Information and Software Technology 68 (2015) 18–33

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

On the capability of static code analysis to detect security vulnerabilities

Katerina Goseva-Popstojanova[a,*], Andrei Perhinschi[b,1]

[a] Lane Department of Computer Science and Electrical Engineering, West Virginia University, PO Box 6109, Morgantown, WV 26506, United States
[b] TASC Inc, Fairmont, WV 26554, United States

ARTICLE INFO

Article history:
Received 2 January 2015
Revised 16 August 2015
Accepted 18 August 2015
Available online 28 August 2015

Keywords:
Security vulnerabilities
Common Weakness Enumeration (CWE)
Static code analysis evaluation
Experiment
Case studies

ABSTRACT

Context: Static analysis of source code is a scalable method for discovery of software faults and security vulnerabilities. Techniques for static code analysis have matured in the last decade and many tools have been developed to support automatic detection.

Objective: This research work is focused on empirical evaluation of the ability of static code analysis tools to detect security vulnerabilities with an objective to better understand their strengths and shortcomings.

Method: We conducted an experiment which consisted of using the benchmarking test suite Juliet to evaluate three widely used commercial tools for static code analysis. Using design of experiments approach to conduct the analysis and evaluation and including statistical testing of the results are unique characteristics of this work. In addition to the controlled experiment, the empirical evaluation included case studies based on three open source programs.

Results: Our experiment showed that 27% of C/C++ vulnerabilities and 11% of Java vulnerabilities were missed by all three tools. Some vulnerabilities were detected by only one or combination of two tools; 41% of C/C++ and 21% of Java vulnerabilities were detected by all three tools. More importantly, static code analysis tools did not show statistically significant difference in their ability to detect security vulnerabilities for both C/C++ and Java. Interestingly, all tools had median and mean of the per CWE recall values and overall recall across all CWEs close to or below 50%, which indicates comparable or worse performance than random guessing. While for C/C++ vulnerabilities one of the tools had better performance in terms of probability of false alarm than the other two tools, there was no statistically significant difference among tools' probability of false alarm for Java test cases.

Conclusions: Despite recent advances in methods for static code analysis, the state-of-the-art tools are not very effective in detecting security vulnerabilities.

1. Introduction

Today's economy is heavily reliant on computer systems and networks and many sectors, including finance, e-commerce, supply chain, transportation, energy, and health care cannot function without them. The growth of the online commercial environment and associated transactions, and the increasing volume of sensitive information accessible online have fueled the growth of cyber attacks by organized criminal elements and other adversaries [1]. According to the 2014 report by the Ponemon Institute, the mean annualized cost of the cyber crime for 257 benchmarked organizations was $7.6 million per year, with average of 31 days to contain a cyber attack [2].

Deficiencies in software quality are among leading reasons behind security vulnerabilities. *Vulnerability* is defined as a property of system security requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited and result in a security failure [3]. Basically, if a security failure has been experienced, there must have been a vulnerability. Based on the estimates made by the National Institute of Standards and Technology (NIST), the US economy loses $60 billion annually in costs associated with developing and distributing patches that fix software faults and vulnerabilities, as well as cost from lost productivity due to computer malware and other problems caused by software faults [4].

Therefore, it is becoming an imperative to account for security when software systems are designed and developed, and to extend the verification and validation capabilities to cover information assurance and cybersecurity concerns. Anecdotal evidence [5] and prior

* Corresponding author. Tel.: + 1 304 293 9691.
E-mail address: Katerina.Goseva@mail.wvu.edu, katerina.goseva@gmail.com (K. Goseva-Popstojanova).
[1] This work was done while Andrei Perhinschi was affiliated with West Virginia University.

http://dx.doi.org/10.1016/j.infsof.2015.08.002
0950-5849/© 2015 Elsevier B.V. All rights reserved.

empirical studies [6,7] indicated the need of using a variety of vulnerability prevention and discovery techniques throughout software development cycle. One of these vulnerability discovery techniques is a static analysis of source code, which provides a scalable way for security code review that can be used early in the life cycle, does not require the system to be executable, and can be used on parts of the overall code base. Tools for static analysis have rapidly matured in the last decade; they have evolved from simple lexical analysis to employ much more complex techniques. However, in general, static analysis problems are undecidable [8] (i.e., it is impossible to construct an algorithm which always leads to a correct answer in every case). Therefore, static code analysis tools do not detect all vulnerabilities in source code (i.e., false negatives) and are prone to report findings which upon closer examination turn out not to be security vulnerabilities (i.e., false positives). To be of practical use, a static code analysis tool should find as many vulnerabilities as possible, ideally all, with a minimum amount of false positives, ideally none.

