Static Analysis and Penetration Testing from the Perspective of Maintenance Teams

Mariano Ceccato
Fondazione Bruno Kessler
Trento, Italy
ceccato@fbk.eu

Riccardo Scandariato
Chalmers and University of Gothenburg
Gothenburg, Sweden
riccardo.scandariato@cse.gu.se

ABSTRACT

Static analysis and penetration testing are common techniques used to discover security bugs in implementation code. Penetration testing is often performed in black-box way by probing the attack surface of a running system and discovering its security holes. Static analysis techniques operate in a white-box way by analyzing the source code of a system and identifying security weaknesses. Because of their different nature, the two techniques report their findings in two different ways. This paper presents an exploratory study meant to determine whether a vulnerability report generated by a security tool based on static analysis is more or less useful than a report generated by a security tool based on penetration testing. The usefulness is judged from the perspective of the developers that have to devise a vulnerability-fixing patch. The initial results show an advantage when using penetration testing in one of the two cases we investigated.

CCS Concepts

•**Software and its engineering** → **Maintaining software;** •**Security and privacy** → *Penetration testing; Vulnerability scanners;* •**Social and professional topics** → *Software maintenance;*

Keywords

Software maintenance; Static analysis; Penetration testing

1. INTRODUCTION

Vulnerabilities are defects that expose a software system to a security threat. As an example of vulnerability, consider a piece of server-side code that executes the following SQL statement, which is built from some user-provided input: `"SELECT * FROM Users WHERE UserId = " + getRequestString("UserId")`. If a malicious user would provide the string `"myUserId; DROP TABLE Suppliers"` as input, the execution of the above statement would cause the deletion of an entire table from the database. This is an example of a SQL Injection attack, which is possible due to a vulnerability in the code. The example above is a hefty programming mistake, as a parametrized statement should have been used instead. In general, however, even small programming error can have a sizeable security impact. The anatomy of the Heartblead vulnerability in OpenSSL exposes this very issue all too well: two missing lines of code have compromised the integrity of the entire Internet in 2014 [13].

Two techniques that are widely used to localize vulnerabilities are static analysis and penetration testing. These techniques are supported by several commercial tools, like IBM AppScan or Klocwork, which automate the process of discovering the vulnerabilities to a large degree. Detecting a vulnerability, although challenging, is just one side of the coin. Once vulnerabilities are detected, their nature need to be understood by the maintenance teams and a fix need to be developed. The information about the discovered vulnerability provided by the tools to the maintenance teams is different depending on the discovery technique used. Typically, static analysis tools report the 'sink' of a vulnerability, i.e., the point where the harm to an information asset could potentially take place. In the example above, a static analysis tool would report the line of code containing the incorrect SQL statement. A penetration testing tool would report the entry point to the application where malicious user input could be used to cause harm. In the example above, the tool would report the URL and the malicious payload used as attack vector, which is the 'source' of the vulnerability. Hence, static analysis narrows down the location of the security issue, while penetration testing provides an executable scenario for the developer to replicate the problem.

The *contribution* of this paper is an *exploratory* controlled experiment involving a small number of master students as participants. The goal of the empirical study is to determine whether a vulnerability report generated by a security tool based on static analysis is more or less useful than a report generated by a security tool based on penetration testing. The usefulness of the report is judged from the perspective of the developers that have to devise a bug-fixing patch to close the vulnerability.

The rest of this paper is organized as follows. In Section 2, we specify the research questions. Section 3 describes the set-up of the experiment. In Section 4, we report the observed results. Section 5 discusses the threats to validity. In Section 6, we overview the related work. Finally, Section 7 presents the concluding remarks.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESEM '16, September 08-09, 2016, Ciudad Real, Spain

© 2016 ACM. ISBN 978-1-4503-4427-2/16/09…$15.00

DOI: http://dx.doi.org/10.1145/2961111.2962611

2. EXPERIMENTAL DEFINITION

In this study we use a representative tool for each of the investigated techniques: HP Fortify Source Code Analyzer (www.fortify.com) for static analisys and Burp Suite (portswigger.net) for penetration testing. These tools have been successfully used in the past to analyze the web applications used as objects in this study. The choice of the tools has been influenced by our previous study comparing static analysis to penetration testing with respect to vulnerability detection [10]. In this study we use the same tools and applications used previously, hence facilitating the aggregation of the experimental results in order to obtain an end-to-end assessment of the techniques.

