Assessing Common Software Vulnerabilities in Undergraduate Computer Science Assignments

Andrew Sanders
Computer and Cyber Science
Augusta University
Augusta, USA
asanders4@augusta.edu

Gursimran Singh Walia
Computer and Cyber Science
Augusta University
Augusta, USA
gwalia@augusta.edu

Andrew Allen
Computer Science
Georgia Southern University
Statesboro, USA
aallen@georgiasouthern.edu

***Abstract*—As the demand for secure coding education grows, there is a need for improvements in how secure coding is taught and in preparing students to develop more secure software. As time in a Computer Science classroom is finite, educational efforts should be placed on targeting the most common types of vulnerabilities to better prepare students to avoid common security pitfalls in coding.**

Existing research in this area mainly focuses on developing vulnerability detection tools rather than analyzing the types of commonly produced vulnerabilities by students. Limited research exists in determining common student-produced vulnerabilities, and the available studies differ from the types of vulnerabilities that are researched in vulnerability detection literature.

Our research works to further establish the types of vulnerabilities produced by students by using a static analysis tool on assignment code submissions in an undergraduate Programming II (CS2) course.

We present our findings on what types of vulnerabilities are commonly produced by students and contrast them with what is commonly researched in the literature. We find there is little overlap between the vulnerability types reported by our study and other studies in the research area. This research has potential implications for secure coding education in a Computer Science curriculum. Further work should be done to establish the contexts in which specific vulnerability types are more likely to be produced and how to best teach students to avoid producing these vulnerabilities.

***Index Terms*—Secure Coding Education, Cyber-Security Education, Vulnerability Analysis**

I. Introduction

While the Computing curriculum has focused on preparing the workforce to develop functional software, the need for secure coding education is growing. The United States Department of Homeland Security has previously stated that 90% of reported security incidents result from exploits against defects in the design or code of software [6]. More recently, Verizon's 2023 Data Breach Investigation Report stated that software code vulnerability exploitation is one of the primary methods by which attackers access an organization [5]. Recommended Computer Science guidelines, such as those proposed by ACM's Curriculum Guidelines and the recent ABET standards [1], have evolved to include principles of secure computing (sometimes called secure coding) in the general curriculum requirements. The increased importance of integrating security principles in the CS/Cybersecurity curriculum has led to the development of the Information Assurance and Security knowledge area by the ACM and IEEE Joint Task Force on Computing Curricula [11]. The development of these guidelines has implications for technical workforce (professionals in industry), which relies on academic institutions to produce graduates that are versed with secure coding education [7].

Traditionally, academic computer science programs have either lacked a software security course requirement [10] or included a stand-alone senior-level area of emphasis course. In a survey of development and IT professionals conducted by Veracode, most developers felt their university-provided software security skills were inadequate for their industry jobs requirements [10]. The current lack of focus on the integration of secure coding education in computer science programs points to a need for improvements in how secure coding is taught and in preparing students to develop more secure software. To train students on understanding and avoiding introducing vulnerabilities during code development, the first step is to collect data on the type of vulnerabilities that students (or developers) introduce at different levels of their program completion.

It is essential to establish commonly used terms, taxonomy, and repositories to aid readers' understanding of the technical terms used in this research. For this work, we are following the Common Weakness Enumerated (CWE) framework [4]. The CWE framework establishes the terms that follow.

- A weakness is a condition that, under certain circumstances, could contribute to the introduction of vulnerabilities.
- A vulnerability is a weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source.

The Common Vulnerabilities and Exposures (CVE) program identifies, defines, and catalogs publicly disclosed cybersecurity vulnerabilities [13]. They maintain a list of CVE records, which is descriptive data about a vulnerability associated with a CVE ID. The National Institute of Standards and Technology (NIST) maintains the Software Assurance Reference Dataset (SARD) and the National Vulnerability Database (NVD). SARD is a collection of test programs with documented weaknesses with test cases that vary from small synthetic programs to large applications [12]. The NVD is a reposi-

tory of standards-based vulnerability management data and is tasked with analyzing each CVE published to the CVE list and associating reference tags, Common Vulnerability Scoring System (CVSS), Common Weakness Enumeration (CWE), and Common Platform Enumeration (CPE) applicability statements [13]. Common Weakness Enumeration (CWE) is a list of weakness types for software and hardware. Vulnerabilities within this list are arranged in a tree-like hierarchy, based on the level of abstraction. The top level is Category, which contains entries that share common characteristics and represent commonly understood areas within software development. When referring to a weakness, it is common to refer to it using its CWE-ID.

