自动化安保漏洞

Towards Automated Security Vulnerability and Software Defect Localization

Nicholas Visalli[1], Lin Deng[1], Amro Al-Suwaida[1], Zachary Brown[1], Manish Joshi[1], and Bingyang Wei[2]
[1]Department of Computer and Information Sciences
Towson University, Maryland, USA
{nvisal1,aalsuw2,zbrown4,mjoshi1}@students.towson.edu, LDeng@towson.edu
[2]Department of Computer Science
Texas Christian University, Fort Worth, Texas, USA
b.wei@tcu.edu

***Abstract*— Security vulnerabilities and software defects are prevalent in software systems, increasingly threatening every aspect of cyberspace. Along with the increased complexity of software systems, security vulnerabilities and software defects become a major target of cyberattacks, which can lead to significant consequences. Manual identification of vulnerabilities and defects in software systems is very time-consuming and tedious. Many tools have been designed to help analyze software systems and to find vulnerabilities and defects. However, the vulnerabilities and defects that are not caught by these tools are usually too complicated to find, do not fall inside of an existing rule-set for identification, or are hidden well. In this undergraduate student research project, we hypothesize that these undiscovered vulnerabilities and defects do not occur randomly, rather, they share certain common characteristics. Thus, we propose a static analysis algorithm that can automatically predict security vulnerabilities and software defects. We use a comprehensive experimental evaluation to assess the algorithm and report our findings.**

Keywords—Cybersecurity, Security Vulnerability, Defect Localization, Static Analysis, Software Defect

I. Introduction

Cyberspace, including systems, information and people, has become increasingly critical to society. However, cyber-attacks are threatening every aspect of cyberspace, for example, by exploiting software vulnerabilities and defects, stealing sensitive information, or attacking critical infrastructure. Among these attacks, the most influential one is to exploit software security vulnerabilities and defects. Today's software is no longer isolated, small programs with thousands of lines of code but, rather, complex, integrated, connected systems that are critical to society, including national security, financial and business transactions, transportation controls, etc. Software security vulnerabilities and defects in existing software systems may lead to unauthorized access to sensitive data (confidentiality), exfiltration or manipulation of sensitive/proprietary information (integrity), or the malfunction or denial of service of systems (availability). These may result in significant consequences, including major loss of finance, resources and even human life. For example, *Conficker*, a computer worm exploited a vulnerability of Windows operating systems, caused a financial loss of more than $9 billion [1]. Unfortunately, the increasing scale of complexity and connectivity of software has resulted in significantly more software security vulnerabilities and defects. It is critical for cybersecurity analysts and researchers to design new techniques and tools for detecting vulnerabilities and defects to ensure the security and quality of software systems.

Finding security vulnerabilities and software defects is a very time-consuming and difficult task, especially considering the size and complexity of today's software systems as well as the number and variety of vulnerabilities and defects. In 2015, more than 16,000 vulnerabilities were discovered [2]. It is unrealistic for software developers, cybersecurity analysts and testers to find these vulnerabilities and defects unaided. Static analysis is one approach that examines software artifacts (e.g., source or binary code) without actual execution [3]. Many static analysis tools have been proposed to analyze software systems to catch similar vulnerabilities and defects. However, these tools tend to detect similar families of vulnerabilities or defects. Consequently, they might miss vulnerabilities and defects that are too complicated to find, do not fall inside of an existing rule-set for finding defects, or are hidden well.

In this paper, we experimentally evaluate four static analysis tools and identify those vulnerabilities and defects always missed by these tools. Then, we extract abstract syntax trees (AST) for these vulnerabilities and defects. Finally, we design a matching algorithm that is able to determine the likelihood of undetected bugs appearing in specific patterns of code. The ultimate goal of this research is to design and implement an automated, reliable, effective, and efficient software security vulnerability and defect analysis algorithm that reduces the workload of cybersecurity analysts and software developers.

