An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing

Roland Croft
University of Adelaide
Cyber Security Cooperative Research Centre
roland.croft@adelaide.edu.au

Dominic Newlands
University of Adelaide
dominic.newlands@student.adelaide.edu.au

Ziyu Chen
Monash University
zche0071@student.monash.edu

M. Ali Babar
University of Adelaide
Cyber Security Cooperative Research Centre
ali.babar@adelaide.edu.au

ABSTRACT

Background: Static Application Security Testing (SAST) tools purport to assist developers in detecting security issues in source code. These tools typically use rule-based approaches to scan source code for security vulnerabilities. However, due to the significant shortcomings of these tools (i.e., high false positive rates), learning-based approaches for Software Vulnerability Prediction (SVP) are becoming a popular approach. **Aims:** Despite the similar objectives of these two approaches, their comparative value is unexplored. We provide an empirical analysis of SAST tools and SVP models, to identify their relative capabilities for source code security analysis. **Method:** We evaluate the detection and assessment performance of several common SAST tools and SVP models on a variety of vulnerability datasets. We further assess the viability and potential benefits of combining the two approaches. **Results:** SAST tools and SVP models provide similar detection capabilities, but SVP models exhibit better overall performance for both detection and assessment. Unification of the two approaches is difficult due to lacking synergies. **Conclusions:** Our study generates 12 main findings which provide insights into the capabilities and synergy of these two approaches. Through these observations we provide recommendations for use and improvement.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*; • **Computing methodologies** → *Machine learning*; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Static Application Security Testing, Machine Learning, Security

ACM Reference Format:
Roland Croft, Dominic Newlands, Ziyu Chen, and M. Ali Babar. 2021. An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '21), October 11–15, 2021, Bari, Italy.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3475716.3475781

ESEM '21, October 11–15, 2021, Bari, Italy
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8665-4/21/10…$15.00
https://doi.org/10.1145/3475716.3475781

1 INTRODUCTION

Software security is vital for organisations to avoid catastrophic exploits, but it is significantly more difficult to detect Software Vulnerabilities (SVs) than regular fault detection [34]. Static Application Security Testing (SAST) tools purport to provide timely automated support for secure software development. SAST tools help developers to detect SVs during the coding phase, where it is relatively inexpensive to identify and fix security problems in source code. SAST tools, which primarily operate through rule-based approaches, statically examine source code for known vulnerable patterns that indicate the presence of potential SVs. Given SAST tools enable developers to quickly and cheaply perform quality assurance steps from a security perspective at an early stage of software development, these tools have gained sizeable traction, particularly in Open Source Software (OSS) communities [2]. However, SAST tools have had mixed success in making inroads in commercial software development practices as developers seem weary of their limitations, particularly the significant amounts of false positives [7, 23, 29].

Recently, another source code security analysis approach, Software Vulnerability Prediction (SVP) [19], has been gaining significant attention for security assurance during implementation stage. An increasing number of research efforts are developing effective learning-based approaches for the identification of vulnerable code modules [19]. These approaches use historic source code modules to devise data-driven approaches for detection or prediction of SVs [9]. Rather than using pre-defined rules like SAST tools, SVP models automatically learn the rules and patterns for SV detection.

Both of these approaches, SAST tools and SVP, are aimed at effectively and efficiently analyzing source code for SVs. However, there are several differences in the techniques of detecting security flaws in source code underpinning these two approaches. SAST tools provide granular warnings, but suffer limited individual coverage [1] and high rates of false positives [20]; SVP models can produce effective SV classifications, but they are difficult to adopt due to demanding data requirements and need for data science expertise. Perhaps due to the disparity of these approaches, their use and analysis largely exists in isolation.

Despite the increasing amount of literature on SAST tools and SVP approaches, there has been little empirical research on providing evidence-based comparison of these two types of security assurance approaches. Hence, we decided to empirically compare these two approaches to investigate their respective capabilities of source code security analysis during early stages of software development. We also aimed to empirically explore the potential viability and effect of combining the two approaches.

An empirical study like ours ought to consider a variety of factors when selecting a SAST approach, such as escaped SVs (false negatives), wasted inspection efforts (false positives), and setup requirements. Different organizations will typically value these factors differently. For instance, a mission critical system would aim to ensure that all SVs are identified; whereas an agile startup would be more oriented towards speed and lower inspection efforts. As tool integration is expensive [7, 23], most organisations would desire to adopt a singular solution. Hence, it is imperative that we outline the relative strengths and weaknesses of each approach in advance, to assist practitioners with their selection.

To empirically determine the relative trade-offs of each approach, we selected and deployed a variety of publicly available SAST tools and replicable SVP models. We then evaluated their SV detection and assessment performance on several open-source datasets to determine the relative capabilities of each approach.

Our empirical study is one of the first, if not the first, efforts aimed at contributing a large-scale assessment of both rule-based and learning-based approaches to identify their comparative capabilities for detecting SVs. The main findings of this study include:

- Both approaches exhibit similar capabilities in terms of recall, but SVP models produce much better overall performance.
- Both approaches are constrained in their capabilities for SV assessment, but most SAST tools are incapable of this task.
- Effective unification of the two approaches is difficult as there is a lack of synergy.

Our findings provide evidence-based insights for both developers and researchers. For developers, we inform the comparative value of these two approaches for source code security analysis; we also provide some recommendations that can be useful for their decisions about selecting and using one of these approaches. For researchers, we identify some necessary research opportunities based on the pain-points we have discovered for each approach. We discuss the details of our findings and their implications in section 5. Our datasets and scripts are publicly available from our reproduction package [10].

2 BACKGROUND AND RELATED WORK

2.1 Static Application Security Testing

Static Application Security Testing (SAST) tools are defined as tools that can statically analyze source code or compiled versions to help identify potential security flaws [15]. SAST is commonly performed as part of code review during the implementation phase of a software development project [50]. These tools are desirable as they provide early and immediate security feedback, which is more efficient and cheaper than finding security flaws in source code at a later stage of softare development.

SAST tools scan source code using a set of pre-defined security weaknesses (rules). Different tools use different techniques for scanning, such as pattern matching [59], data-flow analysis [27], or symbolic execution [64]. The types of security warnings they can produce are often dependent on the used detection techniques.

Researchers have conducted several user studies with software developers to identify major pain-points and areas of improvement [7, 23, 39]. These studies commonly identify that developers are reluctant to use or take appropriate actions on the SAST tools' outputs; these studies report the major reasons for lack of use of SAST tools as the excessive number of outputs, difficulty in customization, and lack of useful warning messages. To better understand the actual tool performance, several researchers have also performed bench-marking of selected SAST tools [1, 12, 24, 60].

However, none of the existing works for analysis of SAST tools considers or evaluates learning based approaches. We aim to extend this analysis by additionally comparing SVP performance. Furthermore, our analysis and assessment criteria is more extensive than the existing bench-marking works, which have only evaluated the raw detection performance on singular data sources.

2.2 Software Vulnerability Prediction

Software Vulnerability Prediction (SVP) is another approach to performing source code analysis to detect SVs or security risks early in software development. That is why SVP approaches have several characteristics similar to SAST [9]. However, rather than searching for a set of predefined security weaknesses, SVP models aim to automatically learn SV knowledge and patterns from historical data. This process has seen continual technical advancement over the last decade through a large number of research efforts [19].

