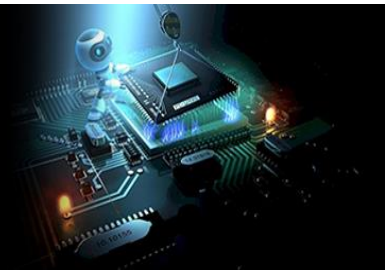
International Journal of Engineering in Computer Science



E-ISSN: 2663-3590
P-ISSN: 2663-3582
www.computersciencejournals.com/ijecs
IJECS 2024; 6(2): 120-124
Received: 16-06-2024
Accepted: 03-08-2024

V Arun Kumar
Assistant Professor, Malla Reddy Engineering College for Women (Autonomous Institution), Hyderabad, Telangana, India

P Keerthi
Student, Malla Reddy Engineering College for Women (Autonomous Institution), Hyderabad, Telangana, India

N Jyoshna
Student, Malla Reddy Engineering College for Women (Autonomous Institution), Hyderabad, Telangana, India

S Shreeja
Student, Malla Reddy Engineering College for Women (Autonomous Institution), Hyderabad, Telangana, India

Corresponding Author:
V Arun Kumar
Assistant Professor, Malla Reddy Engineering College for Women (Autonomous Institution), Hyderabad, Telangana, India

Software vulnerability detection tool using machine learning algorithms

V Arun Kumar, P Keerthi, N Jyoshna and S Shreeja

DOI: https://doi.org/10.33545/26633582.2024.v6.i2b.134

Abstract
There has been a lot of focus on exploitable software vulnerabilities recently because to the seriousness of the damage they may bring to data and computer security. Code inspection has been aided by several suggested vulnerability detection methods. One set of research has shown encouraging outcomes when using machine learning approaches to these strategies. With the goal of demonstrating how these 22 recent research use state-of-the-art neural approaches to identify potential problematic code patterns, this article covers deep learning as a vulnerability detection method. From the papers we looked at, we were able to pick out four that really changed the game when it came to using deep learning for vulnerability identification. We also gave you the lowdown on what these four studies had to say about the field as a whole. Reviewing the remaining studies in light of the four game-changers, we offer their methods and solutions, which either expand upon or build upon the game-changers, and we share our thoughts on the trends that will shape future research. We also talk about possible areas for future study and point out the difficulties encountered in this area. We want to inspire readers to delve more into this emerging yet rapidly expanding field of study.

Keywords: Software vulnerability, vulnerability detection, machine learning algorithms

Introduction
I therefore allow you, without compensation, the right to reproduce in whole or in part, either digitally or by hand, any portion of this work for educational or personal purposes, so long as you do not reproduce or distribute the copies for commercial gain and your copies include this notice and the whole citation on the first page. Parts of this work may belong to parties other than ACM, and their copyrights should be respected. Acknowledgment is required while abstracting. Prior explicit permission and/or payment may be required for any other kind of copying, republishing, posting on servers, or redistribution, including but not limited to lists. found by hostile adversaries and used for evil purposes. Attackers can cause a denial of service (DoS) when they crash a critical operating software. However, there are situations when the attacker can get more rights or even complete control of the system. Compilers and operating systems have evolved to include various safeguards against buffer overflow attacks, which have been used by malevolent hackers for many years. For instance, data execution prevention (DEP) renders the call stack non-executable, meaning that hackers can't run their payloads, and address space layout randomization (ASLR) makes it harder for hackers to insert correct addresses into their payloads by randomly arranging the process's address space [17]. These methods, however, have served only to annoy persistent enemies. Up until now, writing safe code has been the sole option for keeping hackers from carrying out an attack. But even with automatic and manual methods, it is difficult to scan complicated programs for defects, especially those written in a low-level language like C. Although Microsoft invests around 100 machine years annually into automated bug detection techniques [7], their products frequently have multiple bugs due to the complexity of pointer arithmetic and the developers' relentless focus on meeting deadlines. Security experts and developers must stay abreast of new automated vulnerability detection technologies since that is how attackers find program security vulnerabilities. An approach to identifying susceptible and non-vulnerable functions in C source code is presented in this study. Following our discovery of one hundred applications on GitHub, we extracted all of their functionality. Afterwards, we used these functions to extract both non-trivial characteristics (n-grams and suffix trees) and simple features (function length, nesting depth, string entropy,

