Benchmarking Machine Learning Approaches for ICFG-enhanced SAST

Deric Alvarez, Alec Cheng, Gabriel Hervias, Dongfeng Fang
College of Engineering
California Polytechnic State University
San Luis Obispo, USA
dkalvare@calpoly.edu, acheng36@calpoly.edu, ghervias@calpoly.edu, dofang@calpoly.edu

***Abstract*—In this paper, we benchmark and compare various machine learning algorithms used in Static Application Security Testing (SAST). SAST involves utilizing static analysis techniques on program code to identify security-critical software defects early in the development process without execution. Traditionally, this has been accomplished using rule-based approaches, where the code is evaluated against predefined patterns. Recently, however, there has been a growing body of research exploring the application of machine learning methods to enhance the effectiveness of SAST. Our approach for benchmarking and comparison relies on a novel interprocedural control flow graph-based embedding method for program code. Through this approach, we apply different unsupervised machine learning approaches and assess their performance in detecting vulnerabilities. We evaluate these methods based on several criteria, such as detection accuracy, false positive rates, and computational efficiency. Our findings indicate that machine learning-based approaches can significantly improve the accuracy and efficiency of SAST, offering promising directions for future research and practical applications in software security.**

I. Introduction

In a rapidly evolving landscape of software development, ensuring the security and reliability of applications has become increasingly important. One critical aspect of this endeavor is the detection and mitigation of vulnerabilities within code bases. Static Application Security Testing (SAST) has emerged as a crucial methodology in this context. SAST involves analyzing the source code, byte code, or binary code of applications without execution, aiming to identify security vulnerabilities early in the development lifecycle [1]. SAST tools have traditionally applied predefined analysis rules that model different threats [2]. This approach allows developers to address potential security concerns and risks promptly, reducing the risk of exploitations in production environments.

Despite its advantages, SAST faces significant challenges in effectively detecting complex security vulnerabilities. Traditional SAST tools often struggle with analyzing the intricate flow of data and control across different parts of complex programs. This limitation hampers their ability to identify vulnerabilities that arise from the interaction between various components, such as cross-site scripting (XSS), SQL injection, or buffer overflows that may not be confined to a single function. Consequently, there is a pressing need for more sophisticated analysis techniques that can provide a comprehensive understanding of the code's behavior across different functions and modules.

Special thanks to Mihai for managing the server that we ran our experiments on.

To address this challenge, an increasing trend has been observed where researchers apply machine learning techniques to the problem of SAST [3,4]. Hence, the usage of SAST no longer requires rules to be defined explicitly; instead, statistical methods can be used to separate benign from defective code. A majority of studies and research papers that have implemented machine learning for SAST take a supervised learning approach. This approach, however, has faced additional challenges due to the supervised training needing labeled data, dramatically reducing the amount of usable data sets. Additionally, these studies have shown a high rate of false correlations [5].

An alternative approach researchers have turned to involves the construction and utilization of Interprocedural Control Flow Graphs (ICFGs) along with unsupervised learning techniques [6]. An ICFG is a powerful tool that represents the control flow of an entire program, encompassing multiple procedures and their interactions. By incorporating information about function calls and returns, ICFGs enable a more thorough analysis of the program's execution paths. ICFGs provide a detailed map of how different parts of a program interact, facilitating the identification of security issues that might otherwise be missed by conventional SAST techniques. For instance, by analyzing an ICFG, a SAST tool can detect a vulnerability caused by a specific sequence of function calls or the improper handling of data as it passes through multiple layers of the application. This holistic view allows SAST tools to trace the flow of data and control across different functions, significantly enhancing their ability to detect complex vulnerabilities.

Our research project aimed to further explore the use of ICFG-enhanced SAST as presented in "Machine Learning for SAST: A Lightweight and Adaptable Approach" [7]. There are currently very few comprehensive benchmark papers in this area with common criteria. The goal of our research is to examine the limitations of their process, improve its vulnerability detection efficiency, and benchmark various machine learning algorithms using established metrics. Specifically, we aimed to address the limitation of their use of k-means clustering and expand the testing to cover more than the 20 out of 114 vulnerability types from the Juliet test suite that the original paper tested.