The SVP process follows a standard pipeline. First, a model extracts data modules from historical software repositories, such as version control and bug tracking systems. These modules include both *vulnerable* and *clean* code, to help learn the distinction between the two classes. Informative features are then generated from the code, such as software metrics or code tokens, for a model to learn from using a specified classification method (e.g., random forests). The trained prediction model can then be used to classify whether or not incoming code modules are potentially vulnerable.

There are two main approaches for SVP [17]: software metric based approaches, and vulnerable code pattern recognition. The former utilizes software metrics, such as code complexity or development characteristics, to help decide which code modules are at risk of containing SVs. This approach relies on the correlation of these metrics to the emergence of SVs, which has been reported in several studies [6, 51]. However, this assumption often means that these approaches are inaccurate and unable to identify SVs explicity [33]. The latter approach utilises the explicit code tokens to identify vulnerable patterns in source code. This approach has been shown to outperform software metrics [61], and is the more popular approach in literature [17].

2.3 Combined Approaches

Whilst these two approaches largely exist in isolation, some studies have investigated their combination. Learning-based approaches have been used to enhance static analysis tools by reducing their false positives [66], or to assist with the output analysis by ranking tool warnings [41, 46]. Alternatively, the output of static analysis tools has also been used to enhance the capabilities of SVP models, but these attempts have produced uninspiring results [16, 45].

Rahman et al. [45] performed a comparative study of static bug finders and statistical prediction approaches to identify synergistic aspects that may be leveraged by combining the two types of approaches. In terms of performance, they found that the static bug finders perform similar to the prediction models. They also reported that the output of the static tools did not improve the performance of statistical prediction techniques, but prediction models were able to produce better orderings of the static tool outputs than the natural ordering. Whilst our study has an overall goal that is similar to Rahman and colleagues' [45] work of comparing learning-based method to traditional static analysis methods, there are several key differences that has resulted in unique findings from our study.

Rahman et al. [45] only examined software bugs, not vulnerabilities. SVs exhibit different characteristics to regular software bugs as they do not necessarily represent functional flaws [50]. Additionally, SVs are much more scarce [52] and harder to detect [34]. As such, we also expect the characteristics and performance of source code security analysis approaches to differ substantially. SV mitigation also requires better understanding of code and potential consequences [54]. Hence, we have also investigated assessment that is another vital task of SVs. Furthermore, Rahman et al. [45] only provided a comparison of the inspection costs of each approach. We have instead focused on the performance and capabilities of each approach, as SV detection is much more critical; exploited vulnerabilities can lead to catastrophic consequences.

3 RESEARCH METHOD

The goal of this research is to empirically investigate the comparative value of source code security analysis approaches. We aim to answer the following Research Questions (RQs):

- **RQ1** *What is the capability of SAST tools and SVP models for SV detection?*
  We first aim to identify the performance and efficacy of each source code security analysis approach. It is vital that approaches have practical capabilities for detecting SVs.
- **RQ2** *What is the capability of SAST tools and SVP models for SV assessment?*
  Another critical component of SV mitigation is assessment [54]. Once we have detected a vulnerability, we must also be able to identify its type, so that we can infer potential impacts and appropriately plan mitigation. Hence, we seek to identify the capabilities of each approach for SV assessment.
- **RQ3** *Can these approaches complement each other?*
  Finally, we are interested in investigating the potential synergies between SAST tools and SVP models. Despite the differences of these two approaches, we aim to determine whether the similarity of the task can enhance their combined performance for a unified approach; the outputs of the SAST tools are used as inputs for an SVP model.

To address these RQs, we have conducted a large-scale comparative study into the application of these approaches on a variety of SV datasets. Through this study, we have empirically evaluated the operation of a selected set of representative tools and SVP models. Figure 1 presents an overview of the study design.

3.1 Selecting and Extracting Case Study Datasets

The goal of SAST tools is to help developers to identify SVs in source code. Hence, we aimed to collect a representative dataset of a codebase containing SVs. To make our set of SVs more generalizable to a real-world setting, we primarily constructed our datasets using real-world data from open-source repositories.

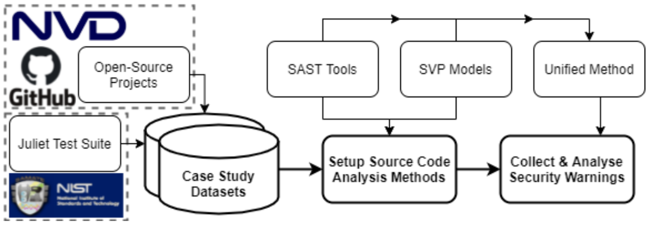
NVD
GitHub
Open-Source Projects
Juliet Test Suite
NIST National Institute of Standards and Technology
Case Study Datasets
SAST Tools
SVP Models
Unified Method
Setup Source Code Analysis Methods
Collect & Analyse Security Warnings

Figure 1: The overall methodology for evaluating SAST tools and SVP models.

Both of the approaches that we have investigated (SAST tools and SVP models) are language specific. It is common knowledge that vulnerable code patterns are difficult to translate across programming languages [36, 50]. That is why we decided to collect the code datasets for C/C++. We made this decision as these two programming languages are generally known to have more SVs due to their lower levels of abstraction [49]. C/C++ are also the 1st and 4th most popular programming languages respectively as per the March 2021 TIOBE index [58].

For data collection from open source projects, we utilized the VulData7 framework developed by Jimenez et al. [22]. This framework automatically collects vulnerability report data (i.e., fixing commits of source code files) from the relevant software archives (Git and NVD reports) for open source projects. We formed our vulnerability dataset from pre-patch file versions. VulData7 is automatically configured to collect data for four repositories: Linux kernel, Wireshark, OpenSSL, and SystemD. However, we excluded SystemD due to the small initial size of this dataset (9 SVs).

Additionally, to improve our data quality we attempted to address the limitations reported for the VulData7 tool [22] by performing two post processing steps. First, we removed all the duplicate data entries introduced from multiple code branches by dropping duplicate file contents. Second, we ensured that the vulnerability reports were actually relevant by removing fixing commits which do not make functional changes to code (i.e., only changes to comments or white space). Three of the authors manually validated a random sample (n=67, significant size, [8]) of entries to ensure that the quality of our data is sufficient (that they contain a relevant vulnerability to the associated security report). Disagreements were resolved through discussions. We found that 90% of our entries were valid at a confidence level of 90% +/-10%.

We also required examples of non-vulnerable files so that we can train SVP models and report false positives. Hence, we obtained a set of C/C++ files which were not labelled as vulnerable to form our non-vulnerable class. As our vulnerable files come from different versions of each repository, we similarly sampled non-vulnerable files from different commits made to each repository. To help ensure that this assumption of non-vulnerability was accurate, we only considered files which did not have any associated vulnerability reports across all versions. This is because we were unable to accurately determine in which version a vulnerability is introduced or removed in a file [14]; fixing commits may only be partial or incomplete. Additionally, we excluded the file versions where the commit message contained a security keyword (using the security keyword list from Le et al. [28]) as these files may be related to SVs. Finally, we only kept unique files to avoid redundancy.

Given the open-source repositories use security advisories to record SV data, they are only representative of SVs detected during the testing or maintenance phases. To this extent, these data sources do not document SVs that have been removed during implementation or code review [40], which is a conundrum as it is these very SVs that SAST tools are oriented towards [15]. Hence, we complemented our dataset with the inclusion of a synthetic test suite to capture more conventional SVs. We selected the Software Assurance and Reference Dataset (SARD) produced by the National Institute of Standards and Technology (NIST) [37]. We downloaded all the test cases from version 1.3 of the Juliet Test Suite for C/C++. This test suite provides labelled examples of both clean and vulnerable code. Hence, to obtain the file examples of both classes, we split the test cases into separate vulnerable and non-vulnerable files. Table 1 reports the summary statistics of our collected datasets.

