The Journal of Systems & Software 185 (2022) 111150

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Understanding security vulnerabilities in student code: A case study in a non-security course☆

Tolga Yilmaz *, Özgür Ulusoy

Department of Computer Engineering, Bilkent University, 06800, Ankara, Turkey

ARTICLE INFO

Article history:
Received 5 March 2021
Received in revised form 14 November 2021
Accepted 18 November 2021
Available online 14 December 2021

Keywords:
Secure coding education
Source code analysis
Data mining
Vulnerability analysis

ABSTRACT

Secure coding education is quite important for students to acquire the skills to quickly adapt to the evolving threats towards the software they are expected to create once they graduate. Educators are also more aware of this situation and incorporate teaching security in their respective fields. An effective application of this is only possible by cultivating the teaching and learning perspectives. Understanding the security awareness and practice of students is useful as an initial step to create a baseline for teaching methods and content. In this paper, we first survey to investigate how students approach security and what could motivate them to learn and apply security practices. Then, we analyze the source code for 6 semesters of coding assignments for 2 tasks using a source code vulnerability analysis tool. In our analysis, we report the types of vulnerabilities and various aspects between them while incorporating the effect of student grades. We then explore the lexical and structural features of security in student code using data analysis and machine learning. For the lexical analysis, we build a classifier to extract informative features and for the structural analysis, we utilize Syntax Trees to represent code and perform clustering in terms of structural features where clusters themselves yield different vulnerability levels.

1. Introduction

Teaching security aspects of programming has gained importance (Taylor et al., 2013; Chi et al., 2013) as today's software requires more connectivity and is more complex while it is inevitable for students as fresh graduates to come across vulnerabilities affecting the security, reliability, and maintainability of their code early in their careers. Therefore, it may be wise enough for them to learn how to bridge the gap towards more secure code as early as possible (Taylor et al., 2013; Cabaj et al., 2018).

With the inclusion of cybersecurity in ACM-IEEE computer science curricula in 2013 (Joint task force on computing curricula, association for computing machinery (ACM) and IEEE computer society, 2013) and the guidelines in cybersecurity curricula by ACM-IEEE Joint Task Force in 2017 (Joint Task Force on Cybersecurity Education, 2017), educators have been advised to include security concepts in the classrooms and initiated such efforts (Taylor and Kaza, 2016). However, educators need to establish an initial understanding of where students struggle first in their coding to teach them about secure coding. A starting point would be to evaluate the security level of code, which is generally done through vulnerability analysis.

Vulnerability analysis is vastly studied in the literature and there are many different techniques (Liu et al., 2012) from predefined rules to machine learning approaches (Ghaffarian and Shahriari, 2017) that help catch certain textual and syntactic patterns in code that are indicative of vulnerabilities. To communicate the analysis results and their additional properties such as the discovered vulnerabilities and their severity score, the security community developed standardized ways such as Common Weakness Enumeration (CWE), Common Vulnerability Exposure (CVE), Common Vulnerability Scoring System (CVSS), and OWASP Top 10. Open-source and commercial code analyzers also integrate most of these standards and some of them use their own classification to augment their findings.

In this work, we investigate a data set comprised of student code from different semesters for two different web application development tasks for the third year "Database Management Systems" course, which covers theoretical foundations of the topic and has assignments as applications of Database Systems and is mandatory for Computer Science students. Security-related courses are occasionally offered as electives and taken by mostly fourth-year students. In the described development tasks of the course, students are required to implement a web application with a database using PHP/MySQL (and Java to create the initial database). They were required to publish their application online.

☆ Editor: W.K. Chan.

* Corresponding author.
E-mail addresses: tolga.yilmaz@bilkent.edu.tr (T. Yilmaz), oulusoy@cs.bilkent.edu.tr (Ö. Ulusoy).

https://doi.org/10.1016/j.jss.2021.111150
0164-1212/

We analyze the submissions in terms of security using data analysis techniques to identify lexical and structural variations.

We believe that analyzing and learning security aspects from the code of university students bring two immediate benefits. Firstly, students focus on implementing only the functionality, are not slightly interested in implementing other features (Taylor et al., 2013). Our grading in the assignments was based on functionality as well. Thus it can be expected in our scenario that students would not implement security features on their own unless required explicitly, even though the nature of the assignment would present a good learning opportunity for such features. Therefore, this may indicate an analysis of a purely security-indifferent approach, providing a worst-case analysis of security. Secondly, having different implementations of the same task can help differentiate and prioritize improvement points from the teaching perspective.

To summarize, we investigate the following research questions (RQs):

1. How do students assess the security of their own code, and is there a gap between their own understanding and the actual results?
2. What are the most common security vulnerabilities in university students' code?
3. How does increasing grade (or functionality) affect security in the code?
4. Is it possible to encourage students towards more secure code using external motivation such as extra points?
5. Is it possible to extract more information on security vulnerabilities by leveraging lexical and structural features of the code?
6. What are the security concepts for teaching university students to bridge the gap towards more secure coding?

As means to answer the research questions, we perform the following:

- We use a dataset that consists of completely security-unaware code thus may provide a worst-case analysis of security.
- We provide an analysis of security vulnerabilities in student code on two programming tasks over 6 semesters which generates the data size and time-span for exploration.
- We investigate the types of vulnerabilities in terms of various aspects. We show that the number of vulnerabilities increases proportionally to the increasing grade. However, we believe this is in fact about the increased functionality.
- We provide the results of a user study, namely a questionnaire about secure coding. Students report a total dismissal of security in their practices.
- We encouraged students to secure their code by adding a bonus section to the assignment. We show that extra points can engage the students into securing their code and the students who submit bonus parts have fewer vulnerabilities in their code.
- We bring lexical and structural perspectives on student code using exploratory machine learning techniques, that can help towards more secure code.

The paper is structured as follows. In Section 2, we review some of the studies related to our work. Section 3 describes the methodology and the data set. Section 4 lists the results of our survey. Section 5 gives the details of our analysis based on security vulnerabilities. Section 6 is about the machine learning techniques employed with the lexical and structural features of student code. In Section 7, we elaborate on the results and Section 8 lists the limitations of this study.

2. Related work

Cyber security education. The importance of cybersecurity education and its integration into computer science curricula attract the attention of researchers. Cabaj et al. (2018) review the evolution of cybersecurity education. Švábenský et al. (2020) review the research and provide a taxonomy in cybersecurity education. Jones et al. (2018) address what students should learn in schools and share the results of a survey targeted at cybersecurity professionals. Parekh et al. (2018) identify the cybersecurity concepts that should be taught using Delphi processes.

