. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th international conference on Software engineering*. ACM, 341–350.

[18] Robert C Seacord. 2014. *The CERT C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems*. Pearson Education.

[19] CERT Software Engineering Institute. [n. d.]. Secure Code Analysis Laboratory (SCALe). https://www.cert.org/secure-coding/products-services/scale.cfm?. ([n. d.]).

[20] David Svoboda, Lori Flynn, and William Snavely. 2016. Static Analysis Alert Audits: Formal Lexicon and Rules. In *Proceedings of the IEEE Cybersecurity Development (SecDev) 2016*. ieee.

[21] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 297–308.

[22] Terry Yin. [n. d.]. Lizard. https://github.com/terryyin/lizard. ([n. d.]). Accessed January 29, 2018.

[23] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. 2016. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 309–320.