Teachers and Students Learn Cyber Security: Comparing Software Quality, Security

Shlomi Boutnaru
School of Education
Tel Aviv University
ISRAEL
boutnaru@mail.tau.ac.il

Arnon Hershkovitz
School of Education
Tel Aviv University
ISRAEL
arnonhe@tauex.tau.ac.il

ABSTRACT

In recent years, schools have added cyber security to their computer science curricula. While doing so, existing teachers are trained with the new material. In this study we explore differences in teachers' and students' learning of cyber security, implementing a multi-way, data-driven approach by comparing measures of software quality and security. Our findings suggest that teachers' codes have a better quality than the students', and that the students' codes are slightly better secured than the teachers'. The findings imply on the teachers benefit from their prior knowledge and experience. Also, findings shed light on the difference between quality and security in today's programming paradigms.

Keywords

Cyber security, code metrics, software quality, software security, hierarchical clustering, decision tree.

1. INTRODUCTION

Educational systems worldwide often adopt "hot", emerging topics to their curricula. Usually, teachers from within the system are quickly trained to teach the new material. The current study aims on understanding the differences in teachers' and students' learning of new material in the case of cyber security (also known as computer security, IT security), that is, the practice of protecting computer systems from unauthorized access, change or destruction. This understanding might contribute to the pedagogy of teaching new materials, as well as to teacher development, shedding light on previous findings regarding novices' and experts' programing knowledge [1,9]. Studies in this field had usually used various measures to assess experts' and novices' programming skills and knowledge, mostly based on qualitative data collection (mainly programming-related tasks and interviews), rather than on assessment of programs written. Our approach is to use automatically extracted software quality and security features.

Explicit metrics for measuring different dimensions of code quality have been developed from the late 1960s, shortly after the development of the then-new domain of software engineering [3,5]. Metrics were defined with their automation in mind. As setting a numerical value for metrics might be time-consuming, subjective and expensive, "one would prefer for large programs an automated algorithm which examine the program and produces a metric value" [3; p. 596]. In recent years, EDM/LA methods have been used along with software metrics, allowing complex structure-based features, as well as adding variables measuring student-computer interaction [e.g., 2,7]. In this paper, we take an EDM approach, together with a comprehensive set of software metrics for both quality and security. We use both quality and security metrics.

As the main purpose of the current study is to explore the way novice students and experienced teachers learn cyber security – most commonly involves learning Python – we chose to focus on software metrics derived from the standards of that language. Therefore, we used the *Style Guide for Python Code (PEP 8)*[1] as the basis to the metrics. (These metrics were not accessible to the participants, and were only assessed in retrospect.) As there is no yet a Python standard for security, and based on the similarities between Python and C++, we based our quality metrics on the *C++ Secure Coding Standard*, by Carnegie Mellon University's CERT[2]. Both these standards are widely used in code evaluation

2. METHODOLOGY

Participants in this study were 31 11th- and 12th-grade students from two Israeli high-schools, 17-18 years old; and 18 high-school computer science teachers from different parts of Israel, 31-53 years old. Two of the latter were the teachers of the participant students.

Each of the participant teachers attended one of two cyber security programs (June 2012 – March 2013 or September 2013 – January 2014). The participant students took a curriculum-based cyber security program, as part of their computer science studies, during 2012/3 school year. Solutions (in Python) to tasks assigned during these programs were collected and analyzed.

Overall, 109 source files were collected – 68 teachers' and 41 students'. The teachers were assigned with four different exercises (writing a UDP echo server, a basic TCP command server, an advanced TCP command server, and a Web server); the students were assigned with three different tasks (writing a UDP echo server, an advanced TCP command server, and a TCP-based Chat).

Number of actual participants in the analysis was decreased to 17 teachers (with 60 files) and 15 students (with 27 files), as sometimes teachers/students worked in pairs or triples. When the same pair/triple had submitted all of the exercises, we arbitrarily left only one of the group in the data set. When pairs/triples had changed over the course of the program, we arbitrarily assigned chose only one representative for each submission.

[1] This guide was co-authored by Python creator, Guido van Rossum. Available on http://legacy.python.org/dev/peps/pep-0008 [accessed 3 May 2014].