This paper is focused on empirical evaluation of static code analysis tools' ability to detect security vulnerabilities, with a goal to better understand their strengths and shortcomings. For this purpose we chose three state-of-the-art, commercial static code analysis tools, denoted throughout the paper as tools A, B, and C. The criteria used to select the tools included: (1) have to be widely used, (2) specifically identify security vulnerabilities (e.g., using the Common Weakness Enumeration (CWE) [9]) and support detection of significant number of different types of vulnerabilities, (3) support C, C++ and Java languages, and (4) are capable of analyzing large software applications, i.e., scale well. An additional consideration in the selection process was to choose one tool from each of the three main classes of static code analysis tools [10] (given here in no particular order): 'program verification and property checking', 'bug finding', and 'security review'.

With respect to the vulnerabilities included in the evaluation, as in works focused on software fault detection [11], synthetic vulnerabilities can be provided in large numbers, which allow more data to be gathered than otherwise possible, but likely with less external validity. On the other side, naturally occurring vulnerabilities typically cannot be found in large numbers, but they represent actual events. Obviously either approach has its own advantages and disadvantages; therefore, we decided to use both approaches.

The first evaluation approach is based on a controlled experiment using the benchmark test suite Juliet which was originally developed by the Center for Assured Software at the National Security Agency (NSA) [12] and is publicly available. Juliet test suite consists of many sets of synthetically generated test cases; each set covers only one kind of flaw documented by the Common Weakness Enumeration (CWE) [9]. Specifically, we used the largest subset of the Juliet test suite claimed to be detectable by all three tools, which consisted of 22 CWEs for C/C++ and 19 CWEs for Java, with 21,265 and 7516 test cases, respectively. Testing static code analysis tools with this benchmark test suite allowed us: to cover a reasonably large number of vulnerabilities of many types; to automate the evaluation and computation of the tools' performance metrics, such as accuracy, recall (i.e., probability of detection), and probability of false alarm; and to run statistical tests.

In addition to the experimental approach, our empirical evaluation includes three case studies based on open source programs, two of which were implemented in C and one implemented in Java. Each program has a known set of vulnerabilities that allow for quantitative analysis of the tools' ability to detect security vulnerabilities. For this part of the study, because of the relatively small number of known vulnerabilities the results were obtained by manual inspection of the static code analysis tools' outputs. The evaluation based on case studies allowed us to gauge the ability of static code analysis to detect security vulnerabilities in more complex settings.

The main contributions of this paper are as follows:

- The experimental evaluation was based on the Juliet test suite, a benchmark for assessing the effectiveness of static code analyzers and other software assurance tools. Previous evaluations based on Juliet either did not report quantitative results [13,14] or used very small sample of test cases related to vulnerabilities in C code only [15].
- Our study reports several performance metrics – accuracy, recall, probability of false alarm, and G-score – for individual CWEs, as well as across all considered CWEs. We used formal statistical testing to compare the tools in terms of performance metrics and determine if any significant differences exist. None of the related works included statistical testing of the results.
- In addition to the experimental approach, three widely-used open source programs were used as case studies. By combining experimentation with case studies, we were able to get sound experimental results supported by statistical tests and verify them in realistic settings.

Main empirical observations include:

- None of the selected tools was able to detect all vulnerabilities. Specifically, out of the 22 C/C++ CWEs, none of the three tools was able to detect six CWEs (i.e., 27%), seven CWEs (i.e., 32%) were detected by a single tool or a combination of two tools, and only nine CWEs (around 41%) were detected by all three tools. The results obtained when running the Java test cases were similar. Out of the nineteen CWEs, two CWEs (i.e., around 11%) were not detected by any tool, thirteen CWEs (i.e., 68%) were detected by a single tool or a combination of two tools, and only four were detected by all three tools. Note that 'detect' in this context does not means detecting 100% of 'bad' functions for that specific CWE. Rather, it means correctly classifying at least one bad function.
- The selected static code analysis tools did not show statistically significant difference in their ability to detect security vulnerabilities for both C/C++ and Java. In addition, the mean, median, and overall recall values for all tools were close to or below 50%, which indicates comparable or worse performance than random guessing.
- For C/C++ vulnerabilities, one of the tools had better performance in terms of probability of false alarm and accuracy than the other two tools. No significant difference existed for Java vulnerabilities.
- No statistically significant difference existed in the values of G-score (i.e., harmonic mean of the recall and 1-probability of false alarm) neither for C/C++ nor for Java vulnerabilities. (G-score combines in a single measure tools' effectiveness in detecting security vulnerabilities with their ability to discriminate vulnerabilities from non-flawed code constructs.)
- The experimental results related to tools' ability to detect security vulnerabilities were confirmed on three open source applications used as case studies.

The rest of the paper is organized as follows. Related work is presented in Section 2, followed by the background description of the structure and organization of the Common Weakness Enumeration (CWE) and the Juliet test suite in Section 3. The design of the experimental study, its execution, and the analysis per individual CWEs and across all CWEs, including the results of statistical tests are given in Section 4. Section 5 presents the findings based on the three open source case studies. The threats to validity are presented in Section 6, followed by the discussion of the results in Section 7 and concluding remarks in Section 8.

2. Related work

Despite the widespread use of static code analysis, only a few public evaluation efforts of static code analysis tools have been undertaken, and even fewer with a focus on detection of security vulnerabilities.