While previous research efforts in the area have mainly focused on developing vulnerability detection tools and methods [8], there seems to be a lack of focus on analyzing types of vulnerabilities produced by students and professional developers. As of 2022, there seems only to exist one paper [15] that talks about code vulnerabilities introduced by students. However, the most common vulnerabilities reported in the paper differ from those commonly researched by vulnerability analysis papers. The most common types of vulnerabilities studied by software vulnerability researchers are as follows [8]:

- (CWE-78) OS command injection
- (CWE-79) Cross-site scripting
- (CWE-89) SQL injection
- (CWE-119) Buffer errors
- (CWE-120) Buffer overflow
- (CWE-190) Integer overflow
- (CWE-306) Missing authentication for critical function

Each of these seven vulnerabilities is contained in the 2022 CWE Top 25 Most Dangerous Software Weaknesses [3], which is based on data from the National Vulnerability Database and officially submitted CVEs. These vulnerabilities contrast with what was reported by [15] in Table I to be the most common vulnerabilities introduced by students. None of the CWE-IDs reported by [15] are represented in the most commonly studied vulnerabilities in the literature, though CWE-89 and CWE-564 are both SQL Injection-related. This mismatch can lead to oversights in Computer Science education in which more uncommon vulnerabilities get more of the educational focus, causing students to be less prepared to make secure software.

This research focuses on using existing static analysis tools and their output to further establish the most common types of vulnerabilities produced by students so that teaching methods and tools can be developed to help students and professional developers develop more secure software. To that end, our research questions are as follows.

1) *RQ1*: What are the most common software vulnerabilities produced by CS2 students in their assignment submission code?
2) *RQ2*: How do these software vulnerabilities compare and contrast with the types of commonly researched vulnerabilities?

The format of this document is as follows. Section II explores the related work in the research area. Section III states the methodology used to answer the research questions. Section IV presents the results. Section V discusses the findings.

II. Related Work

Yilmaz et al. used a source code vulnerability analysis tool to study vulnerabilities introduced by students in a third-year Database Management Systems course [15]. The authors created a private dataset using the source code for two tasks over six semesters of programming assignments. The students used PHP, HTML, and JavaScript for the assignments. The authors stated that for future work, research on student code vulnerability would benefit from datasets where students develop more complex applications that resemble real-world scenarios. Table I shows the most common types of vulnerabilities. Figure 1 plots the grades awarded to each student along with the number of vulnerabilities. The authors state that "better grades indicate more functionality and complexity thus more probability to create security vulnerabilities". They conclude that practical knowledge of various programming aspects such as logging, authorization, exception handling, encryption, and communication protocols are needed to create an effective learning environment. They also find that there are correlations between structure of code and vulnerabilities. Our work builds upon this work by validating its findings and comparing and contrasting them with commonly researched vulnerabilities.

Hanif et al. studied software vulnerability detection methods and created a taxonomy of research interests [8]. The research interests they taxonomize are methods, detection, feature, code, and dataset. They reported a considerable interest in addressing methods and detection problems and showed a considerable interest in using machine learning to detect vulnerabilities. Relevant to this write-up is that the authors found that most existing works targeted specific types of vulnerabilities for detection. These specific types are common because they are frequently targeted by vulnerability detection systems (not necessarily because they are commonly introduced in code). Another finding by the authors is that there is a lack of a large, gold-standard dataset for software vulnerability detection and that the currently available real-world vulnerability dataset is the National Vulnerability Database (NVD). They also note that the NVD dataset involves the manual extraction of source code from repositories, which could have potential mislabeling. Of the 83 cited papers, the dataset breakdown is as follows:

- National Vulnerability Database (NVD): 20 Papers
- Software Assurance Reference Dataset (SARD): 18 Papers
- Open Source Software (OSS): 49 Papers
- Common Vulnerabilities and Exposures (CVEs): 2 Papers
- Code from Competition: 3 Papers
- Private dataset: 12 Papers

TABLE I
VULNERABILITIES OF STUDENT CODE PER CWE TYPE [15]