[2] Available at https://www.securecoding.cert.org [accessed 3 May 2014].

2.1 Feature Engineering

Features were evaluated at the code-level; for the participant-level analysis (descriptive statistics, hierarchical tree), feature values were averaged across each participant's source files.

2.1.1 General Features (6 features)

For each source file, the general features are the following:

- *Number of Statements* (code size);
- *Number of Comment Lines*;
- *Documentation Rate* (= *Comment Lines* / *Statements*);
- *Number of Lines* (statements, comments and empty lines);
- *File Name Length* [characters; excluding the extension .py];
- *File Name Meaningfulness* (1 – file name is not meaningful at all; 2 – partly meaningful; 3 – very meaningful).

2.1.2 Quality Features (20 features)

These were automatically extracted by running *Pylint* (http://pylint.org), a common source code bug and quality checker for Python which follows PEP 8 style guide. Pylint defines five categories of standard violations/errors:

1. **Convention (C; 18 measures)**. Recommendations of software structural quality. Convention measures indicate standard violations (e.g., function/variable name does not match a regular expression defined in the standard);

2. **Warning (W; 61 measures)**. Python-specific problems that do not follow Python's best practices and may cause run time bugs (e.g., an unused import from wildcard import);

3. **Error (E; 32 measures)**. Probable bugs in the code that relate to general programming concepts (e.g., the use of a local variable before its assignment);

4. **Refactor (R; 15 measures)**. A "bad smell" code (derived from the term refactoring. the process of restructuring existing computer code without changing its external behavior). Such violation might be indicated when a function takes too many variables as input;

5. **Fatal**. This are errors in Pylint processing and not in the source file itself, hence were excluded.

Pylint scans the code and returns a list of measures for which violations/errors found, along with their count (we consider 0 for the measures that were not triggered by Pylint). Based on Pylint output, the following features were computed for each category:

- *Mean Count (C/W/E/R)* – mean count of violations/errors across all the category's measures.

- *Normalized Mean Count (C/W/E/R)* – *Mean Count* divided by code size (*Number of Statements*);

- *Rate of Triggered Measures (C/W/E/R)* – number of triggered measures divided by total number of measures;

- *Triggered Category (C/W/E/R)* – indicating whether at least one measure of it was triggered.

- *Normalized Triggered Category (C/W/E/R)* – *Triggered Category* divided by code size (*Number of Statements).*

2.1.3 Security Features (6 features)

These features – extracted using scripts written by the research team – are binary, indicating whether the relevant mechanism was implemented (1) or not (0).

- *Input Validation* (the process of ensuring that a program operates on clean, correct and expected input);
- *Anti-Spoofing Mechanism* (spoofing attack is a situation in which an attacker masquerades as another entity by sending specially crafted data that seems as it was send from the legitimate source);
- *Bound Checking* (checking whether a variable is within some range before it is used);
- *Checking for Errors* (not checking return codes for errors can cause logical security bugs/crashing of the program that can cause Denial of Service attacks);
- *Sensitive Data Encryption*;
- *Client-Side-Only Security* (when the server relies on protection mechanisms placed on the client side only);

Among these, *Client-Side-Only Security* is the only one for which a 0-value denotes a good behavior.

3. RESULTS

3.1 Descriptive Statistics

3.1.1 General Features

Means of four general metrics are significantly different between students and teachers: *Number of Statements*, *Number of Lines*, *File Name Length*, and *File Name Meaningfulness*; on average, students' programs were longer than the teachers', and teachers' file names were longer and more meaningful than the students'. The difference regarding code size (*Number of Statements* and *Number of Lines*) might hint that teachers have a better grasp of the concept of programming with Pyhton, as this language allows far fewer lines compared to other languages. No significant differences were found between the means of the two documentation-related features. Average *Documentation Rate* was 0.1, which shows a reasonable documenting practice in Python. Results are summarized in Table 1.