For each identified vulnerability, the static analysis tool produces a report containing the line number of the affected file and additional contextual information, including (i) the type of vulnerability (e.g., SQL Injection or Cross-Site Scripting) and its estimated severity (on a 1 to 5 range).

The penetration testing report contains the entry point to the application (as URL), the malicious input to be used to replicate the attack, the type of vulnerability and a short description of the attack steps to follow.

Examples of the above-mentioned security reports are provided online in our replication package[4].

The *goal* of this study is to analyze the effect of said security reports with the *purpose* of evaluating how they support corrective maintenance. In particular, the experiment aims at answering the following two research questions:

- RQ1: Is there a difference between static analysis reports and penetration testing reports in supporting a *correct* removal of vulnerability defects?
- RQ2: Is there a difference between static analysis reports and penetration testing reports in supporting a *fast* removal of vulnerability defects?

3. EXPERIMENTAL SETTINGS

3.1 Context

The *participants* of this experiment are 12 second-year master students from the University of Trento, attending the *Security Testing* course. They are fluent in the programming language used in the study (i.e., Java).

The *objects* of this experiment are two similar open-source Java web systems, Pebble and Roller. Both application are fully-featured blogging platforms and support multiple users, customization of blogs via templates, threads of comments from blog visitors, and so on. `Pebble` (pebble.sourceforge.net) contains about 56 thousand lines of code (KLOC). We used version 2.6.3. `Roller` is (roller.apache.org) is very similar to Pebble in terms of size (62 KLOC) and functionality. We used version 5.0.1.

3.2 Tasks and Experimental Design

We selected particular releases of Pebble and Roller that contain some known and documented vulnerabilities, that the participants are supposed to fix. Applications have been subject to static analysis (via Fortify SCA) and penetration testing (via Burp Suite) to collect security reports. For the experiment, we selected two vulnerabilities per application, among those that are detected by both static analysis and penetration testing.

Table 1: Experimental design. Order of tasks (A-D) and treatments (SA, PT).

Lab	System	Group 1	Group 2	Group 3	Group 4
1	Pebble	T_A^{SA},T_B^{SA}	T_B^{SA},T_A^{SA}	T_A^{PT},T_B^{PT}	T_B^{PT},T_A^{PT}
2	Roller	T_C^{PT},T_D^{PT}	T_D^{PT},T_C^{PT}	T_C^{SA},T_D^{SA}	T_D^{SA},T_C^{SA}

As shown in Table 1, throughout the course of the experiment, each participant completes 4 tasks, consisting of fixing a vulnerability (T_A-T_D) using the reports generated by either the static analysis (SA) technique or the penetration testing (PT) technique. The tasks are carried out in two lab sessions, one for each system. Each participant works on both systems and uses both techniques (i.e., treatments). The order of the tasks and treatments is randomized.

3.3 Measures and Hypotheses

The independent variable of the experiment is the type of report used to identify the vulnerability (i.e., generated via static analysis or penetration testing). The measures we collect are:

- *Correctness:* For each task we assess whether the participant correctly fixed the vulnerability;
- *Productivity:* We measure the *time* spent by a participant to complete a task and we compute the productivity as the ratio between the number of correctly completed tasks and the time spent on them.

To track the time, participants are asked to use *toggle*[1], a web application for time keeping. Participants are also asked to submit the edited code to the authors, who verified if the fix was indeed correct or not. The success of each maintenance task has been assessed by one of the authors, who inspected the source code to understand the performed changes and ran a test to verify that the security defects had been removed.