Table 1: Statistics of the selected datasets.

Data Source	Project	% Vulnerable Files	Total Files	# LOC	# CWE Types
Open-Source Projects	OpenSSL	32.37%	2258	1,567,897	15
	Wireshark	9.06%	3731	5,678,133	12
	Linux Kernel	7.66%	34,961	25,600,912	18
Synthetic Test Cases	Juliet Test Suite	49.61%	200,159	16,324,088	13

3.2 Selecting SAST Tools

For SAST tool selection, we first analysed the list of tools documented by NIST [38] and OWASP [15]. We applied two restrictions when selecting the SAST tools for this study. First, the tools must be open source as we aimed to consider the most accessible and widely used tools. Second, we did not consider tools which had usage limits or purely operate through a graphical user interface, as we aimed to conduct a large scale benchmarking of these tools. Based on these requirements, we selected 3 tools: Flawfinder, Cppcheck, and RATS. These tools have been commonly used in prior works [1, 24] or by large organizations [7]. We provide a brief description of each tool:

Flawfinder [62] uses a simple pattern matching method to match source code text with a built-in database of known vulnerable C/C++ functions. The tool is designed to be fast, rather than accurate.

Cppcheck [31] uses a unique bi-directional data flow analysis method to focus on detecting SVs resulting from undefined behaviour. Its goal is to produce very few false positives. Cppcheck additionally provides style and code quality warnings, but these SVs are ignored for our study as they are not included in our datasets.

RATS [4] is a Rough Auditing Tool for Security, which uses a simple pattern matching method similar to Flawfinder. Hence, it is also oriented towards speed rather than performance.

We represented the collected vulnerable and non-vulnerable files as a single codebase for each of the four datasets, which we fed as input to the three selected SAST tools[1].

3.3 Building SVP Models

There are two major approaches for SVP models, determined by the utilised features: software metrics and code tokens. We build SVP models for these two approaches by using the features proposed by Munaiah and Meneely [35], and Scandariato et al. [48]. These models operate at the file-level; the most common granularity of existing SVP techniques [17].

Munaiah and Meneely [35] proposed and evaluated 10 software metrics for SVP. For our replication, we used eight of the software metrics[2]. We excluded the *# Paths* metric as we found it had a high correlation to cyclomatic complexity, and the *Offender* metric as it aligned with the rules of our class separation. The *Contribution, Collaboration* and *Churn* metrics could not be extracted for the Juliet dataset. Although a higher number of software metrics have been used in previous studies [68], these metrics often overlap or correlate, which can hinder the performance of a model [18].

For code tokens, we used text mining to extract Bag-of-Word features as proposed by Scandariato et al. [48]. We built a custom tokenizer for C/C++ to appropriately handle code syntax, which is available in our reproduction package [10]. For our replication, we excluded the comments from tokenization and limited the vocabulary size to 1000 tokens. For the Juliet dataset, we replaced function names with neutral strings as we found the original function names to be separated between the *clean* and *vulnerable* files.

We additionally created a combined SVP model using both software metric and code token features, for the purpose of direct comparison with SAST tools. However, the performance of the combined model did not differ to just using code token features, so we do not report its performance separately. The models were trained using a variety of common machine learning classification algorithms [65]: K-Nearest Neighbors, Support Vector Machine, Random Forest, and AdaBoost. We observed that the Random Forest classifier produced the best performance values for each experiment, except for the Juliet dataset when using code token features, and hence we refer to this algorithm the most.

Additionally, to address RQ3 we experimented with creating a unified method; an SVP model using features created from the output of SAST tools. The SAST tool features were inspired by the features proposed by Ribeiro et al. [46] for ranking SAST tool output. For each file, we considered the number of warnings and mean severity level for each tool. Additionally, we considered the aggregated features for the three tools as we expected multiple warnings to be more likely to represent a true vulnerability. We used the mean number of warnings and normalized severity of the three SAST tools, the ratio of warnings that overlap, and the average number of warnings within four lines of another warning.

To properly construct and evaluate our SVP models, the appropriate hyperparameters[3] of each model were tuned using cross-validated grid search on the training set. To evaluate the performance of our models, we followed the recommendations by Tantithamthavorn et al. [57] to use out-of-sample bootstrap for evaluation. Out-of-sample bootstrap constructs a training set using random sampling with replacement from the original dataset. The model was then trained and validated on the random training set,

[1] For Juliet, we excluded warnings relating to *srand*, as these are located in the main function of Juliet test cases, making them consistent for all test cases.

[2] The used software metrics are available in our reproduction package [10].

[3] The tested hyperparameters and values are listed in our reproduction package [10].

and finally evaluated using the remaining unseen entries of the dataset which were not sampled. Although time-based validation is a recommended evaluation process as it better represents the real world scenario [13], it can produce performance estimates that are often biased or unstable [57]. Hence, we used out-sample-bootstrap to obtain more stable performance evaluations for the purposes of our comparative analysis. Due to the randomness of this process, we averaged the performance of 30 runs.

3.4 Evaluating Tool/Model Outputs

To form a comparison of SAST tools and SVP models, we needed to evaluate both on an equal footing, i.e., at the same level of granularity. To achieve this, we considered the warnings of both approaches at the file-level; whether a file was flagged as vulnerable or not. Whilst SAST tools produce individual warnings at the line-level, SV mitigation does not operate at this granularity in reality. To properly identify a vulnerability, a developer must inspect and understand a much larger code context [53]. For our analysis, we expanded this context to match that of SVP models at the file-level. As SAST tools can produce multiple warnings for a single file, we aggregated the SAST tool outputs into a single classification per file. True positives are vulnerable files with warnings, false positives are non-vulnerable files with warnings, false negatives are vulnerable files without warnings, and true negatives are non-vulnerable files without warnings.

We evaluated the approaches for two main tasks; detection and assessment. Ultimately, the goal of source code security analysis is to identify potential SVs in source code. Hence, detection is the most important capability of each approach. However, unlike other types of code defects that may cause software to function unexpectedly or incorrectly, SVs expose code to potential exploits. Hence, we also need to assess the SV type so that we can better understand the relevant nature and impacts of SVs.

For the assessment metric, we used the Common Weakness Enumeration (CWE) [11] to classify SV type. Through the CWE type, we can understand the potential severity and impacts. As the labels in our dataset come from NVD or synthetic cases, the majority of vulnerable entries have an associated CWE type to serve as a ground truth. CWE uses a hierarchical structure, so we grouped CWEs to their highest level category to lower the dimensionality of our analysis, similar to Paul et al. [40]. For SAST tools, we considered whether they were able to produce a warning of the correct type for a file. For SVP models, we evaluated their ability for assessment through multi-class SVP models, which predict the CWE type (or no type for non-vulnerable cases) of a file.

To measure the effectiveness and assessment, we used the performance measures of precision, recall, and Matthew's Correlation Coefficient (MCC). Although the F1 Score is often used as an overall indicator of model performance, MCC provides a more reliable statistic which considers all four confusion matrix categories [5]. Hence, we considered MCC for selecting optimal SVP models and parameters. Precision and recall have a range from 0 to 1, while MCC has a range of -1 to 1, where 1 is the best value for all metrics.