The rest of the paper is structured as follows: In Section 2, we describe the methodology used to implement the various machine learning models, as well as the embedding method employed to create the ICFGs, providing a comprehensive explanation of the techniques and algorithms utilized in our research. Section 3 covers the ethical considerations relevant to our research, including non-maleficence and contribution to the public good. Section 4 presents the results of our experiments, including a detailed analysis of the findings. It also discusses the challenges faced during the research project, offering insights into the practical aspects of implementing ICFG-enhanced SAST with machine learning. In Section 5, we outline the various limitations of the paper and suggest potential areas for future improvement and research. The final section contains the conclusion of the paper, summarizing our findings and their implications.

II. Methodology

Our aim in implementing this neural network model is to measure its performance for SAST applications. We have 2 main functional requirements for this model.

1) Receive embedded ICFG paths as input.
2) Using a recurrent neural network (RNN), generate 2 outputs: a binary classification of a path as ”vulnerable” or ”safe” and if the path is vulnerable, a classification of the path as being of a certain type of Common Weakness Enumeration (CWE).

A. ICFG Embeddings

Using an ICFG to represent the potential vulnerable source code is a method described in [6]. The choice of using this method build our model upon was based on the presumption that an ICFG has the advantage of maintaing more of the semantic value of the source code. Our embedding method is the one implemented in [6]. Our prototype uses the embedding module, which is the ”SVF” module in the pipeline as seen in Figure 1. The entirety of our prototype is a customization of the model ”ML-Based Analysis,” also seen in Figure 1.

B. Dataset

The training of a neural network requires a substantial set of data to train its parameters on. As done in [6], we use the Juliet Test Suite [7], which is a dataset of code samples demonstrating 118 Common Weakness Enumerations (CWE). This data we use from the Juliet Test Suite has been preprocessed by the authors of [6] and made available to us through their project GitHub repository. Because of time constraints for our project, we were unable to gather data for more test cases than was made available through the GitHub repository made for [6]. The preprocessing involved splitting source code files to individual procedures, labelling code samples as vulnerable or safe, and labelling code with

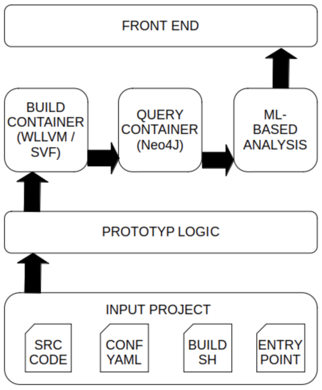
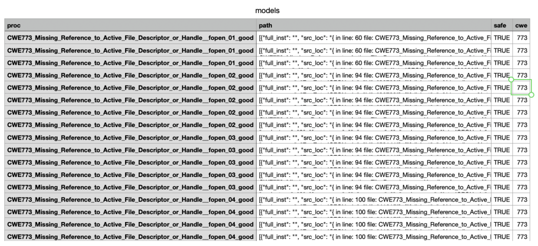


