Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools

Foteini Cheirdari
Department of Information Systems
University of Maryland, Baltimore County (UMBC),
Baltimore, MD 21250, USA
fcheird1@umbc.edu

George Karabatis
Department of Information Systems
University of Maryland, Baltimore County (UMBC),
Baltimore, MD 21250, USA
georgek@umbc.edu

***Abstract*— Static source code analysis for the detection of vulnerabilities may generate a huge amount of results making it difficult to manually verify all of them. In addition, static code analysis yields a large number of false positives. Consequently, software developers may ignore the results of static code analysis. This paper analyzes the results of static code analysis tools to identify false positive trends per tool. The novel idea is to assist developers and analysts identify the likelihood of a finding to be an actual true positive. This paper proposes an algorithm that makes use of a new critical feature, a personal identifier, which assists labeling the findings correctly as true or false. Experiments verified identification of true positives with a higher level of accuracy.**

Keywords- Software assurance, Vulnerability discovery, Data mining

I. INTRODUCTION

Static code analysis has been used for quite a long time to identify software vulnerabilities. Static analysis looks for security risks in source code without the need for compilation and provides a list of potential vulnerabilities. There are many open source and commercial tools used by developers or software assurance analysts to scan source code and identify vulnerabilities, and then examine the results to calculate the risk associated with each vulnerability. Static code analysis offers an overview of the security posture of the system, but there are two main limitations associated with it: First, static analysis may generate an overwhelming number of raw findings. It is quite daunting for analysts and developers to review all of them. Sometimes the developers get discouraged and choose not to review any vulnerabilities at all. Secondly, it may lack the knowledge of how data flows through the system, the dependencies and the overall software architecture. As a result, static analysis may produce a high number of false positive vulnerabilities (potential vulnerabilities identified by the static analysis tools which after human examination are deemed to be non-vulnerabilities). This paper addresses these problems and provides the following contributions:

- It analyzes, identifies, and categorizes the most common false positive types of vulnerabilities generated by several static analysis tools with a goal to identify individual tool trends. There is not a lot of information available on the type of false positives that each tool generates and researchers usually focus on one or very few languages and/or tools. We offer a comparative study of several static code analysis tools evaluated on actual software systems written in different programming languages. We limit the scope of this study solely to false positive results, due to their high number produced by static code analysis tools.
- It proposes an approach that labels static code analysis findings with a high degree of accuracy, by identifying specific important features, and describes a novel algorithm to accurately label the findings that were not manually reviewed by the analysts. In our previous work we identified the most prominent features that boost the accuracy of static code analysis [7]. In addition to those, we now include a new, yet critical feature, the personal identifier (i.e., author, source) as a catalyst in the identification of vulnerabilities.

We have identified the Stochastic Gradient Descend (SGD) algorithm [18] to be the highest performing one in classifying true and false vulnerabilities [7], and we use SGD results as a baseline to compare against our proposed algorithm results. In this study the terms vulnerabilities and findings are used interchangeably to indicate results that are produced by existing static code analysis tools.

Based on discussions with software assurance teams and developers we identified the need to use machine learning on labeling the static code analysis findings as either true or false positive.

While false negatives (findings that the static code analysis tools missed) are very important for system security, false positives can also be a direct threat with the same significance to software security. It is quite common fact that the tool findings may contain a large number of false positives and when an analyst reviews the source code to mitigate a potential vulnerability finds out that it is an actual false positive. This phenomenon occurs quite frequently and the analysts lose their trust in the tool (since it produces a lot of false positives), consequently they choose to ignore the findings. Therefore, several findings in the source code may be overlooked leaving the application vulnerable to security threats.

In addition, the number of findings is usually very high and quite overwhelming for an analyst to review all of them. When the false positives are identified and removed, the number of findings is greatly reduced, making it more feasible for an analyst to concentrate on the fewer and actual findings, increasing the security posture of the system.

II. Related Work

We categorize related work in two relevant areas on static code analysis: Research on the static analysis tools that scan and identify potential source code vulnerabilities, and research related to classification of the static code analysis vulnerability results into true and false positives.