We also conducted additional manual inspection of the output of SAST tools and SVP models on a random 10% sample of files for each dataset (using the remaining 90% of files as training data for SVP models), to better determine the detection capabilities of each of the compared approaches. We refer to this as our comparison set. We manually analysed up to 15 random files that SAST tools and SVP models each flagged as true positives, false positives and false negatives, exclusively and in combination. In total, we manually analysed the classifications of 265 files.

4 RESULTS AND ANALYSIS

4.1 RQ1: What is the capability of SAST tools and SVP models for SV detection?

Table 2 displays the performance metrics of each of the studied approaches on the four projects' datasets. In terms of precision and MCC, each of the SVP models outperformed all three of the SAST tools (0.54 higher precision and MCC on average across the four datasets). We confirmed the performance increase to be significant using the Wilcoxon signed rank test [63] on the performance difference for SAST tools compared to SVP models for each of the four datasets (p = 0.002 for precision and MCC scores). However, the recall rate of SAST tools was comparable to, and in some instances even better than that of SVP models' recall. Likewise, using the Wilcoxon signed rank test [63], we failed to reject the null hypothesis that recall rate is different between SAST tools and SVP models (p = 0.1). However, the high recall rate of SAST tools came at a significant trade-off to precision due to the high number of false positives that these tools produced.

Finding 1: *We cannot conclude that SAST tools and SVP models produce different recall values.*

Finding 2: *SVP Models exhibit significantly better precision and overall performance.*

We examined the similarity of predictions for the vulnerable files in our comparison set for the aggregated SAST tool outputs and the combined SVP model. Table 3 displays the Jaccard similarity coefficient [21]. We observed that the detected vulnerabilities for SAST tools and SVP models were overall 61% similar across the four projects. Considering that the recall value for these two approaches was also not significantly different, we claim that the detection capabilities of these two approaches are similar.

Finding 3: *The detection capabilities of the positive class are similar for SAST tools and SVP approaches.*

Unexpectedly, SAST tools produced the lowest performance (in terms of recall and MCC) on the Juliet dataset. An MCC value that is close to 0 indicates that the approach is doing little better than random guessing. We expected SAST tools to perform better on this dataset as it is of lower complexity; SVs are more distinct and rigid. However, upon manual inspection we discovered the difficulties to stem from the small class separation in this dataset; the vulnerable and non-vulnerable files only differentiate by minor line changes that alter the security. SAST tools were unable to effectively differentiate between vulnerable and non-vulnerable files; they often flagged both files or none of them. This reinforces the notion that SAST tools produce extremely large numbers of false positives that makes it difficult to identify true vulnerabilities [7,

Table 2: Performance comparison of the source code security analysis approaches.

SAST Tools

Project	Flawfinder				Cppcheck				RATS			
	# Warnings	Recall	Precision	MCC	# Warnings	Recall	Precision	MCC	# Warnings	Recall	Precision	MCC
OpenSSL	52.08%	0.732	0.455	0.292	30.96%	0.703	0.735	0.589	55.67%	0.744	0.433	0.261
Wireshark	29.21%	0.609	0.189	0.220	14.90%	0.533	0.324	0.340	27.10%	0.571	0.191	0.213
Linux	37.57%	0.676	0.138	0.178	25.87%	0.710	0.210	0.297	3.55%	0.195	0.421	0.249
Juliet	47.71%	0.488	0.507	0.021	5.80%	0.075	0.638	0.071	11.83%	0.128	0.535	0.029

SVP Models

Project	Software Metrics				Code Tokens			
	# Warnings	Recall	Precision	MCC	# Warnings	Recall	Precision	MCC
OpenSSL	30.59%	0.845	0.892	0.795	30.71%	0.915	0.963	0.904
Wireshark	6.27%	0.495	0.721	0.561	5.90%	0.554	0.856	0.660
Linux	3.67%	0.413	0.863	0.565	4.66%	0.565	0.930	0.709
Juliet	52.60%	0.893	0.843	0.729	53.14%	0.922	0.862	0.778

Note: The # Warnings column shows the percentage of files flagged. For the SVP models, all values are averaged over 30 runs.

Table 3: Jaccard similiarity coefficient of SAST tool and SVP model warnings for vulnerable files.

	OpenSSL	Wireshark	Linux	Juliet	Mean
Similarity	0.86	0.43	0.61	0.53	0.61

23]. However, SVP models were able to achieve good performance values on both the open-source datasets and the Juliet dataset. This shows that they were able to differentiate between minor changes in code as well as detect SVs of a varying complexity.

For SVP models, software metrics performed worse than code tokens. SVP models also generally performed worse for the Wireshark and Linux datasets. This is likely because these datasets suffer from class imbalance issues [55].

Across the three SAST tools, Flawfinder was able to produce the highest recall values. However, Flawfinder also produced the highest number of warnings across the four projects. Contrastingly, Cppcheck achieved the highest precision but produced the lowest number of warnings across the four projects. However, the recall of Cppcheck was not higher than the other two SAST tools, which indicates a trade-off between recall and precision in these tools. Tools which flag abundantly are more likely to identify both true and false positives. It is very difficult for these tools to achieve both high precision and recall.

Finding 4: *SAST tools exhibit a trade-off between precision and recall, based on the number of warnings.*

SVP models generally flagged a lower percentage of files to SAST tools. The flagged number of files was more representative of the true distribution of vulnerable files seen in Table 1. Hence, these models achieved much higher values of precision compared to SAST tools. Despite this disparity, we would actually expect the inspection costs of SAST tools to be much lower due to the finer granularity of the line-specific warnings they provide. However, upon manual inspection we found this not to be the case as the line-level warnings were extremely inaccurate. For the vulnerable files of the open source datasets that SAST tools flagged, we found that only 5% had an appropriate line-level warning.

Finding 5: *SVP models produce a lower number of files to inspect.*

Table 4: Recall of line-level predictions for SAST tools.

Project	Flawfinder	Cppcheck	RATS
OpenSSL	0.009	0	0.016
Wireshark	0.011	0.001	0.008
Linux	0.008	0.002	0.004
Juliet	0.167	0.036	0.057

Table 4 displays the recall of SAST tools at the line-level. We identified vulnerable lines through the vulnerability fixing commit changes for the open source datasets, and through the Juliet metadata. All three tools struggled to produce any true positive warnings for any of the projects. We note that the line warnings were more useful for the Juliet dataset; from our manual inspection, we found that 77% of the true positives had valid line warnings.

We found that most of the SAST tool warnings were incidental; unrelated to the true nature of a vulnerability. For Flawfinder and RATS, the warnings predominantly stemmed from the use of string handling functions, e.g., *memcpy, char* and *strlen.* For Cppcheck, the bulk of errors came from syntax issues. For the open source datasets, however, the true vulnerabilities typically occurred through missing logic and checks, i.e., checking for the length of a buffer or existence of a pointer. As the vulnerable code was often contained in for/if statements, rather than function use, it was very difficult for SAST tools to detect or localize these.

Finding 6: *The majority of SAST tool warnings are incidental.*

Despite these incidental warnings, SAST tools were still able to produce warnings at the file-level with high recall. We suspect this was because files containing vulnerabilities tend to be more complex [35]. Hence, they also tend to use more approaches flagged by SAST tools. This correlation is unlikely to be strong due to the low precision of these methods. However, in this sense SAST tool warnings operate similarly to software metrics as indicators of vulnerabilities. We investigated this correlation further in RQ3.