Fig. 1. ML-SAST system architecture


models

proc	path	safe	cwe
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_01_good	[{"full_inst": "", "src_loc": "{ in line: 60 file: CWE773_Missing_Reference_to_Active_Fi	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_01_good	[{"full_inst": "", "src_loc": "{ in line: 60 file: CWE773_Missing_Reference_to_Active_Fi	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_01_good	[{"full_inst": "", "src_loc": "{ in line: 60 file: CWE773_Missing_Reference_to_Active_Fi	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_02_good	[{"full_inst": "", "src_loc": "{ in line: 94 file: CWE773_Missing_Reference_to_Active_Fi	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_02_good	[{"full_inst": "", "src_loc": "{ in line: 94 file: CWE773_Missing_Reference_to_Active_Fi	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_02_good	[{"full_inst": "", "src_loc": "{ in line: 94 file: CWE773_Missing_Reference_to_Active_Fi	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_02_good	[{"full_inst": "", "src_loc": "{ in line: 94 file: CWE773_Missing_Reference_to_Active_Fi	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_02_good	[{"full_inst": "", "src_loc": "{ in line: 94 file: CWE773_Missing_Reference_to_Active_Fi	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_03_good	[{"full_inst": "", "src_loc": "{ in line: 94 file: CWE773_Missing_Reference_to_Active_Fi	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_03_good	[{"full_inst": "", "src_loc": "{ in line: 94 file: CWE773_Missing_Reference_to_Active_Fi	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_03_good	[{"full_inst": "", "src_loc": "{ in line: 94 file: CWE773_Missing_Reference_to_Active_Fi	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_03_good	[{"full_inst": "", "src_loc": "{ in line: 94 file: CWE773_Missing_Reference_to_Active_Fi	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_03_good	[{"full_inst": "", "src_loc": "{ in line: 94 file: CWE773_Missing_Reference_to_Active_Fi	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_04_good	[{"full_inst": "", "src_loc": "{ in line: 100 file: CWE773_Missing_Reference_to_Active_I	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_04_good	[{"full_inst": "", "src_loc": "{ in line: 100 file: CWE773_Missing_Reference_to_Active_I	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_04_good	[{"full_inst": "", "src_loc": "{ in line: 100 file: CWE773_Missing_Reference_to_Active_I	TRUE	773
CWE773_Missing_Reference_to_Active_File_Descriptor_or_Handle__fopen_04_good	[{"full_inst": "", "src_loc": "{ in line: 100 file: CWE773_Missing_Reference_to_Active_I	TRUE	773

Fig. 2. Preprocessed data from Juliet Test Suite

its CWE. This process was likely done with a script we did not have access to.

C. Neural Network Design

Because we would like our neural network to perform two distinct but related tasks, we saw an opportunity to use multi-output learning, which is a process that simultaneously predict multiple outputs given an input [8]. As seen in Figure 3, our neural network is made of 5 hidden layers and 2 output layers.

The neural network model we designed for this prototype went through many iterations of experimentation of hyper-parameters. These parameters include the order and number of hidden layers, the number of neurons within a layer, the learning rate, dropout values, the number of epochs, and the batch size.

1) SimpleRNN layer: The first hidden layer is a fully connected RNN layer [9]. This was chosen for its usefulness with simple sequential data. RNNs have the ability to model input and/or output consisting of sequences of elements that are not independent [10]. With our embedding method, a time step is node representing a specific instruction within a code path.

2) Dense layers: Two of the hidden layers following the RNN layer are Dense, or fully-connected layers. The goal with these layers is to capture additional complexity and transition the data towards the output shape. To combat overfitting and increase performance [11], there is a dropout layer between the two dense layers.

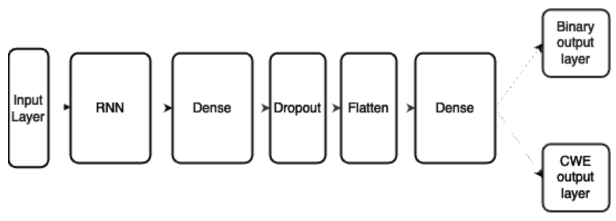


Fig. 3. Neural network diagram

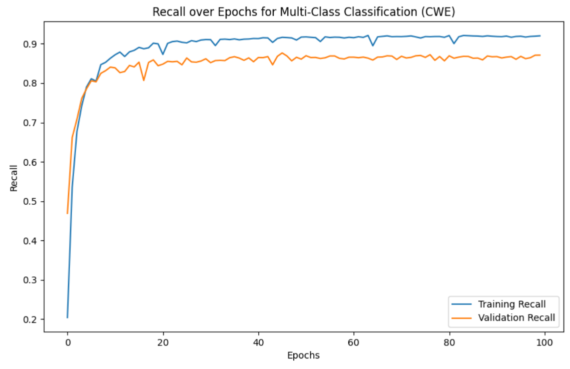


Fig. 4. Recall over Epochs for Multi-Class Classification (CWE)

D. Training

Our neural network is training on a subset of the Juliet Test Suite using a 60-20-20 split. This means that 60% of the dataset is for training, 20% is for testing, and 20% is for validation. This common split was chosen to provide more samples for the training of the model.