Based on the measures chosen, we can formulate the following null hypotheses:

- $\mathbf{H0}_C$: There is no difference in the *correctness* of vulnerability fixing tasks when supported by static analysis and penetration testing;
- $\mathbf{H0}_P$: There is no difference in the *productivity* participants working on vulnerability fixing tasks when supported by static analysis and penetration testing.

3.4 Questionnaires: Measure Co-Factors and Interpret Results

We also asked the participants to answer a profiling pre-questionnaire and a feedback post-questionnaire. The pre-questionnaire collects information about the abilities and experience of the involved participants. This is important to analyse the effect of ability and experience in the successful completion of fixing tasks. Among the co-factors that can potentially affect the results, we measured and analyzed the following ones:

[1]https://toggl.com/

- *The System* to fix: as detailed above, we considered two systems: Pebble and Roller. Although they are comparable in terms of features and complexity, participants may perform differently on different systems;
- *The Task*: for each application, two fix tasks have been defined. Although they have been identified to be quite similar in terms of complexity, performance could be different on different tasks;
- *The Task Order*: there could be a learning effect between the first and the second task and this could influence the performance of participant;
- *The Experience*: The proficiency of a participant with Java, their knowledge of security and their industrial experience might influence accuracy and productivity of fixing tasks;

The post-questionnaire collects information about how each security report helped participants to understand the vulnerability defect and to fix it. The questionnaires are available online [4].

3.5 Experimental Procedure

Training. Before each experiment, participants attended a lecture to recall the notion of both static analysis and penetration testing, and to clarify the experimental procedure.

Before the actual experiment, participants have been involved in a warm-up session where they have been asked to perform some fixing tasks (similar to those in the study) on a sample web application. The objective was to make participants confident with the experimental environment, and to make sure that they are aware of the kind of tasks that they were supposed to perform during the experimental labs. For the experiment, participants used a personal computer equipped with with Apache Tomcat and Mysql sever. The purpose of the training was also to test that their working environment were correctly configured, in order to start the experimental labs with a fully functional setup.

Material. We distributed the following material to our participants:

- A short textual documentation of the system they had to fix, including how to compile, deploy and start it in the local web server;
- The source of the system to fix;
- The description of the two tasks (i.e., the two vulnerabilities to be fixed)(see Table 1).

Execution. The experiment was carried out according to the following procedure. Participants had to: (i) Fill the profiling questionnaire; (ii) Read the application description; (iii) Compile the application with Eclipse, deploy and run the application (Pebble or Roller) to familiarise with it; (iv) Read the first vulnerability report; start the timer; perform the task and fix the vulnerability; stop the timer; (v) Read the second vulnerability report; start the timer; perform the task and fix the vulnerability; stop the timer; (vi) After completing all tasks, create an archive containing the modified source code and send it to the experimenter by email, together with the time report; and (vii) Complete a post-experiment survey questionnaire.

During the experiment, the authors monitored the laboratory to prevent collaboration among participants, and to check that participants properly followed the experimental procedure.

3.6 Analysis Procedure

The difference between the output variable (*Correctness* and *Productivity*) obtained under different treatments (static analysis vs. penetration testing) is tested using non-parametric statistical tests, assuming significance at a 95% confidence level (α=0.05). So, we reject the null-hypotheses when p-value<0.05. All the data processing is performed using the R statistical package [9].

To analyse the differences in terms of *Correctness*, we looked at the frequencies of correct/wrong tasks and we used a test on categorical data, because the tasks can be either correct (completed successfully) or incorrect (completed unsuccessfully). In particular, we used Fisher's exact test [6][2].

To test the differences in *Productivity* we perform the one-tailed Mann-Whitney U test on all samples [11].

To quantify the magnitude of differences among the two labs, we used two kinds of effect size measures, the *odds ratio* for the categorical variable *Correctness* and the Cliff's delta effect size [7] for *Productivity*. The effect size is computed using the effsize package [12].