etc.). Two tables, one for training data and one for test data, were created to include the feature statistics. The test samples were classified using a variety of classifiers, such as Naive Bayes, k-nearest neighbors, k-means, neural network, support vector machine, decision tree, and random forest. Out of all the classification methods tested, the one using trivial features had the highest accuracy (75%), followed by n-grams (69%), and finally, suffix trees (60%). More information on these findings is provided in Section 5. First, some basic ideas are covered in Section 2. Then, in Section 3, past work is reviewed. In Section 4, the testing technique is detailed. Finally, in Section 6, the results are presented.

Related Work

Investigating Interconnections in the Enron Email Database

For three reasons, including (a) being a massive collection of emails from a genuine company and (b) spanning three and a half years, the Enron email corpus is attractive to scholars. Our study in this article adds to the preliminary social network analytics examination of the Enron email dataset. As far as relational data and communication network extraction from the Enron corpus is concerned, we detail our efforts here. Using a variety of network analytic methods, we investigate the Enron networks' structural features and track down important actors across time. The network was denser, more centralized, and more linked during the Enron crisis than it is during normal times, according to our early data. Our data also shows that throughout the crisis, there was more diversified communication among Enron employees based on their official roles. However, the top executives established a close clique, supported each other, and interacted with the rest of the corporation through highly mediated relationships. Organizational crisis scenario modeling and failure indicator research may both benefit from the insights obtained via the analyses we conduct and suggest.

A verification of the efficacy of metrics for object-oriented design as quality markers

This article details the findings of an investigation into the object-oriented (OO) design metrics proposed in (Chidamber and Kemerer, 1994) and their practical application. In particular, we want to find out if these measures may be utilized as early quality indicators by evaluating them as predictors of classes that are prone to errors. Using the same set of criteria to evaluate the frequency of maintenance modifications to classes, this study supplements the work provided in (Li and Henry, 1993). For the purpose of our validation, we gathered information on the creation of eight information management systems for medium-sized businesses that met the same criteria. C++ and the famous OO analysis/design methodology were the tools of choice for all eight projects' development. The benefits and downsides of various OO measures are examined based on quantitative and empirical investigation. At the beginning of a class's lifecycle, it seems that some of the OO metrics proposed by Chidamber and Kemerer can be helpful in predicting the class's fault-proneness. Additionally, they outperform "traditional" code metrics—which cannot be gathered until later in the software development process—as predictors on our dataset.

RICH: Rendering Integer-Based Vulnerabilities Safe by Design

Here we introduce RICH, an acronym for "Run-time Integer Checking," a tool that can efficiently identify integer-based attacks on C programs while they are running. When a variable's value exceeds the range of the machine word used to materialize it, for as when assigning a huge 32-bit int to a 16-bit short, a common programming mistake known as a C integer bug occurs [1–15]. We prove that the well-known sub-typing theory represents all C integer operations, both safe and dangerous. To protect against integer-based assaults, the RICH compiler extension converts C programs to object code that executes self-monitoring. After adding RICH as a GCC plugin, we tested it on several servers in the network and UNIX tools. Integer operations are ubiquitous, yet RICH's performance overhead is a meager 5% on average. While testing for known integer flaws, RICH discovered two new ones and caught all except one. Based on these findings, RICH is an effective and lightweight tool for testing software and a defense mechanism for runtime. Due to its lack of modeling of some C features, RICH has the potential to overlook some integer problems and produce false positives when programmers intentionally employ integer overflows.