Secure coding education: Student perspective. Some studies explore student perspectives in security. Acar et al. (2016) address the impact and the use of information sources with computer science students and professional developers. The results show that not experience but prior security training had a positive effect on code security. In Jain et al. (2014), it is shown that a set of developers selected from computer science students tend to prefer a more privacy-preserving API. Taylor and Kaza (2011) provides a checklist-based model to teach students security aspects. Hooshangi et al. (2015) try to understand the student mindset and behavior in which they write offensive and defensive code in a course on security. They observe that students who wrote good defensive software also wrote good offensive software but not vice-versa. Cappos and Weiss (2014), similarly use a teaching method for security that has a series of assignments to code attack and defense tasks. Williams et al. (2014) point out some of the issues for beginner programmers.

Code analysis and machine learning techniques. There are many aspects to source code analysis and it has been a relevant research field since code has existed (Binkley, 2007). Although the research can mainly be divided into two parts, dynamic and static analysis, our research is done via means of static analysis. An aging but still relevant and comprehensive survey on dynamic program analysis is given by Cornelissen et al. (2009).

One primary use case of static analysis is to detect vulnerabilities (Liu et al., 2012). We can refer to Pistoia et al. (2007) for vulnerability detection using static code analysis techniques for the period before machine learning methods become popular. Many recent studies utilize machine learning and deep learning techniques in this field to detect vulnerabilities. Allamanis et al. (2018) give an overview of machine learning research on code. They provide a taxonomy of papers that includes code generating models, representational models, pattern mining models. Code generating models deal with models that can produce code out of training examples. Representational models are different approaches to the problem of representing source code for machine learning models to be used in various tasks. For instance, Allamanis et al. (2016) use a method to represent code using tokens to summarize code as method names. Alon et al. (2019, 2018) utilize embeddings from Abstract Syntax Tree (AST) paths for general-purpose use. Pattern mining models try to identify recurring patterns and extract meaning from them. For example, Wang et al. (2016) use AST representation to predict bugs in source code and White et al. (2016) use deep learning to identify code clones. Russell et al. (2018) use Convolutional Neural Networks and Recurrent Neural Networks to extract features from code tokens and using these to predict vulnerabilities. The base of this work intersects with representational models and pattern mining models.

3. Methodology

RQ.1 is about students' security perception of their own code. To learn about that, we ask related questions in the form of a survey. We report the results in Section 4.

RQ.2, 3, and 4 investigate the types of vulnerabilities created by students' code. We automatically analyze the code using Sonarqube Community Edition,[1] which is a static code analyzer that performs rule-based analysis for various languages. We chose it for its accessibility and popularity, which is helpful because any further research using other software would have a chance to perform comparisons as long as a community edition exists. We use the default rule-set of the scanner. We only scan student-authored code. The results of the analysis are given in Section 5.

RQ.5 asks whether it is possible to identify additional information by leveraging lexical and structural features of code. The lexical approach is about what the code contains, statements, variables. For this, we utilize a classifier based on Logistic Regression. We then explore the most informative lexical bits in the code, extracted from the classifier. Structural analysis on the other hand requires breaking the code into smaller parts and then utilize a representation that can then be learned from. In our approach, we convert code into Abstract Syntax Trees and are able to perform clustering based on this and show that structure also has a value in terms of security. We explain the details of both lexical and structural studies in Section 6.

The data set contains implementations of two different tasks. Task A is a simple university internship application and Task B is a simple banking application. Tasks are given to students in different semesters. Submissions are graded based on how well the requirements are implemented and not in terms of security. Our data set consists of 6 semesters of student code, containing more than 400 submissions.

4. Student survey

In this section, we explore RQ.1: "*How do students assess the security of their own code, and is there a gap between their own understanding and the actual results?*"

To learn about their perception regarding the security of their code, we added a bonus section to the assignment in the Spring 2019 semester. In the bonus part, we asked students to write a short report in which they comment on how secure their application is. We notified them that they could write about the possible weaknesses in their code, how they could improve their application code. We also asked them to answer the following questions: "To what extent do you consider security in your code?" and "What are the reasons for that in your opinion?".

The bonus part could earn the students up to 50%. (Maximum without the bonus was 100 and with the bonus, it was 150 for a single semester). This way we intended to identify whether adding additional points would encourage more student engagement. Indeed, different types of encouragement have been shown to increase cybersecurity awareness on pedagogical examples for different audiences (Amo et al., 2019; Schumacher and Welch, 2002).

4.1. Survey results

Out of 39 students who submitted their code, 11 of them also did the bonus section. This means extra points could engage 28.2% of students who submit their assignments. Here we report the aggregated survey results.

First of all, as we list in Table 1, students list some of the security vulnerabilities that may occur in their applications. Almost all students reported SQL Injection as their primary security problem. The least reported one was Path Traversal.

[1] https://www.sonarqube.org

Table 1
Vulnerabilities reported by students.

Security Vulnerabilities	Ratio of Students
SQL Injection	0.91
Data Sanitization/Input Validation	0.64
Hardcoded passwords and credentials	0.36
Sensitive Data Exposure	0.36
Encryption	0.36
Authentication and Authorization	0.27
Insecure Logging	0.27
Cross Site Scripting	0.18
Link Misuse	0.18
Path Traversal	0.09

Table 2
Reasons security is neglected, as reported by students (a student can report multiple reasons).

Reason	Ratio of Students
Time concern	0.36
Not required in the assignment	0.27
Did not respond	0.54

Table 3
Students' own security level assessment of their code.

Security Level Perception	Ratio of Students
Low	0.81
Medium	0.18
High	0

Students also reported their thoughts on why security is often neglected in this or generally any other Computer Science assignment (see Table 2). The primary reason was the time concern. It is understandable for students not to spend too much time on security features. The other reason was that security features are not typically required in computer assignments, and they do not want to spend extra time on something that would not bring them explicitly more points even if they were required to publish their application online in the assignment. Nance et al. (2012) also mention that students should be required to implement security, otherwise they are not motivated. Finally, students mostly assess their code to be highly insecure, where there is a group that thinks that their code provides medium-level security (see Table 3). Whether their expectation and the actual results match will be studied in the following sections.

5. Analysis

5.1. Overview

We first investigate the number of vulnerabilities per student. As Fig. 1(a) shows there exist few students with a large number of vulnerabilities whereas there is a long tail of students with fewer vulnerabilities. In Fig. 1(b) vulnerabilities per student per semester shows an increasing trend from the per-task perspective. However, this does not lead us to a conclusion without more data.