Figure 4 shows the recall over epochs for multi-class classification for 100 epochs. Around 25 epochs is where the increase of recall starts to flatten out, so we chose 25 epochs as our limit. Figure 5 shows the same graph for only 25 epochs, while Figure 6 shows the recall over epochs for binary classification for 25 epochs. As seen in the graph, the recall for binary classification reached its limit very early, while multi-class classification takes longer.

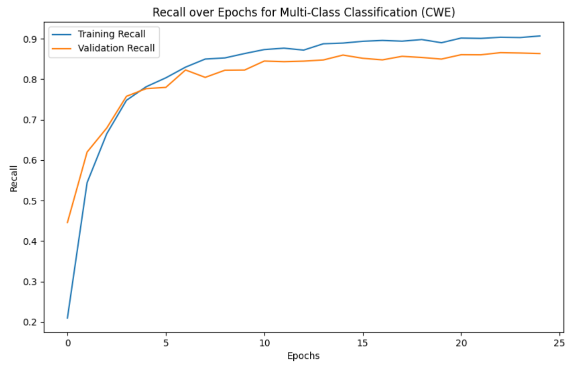


Fig. 5. Recall over Epochs for Multi-Class Classification (CWE)

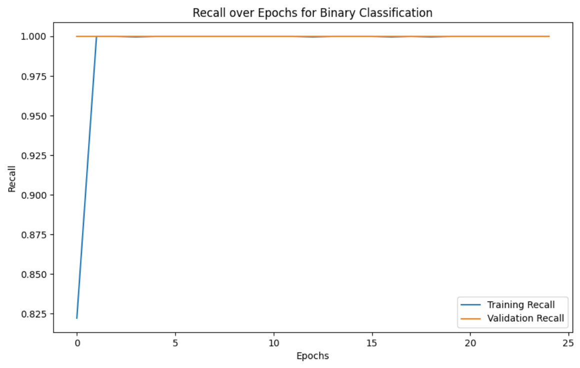


Fig. 6. Recall over Epochs for Binary Classification

E. Threat Model

In the context of software security, several key assests must be protected to ensure the integrity and functionality of the system.

Assets:

- Source code: The actual code being analyzed for vulnerabilities.
- Machine Learning models: These are the trained models used to detect vulnerabilities in the code
- ICFG Data: The ICFG that map the control flow of the application across different procedures.

Threat Agents:

- Malicious Developers: Insiders who might deliberately introduce vulnerabilities
- External Attackers: Hackers who try to exploit weaknesses from outside.
- Compromised System: Systems that have been breached and can be used to further attacks.
- Insider Threats: Authorized users who misuse their access rights.

Threats and Vulnerabilities:

- Path Transversal and command injection: These vulnerabilities occur when an attacker manipulates input to access unauthorized paths or execute arbitrary commands on the host operating system.
- Buffer Errors: Buffer errors involve reading from or writing to a buffer (an area of memory) in a way that exceeds its boundaries, leading to crashes, data corruption, or code execution.
- Format String Vulnerabilities: These vulnerabilities occur when user input is improperly handled in format string functions, leading to crashes or code execution.
- Integer and numeric Errors: These vulnerabilities arise from incorrect handling of numeric values, leading to unexpected behavior or security issues.

- Memory Management Errors: These vulnerabilities involve improper handling of memory allocation, deallocation, and usage, leading to crashes, leaks, or code execution.