The CSSV project is working towards a practical solution that can statically detect all C buffer overflows.

Software viruses sometimes take advantage of security holes in C programs caused by incorrect string manipulations. A tool that statically reveals all string manipulation issues is presented here: C String Static Verifier (CSSV). As a cautious tool, it discloses all such mistakes, even though it occasionally triggers false alarms. Thankfully, the stated false alert rate is low, demonstrating that it is possible to significantly decrease program vulnerability. By dissecting each operation independently, CSSV is able to manage big applications. In order to achieve this goal, the technology permits procedural contracts that are confirmed. To ensure that the actual EADS Airbus code was error-free, we built a CSSV prototype and tested it extensively. The usage of CSSV revealed actual issues with minimal false positives when applied to another widely used string demanding application. Enhancing safety with lightweight, extendable static analysis.

Common types of implementation problems are typically the target of security attacks. These issues occur with disturbing regularity despite developers' best efforts to identify and fix them before software deployment. This is not due to a lack of understanding of these vulnerabilities within the security community, but rather to the fact that methods for avoiding them have not been incorporated into software development. In order to identify typical security flaws, such as format string vulnerabilities and buffer overflows, this paper details an extendable tool that use lightweight static analysis.

Test Automation for Whitebox Fuzz

One reliable method for discovering software security flaws is fuzzy testing. Fuzz testing tools have always tested the outputs of programs by randomly altering their well-formed inputs. Our whitebox fuzz testing alternative is based on the latest innovations in symbolic execution and dynamic test development. We capture restrictions on inputs that capture

how the program utilizes them, symbolically assess the recorded trace, and then record a real execution of the program under test on a well-formed input. New inputs that exercise alternative program control routes are produced by negating and solving each collected constraint using a constraint solver. A code-coverage maximizing heuristic is used to detect problems as soon as possible, and this procedure is repeated with their aid. As a new tool for Whitebox fuzzing arbitrary file-reading Windows applications, SAGE (Scalable, Automated, Guided Execution) uses x86 instruction-level tracing and emulation, and we've integrated this approach into it. In this paper, we detail the essential improvements that are required for dynamic test creation to scale to massive input files and lengthy execution traces including hundreds of millions of instructions. After that, we show extensive trials using a number of Windows programs. Notably, SAGE finds the MS07-017 ANI vulnerability without format-specific knowledge, although static analysis and thorough Blackbox fuzzing failed to do so. Even though it's still early in the development process, SAGE has found thirty or more new flaws in big Windows apps that have been delivered, such as image processors, media players, and file decoders. A number of these issues may include memory access violations that might be exploited.