A. *Comparing Static Analysis Tools*

There has been a lot of research in the area of identifying true and false positive software vulnerabilities. The related work is focused mostly on comparing tool functionality, ease of use, and type of vulnerabilities covered. Below is a list of studies related to static code analysis tools:

In [1] the authors are evaluating open source static analysis tools for Android apps. The study is focused on buffer errors, their frequency and the risk they pose to applications. The authors evaluated six open source tools. The research in [4] used realistic test cases to compare static code tools on cryptography misuse. The authors realized that the tools detect only a small percentage of cryptography misuse but they were able to identify the best tool based on usage scenarios. The researchers in [11] evaluated three static analysis tools by comparing their results, the type of vulnerabilities they identify, coverage and accuracy of findings. The study in [12] ranked 16 tools that analyze C and C++ based on specific checks related to their functionality and features. It concluded that it is difficult to make comparisons between each tool because they can be used for different situations and provide different types of information. Three tools are compared in [21] to find the best performing tool based on metrics like precision, recall and accuracy. It is concluded that a combination of all three tools provides better coverage in identifying the vulnerabilities. The authors in [27] compared the performance of three tools and the ease of use. The performance metrics include the accuracy and number of flaws detected. This study has similar results with the previous studies we examined above in [12] and [21], where it was concluded that each tool has different strengths and weaknesses, but in combination the tools provide more value and vulnerability coverage. In [35], the authors evaluated PMD, an open source tool for Java and the value it provides if it is run on source code before a peer review. That study revealed that it eliminates some of the peer review comments because it identifies the vulnerabilities in advance. An algorithm is proposed in [39] that ranks each tool based on precision, recall and confidence for each type of finding. The best performing tool result is used for the specific type of finding.

B. *Predicting True and false Positive Vulnerabilities*

The research in [2] uses an experimental approach to classify the vulnerabilities based on semantics. It computes a semantic signature for each vulnerability and then groups them together. In [3] several techniques are combined for static analysis (slicing, Iterative Context Extension, loop abstraction for Bounded Model Checking). In [8] thread specialization is introduced for pruning false positives of static data-race detection. In [16] we see a combined abstract interpretation and source code bounded model checking. The research in [17] proposes backward trace analysis and symbolic execution to detect vulnerabilities, taking program execution and vulnerability related paths into consideration.

The study in [19] detects and mitigates vulnerabilities and proposes specific classifiers. The authors in [20] used sensitivity analysis for feature selection and compared Artificial Neural Networks (ANN) to Support Vector Machines (SVMs) showing more accurate results by SVMs. The research in [40][41][34] uses signatures and similarity analysis to detect known vulnerabilities (CVEs) and in some cases identify new vulnerabilities. Even though t they address the code reuse issue we are also exploring, we use commercial source code with not known vulnerabilities yet (CWEs).The authors in [22] utilized SPARROW as a static analysis tool and used Java open source code. A feature vector was created using Abstract Syntax Tree (AST) and the SVM classifier. The study in [23] uses logistic regression models to predict whether the vulnerabilities are actionable. Triaging techniques and checklist assistants are used in [25] to help developers identify false and true positives. In [26] the authors compare different classifiers and use Airac, a bug finding C analyzer. The classifiers that offer the best results are random forest and boosting. The study in [28] shows how often developers use FindBugs during the development process. It identified findings important for the developers and the tool to fix, and in general offered an overview on the effect a code analyzer has in the process and the value of the final product. The authors in [32] experimented with alert patterns and ranking actionable vulnerabilities. The alerts are generated using FindBugs. The research in [33] compared 15 machine learning algorithms and identified 14 alert characteristics with high accuracy (88-97%).

III. Approach

A. *Static Analysis Tools False Positive Trends*

We performed an analysis on the trends of 10 static analysis tools. Nine are open source tools, identified by their names, and one is commercial, identified by "Commercial Tool 1." In order to analyze the findings of the tools, we used 21 commercial software systems. These are production systems written in multiple languages and contain open source components embedded in them. Each system has an average of 2 to 3 million lines of code. We scanned the 21 software systems with the 10 static analysis tools and we collected the individual results (vulnerability findings) from the tools, which generated an average of 30,000 to 50,000 findings per software system scanned. In addition, the results were manually labeled by a team of software assurance analysts. The vulnerabilities were labeled as either true of false positive after manual examination of the source code.