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

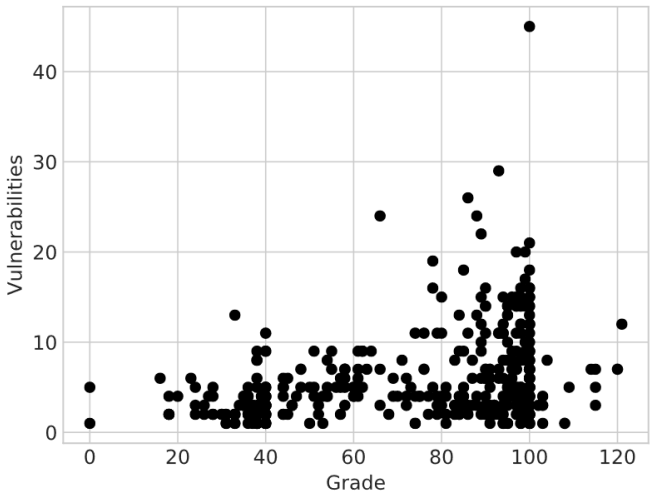

Fig. 1. Vulnerabilities of Student Code and Grades Received [15]

Our Measurement	Book 1	Book 2	Book 3	Book 4
Security Vulnerability	17	13	17	26
Quality Vulnerability	82	36	121	106

Fig. 2. Types of Bugs Reported by FindBugs in Four Java Textbooks Categorized by the Authors' Vulnerability Criteria [9]

If you roll up the datasets of the cited papers, data can be classified as coming from one of three places: National Institute of Science and Technology (NIST), Open Source Software (OSS), and private datasets.

Hu et al. studied vulnerabilities in Java programming textbooks for an undergraduate Java programming course [9]. The authors used an open-source vulnerability analysis tool called FindBugs to analyze the byte code of the sample source codes in four Java textbooks. They find many common bugs in the sample source codes, which raise security concerns. Figure 2 shows the bugs reported by their analysis tool grouped by the authors' vulnerability criteria for the four Java textbooks. If students were to adopt the coding styles of these bugged code samples, they might introduce the same bugs in larger software.

In reviewing software vulnerability detection papers, Hanif et al. [8] broadly divide the approaches into two categories: Conventional approaches and Machine Learning based approaches. Machine learning approaches are the more popular of the two and have more consistent growth in published papers within the last decade. The machine learning papers are further broken down into deep learning, supervised learning, ensemble learning, natural language processing, semi-supervised learning, regression, and tree-based. The conventional papers are broken down into static analysis, hybrid analysis, pattern matching and searching, graph-based, taint analysis, dynamic analysis, formal models, and statistical analysis. Conventional approaches often have low detection performance and high number of false positives and are generally becoming less reliable as vulnerabilities keep evolving [8]. However, they can still be used for conventional vulnerability detection when tools using other methods are unavailable.

While much work has been put into vulnerability detection and secure coding, there has been a lack of focus on analyzing the types of vulnerabilities produced by Computer Science students and graduates. This lack of analyses also pairs with a lack of directed pedagogy toward curbing the kinds of software vulnerabilities produced during the education process.

III. METHODOLOGY

This section provides our methodology for applying vulnerability detection tools to answer the research questions in section I.

To answer *RQ1*, we applied a vulnerability detection tool to each file in the dataset. We generated our dataset by analyzing the Github assignment submissions for a Georgia Southern University Programming Principles II course over the 2017-2023 school years. The total number of assignment submissions, excluding empty projects, was 3537. The data consisted of object-oriented assignments in the Java programming language. Each assignment submission was compiled before analysis. Each submission was grouped by year and

Fig. 3. SonarQube Analysis Example

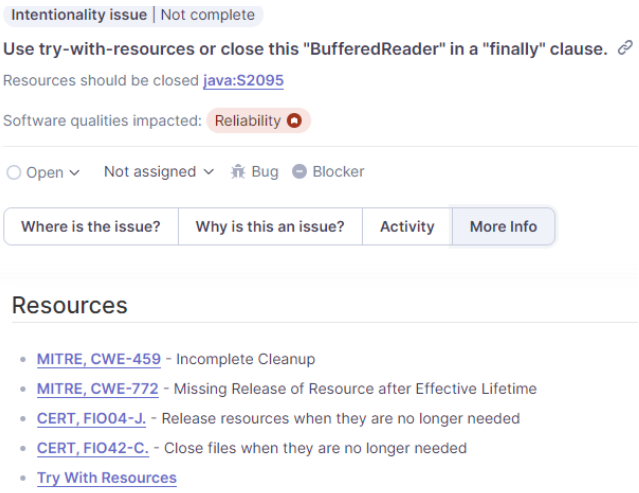

Fig. 4. SonarQube Issue Information Example

semester so a later analysis of vulnerabilities produced over time could be performed. For each file in our dataset, our vulnerability detection tool reported all potential vulnerabilities grouped by CWE-ID. The resulting CWE-ID classifications are grouped per student and per semester to discover the most common software vulnerabilities produced in assignment code. In section IV, we present the results and analysis from the vulnerability tool.