The rest of this paper is organized as follows: Section II goes through the algorithms used in the research; Section III describes the evaluation approach and results; Section IV gives an overview of related research, and the paper concludes and suggests future work in Section V.

II. Algorithms

In this paper, we propose a unique matching algorithm based on two well-known algorithms: Needleman-Wunsch (NW) algorithm [4] and Longest Common Subsequence (LCS) algorithm [5]. NW algorithm is originally designed to be used in bioinformatics to align protein or nucleotide sequences. It can compare biological sequences. For example, suppose we have a large sequence, the NW algorithm divides it into a series of smaller problems and uses the solutions to the smaller problems to reconstruct a solution to the larger problem. LCS algorithm is originally designed for finding the longest subsequence common to all sequences in a set of sequences. It can break large problems down into subproblems. For example, suppose we have two sequences, "ABCDGH" and "AEDFHR." LCS algorithm computes "ADH" of length 3.

These algorithms have been used in practice to search for plagiarism in program source code. By combining NW and LCS algorithms, we design a unique algorithm for matching program source code. The NW algorithm checks for the number of matching nodes within a sequence of strings. The scoring process takes order into account and subtracts a set value from the score in the event of a gap. The LCS algorithm cares only about the number of matching nodes and does not subtract points in the event of a node mismatch. Given the idea that each algorithm focuses on different aspects of a program, we propose to assign equal weights to each of the two algorithms to form our new algorithm. The algorithm calculates a similarity score ranging from 0 to 1, where 0 means two programs have nothing in common, 1 means two programs are identical.

III. Experimental Evaluation

A. Evaluation Approach

Our experimental evaluation is designed to answer the following two research questions:

Research Question 1:

Are there any security vulnerabilities or software defects that always stay undetected by static analysis tools?

Research Question 2:

Is our proposed algorithm able to predict those undetected vulnerabilities and software defects?

Therefore, we divide our experimental evaluation into two major parts to address the above research questions. We first find out those security vulnerabilities and software defects that cannot be found by selected static analysis tools. Then, we apply our proposed algorithm to evaluate the effectiveness of our approach. Figure 1 illustrates the general approach of our study. First, we use four static analysis tools (Jlint [6], SpotBugs [7], PMD [8], and CodePro AnalytiX [9]) to scan for security vulnerabilities and software defects in Juliet dataset [10]. The Juliet dataset contains over 81,000 security vulnerabilities and software defects. In this step, we would like to investigate what types of vulnerabilities and defects that these tools can and cannot find. In order to find get representative results, we select the tools based on their parsing methodologies. Each of the tools finds vulnerabilities and defects with a unique approach. Most of these tools are still well maintained and frequently updated.

After that, we create a list of vulnerabilities and defects that are not found by any tools. The list will be critical to our study. This list will then be further categorized by different types. Then, we generate abstract syntax trees for each program within the list using *JavaParser*, a third-party AST generation tool. We further simplify these abstract syntax trees by using preorder traversal to generate node sequences. Finally, we compare the node sequences against each other in one million random pairs, using the algorithm proposed in Section II to evaluate the effectiveness at identifying vulnerabilities and defects. Specifically, the scores generated by the algorithm will tell whether these undetected source files share similar code structures, i.e., the possibility of being a vulnerability or a defect.

