An Efficient Source Code Auditing using Fuzzy Decision Tree

Rani Sahu
Student of information Technology Department
SATI
VIdisha (M.P) India

Shailendra Kumar Shrivastava, PhD
Asst. Prof.of information Technology Department
SATI
VIdisha (M.P) India

ABSTRACT

Here in this paper the discovery of Vulnerabilities in the Source Codes is proposed. The Proposed Methodology applied is based on the Concept of Fuzzy Based Decision Tree. The Methodology adopted here for the Checking of Codes Vulnerabilities provides efficient discovery of Vulnerabilities and hence provides improved performance and high precision and Recall. The Proposed Methodology Audits the source code and searches the possible vulnerabilities on the basis of Rules generated Fuzzy Decision Tree. Various Experimental results are achieved on numerous datasets and shows that the proposed methodology provides better accuracy in comparison.

Keywords

Software, Auditing, Fuzzing, Vulnerabilities, Fault Prediction, Vulnerabilities Prediction

1. INTRODUCTION

As gradually more secure the ease of information technology the security of computer systems is becoming a developing apprehension. To present it secure network protocols structural designs and cryptographic algorithms are essential. Due to the lack of ability of a program to recognize non-trivial properties of another program, the generic problem of ending software vulnerabilities is undecidable. However, the efficiency of traditional fuzz testing tools is usually very poor due to the blindness of test generation. As working [1] as on involuntary fuzzing organization for software vulnerability detection, this combines fuzzing with symbolic execution and taint analysis techniques to tackle the above problem. As a consequence, modern represents for marking security flaws are either inadequate to specific types of vulnerabilities or build on deadly and manual auditing. Particularly, securing large software projects, for example an operating system kernel, look likes a scaring job as a single flaw may challenge the security of the complete code base. Even though some classes of vulnerabilities reoccurring all the way through the software background exist for a long time, for example buffer overflows and format string vulnerabilities, automatically detecting their incarnations in specific software projects is often still not possible without significant expert knowledge [2]. The techniques that have been proposed include source code auditing, static program analysis, dynamic program analysis, and formal verification [3- 5]. However, many of the methods lie on dangerous conclusions of the range concerning the cost-effectiveness as represented in Figure 1.1. Static sequence investigates are utilized by many developers to test their programs because they are efficient in discovering some trivial bugs that can be trapped by the laws that describe security

Breaches with very tiny store. On the other hand, they are in adequate in that the presentation is only good as the laws.

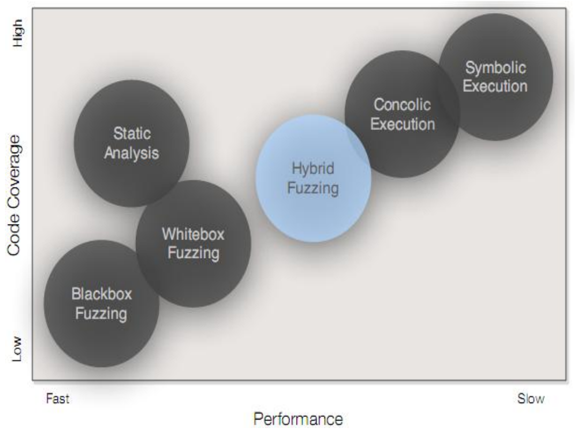


Figure 1: Software Bug Ending Systems Diagram

Mutual system used by many sanctuary scholars is a sequence challenging system called fuzzing. Fuzzing finds bugs in a target program by natively executing it with randomly mutated or generated inputs while monitoring the execution for abnormal crashes. Fuzzing is good at quickly exploring the program code in depth for the reason that it runs the objective program inhabitant with concrete inputs. On the other hand, due to its scenery, fuzzing often suffers from low code reporting trouble. Symbolic execution is another method that has in recent times gotten the awareness of security researchers. On the contrary to fuzzing, representative performance tests a program by pleasuring the program's input as symbols and understanding the program over these symbolic inputs. In presumption, symbolic execution is assurance to be effectual in accomplish high code coverage yet this usually needs exponential reserve which is not realistic for many real-world programs. Our purpose is to discover more bugs faster than conventional approaches. With the purpose of achieve this aim; we need to obtain high code coverage in reasonable resource bound (e.g. computing power and time). High code reporting implies both breadth and depth in exploration of the program. Although not achieve the best code coverage or speed, Aim to find the sweet spot in cost-effective way to gain higher code reporting than the fuzzer and higher speed than the representative executor as shown in Figure 1.1.