We used Sonarqube Community Edition Version 10.2.0.77647 to analyze student assignment submission code for vulnerabilities and weaknesses. It is a self-managed static analysis tool for continuous codebase inspection [2]. The SonarQube quality model has four different types of rules: reliability (bug), maintainability (code smell), and security (vulnerability and hotspot) rules [14]. A security hotspot highlights a security-sensitive piece of code that the developer needs to review. A vulnerability is a problem that impacts the application's security and needs to be fixed immediately. A bug is a coding mistake that can lead to an error or unexpected behavior at runtime. A code smell is a maintainability issue that makes your code confusing and difficult to maintain. For this research, we are only considering issues that have a direct CWE-ID mapping, which is indicated by an issue having "cwe" as tag. The related CWE-IDs are extracted from the issue description. Figure 3 shows an example of an assignment analysis by SonarQube. The figure shows both a bug and a code smell on separate lines. Each issue has a "cwe" tag, indicating that the issue has a CWE-ID mapping. Figure 4 shows a description of the bug issue. The issue is "Use try-with-resources or close this 'BufferedReader' in a 'finally' clause". This issue has a CWE-ID mapping of both CWE-459, "Incomplete Cleanup" and CWE-772, "Missing Release of Resource after Effective Lifetime". For our analysis, we include both of these CWE-IDs for this assignment, in addition to the other CWE-IDs mapped to the other issues.

To answer *RQ2*, we used the findings from *RQ1* and compared the results with the commonly researched vulnerabilities, as established by [8] and reported by [15]. Using CWE-IDs, we compared those present in both papers, specifically CWE-IDs 78, 78, 89, 119, 120, 190, and 306 in the case of [8] and CWE-IDs 259, 20, 564, 943, 480, 315, 117, 532, 778, 521, 311, and 614 in the case of [15]. These comparisons are then used to determine the amount of overlap in the literature.

IV. Results

The results of using SonarQube on the student assignment submission dataset are presented in subsection IV-A. The comparisons with commonly researched vulnerabilities are presented in subsection IV-B.

A. RQ1

The statistical measures of the CWE-IDs introduced in the assignments are as follows.

- Mean: 4.37
- Median: 2.0
- Mode: 0
- Min: 0
- Max: 76
- Range: 76
- Standard Deviation: 6.55
- Variance: 42.92
- Skewness: 2.83

The number and skewness of CWE-IDs introduced per assignment is graphically presented in Figure 5. The assignment index is an increasing value so the distribution is clearer to the reader. Of the 3537 assignments, 1442 assignment submissions did not have a mapped CWE-ID. This can potentially be attributed to a variety of factors, such as simplistic assignment submissions in which hardly any code was contributed or blind spots in the analysis software in which further analysis would need to be performed.

With a skewness of 2.8, the data is heavily right-skewed, meaning only a small portion of assignment submissions have a large number of mapped CWE-IDs. Of the 3537 assignment submissions, only 134 assignments had 20 or more CWE-IDs

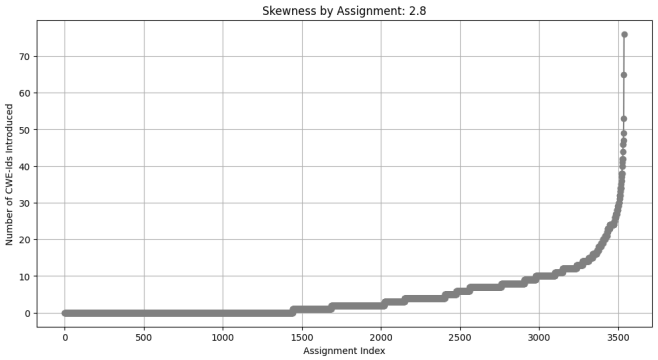

Fig. 5. Number of CWE-IDs Introduced per Assignment

mapped to them and only 558 assignments had 10 or more CWE-IDs mapped to them.