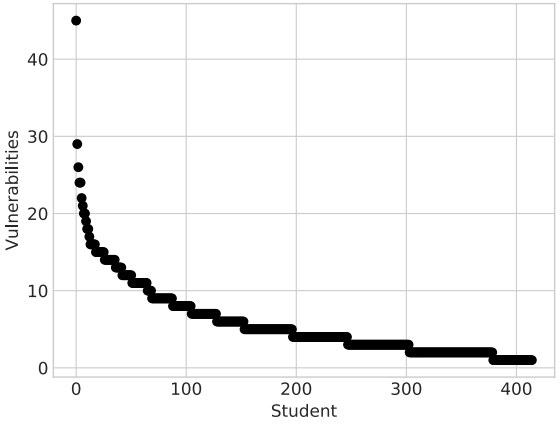
5.2. Understanding the types of vulnerabilities

Here, we target RQ.2:"*What are the most common security vulnerabilities in university students' code?*"

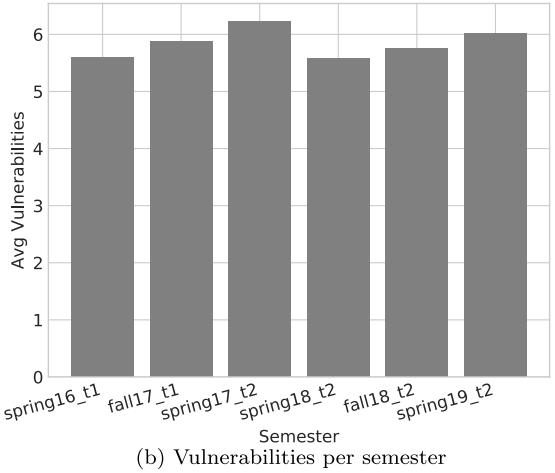
Understanding the most common types of vulnerabilities that exist in the student code is important to prioritize the issues that help to deliver better security training. For this matter, we investigate this in terms of three different aspects (i.e., taxonomies) of security problems: CWEs, OWASP Top 10, and vulnerabilities

Table 4
Vulnerabilities per CWE Type.

Type	Definition	#
CWE-259	Use of Hard-coded Password	829
CWE-20	Improper Input Validation	761
CWE-564	SQL Injection: Hibernate	751
CWE-943	Improper Neutralization of Special Elements in Data Query Logic	751
CWE-489	Active Debug Code	714
CWE-315	Cleartext Storage of Sensitive Information in a Cookie	23
CWE-117	Improper Output Neutralization for Logs	17
CWE-532	Insertion of Sensitive Information into Log File	17
CWE-778	Insufficient Logging	17
CWE-521	Weak Password Requirements	15
CWE-311	Missing Encryption of Sensitive Data	14
CWE-614	Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	14



(a) Vulnerabilities per Student



(b) Vulnerabilities per semester

Fig. 1. Overview.

Table 5
Vulnerabilities per OWASP Top 10 Type.

Type	Description	#
A2:2017	Broken Authentication	843
A3:2017	Sensitive Data Exposure	810
A1:2017	Injection	753
A10:2017	Insufficient Logging & Monitoring	10

Table 6
Vulnerability per Sonar Security Category.

Vulnerability	#
auth	844
sql-injection	751
insecure-conf	737
others	63
log-injection	17
dos	4
weak-cryptography	1

raised by Sonar software. This helps us understand the security problem from different granularity levels. CWEs allow a standardized way to categorize and communicate common vulnerabilities in software and hardware systems and it is organized in a tree hierarchy to provide a highly granular vulnerability taxonomy. OWASP Top 10, on the other hand, provides a big picture view with an emphasis on the most common vulnerabilities from a web application perspective. Both are used in the industry together for a comprehensive understanding of the security situation. There also exist mappings from some CWE codes to OWASP Top 10 categories. In addition, using a taxonomy such as Sonar can also help provide a non-standard view for analysts. In this work, the vulnerabilities with respect to OWASP Top 10, CWE, and Sonar are derived by Sonarqube using various rule sets for each. The rule sets are updated over time by the Sonarqube.

In terms of the most common CWE, we have the following: CWE-798: Use of Hard-coded Credentials, CWE-259: Use of Hard-coded Password, CWE-20: Improper Input Validation, CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'), CWE-943: Improper Neutralization of Special Elements in Data Query Logic, CWE-564: SQL Injection: Hibernate, CWE-489: Leftover Debug Code (see Table 4). Clearly, the most common problems are the use of critical data in the code and not being able to sanitize input and output against malicious operations.

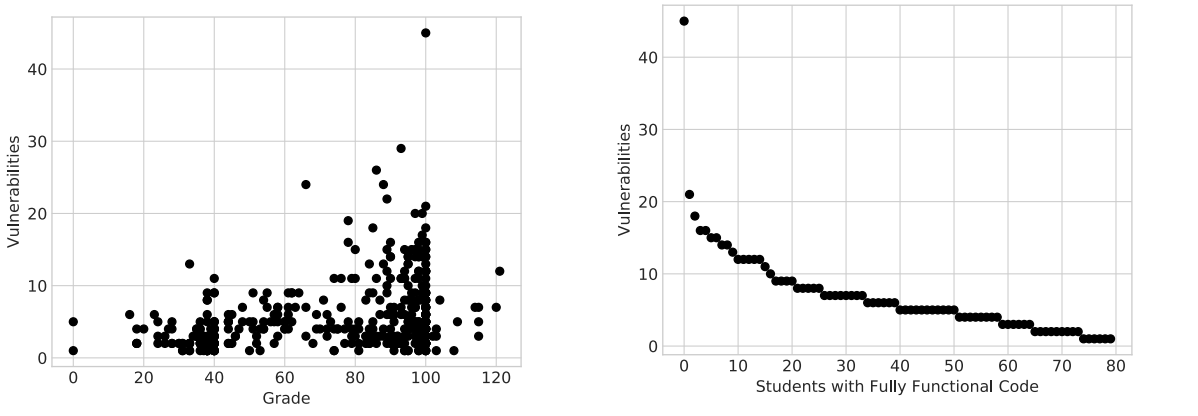
OWASP vulnerabilities show a similar pattern (Table 5); the most common vulnerabilities are A2 Broken Authentication and Session Management, A3 Sensitive Data Exposure, A1 Injection. Sonar software also yields a similar pattern; Auth, SQL-injection, and Insecure configuration are the most common issues (Table 6).

It is interesting to see that students list most of these issues without needing to use any software that analyzed their code. The students who answered the survey questions were aware of the possible vulnerabilities in their code. It should also be noted that most of the found issues result from hurried programming, hence "time concern" as reported by the students.