Naturally, security researchers have been dynamically designing automated vulnerability analysis methods. Many approaches subsist dropping into three main classes: static, active, and concolic examination schemes. These approaches have various advantages and disadvantages. Static analysis systems can give verifiable assurances – that is a static analysis system can demonstrate with confidence that a given piece of binary code is protected. On the other hand, such schemes have two basic problems: they are inaccurate resulting in a huge quantity of false positives and they cannot give "actionable input" (i.e., an illustration of a specific input that can trigger a detected liability). Dynamic analysis systems, for example "fuzzers", check the inhabitant execution of an application to recognize flaws. When flaws are distinguished these systems can give actionable inputs to trigger them. On the other hand, these schemes suffer from they require for "input test cases" to constrain implementation. Without an comprehensive set of test cases, which needs significant guidebook attempt to produce, the usability of such schemes is imperfect. Finally, concolic execution engines use program understanding and restriction solving methods to produce inputs to travel around the circumstances gap of the binary in an effort to accomplish and trigger vulnerabilities. On the other hand, because such schemes are capable to trigger a large amount of paths in the binary (i.e., for a conditional branch, they often produce an input that physically security researchers have been dynamically planning automated vulnerability analysis methods. Many approaches subsist, dropping into three main classes: static, active, and concolic investigation systems. These approaches have various advantages and disadvantages. Static analysis systems can provide provable guarantees – that is, a static analysis system can demonstrate with assurance that a given piece of binary code is protected. On the other hand such systems have two essential disadvantages: they are inaccurate resulting in a huge amount of false positives and they cannot make available "actionable input" (i.e., an example of a specific input that can trigger a distinguished vulnerability). Dynamic analysis schemes, such as "fuzzers", monitor the inhabitant execution of an application to recognize flaws.

When flaws are detected these methods can give actionable inputs to trigger them. On the other hand, these methods suffer from they require for "input test cases" to drive execution. Without incomprehensive set of test cases which needs significant instruction manual attempt to produce the usability of such schemes is inadequate. To conclude, concolic execution engines operate program understanding and restraint solving methods to produce inputs to investigate the state space of the binary in an effort to accomplish and trigger vulnerabilities. On the other hand, because such schemes are proficient to trigger a large number of paths in the binary (i.e., for a conditional branch, they frequently generate an input conditions

In this work, here we focus on vulnerabilities in software, and how they occur and for this reason the schemes regard as are software systems. Additionally, they prohibit vulnerabilities in operation and management from the study to give attention to completely on those flaws observable and fixable in program code i.e., vulnerabilities in a program's design and completion With these constraints in mind, here they focused on vulnerabilities are described to be a subset of flaws making understandable that determining flaws can be think a first step in vulnerability discovery. On the other hand, constrict in on those flaws that infringe security policies is equally significant consequently, in bare distinction to techniques for the detection of defects expanded in software engineering [6, 7], the focus lies on recognizing flaws that are extremely feasible to give the attacker with a certain increase and that can in information be triggered by an attacker.

2. PROPOSED METHODOLOGY

The technique implemented here for the Auditing of Source Codes is based on the concept of feature selection using PSO-SVM and then classification of these features can be done using decision tree generated using Fuzzy logic. The proposed methodology works in the following phases:

2.1 Fuzzy based Decision Tree Classifier

A Decision tree is a recursive form of tree consisting of nodes and leaves on the basis of which a decision is taken from the dataset. It is constructed on the basis of attributes dependency value in which the root node is the most dependent attribute of the dataset. A decision tree is also used for the classification of the dataset. Here Fuzzy Logic is used as the superset of Boolean logic that is used for the identification and classification of classes from the dataset.

2.1.1 Proposed Algorithm

```
Fuzzy Decision Tree (T, F , O)
T- Training Dataset Features
F- PIMA Indian Input Dataset features selected from PSO-SVM
O- Output classified features

1. Initially an empty Tree 't'  $t \to \varphi$
2. For each of the training features or attributes present in the dataset
3. Compute Information & Entropy using
   $I(C1,C2) = -\frac{C1}{(C1+C2)}\log\left(\frac{C1}{(C1+C2)}\right) - \frac{C2}{(C1+C2)}\log\left(\frac{C2}{(C1+c2)}\right)$
   Where I denotes Information of the dataset based on classes C1 & c2.
4. Compute Gain using
   $Gain(attribute) = I(attribute) - E(attribute)$
   Where, I is information & E is the entropy of the attribute
5. Select the attribute with highest information gain
6. Update the tree't' the attribute as the root node to 't'.
7. Remove the attribute from the relation set
8. End
```

Figure 2.1: Fuzzy Decision Tree Algorithm

Now we propose our algorithm to generate a decision tree in the following way.