Table 1 summarizes the results: The column CWE identifies the Common Weakness and Enumeration number [9] an industry standard for software code related vulnerabilities. The column "Rule" is the CWE rule definition, "Tool Name" is the static analysis tool name, and "FP" is the number of false positives identified for each CWE number. The results are listed in decreasing order of the false positive findings. The table below does not compare the tools and their results since each tool usually scans source code written in different languages; it provides a list of the most common false positive finding per tool. The table displays only the results of each tool that had a significant number of false positive vulnerabilities, at least 50 or more. The relationship between each tool and type of vulnerability can be derived from the data gathered as shown in this table. We observe that the highest number of false positive results are generated by Commercial Tool 1 and they are related to CWE 310 (cryptographic issue). It is very common practice for the developers to use **`random()`** to generate a number and the tool will flag it as a potential vulnerability. But this may be a false positive if the random number generator is used for reasons other than cryptography. So the tool is correct in identifying **`random()`** usage but fails to detect that the number generated is not used for cryptographic reasons. Manual examination can correctly detect the above distinction and mark the finding as false positive. In addition, we examined the output results by the static analysis tools to identify the number of software systems (out of the 21 commercial software systems) that each tool generated false positive vulnerabilities, and rank them accordingly.

TABLE 1. FALSE POSITIVES PER CWE AND TOOL

FP	CWE	Rule	Tool Name
2103	310	Cryptographic Issue	Commercial Tool 1
1052	398	7PK - Code Quality	PHP/ CodeSniffer
988	74	Injection	FindBugs
952	398	7PK - Code Quality	Pylint
745	398	7PK - Code Quality	PMD
423	255	Credentials Management	Commercial Tool 1
334	287	Authentication and Authorization	Commercial Tool 1
325	255	Credentials Management	PMD
325	710	Improper Adherence to Coding Standards	PMD
306	94	Code Injection	FindBugs
201	12	Password in Configuration File	Commercial Tool 1
184	399	Resource Management	Commercial Tool 1/PMD
107	93	Improper Neutralization of CRLF Sequences ('CRLF Injection')	FindBugs
98	398	7PK - Code Quality	FindBugs
89	117	Log Forging	Commercial Tool 1
83	465	Pointer Issues	Commercial Tool 1
80	22	Path Traversal	Commercial Tool 1
73	79	Cross-site Scripting (XSS)	Commercial Tool 1
66	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	Cppcheck
63	456	Member variable uninitialized in constructor	Cppcheck
62	465	Pointer Issues	Cppcheck
52	79	Cross-site Scripting (XSS)	FindBugs

Table 2 shows the number of systems with false positive findings per static analysis tool. For example, Commercial Tool 1 generated false positives in 16 out of 21 systems. Although Commercial Tool 1 has the highest number of false positives, it is important to mention that it also generated the highest number of true vulnerabilities.

In addition, the programming language the software system is written in, dictates which tools can run and generate results.

TABLE 2. NUMBER OF SYSTEMS WITH FALSE POSITIVE VULNERABILITIES PER STATIC ANALYSIS TOOL

Tool Name	Number of Systems (out of 21)
Commercial Tool 1	16
FindBugs [13]	9
Cppcheck [10]	7
PMD [30]	7
PHP_CodeSniffer [29]	6
FxCop [14]	3
Pylint [31]	3
Brakeman [5]	2
Checkstyle [6]	2
Gendarme [15]	2

Fig. 1 illustrates a chart providing a visual presentation of the percentage of the false positive results each tool generated. Some tools identified the same findings and are displayed together. We believe the FP trends that we observed so far provide an indication of the correlation between the false positive type (CWE) and the tool that generates the false positive results. We plan to further explore the trends by examining and gathering statistics of additional datasets obtained from static code analysis results.