With respect to answering *RQ1*, *What are the most common software vulnerabilities produced by CS2 students in their assignment submission code?*, Table II presents the overall frequency distribution of CWE-IDs over all assignment submissions. The most frequent CWE-IDs include (CWE-546) Suspicious Comment, (CWE-581) Object Model Violation: Just One of Equals and Hashcode Defined, (CWE-476) NULL Pointer Dereference, (CWE-563) Assignment to Variable without Use, (CWE-489) Active Debug Code, (CWE-215) Insertion of Sensitive Information Into Debugging Code, and (CWE-459) Incomplete Cleanup. CWE-546 generally comes from auto-generated TODO comments that are inserted from the IDE. CWE-581 comes from assignments requiring equals() to be overwritten but not hashcode(). CWE-476 comes from the lack of NULL checking on objects before using them. CWE-563 comes from forgetting to delete variables that are not in use. CWE-489 comes from print statements that are designed to help the programmer but should have been removed before submission. CWE-215 generally comes from catching an exception and printing out the stack trace to the console. CWE-459 comes from try-catch blocks lacking a finally to clean up resources such as a BufferedWriter.

While some of these are of little pedagogical concern to instructors, such as CWE-546 where unremoved TODO comments are left in the code, other weaknesses, such as CWE-476, may be of concern due to the importance of object-oriented design in the Java programming language.

B. RQ2

The commonly researched vulnerabilities, as established by [8], are presented in section I. Of the CWE-IDs listed (78, 78, 89, 119, 120, 190, 306), only (CWE-190) Integer Overflow or Wraparound was found to be represented in student assignment submissions. This CWE-ID only had 538 occurrences in the 3537 assignment submissions. Of the reported weaknesses in [15] (CWE-IDs 259, 20, 564, 943, 480, 315, 117, 532, 778, 521, 311, and 614), only (CWE-259) Use of Hard-coded Password shares a commonality. This CWE-ID only had 41 occurrences in the 3537 assignment submissions. The fact that only these CWE-IDs were found could potentially be attributed to the course level, programming language used, and assignment requirements.

As the assignment submissions were sourced from a Programming II (CS2) course in which Java is used, the requirements and technologies limited the potential for weaknesses to be introduced. In [15], the assignment submissions came from a Database Management Systems Course. The requirements from their assignment involved the internet and database connections, both of which are lacking from the Programming II assignments. Certain weaknesses presented in their study or are commonly researched in the literature are not possible to be introduced in the Programming II assignments, such as SQL Injection and Cross-site scripting. Similarly, the Java programming language, which all students used in their Programming II course, severely limits the possibility of introducing (CWE-120) Buffer overflows due to the bounds-checking of the language. If this same analysis was performed on a Programming II course in which C was the language in use, the results might differ. The key takeaway, when compared to [15], seems to be that the scope and the amount of external data communication of an assignment affect the potential for introducing vulnerabilities.

With respect to answering *RQ2*, *How do these software vulnerabilities compare and contrast with the types of commonly researched vulnerabilities?*, we find the types of software vulnerabilities produced by students have little overlap with both the types of commonly researched vulnerabilities in literature and the types of vulnerabilities reported by [15]. This indicates there is little consensus on what vulnerabilities students produce in their code.

V. Discussion and Conclusion

This paper has studied the use of a static analysis tool on student assignment code in a Programming II course to determine the types of software weaknesses produced by students and how these weaknesses relate to the commonly studied vulnerabilities in prior work. We found that the most common types of weaknesses produced by students are (CWE-546) Suspicious Comment, (CWE-581) Object Model Violation: Just One of Equals and Hashcode Defined, (CWE-476) NULL Pointer Dereference, (CWE-563) Assignment to Variable without Use, and (CWE-489) Active Debug Code. The table of all types of weaknesses produced by students in our dataset is shown in Table II. We also found that the types of vulnerabilities produced by students in our dataset have little consensus with the types that are commonly researched and the types reported in previous work in analyzing student code vulnerabilities. This indicates that further work needs to be done to establish the context in which vulnerabilities are produced, such as programming level, programming language, developer experience, and software requirements.