3. EXPERIMENTAL RESULT

Describes the execution details and parameter detail used in work and results are shown and analysed. Results are compared with fuzzy decision tree algorithm based on Particle Swarm Optimism (PSO)-Support Vector machine (SVM) techniques.

3.1 Formula Used in Result

The classification algorithms based on Precision, Recall, F-measure, and Area Under Curve (AUC) or ROC (Receiver Operating Characteristics) as argues that AUC is the best measure to report the classification accuracy.

Precision measures how many of the vulnerable instances returned by a model are actually vulnerable. The higher the precision is, the fewer false positives exist.

Recall measures how many of the vulnerable instances are actually returned by a model. The higher the recall is, the fewer false negatives exist.

F-Measure is the harmonic mean of Precision and Recall.

A binary classifier, which makes two possible errors:

1. False positive (FPR)
2. True Positive(TPR)

False Positive Rates (FPR): A correctly classified vulnerable class is a False positive (FP).

FP=FP/(FP+TP)

True Positive Rates (TPR): A correctly classified non-vulnerable class is a True positive(TP).

TP=TP/(TP+TP)

Receiver Operating Characteristic (ROC) Curve:

It is a graphical approach for displaying the trade off between true positive rate (TPR) and false positive rate (FPR) of a classifier.

TPR = positives correctly classified/total positives.

FPR = negatives incorrectly classified/total negatives.

Experimental Setup

The following software require to an implementation of the proposed system.

Software Requirement:
The planned work is implemented on Netbeans-6.9-ml windows. All analysis and graphs are planned in using Netbeans. Netbeans provides tools to accumulate, analyse, and visualize information, sanctioning you to realize insight into your information in a fraction of the time it would take using spreadsheets or traditional programming languages. Additionally document and results sharing through plots and reports or as revealed Netbeans code is also possible in Netbeans.

The Datasets

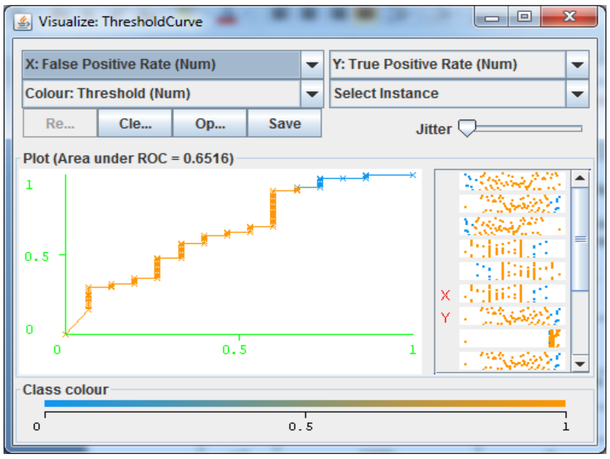
This is a Promise Software Engineering Repository data set made publicly. Available in order to encourage repeatable, refutable, verifiable, and/or Improvable predictive models of software engineering. If you publish material based on PROMISE data sets then, please follow. The acknowledgment guidelines posted on the PROMISE repository web page http://promise.site.uottowa.ca/SERepository.

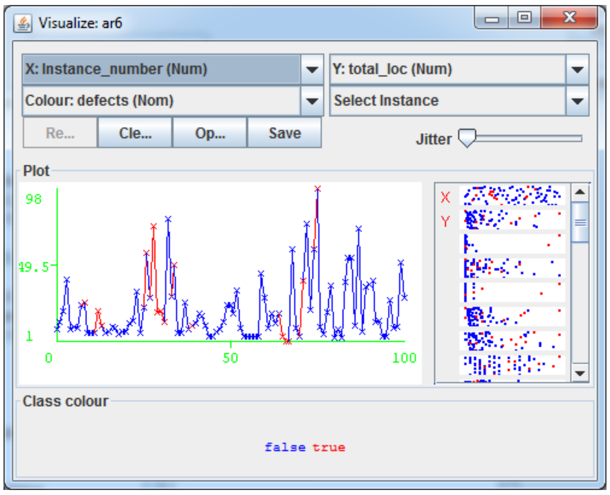
a. Title: AR1 /Software Defect Prediction
b. Date: February, 4th, 2009
c. Data from a Turkish white-goods manufacturer
d. Donated by: Software Research Laboratory (Soft lab),
e. Bogazici University, Istanbul, Turkey