An odds ratio of 1 indicates that the condition or event under study is equally likely in both groups (participants using SA and those using PT). An odds ratio greater than 1 indicates that the condition or event is more likely in the first group. An odds ratio less than 1 indicates that the condition or event is less likely in the first group.

For independent samples, Cliff's delta provides an indication of the extent to which two (ordered) data sets overlap, i.e., it is based on the same principles of the Mann-Whitney test. Cliff's Delta ranges in the interval $[-1, 1]$. It is equal to $+1$ when all values of one group are higher than the values of the other group and -1 when reverse is true. Two overlapping distributions would have a Cliff's Delta equal to zero. The effect size is considered small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$ [5].

The analysis of co-factors is performed using a General Linear Model (GLM). It consists in fitting a linear model of the *dependent* output variables (Correctness or Productivity) as a function of the *independent* input variables (all factors, including the treatment, i.e., the vulnerability detection tool). A general linear model allows to test the statistical significance of the influence of all factors on the output variable. In case of relevant factors, interpretations are formulated by visualising the associated interaction plots.

4. RESULTS

In this section, we present and analyse the results of the study.

4.1 Analysis of Correctness

First of all, we consider the *correctness* of the tasks, which consist of removing code-level security defects. Table 2 reports the number of correct and wrong tasks delivered by

[2]Fisher's exact test is more accurate than the χ^2 test for small sample sizes, which is another possible alternative to test the presence of differences in categorical data. The same analysis was conducted by [3].

Table 2: Analysis of Correctness (Fisher test and Odds ratio).

	Static Analysis		Pen. Testing			
	Correct	Wrong	Correct	Wrong	P	OR
Pebble	5	7	2	8	0.38	2.72
Roller	3	3	8	0	0.05	0.00

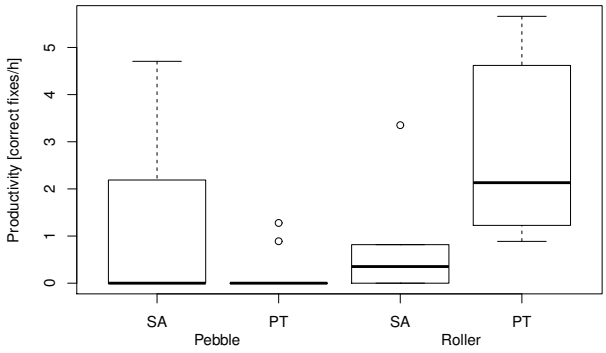

Figure 1: Box plot of Productivity.

participants[3], organized according to the kind of security report they received (static analysis or penetration testing). Different *Systems* are reported separately in distinct rows.

For *Pebble*, there is no clear trend, because the number of incorrect fixes is consistently higher than the number of correct tasks both with static analysis and with penetration testing. Conversely, a trend seems to be present when working with *Roller*. In this second case, when participants are provided with static analysis reports the number of correct and incorrect fixes are the same (3 versus 3 tasks), while when they are provided with penetration testing reports the number of correct tasks is higher (8 versus 0 tasks).

We apply Fisher's test to check if the observed trend is statistically significant. The absence of a clear trend for Pebble is confirmed by the test, as no statistically significant difference is achieved. In the case of Roller, the p-value ("P" column) is still not <0.05, but the odds ratio is very small ("OR" column). Possibly, the lack of significance is due to the limited number of participants. Thus, we can answer to RQ_1 in this way:

The correctness when fixing vulnerabilities (on Roller) is higher when the participants are provided with vulnerability reports generated by penetration testing. However, the difference is not statistically significant.

4.2 Analysis of Productivity

In Figure 1, we report the box-plot of productivity, measured as the number of correct tasks divided by the time required to elaborate them. Since we pre-filter security reports, productivity does not include detection of false positives. While for Pebble, the productivity of the two approach looks similar, on Roller the productivity of participants working with penetration testing looks higher than the productivity of those who work with static analysis.