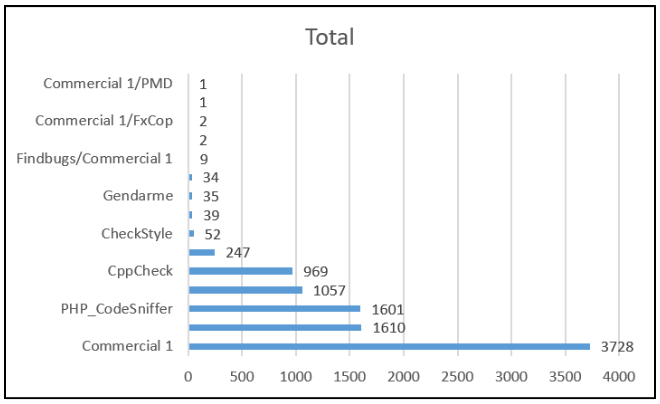

Fig. 1. Number and percentage of false positive results per Tool

B. Predicting True and False Positive Vulnerabilites

It is very common to identify the same type of vulnerabilities across different parts in the source code. If two vulnerabilities are found with the same author name and location, then there is a likelihood that the vulnerabilities are similar. The justification for this is that developers usually cut and paste the same code in different parts of the code base, or use the same programming techniques repeatedly. This common sense observation led us to take advantage of such information and search for vulnerabilities in the code, by adding the personal identifier (e.g., author name) as an additional set of features in a classification algorithm. In a previous work [7] we identified that the Stochastic Gradient Descend (SGD) algorithm generated the best results in classifying true positives and false positive vulnerabilities. The SGD, also known as incremental gradient descent, is a stochastic approximation of the gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions [38].

We propose a novel algorithm called Software Assurance Personal Identifier (SAPI) as a new classifier and we compare SGD and SAPI.

In order for the comparison between the SGD and the SAPI algorithm to be more accurate we used the author information/features on both SGD and SAPI.

Below is a list of metrics we used to evaluate our results:

- **TP**= True Positive (Correctly identifying a static code analysis finding code as a True vulnerability)
- **FN**= False Negative (Not detecting a vulnerability)
- **FP**= False Positive (Falsely identifying a suspected code anomaly as a vulnerability)
- **TN**= True Negative (Truly identifying a valid code segment as a non-vulnerability)
- **TF**= True Finding (Finding that is either true positive or true negative)
- **FF**= False Finding (Finding that is either a false positive or false negative)
- **Accuracy** is the percentage of the vulnerabilities that were identified correctly as either true positive or true negative. **Recall** is the percentage of true positive vulnerabilities that were identified
- **Precision** is the percentage of the vulnerabilities identified as true positive that were correct
- **F-measure** of the system is defined as the weighted harmonic mean of its precision P and recall R [36]

We utilized a set of open source tools to identify software vulnerabilities. The dataset used for this research is derived by static code analysis tools using an actual production software system, currently in use, as input. The output of the tools is a list of vulnerabilities and it is used as our input dataset, which contains the following attributes/features per vulnerability: ID, Severity, CWE, Rule, Description, Tool, Location, Path.

In order to obtain the ground truth on the dataset with the list of vulnerabilities, an independent software assurance team manually reviewed and labeled these vulnerability findings, which are used in our experiment. In this dataset, 1481 findings were manually evaluated. The majority of the findings are located in Java files. We added the personal identifier information (e.g., author or source) on each finding when applicable. Findings with no author information are identified as "No author" values. We used Random Sampling with Waikato Environment for Knowledge Analysis (WEKA) [37], and selected approximately 70% of the dataset (using Random Sampling with WEKA [37]) for training purposes. We trained the SGD classifier using our training set and then evaluated it using the remaining 30% test set and recorded the results.

In parallel we used our own SAPI classifier on the same training and testing datasets.

B.1 Algorithm to Calculate True and False Positive Vulnerabilities

For our calculations we used the CWE, Location (file path) and Subtype (unique identifier of the Description attribute for example method or parameter name) attributes that we identified in a previous research [7] in addition to personal identifiers.