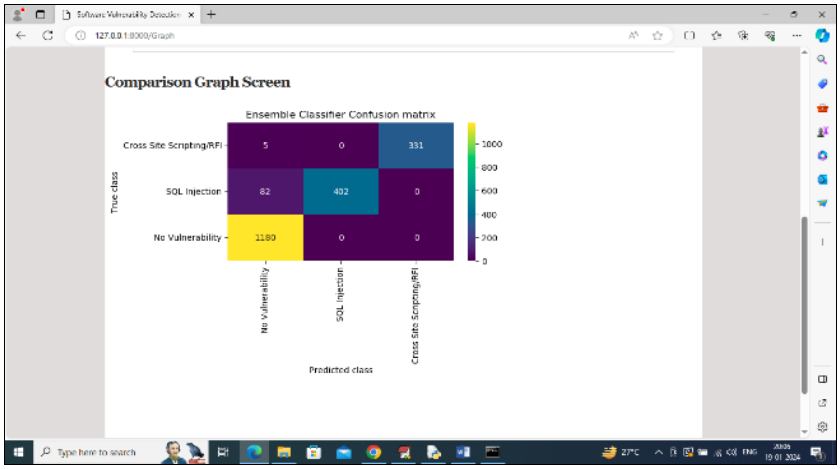
Methodology

To implement this project we have designed following Modules

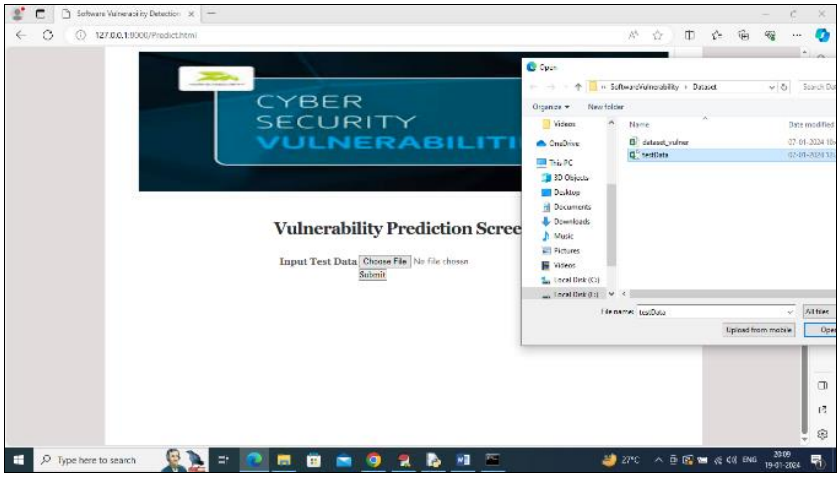
1) **New User Register:** new user can register with the application
2) **User Login**: after sign up user can login to application
3) **Load Dataset:** Following successful login, users will be able to import datasets into the program. They will then be able to extract labels and queries from the dataset. One important step is to eliminate stop words such as "and," "the," "what," and many more. The application will contain core query terms when stop words are removed. The Natural Language Processing Toolkit will be used to process the dataset for the key terms.
4) **Run Ensemble Algorithms:** We will train a model using the processed dataset and then apply it to test data to determine accuracy and other metrics using the Ensemble Machine Learning technique.
5) **Confusion Matrix Graph:** Using this module, we may visualize the algorithm's prediction capacity through a confusion matrix graph.
6) **Predict Vulnerability:** This module allows users to contribute new test data queries, which are analyzed by a machine learning system to forecast the type of vulnerability.

Results and Discussion

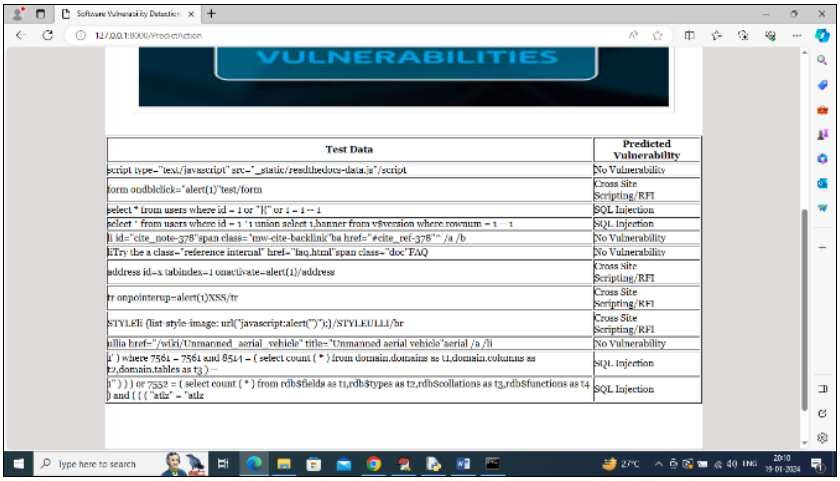
In above result Ensemble Machine Learning algorithm training completed and can see its prediction accuracy as 95% and can see other metrics like precision, recall and FCSORE. Now click on 'Confusion Matrix Graph' link to view visually how many records ensemble predicted correctly and incorrectly.