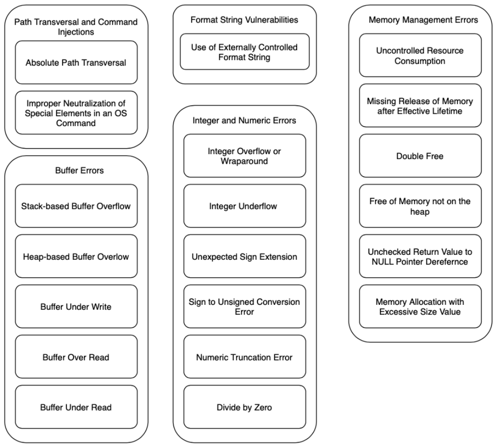

Path Transversal and Command Injections
Absolute Path Transversal
Improper Neutralization of Special Elements in an OS Command
Buffer Errors
Stack-based Buffer Overflow
Heap-based Buffer Overlow
Buffer Under Write
Buffer Over Read
Buffer Under Read
Format String Vulnerabilities
Use of Externally Controlled Format String
Integer and Numeric Errors
Integer Overflow or Wraparound
Integer Underflow
Unexpected Sign Extension
Sign to Unsigned Conversion Error
Numeric Truncation Error
Divide by Zero
Memory Management Errors
Uncontrolled Resource Consumption
Missing Release of Memory after Effective Lifetime
Double Free
Free of Memory not on the heap
Unchecked Return Value to NULL Pointer Derefernce
Memory Allocation with Excessive Size Value

Fig. 7. Tested Vulnerabilities

III. Ethics Statement

In conducting and presenting this research, we are committed to upholding the highest standards of ethical conduct. This ethics statement outlines our adherence to ethical principles throughout the research process.

Non-Maleficence

Our research is designed with the intent of improving security in Static Application Security Testing. We avoid any implementation or demonstration of actual attacks on real systems. Instead, our focus is on benchmarking several different machine learning approaches on prelabeled vulnerabilities included in the Juliet dataset suite.

Contribution to the Public Good

The ultimate goal of our research is to contribute positively to the field of cybersecurity. By benchmarking machine learning approaches for ICFG-enhanced SAST, we aim to advance the understanding and effectiveness of security testing tools, thereby helping to create more secure software systems for society.

By adhering to these ethical principles, we strive to conduct research that is not only scientifically rigorous but also ethically sound, ensuring that our contributions to the field of machine learning and cybersecurity are responsible and beneficial to the wider community.

IV. Results, Analysis, and Challenges

A. Metrics

To compare the RNN with the k-Means model employed in [6], we documented its performance in terms of recall, precision, F1-score, and accuracy. Notably, while recall is an essential metric, the latter three - precision, F1-score, and accuracy - are frequently utilized to assess the performance of machine learning models in Static Application Security Testing SAST, as highlighted in [5].

We identified the most valuable metric to be recall, or the true positive rate. In our context, recall is effectively the measure of how well our model minimizes false negatives. For binary classification, false negative would be a sample of vulnerable code that is incorrectly classified as safe. For multi-class classification, a false negative is the labeling of a vulnerable code sample as the wrong type of CWE. Both of these instances could be harmful in a SAST environment as they may lead to vulnerabilities being pushed to production or handled incorrectly.

B. Results and Analysis

The following tables contain the recorded metrics of the RNN and the k-Means of [6], for the same CWEs. It should be noted that problems with replicating the training dataset used in [6] resulted in only a subset being used to train our model, which may mean a misrepresentation of the results.

TABLE I
RNN Results

Test Suite	Recall	Precision	F1-Score	Accuracy
CWE-78	0.99	0.99	0.995	0.99
CWE-121	0.994	0.994	0.997	0.994
CWE-122	0.9	0.9	0.947	0.9
CWE-126	0.927	0.927	0.962	0.927
CWE-134	0.98	0.98	0.99	0.98
CWE-190	0.577	0.577	0.732	0.577
CWE-191	0.661	0.661	0.835	0.661
CWE-194	0.95	0.95	0.97	0.95
CWE-195	0.917	0.917	0.957	0.917
CWE-197	0.748	0.748	0.856	0.748
CWE-369	0.629	0.629	0.773	0.629
CWE-400	0.787	0.787	0.881	0.787
CWE-401	0.909	0.909	0.952	0.909
CWE-690	0.919	0.919	0.958	0.919

TABLE II
k-Means Results from [6]