The SAPI algorithm labels the findings as true or false using four attributes and their combinations as explained below:

- **C** is the CWE vulnerability, **S** is the CWE subtype as identified in our previous study [7], **A** is the author, **L** is the location.
- **CA** is the combination of CWE and author. We distinguish the finding based on the CWE and author or source information (personal identifier). Two findings are different, for example, if the CWE is the same but have different authors or vice versa. **CL** is the combination of CWE and Location. We consider two findings different with the same CWE but in different location in the code.
- **CLA** is the combination of CWE, Location and Author, **CAS** is the combination of CWE, Subtype and Author

Algorithm 1 explanation: We calculate C probability of occurring as $P(C) = \frac{TF}{FF+TF}$ where TF are the true findings (including TP and TN) and FF are the false findings (including the FN and FP) based on the sample training set. The same principle is used to calculate the rest of the probabilities for the selected attributes as it applies to the S, L, A, CA, CL, CLA, and CAS probability calculations. Then we average all the above probabilities together but we utilize the CLA and CAS twice in our algorithm as a heuristic rule, based on the observation and experience in the field, that identifying the author, the type, subtype and location information in combination, it increases the possibility to correctly classify the finding. As we mentioned earlier it is very common for the author to repeat code multiple times (without the knowledge that it contains a vulnerability) since developers reuse code segments, thus, propagating the vulnerability.

Developer's experience, knowledge, and programming style are additional factors that make the personal identifier an important attribute/feature to consider when looking at the static code analysis results and making predictions based on them. Below is the proposed algorithm. The algorithm goes through each individual vulnerability and calculates the probability of being a true positive. We use a threshold Θ to identify a cut-off point between a true or false vulnerability, declaring anything above the threshold to be a true vulnerability. The threshold is user-defined based on expert input, dataset, experimentation, etc. Currently, the default threshold value is 0.50.

The SAPI algorithm is a heuristic approach that utilizes the four most important attributes of each vulnerability according to our research in order to classify a finding as either true or false positive. The algorithm employs the probabilities of each vulnerability attribute to be true or false positive in conjunction with different combinations of the attributes that are shown through our experiments to improve the accuracy. The combinations that contribute more significantly into the prediction are used twice in the heuristic approach. A threshold is used to differentiate the false from the true positive as explained above.

```
ALGORITHM 1
Input: Import all Findings
Output: Label Findings as True or False Positive
"Read All Findings"
1  While Not EOF do
   "Go through each finding where Ci exists where i=1,…n,
   j=1, …n and g=1,…n."
2  Initialize the settings k =4, a =1
   "k starts at 4 because each finding has at least 4 attributes,
   n is the number of findings, a is used to count number of time
   the combinations of CiLjAg and CiAjSg appear"

3      Get Ci
4      While Ci exists do
5      Calculate Probabilities P(Ci), P(Ai), P(Li), PCi(Si)
       "Check if the following combinations exist"
       "Calculate probabilities of combinations of attributes"
6      begin
7      switch 1 do
8                  case CiLj Exists do
9                    Calculate P(CiLj), k=k+1
10                 end
11                 case CiAj Exists do
12                   Calculate P(CiAj), k=k+1
13                 end
14                 case CiLjAj Exists do
15                   Calculate P(CiLjAg), k=k+1, a=a+1
16                 end
17                 case CiAjSg Exists do
18                   Calculate P(CiAjSg), k=k+1, a=a+1
19                 end
20      end
21     end
22     SAPI(Ci)=
       (((P(Ci)+P(Si)+P(Li)+P(Ai)+P(CiLj)+P(CiAj)+
         P(CiLjAg)+P(CiAjSg))/k)+P(CiLjAg)+P(CiAjSg))/a
       "θ value decided by the user"
23     if SAPI(Ci) >ϑ then
24       Finding = True Positive (Vulnerabiltiy)
25     else
26       Finding = False Positive (Non Existent Vulnerability)
27     end
28   end
29 end
```

B.2. Experimental Results

We conducted a series of experiments with different values lower and higher than the default 0.50 threshold. We display the results of the 0.59 because they offer a clear view on how the results are affected when the threshold is increased. Any findings above the threshold are classified as true positive, while any findings below the threshold were labeled as false findings. In the future, users can determine the value of the threshold based on the focus of their prediction. The threshold can be increased or decreased depending on the priority given to the type of findings. For example, if the users have a preference for more false positives, they will have to raise the threshold. If the users prefer to identify a higher percentage of true positive vulnerabilities, they will have to use a lower threshold value. As we increase the threshold, the algorithm will identify a higher number of true negative findings, and will misclassify a higher number of true positive findings.