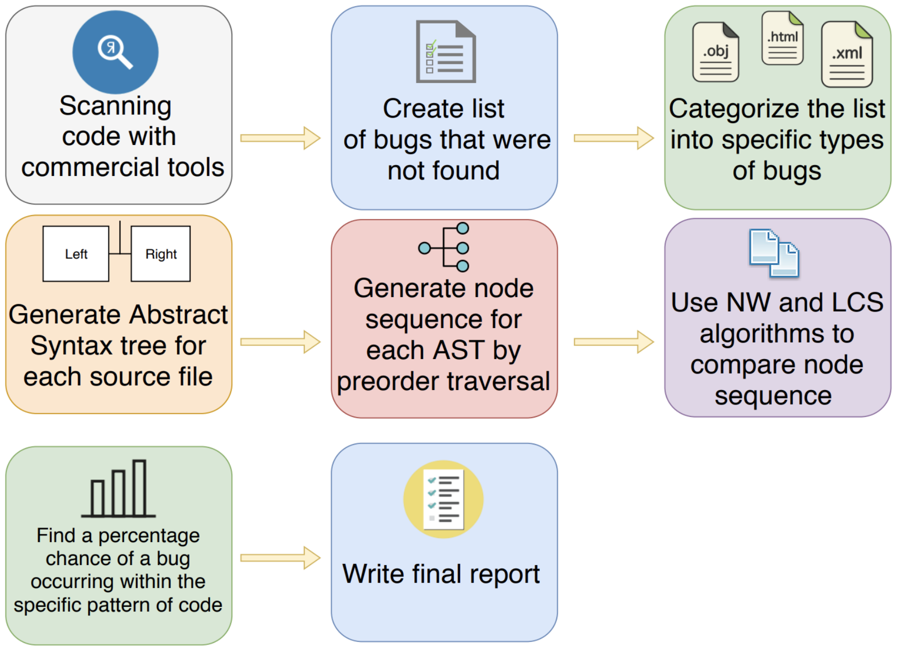
Scanning code with commercial tools
Create list of bugs that were not found
.obj
.html
.xml
Categorize the list into specific types of bugs
Left
Right
Generate Abstract Syntax tree for each source file
Generate node sequence for each AST by preorder traversal
Use NW and LCS algorithms to compare node sequence
Find a percentage chance of a bug occurring within the specific pattern of code
Write final report

Figure 1: General Approach

B. *Evaluation Results and Discussion*

In this section, we present the evaluation results and answer the research questions.

1) Research question 1:

When we ran four different tools against the Juliet dataset, we obtained 2,016,564 results. These results also include syntactical errors, in addition to the vulnerabilities and defects in the Juliet dataset. After filtering through these results, we found that there were over a stunning 33,000 vulnerabilities or defects that none of the tools were able to detect, which indicates that there are a large number of security vulnerabilities and software defects that cannot be detected by selected static analysis tools.

2) Research question 2:

We used our proposed algorithm to compare these undetected vulnerabilities and defects. With the algorithm, we were able to get scores that represent the probability of a bug existing in that code. We compared a million random pairs of classes in the list using the proposed algorithm and obtained a huge output with scores for every program. The average similarity score we obtain is 0.7222, with the highest score at 0.93 and the lowest score at 0.0833. After further comparing the results, we concluded that using the scores of the new algorithm and calculating the average would not produce an accurate score for prediction. Each of the two algorithms would rather serve better depending on the scope of the code being compared. The LCS algorithm works better with the class level comparison, which is the scope we used for our tests. So the LCS algorithm performs better than the NW algorithm at predicting the probability of having vulnerabilities or defects. The NW algorithm works a little differently and does not produce accurate results when comparing so many classes as the score is distorted. We concluded that NW algorithm scoring would be better suited for a method-level comparison.

IV. Related Work

Static analysis is a widely used approach in identifying software defects and security vulnerabilities. The paper of Rutar et al. [11] is the first detailed comparison of several different bug finding tools. The study tested five tools against five Java programs. These tools were Bandera, which uses model checking, ESC/Java (Extended Static Checking System for Java), which focuses on theorem proving, FindBugs and JLint, which both rely on syntactic bug pattern detection and a data flow component, and PMD, which only uses syntactic bug pattern detection. All of these tools follow different rule sets for finding bugs. The study of Pearson et al. [12] focused on real faults, instead of artificial faults. They conducted an empirical study with seven fault localization techniques on 2995 artificial faults, as well as 310 real faults. The study then proposed a new methodology for fault localization. Along with the popularity of machine learning techniques, researchers start to design machine learning based techniques to help identify security vulnerabilities automatically. Li et al. [13] designed VulDeePecker, a deep learning-based vulnerability detection system. They used code gadgets as the representation of software programs and transformed them into vectors, so that data can be fed into the VulDeePecker. The results of their experimental study show that VulDeePecker performed very effective at detecting vulnerabilities and even identified vulnerabilities missed by the National Vulnerability Database and other similar tools. Pan et al. [14] studied an unsupervised/semi-supervised approach for web attack detection based on the Robust Software Modeling Tool (RSMT). Their results show that RSMT can efficiently and accurately detect attacks using synthetic datasets and production applications. The study of Sultana [15] analyzed and compared traceable patterns of software security vulnerability and proposed a vulnerability prediction model based on machine learning and statistical techniques.