Test Suite	Recall	Precision	F1-Score	Accuracy
CWE-78	0.26	0.912	0.405	0.272
CWE-121	0.393	0.778	0.522	0.483
CWE-122	0.345	0.789	0.48	0.426
CWE-126	0.43	0.86	0.573	0.468
CWE-134	0.468	0.979	0.633	0.483
CWE-190	0.741	0.965	0.838	0.752
CWE-191	0.736	0.965	0.835	0.748
CWE-194	0.332	0.602	0.428	0.402
CWE-195	0.375	0.604	0.463	0.39
CWE-197	0.236	0.654	0.347	0.292
CWE-369	0.781	1.0	0.877	0.783
CWE-400	0.632	0.978	0.768	0.646
CWE-401	0.953	0.994	0.973	0.955
CWE-690	0.855	0.905	0.879	0.895

A comparison of these results shows that of the four metrics recorded, the RNN scored a higher average on recall, F1 score, and accuracy.

For recall, the RNN demonstrated an average increase of 0.311, meaning it was less likely to miss a vulnerability and its class. However, it's worth noting that there were decreases observed in four CWEs, which might suggest areas for targeted refinement.

The F1 score showed an average increase of 0.27, suggesting that the RNN enhanced vulnerability detection without significantly compromising the ability to distinguish safe and unsafe code. This metric saw only a decrease in 3 CWEs.

Accuracy increased by an average of 0.278 for the RNN, indicating better performance in correctly classifying instances. Similar to recall, there were decreases across 4 CWEs, again underscoring specific vulnerabilities that might benefit from further attention.

Regarding the only metric with an average decrease, precision, the RNN had an average decrease of 0.006. This decrease was accompanied by a decline in the performance related to 6 CWEs, highlighting potential challenges in distinguishing between vulnerable and secure code.

Considering the significant improvements in recall, F1 score, and accuracy, RNN is indicated to be a superior choice compared to k-Means for SAST. The increase in recall is particularly noteworthy given our previous discussion on metrics. Although there was a marginal decrease in precision, the overall gains in the other metrics outweigh this drawback. Again, due to the difference in datasets used for the RNN and k-Means algorithm, these conclusions should be taken with reservation.

C. Challenges

Throughout our experiment and data collection process, we faced several challenges that took away much of our limited time.

- Requesting and maintaining a server: Due to the setup of the container, the code from [6] required an x86 architecture that wasn't available to most of our team. To ensure everyone had access to the code, we had to request a server from our university. Unfortunately, the first few weeks of using the server was met with continous maintenance issues that were finally resolved later in the timeframe.
- Fixing and updating dependencies: Although the server had the correct architecture, some dependencies still had to be fixed and updated for the base code to run.
- Refactoring the code: To implement an RNN, we had to edit some of the original code so that the data was fed into the RNN instead of the k-Means algorithm.
- Correctly measuring metrics: Given the size of the dataset, it was difficult to verify the metrics other than recalculating them.

V. Conclusion

A. Limitations and Threats to Validity

We have identified a number of limitations within our methodology that are potential areas for future research.

1) Different training splits: Our project did not include the testing of any other training splits besides 60-20-20. Other splits could potentially produce a better-trained model.

2) Tenfold cross-validation: Tenfold cross-validation is a technique used to provide a more robust estimate of model performance. Our project does not utilize this technique.

3) Dataset diversity: Because of our inability to preprocess new code samples, our trained model is limited to a small subset of 14 CWEs consisting of 22,664 code paths.

4) Benchmark paper mismatch: The dataset provided by the authors of [6], the paper which we are benchmarking against, does not include all of the CWEs the model in [6] is trained on, only 14 out of 20. Thus, are benchmarking results are mistmatched.

B. Future Works

Several promising avenues for future research emerge from the limitations identified in our current study.

1) Exploration of different training splits: Future investigations could explore alternative splits, such as 70-15-15 or 80-10-10, to determine if they yield more effective model training and performance. This exploration could provide insights into optimal data allocation for different phases of model development.

2) Implementation of tenfold cross-validation: To enhance the reliability and generalizability of our model's performance, integrating tenfold cross-validation into our methodology is a worthwhile endeavor. This technique would involve dividing the dataset into ten equal subsets and iteratively using each subset for validation while training on the remaining nine. This approach can help in mitigating the variance in model evaluation and provide a more accurate assessment of its performance.