SAST tools were also unable to predict vulnerabilities stemming purely from methods calls; functions defined externally in separate files. This is because SAST tools are unable to infer any information about the surrounding semantics of these calls. SVP models were able to correctly classify some of these limited context files as vulnerable as they analyse the file as a whole. However, they similarly struggled in their classification as they produced many false positives and negatives.

In continuation of our manual analysis, we found it was very difficult to discern between what SVP models could and could not predict. This is because they made predictions at the whole file level, whilst the actual vulnerable lines were usually small and consistent. For instance, the vulnerable code may be the same for multiple files but SVP models produced different positive and negative predictions for each file. The predictions were not reliable; and there was no way of interpreting why a model had made a particular prediction.

Finding 7: *The prediction capabilities of SVP models are not transparent.*

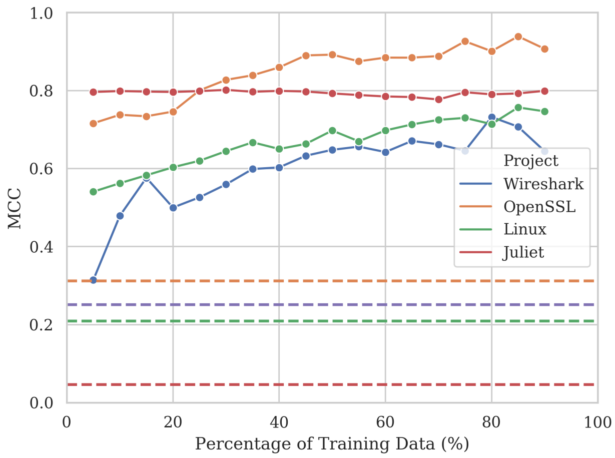

Figure 2: Model Performance for Increasing Percentages of Training Data. Note: MCC of the SAST tools is marked via a dashed line

It should be noted that SVP models were only evaluated on a portion of the data, whereas SAST tools were applied to the entire dataset. This is due to the large training requirements and data hungriness of SVP models. Hence, we also examined the impacts of data requirements on SVP models, i.e., how much data an organisation actually requires in order to use these models. Using the best SVP model configuration, we measured the performance impacts of reducing the amount of training data in increments of 5%. Figure 2 displays the MCC value for each model for varying percentages of training data, as well as the MCC of using SAST tools in combination.

SVP model performance was still greater than SAST tools even when using as little as 5% of the available data. Hence, early application of these models is not unreasonable, and efforts should be made to implement models as soon as data is made available. However, as SAST tools do not have data requirements, these tools can even be used at project inception, potentially to assist in acquiring training data for SVP models.

Using the Kendall rank correlation coefficient [25], we observed a positive correlation between the amount of training data and the model performance for each of the open source datasets (p <0.01), but not the Juliet dataset. This increasing performance shows that SVP models continue to improve with the maturity of a system, as new data becomes available.

Finding 8: *SAST tools can be used by a project immediately. However, SVP models exhibit better performance than SAST tools even with low data requirements.*

4.2 RQ2: What is the capability of SAST tools and SVP models for SV assessment?

Vulnerability assessment is a vital task to assist with SV mitigation [26]. Table 5 reports the assessment performance of each approach.

Table 5: MCC of the assessment task for source code security analysis approaches. Note: RATS does not produce assessment information.

	SAST Tools		SVP Models	
Project	**Flawfinder**	**Cppcheck**	**Software Metrics**	**Code Tokens**
OpenSSL	0.208	-0.033	0.583	0.661
Wireshark	0.215	-0.012	0.359	0.435
Linux	0.156	-0.022	0.346	0.437
Juliet	0.235	0.102	0.575	0.769

SVP models again performed well for the assessment task, but experienced a performance decrease. This is to be expected as the class imbalance is further exacerbated in the multi-class problem, making assessment a more difficult task for learning-based approaches.

However, SAST tools were largely unable to effectively perform assessment; only Flawfinder produced the values which were a little better than random guessing. This incapability could be due to a few reasons: i) as discussed in section 4.1, the majority of the SAST tool warnings for the open source datasets were incidental; hence, it did not reflect the true nature of the vulnerability. This would be similarly perpetuated for SV assessment; ii) due to the rule-based nature of the SAST tool approach, the tools are only setup to detect certain types of SVs, which does not cover all the SVs present in a real-world scenario. This would similarly explain the poor performance of SAST tools on the Juliet dataset, as this dataset contains a wide array of different SV types; iii) assessment is not the major objective of tools, i.e., RATS does not even provide information for this task. Whilst the tools will often provide their own form of classification for their output (e.g., *fixed global buffer size*), these do little to assist with assessment and understanding of the impacts.

Finding 9: *Assessment performance is worse than detection performance for both approaches.*

Finding 10: *SAST tools are generally incapable of assessment.*

SVP models have a similar prediction constraints as they are limited to the contents of the training data. They can only provide accurate classifications for CWE types that have regularly occurred in the past. Hence, models struggle to predict SV types with few examples, which is exacerbated by the scarcity and imbalance of SV data, and are completely incapable of predicting unseen types.

The presence of these limitations was reinforced through our manual inspection, as we observed both SVP models and SAST tools struggled to flag more obscure vulnerabilities. False negatives often came from vulnerabilities that were not as well represented in the data, such as race conditions and timing attacks.

Finding 11: *Both approaches are constrained in the types of vulnerabilities that they can assess.*

4.3 RQ3: Can these approaches complement each other?

To evaluate the combined performance of these two approaches, we considered two approaches for unification: a naive approach in which we simply merge the outputs of the two approaches, and a more sophisticated approach in which we use the outputs of SAST tools as features for SVP models (as described in Section 3.3).

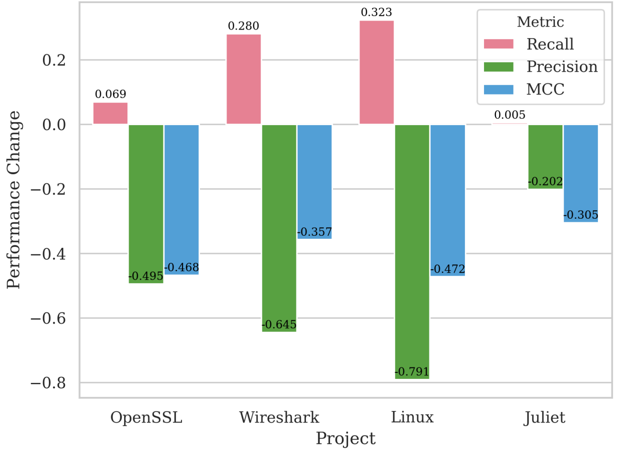

Figure 3: Difference in performance for merging SAST tool and SVP model outputs in comparison to just using SVP models

We first investigated the simple approach of using both approaches in parallel. We considered the merged outputs of all three SAST tools and the combined SVP model to identify flagged files. Figure 3 displays the mean performance change in each dataset in comparison to the original SVP model performance.

By merging the outputs from both approaches, the recall of these approaches improved significantly (except for Juliet due to the low recall for SAST tools on this dataset). For the Wireshark and Linux dataset, there were a high number of false negatives for SVP models, which SAST tools were able to flag. Hence, if detecting as many vulnerabilities as possible is the sole objective of source code security analysis regardless of the inspection efforts, then using multiple approaches is a good approach. However, the increase in recall came at a significant trade-off to precision due to the general imprecision of SAST tools. The use of multiple approaches results in an overall performance decrease.