In above graph x-axis represents Predicted Labels and y-axis represents True Labels and then all different colour boxes in diagnol represents correct prediction count and remaining all blue boxes represents incorrect prediction count which are very few. Now click on 'Predict Vulnerability' link to upload test data and predict Vulnerability.



In above result selecting and uploading 'testData.csv' file which contains SQL, XSS and RFI coding commands and then click on 'Submit' button to get below output.



In above table in first column can see SQL queries, XSS and RFI coding commands and in second column can see predicted vulnerability.

The aforementioned program makes it easy to find any security hole, and the 'testData.csv' file in the 'Dataset' folder is where you may insert a new test command.

Conclusion

Several conclusions may be drawn from the thorough testing of function vulnerability categorization using n-grams, suffix trees, and trivial characteristics. First, taking the 74% accuracy we achieved from "character diversity "as a baseline criterion, it is clear that extracting multiple n-grams does not appear to provide strong classification results at this time. We also found that the overall outcome remained unchanged even when n-gram combinations were manually picked (in a way that would often be considered unlawful and result in overfitting). Nonetheless, the study serves as a solid proof-of-concept for a crucial point: insignificant traits might provide significant insight on a function's vulnerability. To make the outcomes even better, this study may be done in a few different ways. To start, it may be easy to come up with more insignificant qualities to look into. Second, aside from the default settings, it may be prudent to try out different n-gram selection methods and the various categorization parameters available in the SciKit library. Third, rather of relying just on "character diversity," it would be instructive to zero down on the most crucial characters (or strings). One approach would be to do the character variety tests again after pre-processing removes certain strings, such as square brackets, curly brackets, ++, etc. Lastly, it is feasible to evaluate if the methods discussed in this article may effectively identify security flaws in languages other than C.

References

1. Enron email dataset. Available from: https://www.cs.cmu.edu/~enron/. Accessed: 2017-07-01.
2. National vulnerability database. Available from: https://nvd.nist.gov. Accessed: 2017-07-01.
3. Basili VR, Briand LC, Melo WL. A validation of object-oriented design metrics as quality indicators. IEEE Trans Softw Eng. 1996;22(10):751-761.
4. Brumley D, Chiueh T-C, Johnson R, Lin H, Song D. Rich: Automatically protecting against integer-based

vulnerabilities. Department of Electrical and Computing Engineering; c2007. p. 28.
5. Dor N, Rodeh M, Sagiv M. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In: ACM Sigplan Notices. 2003;38:155-167.
6. Evans D, Larochelle D. Improving security using extensible lightweight static analysis. IEEE Softw. 2002;19(1):42-51.
7. Godefroid P, Levin MY, Molnar D. Sage: whitebox fuzzing for security testing. Queue. 2012;10(1):20.
8. Haller I, Slowinska A, Neugschwandtner M, Bos H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: USENIX Security Symposium; c2013. p. 49-64.
9. Hassan AE. Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society; c2009. p. 78-88.
10. Kim S, Zimmermann T, Whitehead EJ Jr, Zeller A. Predicting faults from cached history. In: Proceedings of the 29th International Conference on Software Engineering. IEEE Computer Society; c2007. p. 489-498.
11. Larochelle D, Evans D, *et al*. Statically detecting likely buffer overflow vulnerabilities. In: USENIX Security Symposium: Washington DC. 2001;32.
12. Lathar P, Shah R, Srinivasa K. Stacy-static code analysis for enhanced vulnerability detection. Cogent Eng. 2017;4(1):1335470.
13. Ma R, Yan Y, Wang L, Hu C, Xue J. Static buffer overflow detection for C/C++ source code based on abstract syntax tree. J Resid Sci Technol. 2016;13(6).
14. Moser R, Pedrycz W, Succi G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th International Conference on Software Engineering. ACM; c2008. p. 181-190.
15. Pampapathi RM, Mirkin BG, Levene M. A suffix tree approach to antispam email filtering. Mach Learn. 2006;65(1):309-338.