Figure 2 compares the SGD and the SAPI algorithms for classification. It displays the precision, recall, F measure, and accuracy results we obtained using the SGD algorithm and the SAPI algorithms. We observe the F-Measure is higher using the SAPI algorithm. The accuracy is higher when $\Theta = 0.59$, and recall is also higher. Precision is lower compared to SGD, which means fewer false positives were identified correctly, but for software assurance purposes we prefer a higher recall which is obtained by the SAPI algorithm. These results verify that more true findings were correctly identified using our proposed algorithm.

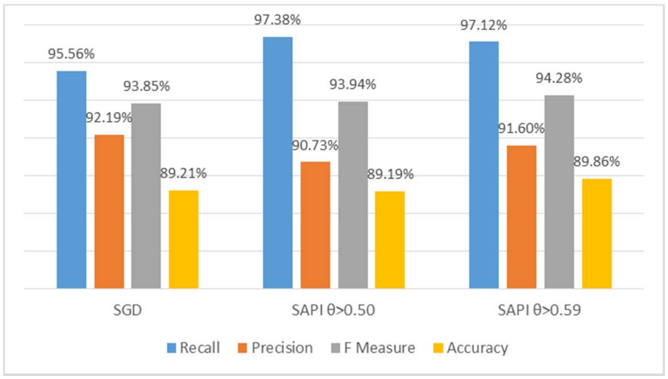

Fig. 2. SAPI vs SGD

IV. CONCLUSIONS

Static code analysis generates a lot of raw vulnerability findings and a significant subset of those generated are false positives. This large volume of results makes it hard, if not impossible for developers and software assurance analysts to examine all vulnerabilities. This research describes an attempt to compare static analysis tools based on the number of false positive results as prescribed by CWE type, in order to understand the connection between the tools and the false positive results in static code analysis. The observations made are based on an analysis and experiments, which offer valuable insight into the type of false positives typically generated during static code analysis.

We also designed and implemented an algorithm that identifies the false and true positive findings. The results of our experiments are very encouraging for future exploration and study. We observed a significant correlation between the personal identifier (author, source of code) and the possibility that the result is true or false. The personal identifier was added to the three attributes we identify as significant (CWE, Subtype, and Location) for true or false positive prediction in a previous study. These four attributes, fed into our SAPI algorithm, predict true or false positive vulnerabilities with a high accuracy. Experience and observations made from actual commercial setting were significant contributing factors in identifying patterns and realizing the significance of author, subtype and location in identifying potential true vulnerabilities. We estimated that our proposed approach can save a significant amount of time and resources in identifying the true vulnerabilities.

REFERENCES

[1] Aloraini, B.; Nagappan, M.,” Evaluating State-of-the-Art Free and Open Source Static Analysis Tools Against Buffer Errors in Android Apps” 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, ICSME

[2] Andreas Podelski, Martin Schaf, Thomas Wies, “Classifying Bugs with Interpolants”

[3] Bharti Chimdyalwar, Priyanka Darke, Anooj Chavda, Sagar Vaghani, Avriti Chauhan, “Eliminating Static Analysis False Positives Using Loop Abstraction and Bounded Model Checking”, in 2015 FM

[4] Braga, Alexandre; Dahab, Ricardo; Antunes, Nuno; Laranjeiro, Nuno; Vieira, Marco, “ Practical Evaluation of Static Analysis Tools for Cryptography: Benchmarking Method and Case Study” 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, ICSME

[5] Brakeman, https://brakemanscanner.org/

[6] Checkstyle, http://checkstyle.sourceforge.net/

[7] Cheirdari Foteini, George Karabatis, “On the Verification of Software Vulnerabilities During Static Code Analysis Using Data Mining Techniques”, in 2017 OTM

[8] Chen Chen, Kai Lu, Xiaoping Wang, Xu Zhou, Li Fang, “Pruning False Positives of Static Data-Race Detection via Thread Specialization”, in 2013 APPT

[9] Common Weakness Enumeration, https://cwe.mitre.org/

[10] Cppcheck, http://cppcheck.sourceforge.net/

[11] Emanuelsson, Pär; Nilsson, Ulf, “A Comparative Study of Industrial Static Analysis Tools”, Proceedings of the 3rd International Workshop on Systems Software Verifcation (SSV 2008), Electronic Notes in Theoretical Computer Science. 21 July 2008 217:5-21