[3]The sum of tasks does not amount to the number of participants, because not all the participants attended all the labs not all tasks have been delivered.

Table 3: Analysis of Productivity (Mann-Whitney test).

	Static Analysis		Pen. Testing			
	Mean	Std dev	Mean	Std dev	P	ES
Pebble	1.19	1.77	0.22	0.47	0.22	0.35
Roller	0.81	1.30	2.81	1.93	**0.02**	-0.79

Table 4: Co-factors of Correctness (general linear model).

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	0.1077	0.4930	0.22	0.8286
TreatmentSA	-0.0914	0.1674	-0.55	0.5894
SystemRoller	0.4451	0.1722	2.58	**0.0152**
TaskB	0.0551	0.1676	0.33	0.7446
Task.order	0.1163	0.1681	0.69	0.4948
java.exp	0.2064	0.1906	1.08	0.2882
security.exp	-0.0221	0.2071	-0.11	0.9157
work.exp	0.0007	0.2318	0.00	0.9976

Table 3 reports descriptive statistics and statistical analysis of productivity. While on Pebble no statistical significance is observed, on Roller the difference in productivity is statistically significant (p-value <0.05) with a *large* effect size ("ES" column). In particular, the average productivity with penetration testing is more than trice than the productivity of static analysis (2.81 vs 0.18 correct tasks per hour). So we can answer to RQ_2 in this way:

The productivity when fixing vulnerabilities (on Roller) is higher when the participants are provided with vulnerability reports generated by penetration testing.

4.3 Analysis of Co-Factors

In this section, we report about the co-factors that could have influenced the dependent variables of our experiment (Correctness and Productivity).

Table 4 reports the analysis of co-factors of *Correctness* obtained by applying the general linear model method. Statistically significant coefficients are in boldface. The particular *System* used in the experimental session influenced significantly the Correctness of vulnerability-fixing tasks.

From the interaction plot in Figure 2 we can notice that vulnerability-fixing tasks on Roller (first lab) are consistently completed with higher correctness than on Pebble (second lab). This effect is particularly evident for Penetration Testing. This could be interpreted as an asymmetric learning effect among successive labs: when participants already practised with Static Analysis in the first lab, in the second lab they achieved higher correctness with Penetration Testing. Conversely, previous practice with Penetration Testing in the first lab does not help in completing correct fixing tasks when working with Static Analysis in the second lab.

Previous practice in fixing vulnerabilities with the support of static analysis yields higher correctness when subsequently working with the support of penetration testing.

As shown in Table 5, we then consider the relation be-

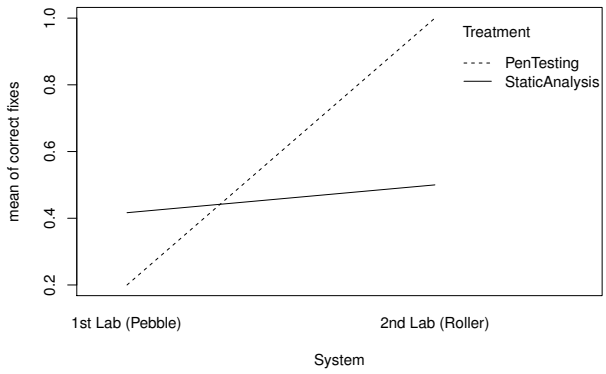

Figure 2: Interaction plot of system and treatment in correctness.

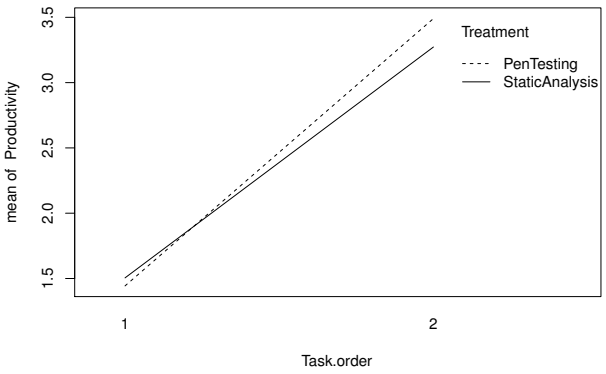

Figure 3: Interaction plot of task order and treatment in productivity.