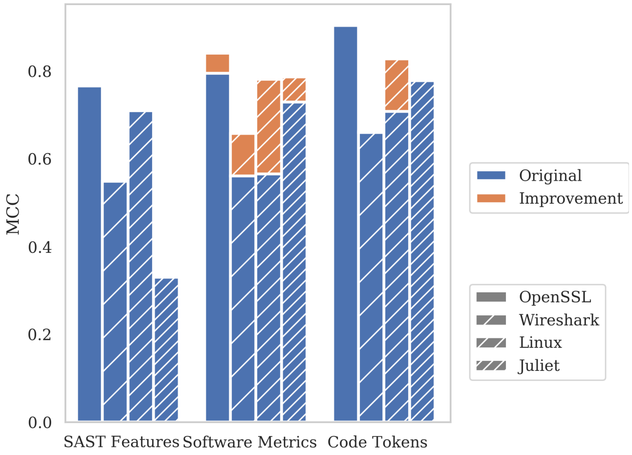

Figure 4: Model performance improvement for the addition of SAST features.

For our two main feature sets (software metrics and code tokens), we inspected model performance increase of adding SAST based features to SVP models, shown in Figure 4. We also considered the use of SAST features alone (the left-hand values of Figure 4) to represent the scenario in which we used learning-based approaches to enhance the SAST tool outputs, through the reduction of false positives [41, 66]. Like the previous studies on such topics, we also observed that this approach can improve the performance of SAST tools. However, SAST features alone did not outperform regular SVP models, despite introducing the same weakness of data requirements and coarse granularity. These approaches only serve to help reduce the gap in performance between SAST and SVP. They do not reduce false negatives, hence, they are unable to produce a approach that is greater than the sum of its parts.

A more promising approach is to use SAST features in combination with regular SVP models. As seen in Figure 4, SAST features were able to improve the software metrics model due to their similar individual performances. However, SAST features still performed worse than the code tokens (with the exception of the Linux dataset).

Whilst the unification of two approaches (i.e., SAST tools and SVPs) appeared useful in some circumstances, it was generally unable to elevate the potential of SVP. For the use of a combined approach to be cost effective for an organisation, we would require their combined use to contribute more value than using one of the approaches individually, which we find not to be the case in terms of detection performance. Hence, our investigation of the potential viability of unifying the two approaches concluded insufficient outcomes; that means there is a need of developing more sophisticated approaches in the future.

Finding 12: *The two approaches lack synergy. It is difficult to improve the overall performance through unification.*

5 DISCUSSION

This study aimed at empirically investigating the capabilities of learning-based approaches for SAST in comparison to rule-based tools for determining the potential synergies between them. Through RQ1 and RQ2, we have found that SVP models exhibit better *overall* performance for both detection and assessment. However, as SVP models have some caveats that SAST tools do not, such as stringent data requirements, coarse granularity, and poor transparency, SAST tools can still be used to achieve recall rates similar to the learning-based approaches. For RQ3 however, we have identified that these two approaches actually lack synergy, through the insignificant and even negative results we obtained for their unification.

However, both of the compared SAST approaches are far from achieving specific SV detection and localization. The actual vulnerable lines and components of a file are usually small (1-5 lines), but the average open source code file is very large (1000+ lines). Hence, inspection from a file-level warning requires significant effort. Whilst SAST tools produce raw warnings at the line-level, the warnings are not accurate; there is a trade-off between granularity and effectiveness. Without effective localization, SAST approaches are limited in their actual testing capabilities. They are best used to assist with manual inspection or to direct more expensive dynamic testing efforts. Whilst this is useful for reducing the effort required for software quality assurance, developers still desire stronger capabilities from source code security analysis approaches in the modern agile software development paradigms [39].

5.1 Observations on Method Usability

Developers also require tools to be convenient. Whilst tool performance is the most important capability of a SAST approach, the usability of tools also serve as barriers to adoption [7, 23, 39]. Hence, to provide an additional view of each approaches' capabilities for adoption, we contribute a discussion of their comparative convenience and ease of application gained through our experiences from conducting this empirical study. We frame our discussion based on the desired properties outlined by Poel [43].

Responsiveness. The responsiveness of an approach is the required run-time to scan files. The pattern matching based SAST tools (Flawfinder and RATS) only took a few minutes to scan the datasets. This speed does not slow down or require stopping of development processes, which is optimal [7]. SVP models operated even faster, taking an average of 6 seconds to make their predictions. However, this required the model to already be setup, which is a time consuming process (training time took an average of 86 minutes per model). Cppcheck, which uses a more sophisticated data flow analysis method, was considerably slower as it took an average of a few hours to scan the entire dataset, which is not scalable. Codebase scans would need to be conducted overnight or at implementation stopping points, making integration difficult.

Error-proneness. During our benchmarking process, Flawfinder produced several run-time errors, but was able to complete its scan. However, Cppcheck occasionally crashed due to code scanning errors. This error-proneness is likely from the generalisability of these tools, which are designed to operate on heterogeneous codebases. Hence, there will inevitably be errors for certain applications' contexts. SVP models did not produce any errors during run-time as they are built towards their respective context.

Interpretability. Interpretability has two dimensions [32]: the local interpretability of individual predictions, and the global interpretability of the overall tool/model. In terms of local interpretability, SAST tools provide a description for each flagged warning. This makes the warnings more actionable by developers during manual verification, as it assists with understanding and mitigation. SAST tools also have global interpretability as their approaches are transparent. Their documentation describe the rules used to produce warnings, and the checks they are capable of. The SVP models we implemented were not interpretable or transparent (*Finding 7*). Although methods exist for interpretable machine learning, we did not incorporate these as they have not yet been properly explored in existing SVP literature [67]. Making SVP models interpretable is important however, to increase model trust and actionability [56].

Setup. For SAST tools, we have found that the setup and configuration were generally easy. All three tools had minor customizability. However, this may only be reflected in open source tools, as the prior studies have found the configuration to be a pain point [7, 23, 39]. We found the setup and configuration to be much more of a challenge for SVP models. Although the implementation itself is relatively straightforward, the construction also needs model requirement identification, tuning, and validation, which would put further strain on developers. The optimisation requirements of SVP model setup are particularly important, as we observed the prediction performance to vary heavily across different classifier algorithms and whilst tuning.

5.2 Implications

Our findings have produced implications for both developers and researchers, based on the discovered limitations of each approach.

5.2.1 Developers.

For developers, we provide insights into the comparative performance and use of the investigated SAST approaches. Additionally, we propose some preliminary recommendations for the use of these approaches based on our findings.

We suggest that these two approaches for source code security analysis should be used separately, at least until better approaches of unification are developed. Tool integration is expensive [7, 23], and hence most organisations will desire to only adopt a singular approach. From *Finding 3 & 12*, we find that some of the detection capabilities overlap and hence one approach is sufficient to replace the other. As we have identified in *Finding 8*, SAST tools are a good initial option for timely software quality assurance, but SVP models have shown better performance as a project matures.

If SAST tools are used, they should be applied to a reduced range of code, rather than to all files iteratively. In *Finding 2 & 5*, we have identified that SAST tools incur heavy inspection costs due to their high false positive rates. Furthermore, alongside *Finding 6*, we find that SAST tools struggle to discern code or security context themselves. Hence, to reduce the subsequent inspection efforts, developers should only use SAST tools on areas of code that they are willing to manually inspect or test, and should not apply these tools to code they confidently consider secure.