Table 1. Descriptive statistics, t-test results for *general features* (one decimal place representation unless mean<0.1)

Variable	Mean (SD) N=32	Mean (SD), Teach. N=17	Mean (SD), Stud. N=15	t(30)[a]
Number of Statements	51 (28.3)	40.5 (19.7)	62.9 (32.3)	2.3*, df=22.6[b]
Number of Comments	6.1 (7.4)	5.5 (7.8)	6.8 (7.0)	0.5
Documentation Rate	0.1 (0.1)	0.1 (0.2)	0.1 (0.1)	-0.4
Number of Lines	56.9 (29.6)	45.4 (23.8)	69.7 (30.9)	2.5*
Name Length	10.8 (5.1)	12.9 (4.0)	8.4 (5.2)	-2.8**
Name Meaning.	1.3 (0.5)	1.6 (0.4)	0.9 (0.5)	-4.3**

* $p<0.05$, ** $p<0.01$. [a] Unless otherwise stated, df=30.

[b] Levene's test for equality of variance resulted with a significant result, hence equal variances not assumed.

3.1.2 *Quality Features*

Means of eight quality metrics of convention (C) and warning (Q) type are significantly different between students and teachers (see Table 2): *Mean Count*, *Normalized Mean Count*, *Rate of Triggered Measures* – for both C and W; *Trigged Category W*, and *Normalized Triggered Category C*. On average, students had more convention- and warning-type violations than the teachers. As convention guidelines improve code readability and maintainability, these differences might indicate on the teachers' smoother migration to programming in a new language.

Table 2. Descriptive statistics, t-test results for *quality features* (one decimal place representation unless mean<0.1 or difference needs to be shown)

Variable	Mean (SD) N=32	Mean (SD), Teach. N=17	Mean (SD), Stud. N=15	t(30)[a]
Mean Count C	72.3 (56.7)	40.8 (28.8)	108.0 (60.0)	4.0**, df=19.6[b]
Mean Count W	56.9 (70.9)	20.7 (37.6)	97.8 (78.4)	3.5**, df=19.5[b]
Mean Count E	1.4 (1.5)	1.3 (1.0)	1.5 (2.0)	0.5, df=19.5[b]
Mean Count R	0.2 (0.3)	0.1 (0.4)	0.2 (0.3)	0.6
Normalized Mean Count C	0.11 (0.04)	0.09 (0.04)	0.13 (0.03)	3.6**, df=26.7[b]
Normalized Mean Count W	0.02 (0.03)	0.01 (0.02)	0.04 (0.03)	2.9**, df=21.6[b]
Normalized Mean Count E	– [c]	– [c]	– [c]	0.05
Normalized Mean Count R	– [c]	– [c]	– [c]	1.1
Rate of Triggered Measures C	0.4 (0.1)	0.3 (0.1)	0.4 (0.1)	4.5**
Rate of Triggered Measures W	0.05 (0.04)	0.03 (0.03)	0.07 (0.04)	3.5**
Rate of Triggered Measures E	0.01 (0.01)	0.01 (0.01)	0.01 (0.01)	-0.1, df=21.6[b]
Rate of Triggered Measures R	0.01 (0.02)	0.01 (0.02)	0.01 (0.01)	0.7
Triggered Category C	1 (0)	1 (0)	1 (0)	N/A
Triggered Category W	0.7 (0.4)	0.6 (0.4)	0.9 (0.3)	2.7*, df=28.1[b]
Triggered Category E	0.4 (0.3)	0.4 (0.3)	0.4 (0.4)	-0.6, df=23.8[b]
Triggered Category R	0.2 (0.3)	0.1 (0.3)	0.2 (0.3)	1.2
Normalized Triggered Category C	0.03 (0.02)	0.04 (0.02)	0.02 (0.01)	-3.1**
Normalized Triggered Category W	0.02 (0.01)	0.02 (0.02)	0.02 (0.01)	0.6
Normalized Triggered Category E	0.01 (0.01)	0.01 (0.01)	0.01 (0.01)	-1.1
Normalized Triggered Category R	– [c]	– [c]	– [c]	0.2

* $p<0.05$, ** $p<0.01$. [a] Unless otherwise stated, df=30.

[b] Levene's test for equality of variance resulted with a significant result, hence equal variances not assumed. [c] Value < 0.01.

Pay attention to the opposite direction difference between students and teachers in *Normalized Triggered Category C*. This is a direct result of *Triggered Category C* getting a 1-value for both students and teachers and of *Number of Statements* being larger for students that it is for teachers (*Normalized Triggered Category C* is a ratio of these two variables).