5.3. Effect of student grade on security

RQ.3: "*How does increasing grade (or functionality) effect security in the code?*" and RQ.4: "*Is it possible to encourage students towards more secure code using external motivation such as extra points?*" are explored in this section.

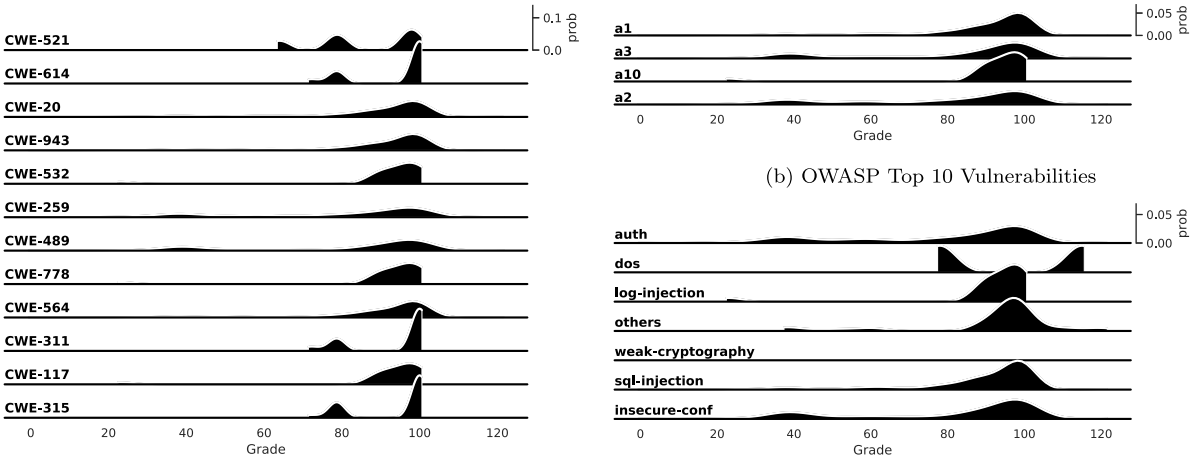
It may be expected to think that students with better grades may incorporate more security into their code, but the results



(a) Vulnerabilities and Student Grades

(b) Vulnerabilities of Fully Functional Assignments (Grade >= 100)

Fig. 2. Grades and Vulnerabilities.



(a) CWE Vulnerabilities

(b) OWASP Top 10 Vulnerabilities

(c) Sonar Security Vulnerabilities

Fig. 3. Distribution of Vulnerability Types w.r.t. Grades.

show otherwise (Fig. 2(a)). We see a clear pattern of increasing vulnerability count with increasing grades. In the evaluation of the assignments, a grade represents how well the functionality is implemented, and 100 means a perfect score and in these assignments, security functionality was not explicitly required. As a result, better grades indicate more functionality and complexity thus more probability to create security vulnerabilities. It can be seen in the figure that some students have better grades than 100. These are the students who submitted the bonus section and some of them upgraded their code for better security as the figure shows fewer vulnerabilities for these students as an exception.

We also investigate the number of vulnerabilities for students with perfect score. The distribution we see in Fig. 2(b) resembles the distribution of all students we depict in Fig. 1(a). This indicates that doing the same job is possible with different security levels.

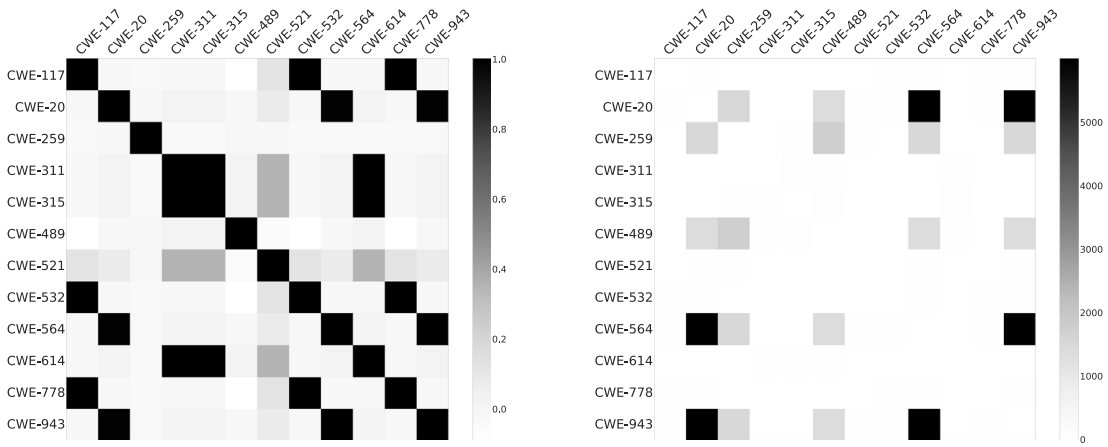
To understand the distribution of vulnerabilities not just by counts but rather with types, we do an analysis from the three perspectives as we did for the overall data set; CWE, OWASP Top 10, and Sonar vulnerabilities. This time we show the occurrence probabilities of different vulnerability types with respect to student grade in Fig. 3. It can be seen that the probability of vulnerability increases with higher grades for various types of vulnerabilities, especially after the grade of 80. In addition, vulnerabilities become more diverse as the grade increases and some of the vulnerabilities start to occur at higher grades. Finally, we see the same effect of bonus submission with lower probabilities for vulnerabilities.

5.4. Relation between vulnerabilities

Understanding the relationship between vulnerabilities is important in order to determine which ones to tackle together. To achieve that we calculate correlation (using Kendall's method) and co-occurrence of vulnerability types for CWE categories per submission as CWE is the most granular one. Fig. 4 shows the two metrics for CWE types. The metrics are able to identify directly related types (e.g., CWE-117: Improper Output Neutralization for Logs and CWE-532: Inclusion of Sensitive Information in Log Files) and not directly related but somehow connected types (e.g., CWE-521: Weak Password Requirements, CWE-311: Missing Encryption of Sensitive Data) and even seemingly unrelated ones (e.g., CWE-521: Weak Password Requirements, CWE-614: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute).

6. Mining patterns in code

We try to find answers to RQ.5.: "*Is it possible to extract more information on security vulnerabilities by leveraging lexical and structural features of the code?*" in this section.



(a) Correlation of CWE types (b) Coocurrence of CWE types

Fig. 4. Relation between CWE Types.