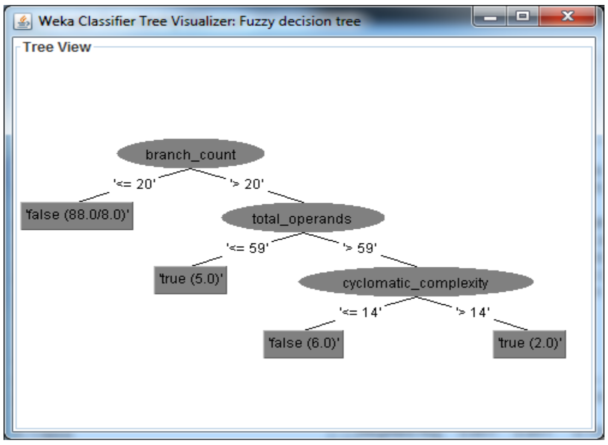
Description:

1. Embedded software in a white-goods product.
2. Implemented in C.
3. Consists of 121 modules (9 defective / 112 defect-free)
4. 29 static code attributes (McCabe, Halstead and LOC measures) and 1 defect information(false/true)
5. Function/method level static code attributes are collected using

4. RESULTS ANALYSIS


Visualize: ThresholdCurve
X: False Positive Rate (Num)
Y: True Positive Rate (Num)
Colour: Threshold (Num)
Select Instance
Re...
Cle...
Op...
Save
Jitter
Plot (Area under ROC = 0.6516)
1
0.5
0
0
0.5
1
X
Y
Class colour
0
0.5
1





Datasets	Correctly Classified Instances	Mean Absolute Error	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC
ar1	99.17%	0.0083	0.889	0.111	0.991	0.889	0.941	0.944
ar3	100%	0	1	0	1	1	1	1
ar4	81.31%	0.1869	1	0	0.813	1	0.897	0.5
ar5	77.77%	0.22	1	0	0.778	1	0.875	0.5
ar6	85.15%	0.1485	1	0	0.851	1	0.92	0.5

5. CONCLUSION AND FUTURE WORK

The paper deals with the basic concept of software auditing and various techniques implemented for auditing as well as finding their advantages and limitations. A major cause of this is that many developers are not equipped with the right skills to develop secure code. Because of limited time and resources, web engineers need help in recognizing vulnerable components. The proposed Methodology adopted here for the Auditing of Source codes using Fuzz based Decision Tree provides better results in terms of Precision and Recall and Accuracy.

6. REFERENCES

[1] Jun Cai, Jinquan Men, Automatic Software Vulnerability Detection Based on Guided Deep Fuzzing", IEEE 2014.

[2] S. Heelan. Vulnerability detection systems: Think cyborg, not robot. IEEE Security & Privacy, 9(3):74–77, 2011.

[3] Hong-Zu Chou, I-Hui Lin, Ching-Sung Yang, Kai-Hui Chang, and SyYenKuo. Enhancing bug hunting using high-level symbolic simulation. In Proceedings of the 19th ACM Great Lakes symposium on VLSI, GLSVLSI '09, pages 417–420, NewYork, NY, USA, 2009.

[4] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. Queue, 10(1):20:20–20:27, January 2012.

[5] Daniel Quinlan and Thomas Panas.Source code and binary analysis of software defects. In Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies, CSIIRW '09, pages 40:1–40:4, New York, NY, USA, 2009.

[6] Cadar, C., Dunbar, D., and Engler, D. R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2008.

[7] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. (2006).Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on Software Engineering, 32:176-192

[8] Ferreira AL, Machado RJ, Paulk MC. Size and complexity attributes for multimodal improvement framework taxonomy. Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on.2010; 306–309. DOI: 10.1109/ICSEA.2009.80.

[9] M. Gegick, L. Williams, J. Osborne and M. Vouk, "Prioritizing software security fortification through code-level metrics", In Proceedings of the 4th ACM workshop on Quality of protection, pages 31–38. ACM, (2008).

[10] K.-S. Joo and J.-W. Woo, "Development of object-oriented analysis and design methodology for secure web applications", International Journal of Security and Its Applications, vol. 8, no. 1, (2014), pp. 71–80.

[11] Y. Shin, A. Meneely, L. Williams and J. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities", IEEE Transactions on Software Engineering, vol. 37, no. 6, (2011), pp. 772–787.

[12] Wheeler, David A. and Rama S. Moorthy, "SOAR for Software Vulnerability Detection, test and Evaluation," IDA paper P-5061, July 2014.

[13] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities", Journal of Systems Architecture, vol. 57, no. 3, (2011), pp. 294–313.

[14] I. Medeiros, N. F. Neves and M. Correia, "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives", In Proceedings of the 23rd international conference on World wide web, ACM, (2014), pp. 63–74.

[15] Siviy J, Kirwan P, Marino L, Morley J. Maximizing your process improvement ROI through harmonization. 2008; Available from: http://www.sei.cmu.edu/library/assets/multimodelExecutive_wp_harmonizationROI_032008_v1.pdf [27 February 2009].