3.1.3 *Security Features*

Overall, both teachers and students showed low levels of implementing security mechanisms, as summarized in Table 3. Both implemented no security mechanism regarding *Anti-Spoofing Mechanisms* and *Sensitive Data Encryption*. As for *Input Validation* and *Checking for Errors* – on average, students statistically significantly implemented more mechanisms than teachers regarding these features. It might be that teachers, learning from their own fresh experience, emphasized these subjects to their students.

As for *Client-Side-Only Security*, recall that a 0-value for this feature denotes a proper security implementation. As seen in Table 3, teachers' mean value for this feature was 0; however, as they had barely implemented any security mechanism, this value cannot be interpreted as a good practice. The students, with relatively a high mean value (0.5), demonstrate poor security design that is focused mostly at the client-side.

Table 3. Descriptive statistics, t-test results for *security features* (one decimal place representation unless mean<0.1)

Variable	Mean (SD) N=32	Mean (SD), Teach. N=17	Mean (SD), Stud. N=15	t[a]
Input Validation	0.06 (0.17)	0 (0)	0.13 (0.23)	2.3*, df=14.0
Anti-Spoofing Mechanism	0 (0)	0 (0)	0 (0)[b]	N/A
Bound Checking	0.10 (0.20)	0.04 (0.12)	0.17 (0.25)	1.9, df=20.1
Checking for Errors	0.18 (0.35)	0.04 (0.12)	0.33 (0.45)	2.5*, df=15.8
Sensitive Data Encryption	0 (0)	0 (0)	0 (0)	N/A
Client-Side-Only Security	0.21 (0.42)	0 (0)[c]	0.5[2] (0.52)	3.3**, df=11.0

* $p<0.05$, ** $p<0.01$. [a] Levene's test for equality of variance resulted with a significant result, hence equal variances not assumed. [b] For this case, N=12. [c] For this case, N=16.

3.2 Hierarchical Clustering

A hierarchical cluster analysis was performed, using Ward's method for clustering by Pearson correlation. Features were standardized using Z-scores before clustering. Analysis was computed using SPSS 18. The results, presenting two clusters, are strikingly clear: One cluster (N=9) holds only teachers, the other (N=23) holds all the 15 students and 8 additional teachers.

Examining features' mean values between the two clusters adds to previous student-teacher comparison. The most striking difference is in refactor (R) features, which did not show up earlier: a) *Rate of Triggered Measures R*, with t(df=20.5)=2.2, at $p<0.05$; b) *Triggered Category R*, with t(df=24.4)=2.2, at $p<0.05$; and c) *Normalized Triggered Category R*, with t(df=20.0)=2.4, at $p<0.05$. Levene's test for equality of variance resulted with significant results, hence equal variances were not assumed. Means in the teachers-only cluster were lower than in the mostly-students cluster (i.e., the teachers had demonstrated better security design). Hence, it might be that teachers are more experienced than students in regulating their own programming and recognizing seemingly-suspicious code.

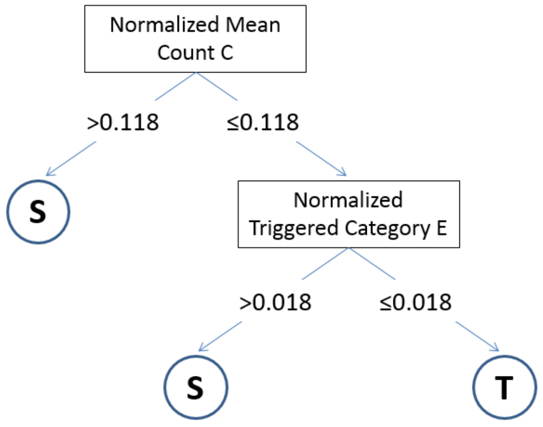
Bound Checking was also found significantly different between the two clusters, with t(df=19.1)=2.2, at $p<0.05$ (here also, equal variances were not assumed as for Levene's test significant result). Mean value for the teachers-only cluster is lower than the mostly-students cluster, in line with previous findings.