SVP models should be regularly retrained and updated. From *Finding 8*, we observe that SVP models can be adopted relatively early. On the OpenSSL dataset, we found that even when using just 110 files for training (35 labelled as vulnerable), the model still achieved >0.7 MCC. However, we note that SVP model performance increasingly improves as more data is used to train the model, and SV assessment is also constrained to the training data (*Finding 11*); hence, models should be regularly updated.

5.2.2 Researchers. For researchers, we have identified some promising research directions based on the several pain points we have identified for each of the compared approaches.

For SAST tools, the main pain point is their performance. Whilst their capabilities for detection are decent (*Finding 1*), they produce a large number of false positives (*Finding 2*). Naturally, researchers and tool-makers should continue to develop better performing methods. We observe that SAST tools exhibit a trade-off between precision and recall (*Finding 4*), so overcoming this hurdle is a must.

For SVP models, the main pain points come from the required knowledge and experience of using them, rather than performance. SVP models require more transparency (*Finding 7*); developers need to be able to understand the capabilities and limitations of the software quality assurance approaches they use. Additionally, local prediction interpretation will assist developers in the inspection of the predicted modules. Many approaches for interpretable machine learning exist [32], but they have so far been under-explored in the context of SVP. More research needs to be conducted in this area.

Additionally, SVP models need better approaches for localization. Although SVP models produce a lower number of files to inspect than SAST tools (*Finding 5*), they still have high inspection costs due to the lack of localization. Models have been developed that predict vulnerable components at finer levels of granularity, through code slices [30] or commits [42]. However, these techniques often require the enclosing context or scope of a vulnerability [47], and hence are not perfect solutions. Recent techniques have been proposed for fault localization of defect prediction models [44]. Researchers should continue to advance these approaches and identify how the approaches transfer to SVP.

Finally, more focus needs to be put on the assessment task. SV assessment is an important task for SV mitigation, but it is not considered as a main priority for either of the approaches, hence, the performance suffers (*Finding 9*). More focus needs to be put into this task to achieve the performance that is more consistent with that of SV detection. Another important component of SV assessment is prioritization; assessing the risk of each SV. Although SAST tools often produce an indicative ranking of their warnings, these are not based on the actual severity or exploitability of SVs. For SVP models, little work has been conducted on using learning-based approaches to predict SV risk at the source code level.

6 THREATS TO VALIDITY

External Validity. We only investigated 3 open source projects and SARD test cases for vulnerabilities of the C/C++ programming language. We acknowledge that our findings may not generalize to other projects or programming languages.

Internal Validity. Our tuning of SVP models and configuration of SAST tools are potentially sub-optimal. To lessen this threat, we tuned a wide range of hyperparameters for our SVP models to optimize them in relation to our dataset. For the SAST tools, we manually analyzed the configuration options and initial outputs to determine the best configuration.

Construct Validity. The range of tools and models we considered is imperfect. We only selected open source SAST tools, as these are the most readily available and widely used. Commercial tools which use more sophisticated techniques will likely produce different performances [1]. Concurrently, the three SVP models we built were also relatively basic. However, selecting simplistic base approaches gives us a more general view for comparison, as they are the most common representation. Our open source datasets only contained documented post-release vulnerabilities from NVD. This largely conceals vulnerabilities already detected or removed during the implementation phase from our analysis. Similarly, we have found that all three of our open source projects have existing documentation of SAST tool usage, including *Cppcheck*, but the usage of these tools is not consistent or thorough[4]. To help overcome this limitation, we also evaluated the compared approaches on the Juliet Test Suite [3], to obtain a more complete view of development vulnerabilities. Our evaluation of SAST tools and SVP approaches included qualitative assessment and manual analysis. Such investigation has the potential of being impacted by subjectivity and human bias. To ensure more reliable human evaluation, we used multiple assessors in this study.

Conclusion Validity. To help strengthen conclusion validity, we confirmed our results using non-parametric statistical tests [63], and did inspection on statistically significant sample sizes [8].

7 CONCLUSION

This study has conducted the first large-scale comparative analysis of rule-based (SAST tools) and learning-based (SVP models) approaches to source code security analysis. Through this analysis, we have identified their comparative capabilities and uses, which we present through 12 main findings. From the findings of this study, we have also derived several implications for both researchers and practitioners to support the selection and use of SAST approaches, as well as direct future research efforts in this area. We conclude that SAST tools and SVP models provide similar detection capabilities, but SVP models provide better overall performance for SV detection and assessment. However, SVP models exhibit some caveats that SAST tools do not, such as data requirements, coarse granularity, and difficult interpretation.

In the future, we aim to conduct a user-survey to see how developers consider the comparative trade-offs that we have identified.

ACKNOWLEDGMENTS

This work has been supported by the Cyber Security Cooperative Research Centre Limited whose activities are partially funded by the Australian Government's Cooperative Research Centre Programme.

[4] https://github.com/openssl/openssl/issues/5013

REFERENCES

[1] Bushra Aloraini, Meiyappan Nagappan, Daniel M German, Shinpei Hayashi, and Yoshiki Higo. 2019. An empirical study of security warnings from static application security testing tools. *Journal of Systems and Software* 158 (2019), 110427.

[2] 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.

[3] Tim Boland and Paul E Black. 2012. Juliet 1.1 C/C++ and Java test suite. *IEEE Computer Architecture Letters* 45, 10 (2012), 88–90.

[4] CERN. [n.d.]. Rough Auditing Tool for Security (RATS). https://security.web.cern.ch/recommendations/en/codetools/rats.shtml

[5] Davide Chicco and Giuseppe Jurman. 2020. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC genomics* 21, 1 (2020), 1–13.

[6] Istehad Chowdhury and Mohammad Zulkernine. 2011. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture* 57, 3 (2011), 294–313.

[7] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 332–343.

[8] William G Cochran. 2007. *Sampling techniques*. John Wiley & Sons.

[9] Rory Coulter, Qing-Long Han, Lei Pan, Jun Zhang, and Yang Xiang. 2020. Code analysis for intelligent cyber systems: A data-driven approach. *Information sciences* 524 (2020), 46–58.

[10] Roland Croft, Dominic Newlands, Ziyu Chen, and Ali Babar. 2021. Reproduction package for "An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing". https://doi.org/10.6084/m9.figshare.14585076.v1

[11] CWE. [n.d.]. Common Weakness Enumeration. https://cwe.mitre.org/

[12] Gabriel Díaz and Juan Ramón Bermejo. 2013. Static analysis of source code security: Assessment of tools against SAMATE tests. *Information and software technology* 55, 8 (2013), 1462–1476.

[13] Davide Falessi, Jacky Huang, Likhita Narayana, Jennifer Fong Thai, and Burak Turhan. 2020. On the need of preserving order of data when validating within-project defect classifiers. *Empirical Software Engineering* 25, 6 (2020), 4805–4830.

[14] Yuanrui Fan, D Alencar da Costa, D Lo, AE Hassan, and L Shanping. 2020. The impact of mislabeled changes by szz on just-in-time defect prediction. *IEEE Transactions on Software Engineering* (2020).