Table 5: Co-factors of Productivity (general linear model).

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	2.9656	2.9207	1.02	0.3339
TreatmentSA	-0.6948	0.8360	-0.83	0.4253
SystemRoller	-0.6659	0.8381	-0.79	0.4454
TaskB	-0.0440	0.7716	-0.06	0.9556
Task.order	2.2303	0.7945	2.81	**0.0186**
java.exp	-0.9411	1.1161	-0.84	0.4188
security.exp	-1.9509	0.9000	-2.17	0.0554
work.exp	-1.2714	1.2865	-0.99	0.3463

tween co-factors and the *Productivity*. We observe that the *Order* of the tasks influences productivity in a statistically-significant way (p-value <0.05).

From the interaction plot in Figure 3, we can see that both for Static Analysis and for Penetration Testing in the 2^{nd} task productivity is consistently higher. Not surprisingly, this can be explained as a learning effect between successive tasks in the same system (while using the same technique). During the first task, the productivity is possibly lower because participants spent some time to familiarize with the new System under maintenance. The acquired knowledge helps to complete the second task with higher productivity.

The productivity in fixing vulnerability defects (on the same system and with the same type of vulnerability report) improves along subsequent fixing tasks.

4.4 Analysis of Post-Questionnarie

At the end of the study, we asked the participants to fill in a short questionnaire, which is available online [4]. Only 8 participants provided feedback.

The respondents agreed that a static analysis report makes it easier to understand the cause of a vulnerability with respect to a penetration testing report (6 respondents in favor of SA vs 2 respondents for PT). In particular, the respondents mentioned in an open question that a static analysis report gives an advantage when it comes to localizing the portion of the code that needs to be fixed.

The answers to the questionnaire did not show any preference in the respondents for what concerns the actual correction of the vulnerabilities. None of the two types of report seems to simplify the devising of a fix for the vulnerability (2 respondents prefer static analysis report, 2 prefer penetration testing, 4 are undecided).

5. THREATS TO VALIDITY

We identified the main threats to the validity that can affect our results [14] as conclusion, internal, construct, and external validity threats.

To limit the *conclusion validity* threats (relationship between the treatment and the outcome) we used objective statistical tests. In particular, the Fisher's exact test is more accurate than the χ^2 test for small sample sizes and Mann-Whitney test is non-parametric, very robust and sensitive [8].

Among the *internal validity* threats (additional factors that may affect an independent variable) we considered learning and fatigue effects. The raining sessions was meant to limit learning effect, splitting the experiment in two labs was indented to limit fatigue. The adopted design allowed measure the learning among consecutive tasks as a co-factor.

To limit the *Construct validity* threats (relationship between theory and observation), we tried to measure time and correctness as objectively as possible. Moreover, subjects were not aware of the study hypotheses, and they were told they would not be evaluated on their performance.

Eventually, the treats to the *external validity* concern the generalization of the findings. Our study was limited to two tools for security review (Fortify SCA and Burp Suite), chosen mainly for uniformity with previous studies [10], two web systems and just students as participants. All in all, young software developers that might be assigned maintenance tasks are not so different from last year master students in Computer Science. Clearly, further studies with more systems and different participants are needed to confirm or contradict the results from this study.

6. RELATED WORK

To the best of our knowledge, this is the first study investigating the effect of different types of vulnerability reports (e.g., entry points vs. sinks) in the process of understanding and fixing the corresponding security issues.

Instead, the two techniques used in this study to generate the vulnerability reports have been investigated extensively.