Some features' means were statistically significantly different when compared between teachers and students, but not different when comparing between clusters: *Number of Lines*, *Number of Statements*, and *Normalized Triggered Category C*. As *Normalized Triggered Category C* is the ratio of *Triggered Category C* – for which all of the participants got a value of 1 – to *Number of Statements*, and as *Number of Statements* and *Number of Lines* are highly correlated– with Pearson's r=0.983, at $p<0.01$ – it is enough to look at *Number of Statements*; therefore, we might conclude that the original difference in *Number of Statements* might have been arbitrary.

3.3 Prediction Model

Finally, we built a classifier at the code-level, predicting whether a program was submitted by a student or a teacher. 87 source codes were used. We ran a Decision Tree algorithm, using RapidMiner 5.3 (default parameters), with a manual forward feature selection. The best model found (with LOOCV kappa=0.751) is relatively simple, having only two features – *Normalized Mean Count C*, and *Normalized Triggered Category E* – three leaves and a total height of two (see Figure 1). It highlights the already known difference in convention violations between teachers and students. However, it adds an interaction of a convention feature with an error-related feature; the latter did not show up earlier. This interesting result suggests that students and teachers that are relatively good in convention-keeping might still pay attention differently to probable bugs.

Figure 1. Best prediction model (S=Student, T=Teacher)



4. DISCUSSION

Overall, we found that the teachers did better than students with regards to software quality metrics of a new programming language. However, the very existence of violations/errors in these metrics may hint that the teachers had struggled with the new material just like novices do. These findings support preliminary findings about computer science teachers being "regressed experts" when coping with new material [4]. Supporting computer science learners in improving their code might be relatively easily, by measuring software quality and security while writing the code and enabling a contextual feedback; this might produce a better code and, more importantly, a better learning [cf. 6, 8]. Popular IDEs (Integrated Development Environments) already provide integration with tools like Pylint (e.g., Emcas, VIM, Eclipse, Komodo, WingIDE, and gedit), so using such software might ease the measuring task.

As our results suggest, codes with higher software quality are not necessarily better secured. Overall, teachers' codes were of higher quality comparing to the students' codes, however with regards to the measurable security features – the opposite was true. If we want future software engineers to implement appropriate security mechanisms, we need to educate them in secure programming while teaching them programming practices.

5. REFERENCES

[1] Bateson, A.G., Alexander, R.A., and Murphy, M.D. 1987. Cognitive processing differences between novice and expert computer programmers. *Int. J. Man-Machine Studies, 26*(6), 649-660.

[2] Blikstein, P. 2011. Using learning analytics to assess students' behavior in open-ended programming tasks. *In Proceedings of the 1st International Conference on Learning Analytics and Knowledge (Banff, AB)*, 110-116.

[3] Boehm, B.W., Brown, J.R., and Lipow, M. 1976. Quantitative evaluation of software quality. *In Proceedings of the 2nd International Conference on Software Engineering (San Francisco, CA)*, 592-605.

[4] Liberman, N., Ben-David Kolikant, Y., and Beeri, C. 2012. "Regressed experts" as a new state in teachers' professional development: lessons from Computer Science teachers' adjustments to substantial changes in the curriculum. *Computer Science Education, 22*(3), 257-283.

[5] McCall, J.A., Richards, P.K., and Walters, G.F. 1977. *Factors in software quality*. General Electric Company, Technical Report RADC-TR-77-369.

[6] Truong, N., Roe, P., and Bancroft, P. 2005. Automated feedback for "fill in the gap" programming exercises. *In Proceedings of the 7th Australasian Computing Education Conference, (Newcastle, NSW, Australia)*, 117–126.

[7] Vihavainen, A., Luukkainen, M., and Kurhila, J. 2013. Using students' programming behavior to predict success in introductory mathematics course. *In Proceedings of the 6th International Conference on Educational Dada Mining (Memphis, TN)*, 300-303.

[8] Wang, T., Su, X., Ma, P., Wang, Y., and Wang, K. 2011. Ability-training-oriented automated assessment in introductory programming course. *Computers & Education, 56*(1), 220-226.

[9] Wiedenbeck, S. 1985. Novice/expert differences in programming skills. *Int. J. Man-Machine Studies, 23*(4), 383-390.