[15] OWASP Foundation. [n.d.]. Static Code Analysis. https://owasp.org/www-community/controls/Static_Code_Analysis

[16] Michael Gegick and Laurie Williams. 2007. Toward the use of automated static analysis alerts for early identification of vulnerability-and attack-prone components. In *Second International Conference on Internet Monitoring and Protection (ICIMP 2007)*. IEEE, 18–18.

[17] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 1–36.

[18] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. 2017. A large-scale study of the impact of feature selection techniques on defect classification models. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 146–157.

[19] Hazim Hanif, Mohd Hairul Nizam Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. 2021. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *Journal of Network and Computer Applications* (2021), 103009.

[20] Nasif Imtiaz, Akond Rahman, Effat Farhana, and Laurie Williams. 2019. Challenges with responding to static analysis tool alerts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 245–249.

[21] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.

[22] Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. 2018. Enabling the continous analysis of security vulnerabilities with vuldata7. In *IEEE International Working Conference on Source Code Analysis and Manipulation*.

[23] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[24] Arvinder Kaur and Ruchikaa Nayyar. 2020. A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. *Procedia Computer Science* 171 (2020), 2023–2029.

[25] Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1/2 (1938), 81–93.

[26] Saad Khan and Simon Parkinson. 2018. Review into state of the art of vulnerability assessment using artificial intelligence. In *Guide to Vulnerability Analysis for Computer Networks and Systems*. Springer, 3–32.

[27] Gary A Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 194–206.

[28] Triet Huynh Minh Le, David Hin, Roland Croft, and M Ali Babar. 2020. PUMiner: Mining Security Posts from Developer Question and Answer Websites with PU Learning. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 350–361.

[29] Triet Le Le Huynh Minh, Roland Croft, David Hin, and Muhammad Ali Ali Babar. 2021. A Large-scale Study of Security Vulnerability Support on Developer Q&A Websites. In *Evaluation and Assessment in Software Engineering*. 109–118.

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Symposium*.

[31] Daniel Marjamaki. [n.d.]. Cppcheck. http://cppcheck.sourceforge.net/

[32] Jian-Xun Mi, An-Di Li, and Li-Fang Zhou. 2020. Review Study of Interpretation Methods for Future Interpretable Machine Learning. *IEEE Access* 8 (2020), 191969–191985.

[33] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. 2015. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*. 1–9.

[34] Patrick J Morrison, Rahul Pandita, Xusheng Xiao, Ram Chillarege, and Laurie Williams. 2018. Are vulnerabilities discovered and resolved like other defects? *Empirical Software Engineering* 23, 3 (2018), 1383–1421.

[35] Nuthan Munaiah and Andrew Meneely. 2019. Data-driven insights from vulnerability discovery metrics. In *2019 IEEE/ACM Joint 4th International Workshop on Rapid Continuous Software Engineering and 1st International Workshop on Data-Driven Decisions, Experimentation and Evolution (RCoSE/DDrEE)*. IEEE, 1–7.

[36] Zaigham Mushtaq, Ghulam Rasool, and Balawal Shehzad. 2017. Multilingual source code analysis: A systematic literature review. *IEEE Access* 5 (2017), 11307–11336.

[37] National Institute of Standards and Technology. [n.d.]. Software Assurance and Reference Dataset. https://samate.nist.gov/SARD/testsuite.php

[38] National Institute of Standards and Technology. [n.d.]. Source Code Security Analyzers. https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

[39] Tosin Daniel Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela Soares Cruzes. 2018. Myths and facts about static application security testing tools: an action research at telenor digital. In *International Conference on Agile Software Development*. Springer, Cham, 86–103.

[40] Rajshakhar Paul, Asif Kamal Turzo, and Amiangshu Bosu. 2021. Why Security Defects Go Unnoticed during Code Reviews? A Case-Control Study of the Chromium OS Project. In *2021 43rd International Conference on Software Engineering (ICSE)*. IEEE.

[41] Jose D'Abruzzo Pereira, João R Campos, and Marco Vieira. 2019. An exploratory study on machine learning to combine security vulnerability alerts from static analysis tools. In *2019 9th Latin-American Symposium on Dependable Computing (LADC)*. IEEE, 1–10.

[42] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 426–437.

[43] Nico Poel. 2010. *Automated Security Review of PHP Web Applications with Static Code Analysis*. Master's thesis. University of Groningen.

[44] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2021. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. *arXiv preprint arXiv:2103.07068* (2021).

[45] Foyzur Rahman, Sameer Khatri, Earl T Barr, and Premkumar Devanbu. 2014. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*. 424–434.

[46] Athos Ribeiro, Paulo Meirelles, Nelson Lago, and Fabio Kon. 2019. Ranking warnings from multiple source code static analyzers via ensemble learning. In *Proceedings of the 15th International Symposium on Open Collaboration*. 1–10.

[47] Emre Sahal and Ayse Tosun. 2018. Identifying bug-inducing changes for code additions. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–2.

[48] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering* 40, 10 (2014), 993–1006.

[49] Robert C Seacord. 2005. *Secure Coding in C and C++*. Pearson Education.

[50] Hossain Shahriar and Mohammad Zulkernine. 2012. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 1–46.

[51] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. 2010. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering* 37, 6 (2010), 772–787.

[52] Yonghee Shin and Laurie Williams. 2013. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering* 18, 1 (2013), 25–59.

[53] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2018. How developers diagnose potential security vulnerabilities with a static analysis tool. *IEEE Transactions on Software Engineering* 45, 9 (2018), 877–897.

[54] Vincent Smyth. 2017. Software vulnerability management: how intelligence helps reduce the risk. *Network Security* 2017, 3 (2017), 10–12.

[55] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1200–1219.

[56] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John Grundy. 2020. Explainable AI for Software Engineering. *arXiv preprint arXiv:2012.01614* (2020).

[57] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2016), 1–18.

[58] TIOBE. [n.d.]. TIOBE Index. https://www.tiobe.com/tiobe-index/

[59] John Viega, JT Bloch, Tadayoshi Kohno, and Gary McGraw. 2002. Token-based scanning of source code for security problems. *ACM Transactions on Information and System Security (TISSEC)* 5, 3 (2002), 238–261.

[60] Andreas Wagner and Johannes Sametinger. 2014. Using the Juliet test suite to compare static security scanners. In *2014 11th International Conference on Security and Cryptography (SECRYPT)*. IEEE, 1–9.

[61] James Walden, Jeff Stuckman, and Riccardo Scandariato. 2014. Predicting vulnerable components: Software metrics vs text mining. In *2014 IEEE 25th international symposium on software reliability engineering*. IEEE, 23–33.

[62] David Wheeler. [n.d.]. Flawfinder. https://dwheeler.com/flawfinder/

[63] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[64] Yichen Xie, Andy Chou, and Dawson Engler. 2003. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. 327–336.

[65] Yanming Yang, Xin Xia, David Lo, Tingting Bi, John Grundy, and Xiaohu Yang. 2020. Predictive Models in Software Engineering: Challenges and Opportunities. *arXiv preprint arXiv:2008.03656* (2020).

[66] Jongwon Yoon, Minsik Jin, and Yungbum Jung. 2014. Reducing false alarms from an industrial-strength static analyzer by SVM. In *2014 21st Asia-Pacific Software Engineering Conference*, Vol. 2. IEEE, 3–6.

[67] Peng Zeng, Guanjun Lin, Lei Pan, Yonghang Tai, and Jun Zhang. 2020. Software Vulnerability Analysis and Discovery using Deep Learning Techniques: A Survey. *IEEE Access* (2020).

[68] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. 2010. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 421–428.