We deepen our analysis by making sense of patterns in student code, which can help us identify the root causes and lead us to better security training for students. In this part, we focus on the web application code written in PHP. We approach this problem from two perspectives. First, we use a lexical perspective by looking at the sentence/statement structure of code line by line or segment by segment and words used by the students. Second, we utilize a bird-eye view of the structure of code by its building blocks which is represented with Abstract Syntax Trees in our case. Below, we explain the details of the two approaches and the results obtained.

6.1. Lexical interpretation

In order to attain a lexical interpretation of code, we build a simple classifier using logistic regression. For this, we first compose a data set consisting of the vulnerable code segments we extract from the student code accompanied by the same number of random segments that are not found vulnerable. We use a simple bag of words model with 10-fold stratified cross validation, repeated 10 times. We obtain 94.4% accuracy. We obtain this otherwise not achievable accuracy by having a codebase of submissions that solve the same two tasks hence expected to have similar vulnerabilities. Pang et al. (2015) report a similar accuracy score in their work which predicts vulnerabilities using a similar model to ours but with Support Vector Machines. Using this classifier we are able to learn about the most informative features pointing at vulnerabilities. The most informative vulnerability predictors as tokens were: *printstacktrace, password, mysql_query, mysqli_query, dbpassword, setcookie, alert, drivermanager, ini_set, mypassword, getconnection, preparestatement, scanner, db_password, mysqli, display_errors*. These keywords are generally used in input/output operations and a lack of validation in these is causing the vulnerabilities.

6.2. Syntactic interpretation

In order to comment on the structural patterns of vulnerable code, we first need to extract its building blocks. A building block in programming languages may correspond to many parts of the program such as functions, conditional and loop blocks. Abstract Syntax Trees (ASTs) provide a standardized way of representing the structural composition of source code in a tree format containing nodes with associated data. In this part, we use a parser that generates the ASTs of the submissions, and using these ASTs we intend to learn the structural indicators of vulnerabilities and generate advice towards more secure code.

Utilizing ASTs to learn about vulnerabilities has been studied in the literature (Mou et al., 2014). One of the most important aspects of this problem is how we represent the code and use the suitable learning algorithm that learns from the respective representation. The most recent work on representational modeling focuses on learning deep neural networks by learning embeddings from ASTs and paths extracted from ASTs (Alon et al., 2019, 2018). A path is defined as a sequence of nodes from a starting node to another node. Then these paths are transformed to be fed to a learning algorithm. In our work, we use AST-paths for vulnerability analysis but without any deep representations. In addition, most of the work focuses on functions as building blocks. In our work, we use files as a whole as building blocks since we need to evaluate vulnerability level per submission and the PHP language at hand is a scripting language with which a lot of functionality can be achieved without function blocks. Therefore, we extract ASTs of individual files using the PHP-Parser tool,[2] thus a submission is a collection of ASTs each corresponding to a file. The AST can be extracted in JSON format where nodes and their relationships are preserved in a tree structure. Every possible path in the AST of a submission is treated as a feature to be used in a learning algorithm, similar to a bag-of-words model except it is bag-of-paths.

In this part of our work, after treating each submission as a collection of paths between every possible node, we investigate the possible formation of similarities between submissions and their relation to the presence of vulnerabilities. We use Latent Semantic Analysis (LSA) and K Means as an exploratory framework to identify clusters of submissions that exhibit different vulnerability levels. Latent Semantic Analysis is used as a dimensionality reduction technique with the help of Singular Value Decomposition (SVD). With SVD, it is possible to reduce the data into manageable and meaningful (informative) components, which makes it possible to analyze data with large dimensions (e.g., AST paths). In this work, we form a bag-of-paths model with the AST paths using TF-IDF vectorization and apply LSA and use the first three components from LSA as these capture most of the variation in the data. After reducing the data to three components, we are able to apply K-Means clustering to observe

[2] https://github.com/nikic/PHP-Parser

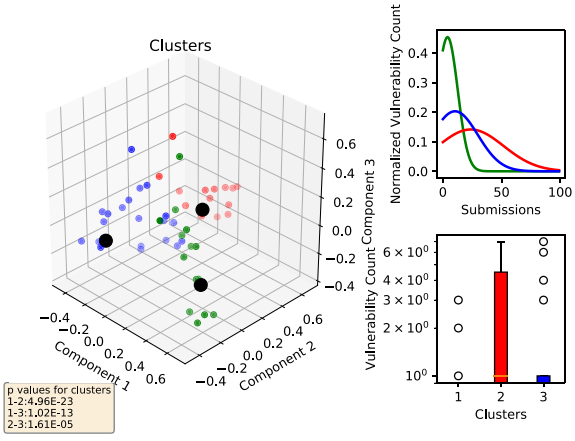


Fig. 5. Clustering submissions for Task A with grades between 90 and 100 into 3 Clusters.

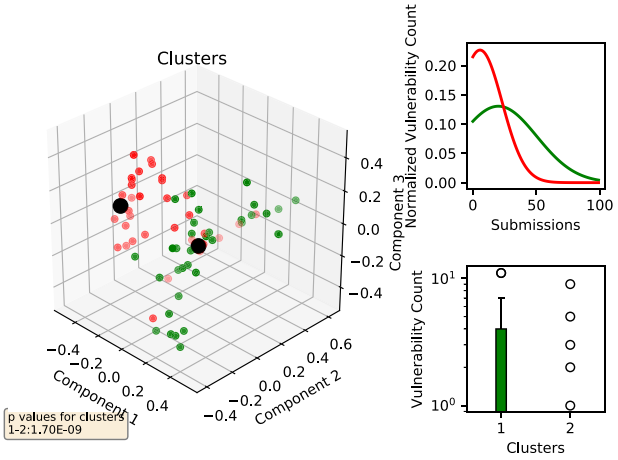


Fig. 6. Clustering submissions for Task B with grades between 90 and 100 into 2 Clusters.

clusters that have different vulnerability distributions as we test the difference using the Kolmogorov–Smirnov test with which we calculate the corresponding p-values where $p \ll 0.05$, which denotes that the samples drawn from the clusters are statistically different. The optimal number of clusters are determined using the elbow technique based on the inertia of the model for various number of clusters. We do not use deep learning models as our data set size is not suitable for such a setting and our analysis is exploratory.