3) Increasing dataset diversity: Future work should focus including more of the Juliet Test Suite than just the small dataset we used. This expansion would enable the training of a more versatile and effective model capable of identifying a wider array of software vulnerabilities.

4) Alignment of benchmark datasets: A significant complication in our study is the mismatch between the dataset provided by the authors of [6] and the CWEs their model was originally trained on. To address this, future research may create their own preprocessing script to encompass all the CWEs from the original paper, or test on their own dataset for all models.

5) Recording time to run as a metric: Although the metrics we recorded were very important in measuring the performance between RNN and k-Means, we were not able to measure the time it took to run the different models. Time to run could play an important role in determining the best model for SAST, as time can be very valuable in situations where code has to be pushed to production in a short amount of time.

6) Expanding and improving selection of models: While we determined RNNs to be the most suited model to use, other approaches such as Density-Based Spatial Clustering of Applications with Noise (DBSCAN) and Long- Short-Term Memory (LSTM) may provide better results. Future work should look into these other algorithms along with optimizing the layers of the RNN.

By addressing these limitations and broadening the models of interest, we anticipate significant advancements in the capability and applicability of our approach to detecting software vulnerabilities.

C. Findings

Our comparative analysis of the RNN and k-Means algorithms for SAST revealed that the RNN outperformed k-Means in three out of four key metrics: recall, F1 score, and accuracy. Considering the significant improvements in recall, F1 score, and accuracy, the RNN appears to be a superior choice compared to k-Means for SAST. However, due to the difference in datasets used for the RNN and k-Means algorithms, these conclusions should be interpreted with caution.

References

[1] Ayewah, N., & Pugh, W. (2008, October 1). Using static analysis to find bugs. *IEEE Journals & Magazine* — IEEE Xplore. Retrieved from https://ieeexplore.ieee.org/document/4602670/citations#citations

[2] Croft, R., Newlands, D., Chen, Z., & Babar, M. A. (2021). An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing. *Association for Computer Machinery*. https://doi.org/10.1145/3475716.3475781

[3] Giray, G., Bennin, K.E., Köksal, Ö., Babur, Ö., Tekinerdogan, B. (2023). On the use of deep learning in software defect prediction. *Journal of Systems and Software*, 195, 111537. https://doi.org/10.1016/j.jss.2022.111537

[4] Hüther, L., et al. (2022). Machine learning in the context of static application security testing - ML-SAST. Technical report, Federal Office for Information Security, Federal Office for Information Security, P.O. Box 20 03 63, 53133 Bonn.

[5] Marjanov, T., Pashchenko, I., & Massacci, F. (2022). Machine Learning for Source Code Vulnerability Detection: What Works and What Isn't There Yet. *IEEE Security & Privacy*, 20(5), 60–76. https://doi.org/10.1109/msec.2022.3176058

[6] Hüther, L., Sohr, K., Berger, B. J., Rothe, H., & Edelkamp, S. (2024). Machine Learning for SAST: A lightweight and adaptable approach. In *Lecture notes in computer science* (pp. 85–104). https://doi.org/10.1007/978-3-031-51482-1_5

[7] Black, P. E. (2018). Juliet 1.3 test suite: changes from 1.2. In *Technical report, NIST TN 1995*, National Institute of Standards and Technology, Gaithersburg, MD. https://doi.org/10.6028/NIST.TN.1995

[8] Xu, D., Shi, Y., Tsang, I. W., Ong, Y.-S., Gong, C., & Shen, X. (2020). Survey on multi-output learning. *IEEE Transactions on Neural Networks and Learning Systems*, 31(7), 2409-2429. https://doi.org/10.1109/TNNLS.2019.2945133

[9] TensorFlow. (2023). *tf.keras.layers.SimpleRNN*. Retrieved from https://www.tensorflow.org/api_docs/python/tf/keras/layers/SimpleRNN

[10] Lipton, Z. C. (2015). A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019. Retrieved from http://arxiv.org/abs/1506.00019

[11] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56), 1929-1958. Retrieved from http://jmlr.org/papers/v15/srivastava14a.html