[12] Fatima Anum, Bibi Shazia, Hanif Rida, “A Comparative study on static code analysis tools for C/C++”2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST) Applied Sciences and Technology (IBCAST), 2018 15th International Bhurban Conference on. :465-469 Jan, 2018

[13] FindBugs, http://findbugs.sourceforge.net/

[14] FxCop,https://docs.microsoft.com/en-us/visualstudio/code-quality/install-fxcop-analyzers?view=vs-2017

[15] Gendarme, https://www.mono-project.com/docs/tools+libraries/tools/gendarme/

[16] Hendric Post, Carsten Sinz, Alexander Kaiser, Thomas Gorges, “Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking”, in 2008 IEEE

[17] Hongzhe Li, Taebeom Kim, Munkhbayar Bat-Erdene, Heejo Lee, “Software Vulnerability Detection using Backward Trace Analysis and Symbolic Execution”

[18] http://curtis.ml.cmu.edu/w/courses/index.php/Stochastic_Gradient_Descent

[19] Ibéria Medeiros, Nuno Neves , Miguel Correia, “Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining”, in 2016 IEEE

[20] Iker Gondra, “Applying machine learning to software fault-proneness prediction”, in JSS 2007

[21] Imparato, Alfredo; Maietta, Raffaele Rodolfo; Scala, Stefano; Vacca, Vladimiro,” A Comparative Study of Static Analysis Tools for AUTOSAR Automotive Software Components Development”, 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Software Reliability Engineering Workshops (ISSREW), 2017 IEEE International Symposium on, ISSREW

[22] J. Yoon, M. Jin, and Y. Jung, “Reducing false alarms from an industrial strength static analyzer by SVM,” in APSEC 2014

[23] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, Gregg Rothermel, “Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach”, in 2008 ACM

[24] JSHint, https://jshint.com/install/

[25] Khoo Yit Phang, Jeffrey S. Foster, Michael Hicks, Vibna Sazawal, “Triaging Checklists: a Substitute for a PhD in Static Analysis”

[26] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, Yongdai Kim, “An empirical study on classification methods for alarms from a bug-finding static C analyzer”, In Inf. Process. Lett. 2007

[27] Mantere, M.; Uusitalo, I.; Roning, J., “Comparison of Static Code Analysis Tools”, 2009 Third International Conference on Emerging Security Information, Systems and Technologies, Emerging Security Information, Systems and Technologies, 2009. SECURWARE '09.

[28] Nathaniel Ayewah, William Pugh, “ The Google FindBugs Fixit”

[29] PHP_CodeSniffer , http://pear.php.net/package/PHP_CodeSniffer

[30] PMD, https://pmd.github.io/

[31] Pylint, https://www.pylint.org/

[32] Q. Hanam, L. Tan, R. Holmes, and P. Lam, “Finding patterns in static analysis alerts: Improving actionable alert ranking,” in MSR 2014

[33] Sarah Heckman, Laurie Williams, “A Model Building Process for Identifying Static Analysis Alerts”, in 2009 IEEE

[34] Seulbae Kim, Seunghoon Woo, Heejo Lee∗, Hakjoo Oh, “VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery”, IEEE SP 2017

[35] Singh, Devarshi; Sekar, Varun Ramachandra; Stolee, Kathryn T.; Johnson, Brittany, “Evaluating how static analysis tools can reduce code review effort” 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium

[36] SpringerLink, “F Measure”, https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-39940-9_483

[37] Weka 3: Data Mining Software in Java, http://www.cs.waikato.ac.nz/ml/weka/

[38] WIKIPEDIA, “Stochastic Gradient Descent”, https://en.wikipedia.org/wiki/Stochastic_gradient_descent

[39] Xypolytos, Achilleas; Xu, Haiyun; Vieira, Barbara; Ali-Eldin, Amr M.T., “A Framework for Combining and Ranking Static Analysis Tool Findings Based on Tool Performance Statistics” 2017 IEEE Int’l Conference on Software Quality, Reliability and Security Companion (QRS-C)

[40] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, Jie Hu, “VulPecker: an automated vulnerability detection system based on code similarity analysis” ACSAC '16

[41] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, Yuyi Zhong, “VulDeePecker: A Deep learning-Based System for Vlnerability Detection”, NDSS 2018