Our empirical analysis shows that clustering with respect to structural features could also yield different vulnerability levels. We cannot deduct a causal relationship between structural variations and vulnerability levels as vulnerabilities can be quite complex and cannot be detected by merely looking at the text and they may exist in well-written code. However, such a correlation starts to appear according to our findings for submissions with grade 80 or more. We chose to depict the results of submissions with grades between 90 and more because these would be the most functional and comparable since it would be hard to conceptualize an analysis with a submission with grade 30 (hence limited functionality) with a submission with a grade of 100 (full functionality). Comparing these would not yield the desired outcome since they are expected to be complexity-wise quite different. For instance, in Fig. 5, we illustrate the 3 clusters of submissions from Task A with grades between 90 and 100 that exhibit different vulnerability distributions with the corresponding p-value. Fig. 7 depicts the structure of centroids. We can see different structural irregularities and complexity in these submissions that achieve the same task at similar functionality levels. Cluster centroid 1 (Fig. 7(a)) depicts a submission that started with relatively regular structure but could not maintain it and cluster centroid 2 (Fig. 7(b)) has the most irregular structure whereas cluster centroid 3 (Fig. 7(c)) has a more regular structure but with repetitions.

The same observations are available for Task B with 2 clusters and Grades between 90–100 as shown in Fig. 6. Centroids for this in Fig. 8 show radical differences in structure.

In addition, in order to test the idea of a correlation between structural features for student code and vulnerability level, we further train a classifier using Logistic Regression with Stochastic Gradient Descent training where vulnerability level is the target. For this, we first normalize the number of vulnerabilities to three categories: Low, Medium, and High. After sampling down the majority of instances of the low category to eliminate bias, and applying 10-fold stratified cross validation which is repeated 10 times, we obtain an accuracy of 61.5%, which denotes a correlation between vulnerability level and structural features, compared to a 33.3% accuracy if there was no correlation. We note that we achieve this accuracy for submissions with a grade of 80 or more. We did not observe any correlation for submissions with grades lower than this. One intuitive explanation would be that the submissions below this grade did not reach the maturity level or functionality level so that they could be clustered into groups. The structural distance between these assignments is also high. Furthermore, we list some examples of informative AST paths as determined by the classifier along with corresponding code segments from some of the assignments in Figs. 9 and 10.

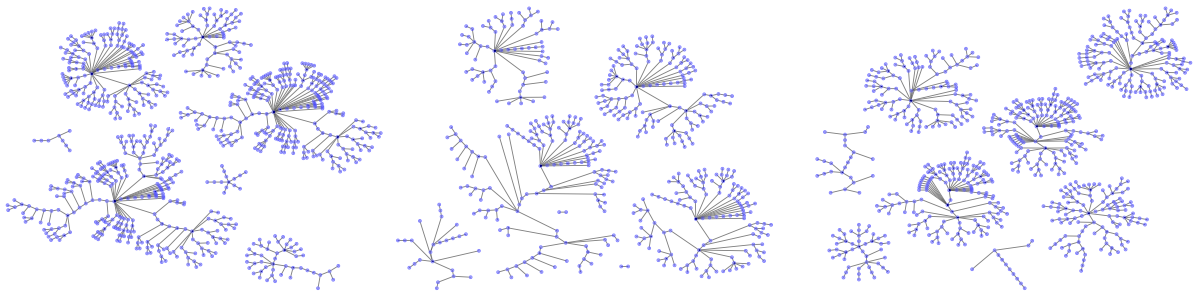
In Figs. 11 and 12, we give examples on how following a structured coding technique may help securing the code. Fig. 11 is taken from a cluster with highly irregular code structure but still functional with more number of vulnerabilities whereas Fig. 12 is taken from a cluster with less. We can go through the possible important points for Fig. 11:

- Between lines 22–25, the author tries to mitigate the possibility of SQL-Injection. However, by trying to achieve it himself as opposed to using known techniques or framework, he again fails.
- Lines 22–29 contain database credentials, which should not be stored in the file in plain-text form.
- Database credentials is repeated in lines 58–61, pointing out that duplicating code with vulnerabilities also duplicates the vulnerability.
- Page tries to do many things; including keeping track of information, trying to print out rows, binding buttons to other files.
- There is an highly risky authentication scheme seen between lines 3–8, 43, 48. Similar is also done with various session variables in lines 50, 54

For Fig. 12, we can state the following:

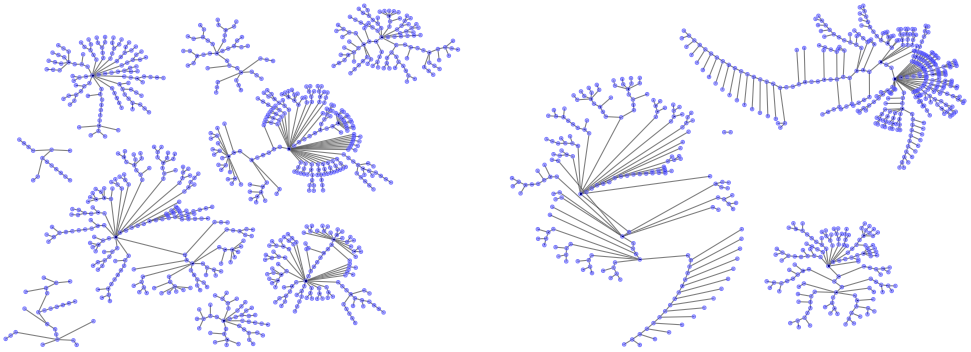
- Page tries to do one thing: Login
- Still no protection from SQL-Injection at line 11.
- Defers other functionality (e.g., database config) to other modules at line 2.

This shows a glimpse of what code looks like underneath. It should be stated that in our examples all other modules/pages follow the similar pattern and as the code becomes more modular and duplication free to start with, it contains potentially less vulnerabilities.



(a) Cluster centroid 1 (b) Cluster centroid 2 (c) Cluster centroid 3

Fig. 7. Task A Cluster Centroid ASTs as Graphs.



(a) Cluster centroid 1 (b) Cluster centroid 2

Fig. 8. Task B Cluster Centroid ASTs as Graphs.

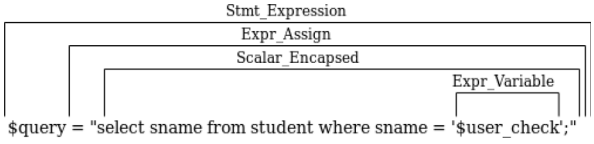


Fig. 9. Example 1 of an Informative AST path and related code snippet.

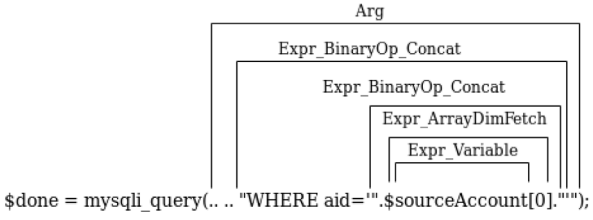


Fig. 10. Example 2 of an Informative AST path and related code snippet.