Scandariato et al. [10] conducted an empirical study with 9 master students to compare static analysis and penetration testing as tools for security code review. Results show that when a participant uses static analysis, on average they detect more vulnerabilities then when using penetration testing, however with a comparable number of false alarms. Additionally, static analysis scores a higher productivity, meaning that participants who use static analysis report a higher number of correct results per hour than participants who use penetration testing.

Both Austin and Williams [2] and Antunes and Viera [1] performed a case study where security experts applied the two techniques to identify the vulnerabilities of several applications. Both studies concur in saying that static analysis yields more results (in terms of actual vulnerabilities found) but at the cost of a lower precision (due to more false alarms).

7. CONCLUSION

Previous studies supported the thesis that static analysis allows a faster detection of more security defects than penetration testing (possibly at the cost of more false positives). In this paper, we focused on the subsequent problem: after they have been successfully detected, defects should be also removed. We presented an exploratory controlled experiment to compare static analysis and penetration testing from a corrective maintenance point of view.

Preliminary results suggest that those participants who work on security reports that are produced by the penetration testing technique might fix security defects with higher correctness and in faster way.

Our interpretation of this result is that penetration testing represents a better support for maintenance, because it gives richer information to developers. In fact, a static analysis report mostly indicates the point in the code that is considered defective. So, a developer has to figure out by theirselves how it can be exploited by an attack. While in some cases this is enough to remove a defect quickly, in complex cases more information could be crucial. In particular, penetration testing provides an executable scenario to replicate the dynamics of an attack. Moreover, the test case can be executed *after* patching the code, allowing the developer to quickly assess the correctness of change in a reliable way.

Despite more replications being required to confirm this finding, this exploratory study was important to assess our experimental design and to formulate suggestions to improve it. For instance, we noticed *mortality*, i.e. not all the participants attended the second lab. Probably, the least skilled students perceived the fist lab as very demanding, so they decided to skip the second lab (as participation was on a voluntary basis). As a result, the second lab was attended by less participants but with more homogeneous background, i.e. consistently more skilled. Lower variability in the participants' profile could be one reason for higher significance to the second lab. In future replications, we will enforce a preliminary filtering of participants, e.g. with a small test, in order to select them based on their actual skills.

More replications of this experiment are in fact required to confirm our findings and overcome the limitations of the present study. The limited number of participants (12 in our study) should be increased and participants with more diverse profiles (last year master students, in this study) should be involved.

Nonetheless, to the best of our knowledge, even if just with an exploratory objective, this is the first controlled experiment aimed at comparing static analysis and penetration testing as support for security maintenance.

Acknowledgment

The research leading to these results has received funding from European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 609734.

8. REFERENCES

[1] N. Antunes and M. Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of SQL injection vulnerabilities in web services. In *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2009.

[2] A. Austin and L. Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, '11.

[3] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19(4):1040–1074, 2014.

[4] M. Ceccato and R. Scandariato. Replication package: Static analysis vs penetration testing. http://selab.fbk.eu/ceccato/replication-packages/SAvsPT-replication-package.zip, 2016.

[5] J. Cohen. *Statistical power analysis for the behavioral sciences (2nd ed.)*. Lawrence Earlbaum Associates, Hillsdale, NJ, 1988.

[6] J. L. Devore. *Probability and Statistics for Engineering and the Sciences*. Duxbury Press; 7 edition, 2007.

[7] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.

[8] H. Motulsky. *Intuitive biostatistics: a nonmathematical guide to statistical thinking*. Oxford University Press, 2010.

[9] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[10] R. Scandariato, J. Walden, and W. Joosen. Static analysis versus penetration testing: A controlled experiment. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 451–460, Nov 2013.

[11] D. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures (4th Ed.)*. Chapman & All, 2007.

[12] M. Torchiano. *effsize: Efficient Effect Size Computation*, 2015. R package version 0.5.5.

[13] C. Williams. Anatomy of openssl's heartbleed: Just four bytes trigger horror bug. http://www.theregister.co.uk/2014/04/09/heartbleed_explained/, 2014.

[14] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.