V. Conclusion and Future Work

Nowadays, software is everywhere in people's daily life. Today's software becomes more and more complex, inter-connected, and integrated, which leads to an increasing number of security vulnerabilities and software defects. Meanwhile, attackers start to gain access to more sophisticated hacking tools that increase the number of cyber-related threats. Identifying vulnerabilities and defects is a very time-consuming task. Different analysis tools are designed to help reduce the human efforts. However, the results of our research find that many vulnerabilities and defects stay undetected. Using an experimental evaluation, we find that the LCS algorithm is good at detecting at the class level, while the NW algorithm works better at the method level.

For the future work, as discussed in Section IV, along with the rapid advancement of machine learning techniques, we believe that an approach designed upon machine learning should be a good fit to solve the problem of predicting vulnerabilities and defects. Thus, we plan to investigate different machine learning techniques and design a more effective approach.

Acknowledgment

This undergraduate student research project is supported via the Information Security Research and Education (INSuRE) project [16]. We thank our technical directors, Dickie George and James Pendergrass from Applied Physics Laboratory, for their help and mentorship.

References

[1] "Conficker's estimated economic cost? $9.1 billion | ZDNet." [Online]. Available: https://www.zdnet.com/article/confickers-estimated-economic-cost-9-1-billion/. [Accessed: 23-Apr-2018].

[2] Flexera Software, "Vulnerability review 2016," 2016.

[3] B. Chess and G. Mcgraw, "Static analysis for security," *IEEE Security and Privacy*, vol. 2, no. 6. pp. 76–79, Nov-2004.

[4] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, Mar. 1970.

[5] D. S. Hirschberg, "Algorithms for the Longest Common Subsequence Problem," *J. ACM*, vol. 24, no. 4, pp. 664–675, Oct. 1977.

[6] "Jlint - Find Bugs in Java Programs," 2016. [Online]. Available: http://jlint.sourceforge.net/. [Accessed: 17-Aug-2018].

[7] "SpotBugs." [Online]. Available: https://spotbugs.github.io/. [Accessed: 17-Aug-2018].

[8] "PMD." [Online]. Available: https://pmd.github.io/. [Accessed: 17-Aug-2018].

[9] "CodePro AnalytiX :," pp. 1–12, 2006.

[10] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and java test suite," *Computer (Long. Beach. Calif).*, vol. 45, no. 10, pp. 88–90, 2012.

[11] N. Rutar, C. B. Almazan, and J. S. Foster, "A Comparison of Bug Finding Tools for Java," in *15th International Symposium on Software Reliability Engineering*, 2004, pp. 245–256.

[12] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and Improving Fault Localization," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 609–620.

[13] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," in *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.

[14] Y. Pan, F. Sun, J. White, D. C. Schmidt, J. Staples, and L. Krause, "Detecting Web Attacks with End-to-End Deep Learning," pp. 1–14.

[15] K. Z. Sultana, "Towards a software vulnerability prediction model using traceable code patterns and software metrics," in *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 1022–1025.

[16] A. Sherman, M. Dark, A. Chan, R. Chong, T. Morris, L. Oliva, J. Springer, B. Thuraisingham, C. Vatcher, R. Verma, and S. Wetzel, "INSuRE: Collaborating Centers of Academic Excellence Engage Students in Cybersecurity Research," *IEEE Secur. Priv.*, vol. 15, no. 4, pp. 72–78, 2017.