7. Discussion

The notion of teaching secure coding to students can be approached from two perspectives; the students' learning perspective and the teachers' teaching perspective.

Students establish a good framework via the survey study which identifies the main problems in terms of lack of security in their code. Firstly, one can expect the code developed by students to contain many vulnerabilities, which is what we discovered as well, one does not expect students to identify possible vulnerabilities as well as they did in our survey without using any analysis techniques prior to their submission of their code. Secondly, with some extra encouragement, which is the "bonus" part in our case, students can significantly improve the security levels of their code. Finally, students provide the real reason as "time concern" they neglect security. These three aspects create a promising situation from the learning perspective.

The teaching perspective should take into account the learning aspects. In other words, it should encourage more student engagement via various means by incorporating security into regular course work or provide secure coding as a separate course in order to solve the "time concern" problem of students. In addition, our analysis may help create better content for addressing "what to teach" aspect of teaching security. We could incorporate teaching best practices of various paradigms which are reportedly been neglected in computer science curricula (Taylor et al., 2013; Lalande et al., 2019) such as logging, authentication, authorization, input/output validation, secrets management, encryption, communication protocols, and clean coding in general. This way, students will have less trouble integrating into the industry and produce more secure code from the beginning. We summarize our suggestions for potential teaching aspects for RQ.6: "*What are the security concepts for teaching the university students to bridge the gap towards more secure coding?*":

- Clean coding practices (Simplicity, Elimination of code duplication). If students are informed and educated with clean coding practices in mind, repetition of the mistakes and snowballing effect will be minimized when it comes to creating vulnerabilities.
- Web application security (SQL Injection, Cross-Site Scripting, Path Traversal). These main topics should be taught as these were highly occurring in our findings.
- Authentication and authorization concepts in practice (Client and Server-side authentication and authorization schemes). Students should be well informed on these concepts and how they are achieved first and the potential mistakes that cause vulnerabilities in such mechanisms.

Fig. 11. Code for a login page from one of the submissions with structural problems and high number of vulnerabilities.

Fig. 12. Code for a login page from one of the submissions with less structural problems and less number of vulnerabilities.

- Secure log management practice (Secure debugging, secure log file management). Students should be equipped with securing log management practices as these were present in the results as well.
- Language/Framework related security practices (Security features of used protocols (e.g., HTTP, TCP), prevention of SQL injection in corresponding languages). While teaching different frameworks and languages, students could benefit from the security features and concepts of such frameworks and languages.
- Practical Cryptography (Secure encryption of sensitive information). Students should be taught about cryptography concepts and how they are implemented and how they play a role in securing information systems.

8. Limitations

We utilize Sonarqube Community Edition to scan the student submissions. False positive rate is one of the top selling points of commercial vulnerability detection software, however even the most successful ones come with a formidable false positive rate and confirming the existing vulnerability may require the development of an actual exploitation strategy. Thus it is imperative to work while false positives are still present, and be able to derive statistically significant results. Nonetheless, to give an idea to the reader how accurate Sonarqube is in this dataset, we have randomly sampled a set of vulnerabilities for each CWE category and manually gone over them. In the process, two security experts were employed where mutual agreement is considered for marking the vulnerability as a false-positive or not. Table 7 shows the types of sampled vulnerabilities and the false-positive counts as the result of the manual assessment. It should be noted that the false positive rate would have been higher for more complex datasets containing industrial applications.

Another limitation to address is the data set size. If it was possible to collect more student assignments, we could train deep learning models to get possibly more accurate results since, for instance, recent studies of vulnerability analysis incorporate such models. However, the time span of our data set is 6 semesters which creates a reliable enough variation in the data. We also analyze 2 different but similar tasks that may help tackle the task bias problem.

This work used a dataset of student coded web-applications from a "Database Management Course". In addition, the implementation language was PHP/HTML/Javascript. As future work, research on this matter would benefit from similar or more elaborate datasets where students develop more complex applications that resemble real-world scenarios better. In addition, a study that compares student code with the code of professionals with respect to security would be a contribution to further closing the gap between education and industry. Another contribution that could highlight the way for secure coding education would be a comparative analysis of how students transform their code when they become professionals.

9. Conclusion

Teaching students to write more secure code and preparing them for the industry as much as possible is useful for their careers and the promotion of security practices in general. In this work, we analyzed a data set of student code and investigated the nature of vulnerabilities present in their code. We utilized a survey to understand the students' point-of-view. Then, we used

Table 7
False Positive (FP) Counts for Sampled (S) CWE types.

Type	Definition	#	# S	# FP
CWE-259	Use of Hard-coded Password	829	20	0
CWE-20	Improper Input Validation	761	20	0
CWE-564	SQL Injection: Hibernate	751	20	0
CWE-943	Improper Neutralization of Special Elements in Data Query Logic	751	20	0
CWE-489	Active Debug Code	714	20	0
CWE-315	Cleartext Storage of Sensitive Information in a Cookie	23	23	0
CWE-117	Improper Output Neutralization for Logs	17	17	0
CWE-532	Insertion of Sensitive Information into Log File	17	17	0
CWE-778	Insufficient Logging	17	17	0
CWE-521	Weak Password Requirements	15	15	0
CWE-311	Missing Encryption of Sensitive Data	14	14	1
CWE-614	Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	14	14	1

data analysis and machine learning techniques to learn about the lexical and structural variations in student code with respect to vulnerability levels.

In order to create an effective learning environment for students, we need to incorporate security into our classes and teach students, not just about theoretical knowledge on computer science but also practical experience on various programming and systems aspects such as logging, authorization, exception handling, encryption, communication protocols. Teaching students about writing clean code is also important as there are indications of a correlation between the structure of code and vulnerabilities.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M.L., Stransky, C., 2016. You get where you're looking for: The impact of information sources on code security. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 289–305.

Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C., 2018. A survey of machine learning for big code and naturalness. ACM Comput. Surv. 51 (4), 81.

Allamanis, M., Peng, H., Sutton, C., 2016. A convolutional attention network for extreme summarization of source code. In: International Conference on Machine Learning, pp. 2091–2100.

Alon, U., Brody, S., Levy, O., Yahav, E., 2018. Code2seq: Generating sequences from structured representations of code. arXiv preprint arXiv:1808.01400.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. 3 (POPL), 1–29.

Amo, L.C., Liao, R., Frank, E., Rao, H.R., Upadhyaya, S., 2019. Cybersecurity interventions for teens: Two time-based approaches. IEEE Trans. Educ. 62 (2), 134–140.

Binkley, D., 2007. Source code analysis: A road map. In: Future of Software Engineering. FOSE'07, IEEE, pp. 104–119.

Cabaj, K., Domingos, D., Kotulski, Z., Respício, A., 2018. Cybersecurity education: Evolution of the discipline and analysis of master programs. Comput. Secur. 75, 24–35. http://dx.doi.org/10.1016/j.cose.2018.01.015, URL http://www.sciencedirect.com/science/article/pii/S0167404818300373.

Cappos, J., Weiss, R., 2014. Teaching the security mindset with reference monitors. In: Proceedings of the 45th ACM Technical Symposium on Computer Science Education. In: SIGCSE '14, Association for Computing Machinery, New York, NY, USA, pp. 523–528. http://dx.doi.org/10.1145/2538862.2538939, URL https://doi.org/10.1145/2538862.2538939.

Chi, H., Jones, E.L., Brown, J., 2013. Teaching secure coding practices to STEM students. In: Proceedings of the 2013 on InfoSecCD '13: Information Security Curriculum Development Conference. In: InfoSecCD '13, Association for Computing Machinery, New York, NY, USA, pp. 42–48. http://dx.doi.org/10.1145/2528908.2528911, URL https://doi.org/10.1145/2528908.2528911.

Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., Koschke, R., 2009. A systematic survey of program comprehension through dynamic analysis. IEEE Trans. Softw. Eng. 35 (5), 684–702.

Ghaffarian, S.M., Shahriari, H.R., 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Comput. Surv. 50 (4), http://dx.doi.org/10.1145/3092566, URL https://doi.org/10.1145/3092566.

Hooshangi, S., Weiss, R., Cappos, J., 2015. Can the security mindset make students better testers? In: Proceedings of the 46th ACM Technical Symposium on Computer Science Education. In: SIGCSE '15, Association for Computing Machinery, New York, NY, USA, pp. 404–409. http://dx.doi.org/10.1145/2676723.2677268, URL https://doi.org/10.1145/2676723.2677268.

Jain, S., Lindqvist, J., et al., 2014. Should I protect you? Understanding developers' behavior to privacy-preserving APIs. In: Workshop on Usable Security. USEC'14, Internet Society.

Joint task force on computing curricula, association for computing machinery (ACM) and IEEE computer society, 2013. Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. Association for Computing Machinery, New York, NY, USA.

Joint Task Force on Cybersecurity Education, 2017. Cybersecurity curricular guidelines: CSEC 2017. Joint Task Force on Cybersecurity Education, URL https://cybered.hosting.acm.org/wp/.

Jones, K.S., Namin, A.S., Armstrong, M.E., 2018. The core cyber-defense knowledge, skills, and abilities that cybersecurity students should learn in school: Results from interviews with cybersecurity professionals. ACM Trans. Comput. Educ. 18 (3), URL https://doi.org/10.1145/3152893.

Lalande, J.-F., Viet Triem Tong, V., Graux, P., Hiet, G., Mazurczyk, W., Chaoui, H., Berthomé, P., 2019. Teaching android mobile security. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education. In: SIGCSE '19, Association for Computing Machinery, New York, NY, USA, pp. 232–238, URL https://doi.org/10.1145/3287324.3287406.

Liu, B., Shi, L., Cai, Z., Li, M., 2012. Software vulnerability discovery techniques: A survey. In: 2012 Fourth International Conference on Multimedia Information Networking and Security. IEEE, pp. 152–156.

Mou, L., Li, G., Jin, Z., Zhang, L., Wang, T., 2014. TBCNN: A tree-based convolutional neural network for programming language processing. Citeseer, arXiv preprint arXiv:1409.5718.

Nance, K.L., Hay, B., Bishop, M., 2012. Secure coding education: Are we making progress? In: Proceedings of the 16th Colloquium for Information Systems Security Education.

Pang, Y., Xue, X., Namin, A.S., 2015. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In: 2015 IEEE 14th International Conference on Machine Learning and Applications. ICMLA, IEEE, pp. 543–548.

Parekh, G., DeLatte, D., Herman, G.L., Oliva, L., Phatak, D., Scheponik, T., Sherman, A.T., 2018. Identifying core concepts of cybersecurity: Results of two delphi processes. IEEE Trans. Educ. 61 (1), 11–20.

Pistoia, M., Chandra, S., Fink, S.J., Yahav, E., 2007. A survey of static analysis methods for identifying security vulnerabilities in software systems. IBM Syst. J. 46 (2), 265–288.

Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M., 2018. Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications. ICMLA, IEEE, pp. 757–762.

Schumacher, J., Welch, D., 2002. Educating leaders in information assurance. IEEE Trans. Educ. 45 (2), 194–201.

Taylor, B., Bishop, M., Hawthorne, E., Nance, K., 2013. Teaching secure coding: The myths and the realities. In: Proceeding of the 44th ACM Technical Symposium on Computer Science Education. In: SIGCSE '13, Association for Computing Machinery, New York, NY, USA, pp. 281–282, URL https://doi.org/10.1145/2445196.2445280.

Taylor, B., Kaza, S., 2011. Security injections: Modules to help students remember, understand, and apply secure coding techniques. In: Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education. In: ITiCSE '11, Association for Computing Machinery, New York, NY, USA, pp. 3–7, URL https://doi.org/10.1145/1999747.1999752.

Taylor, B., Kaza, S., 2016. Security injections@towson: Integrating secure coding into introductory computer science courses. ACM Trans. Comput. Educ. 16 (4), URL https://doi.org/10.1145/2897441.

Švábenský, V., Vykopal, J., Čeleda, P., 2020. What are cybersecurity education papers about? A systematic literature review of SIGCSE and ITiCSE conferences. In: Proceedings of the 51st ACM Technical Symposium on Computer Science Education. In: SIGCSE '20, Association for Computing Machinery, New York, NY, USA, pp. 2–8, URL https://doi.org/10.1145/3328778.3366816.

Wang, S., Liu, T., Tan, L., 2016. Automatically learning semantic features for defect prediction. In: 2016 IEEE/ACM 38th International Conference on Software Engineering. ICSE, IEEE, pp. 297–308.

White, M., Tufano, M., Vendome, C., Poshyvanyk, D., 2016. Deep learning code fragments for code clone detection. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 87–98.

Williams, K.A., Yuan, X., Yu, H., Bryant, K., 2014. Teaching secure coding for beginning programmers. J. Comput. Sci. Coll. 29 (5), 91–99.