The findings in this paper could potentially be used to inform the Computer Science curriculum design in terms of software security and secure coding. With the knowledge that certain software weaknesses are more represented in student

TABLE II
FREQUENCY DISTRIBUTION OF CWE-IDS

CWE-ID	Description	Frequency
546	Suspicious Comment	1502
581	Object Model Violation: Just One of Equals and Hashcode Defined	1222
476	NULL Pointer Dereference	1089
563	Assignment to Variable without Use	1049
489	Active Debug Code	870
215	Insertion of Sensitive Information Into Debugging Code	870
459	Incomplete Cleanup	833
772	Missing Release of Resource after Effective Lifetime	833
1241	Use of Predictable Algorithm in Random Number Generator	681
326	Inadequate Encryption Strength	681
330	Use of Insufficiently Random Values	681
338	Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	681
595	Comparison of Object References Instead of Object Contents	606
597	Use of Wrong Operator in String Comparison	606
478	Missing Default Case in Multiple Condition Expression	588
190	Integer Overflow or Wraparound	538
493	Critical Public Variable Without Final Modifier	517
500	Public Static Field Not Marked Final	269
397	Declaration of Throws for Generic Exception	176
570	Expression is Always False	153
571	Expression is Always True	153
369	Divide By Zero	125
607	Public Static Final Field References Mutable Object	110
582	Array Declared Public, Final, and Static	110
483	Incorrect Block Delimitation	95
1333	Inefficient Regular Expression Complexity	80
400	Uncontrolled Resource Consumption	80
481	Assigning instead of Comparing	77
798	Use of Hard-coded Credentials	44
259	Use of Hard-coded Password	41
484	Omitted Break Statement in Switch	32
477	Use of Obsolete Function	14
594	J2EE Framework: Saving Unserializable Objects to Disk	11
486	Comparison of Classes by Name	7
584	Return Inside Finally Block	7
754	Improper Check for Unusual or Exceptional Conditions	6
391	Unchecked Error Condition	5
377	Insecure Temporary File	1
379	Creation of Temporary File in Directory with Insecure Permissions	1

assignment submissions than others, more effort could be placed on teaching to avoid these common pitfalls in software design. This research area can be expanded by integrating the most common types of software weaknesses into existing pedagogy and studying the resulting effects on the types of weaknesses produced by students in their assignment code.

References

[1] *Accreditation Changes — ABET*. URL: https://www.abet.org/accreditation/accreditation-criteria/accreditation-changes/ (visited on 02/02/2023).

[2] *Code Quality Tool & Secure Analysis with SonarQube*. URL: https://www.sonarsource.com/products/sonarqube/ (visited on 06/19/2023).

[3] *CWE - Common Weakness Enumeration*. URL: https://cwe.mitre.org/index.html (visited on 02/22/2023).

[4] *CWE - Frequently Asked Questions (FAQ)*. URL: https://cwe.mitre.org/about/faq.html (visited on 03/08/2023).

[5] *DBIR Report 2023 - Master's Guide*. Verizon Business. URL: https://www.verizon.com/business/resources/reports/dbir/2023/master-guide/ (visited on 07/06/2023).

[6] Department of Homeland Security, US-CERT. *Software Assurance*. URL: https://www.cisa.gov/sites/default/files/publications/infosheet_SoftwareAssurance.pdf (visited on 07/11/2023).

[7] Tiago Gasiba et al. *Is Secure Coding Education in the Industry Needed? An Investigation Through a Large Scale Survey*. May 1, 2021, p. 252. 241 pp. DOI: 10.1109/ICSE-SEET52601.2021.00034.

[8] Hazim Hanif et al. "The Rise of Software Vulnerability: Taxonomy of Software Vulnerabilities Detection and Machine Learning Approaches". In: *Journal of Network and Computer Applications* 179 (Apr. 1, 2021), p. 103009. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2021.103009. URL: https://www.sciencedirect.com/science/article/pii/S1084804521000369 (visited on 01/11/2023).

[9] Yen-Hung Hu and Thomas Kofi Annan. "Assessing Java Coding Vulnerabilities in Undergraduate Software Engineering Education by Using Open Source Vulnerability Analysis Tools". In: *Journal of The Colloquium for Information Systems Security Education* 4.2 (2 Feb. 19, 2017), pp. 33–33. ISSN: 2641-4554. URL: https://cisse.info/journal/index.php/cisse/article/view/60 (visited on 02/01/2023).

[10] John Zorabedian. *Veracode Survey Research Identifies Cybersecurity Skills Gap Causes and Cures*. Veracode. URL: https://www.veracode.com/blog/security-news/veracode-survey-research-identifies-cybersecurity-skills-gap-causes-and-cures (visited on 07/12/2023).

[11] Association for Computing Machinery (ACM) and IEEE Computer Society Joint Task Force on Computing Curricula. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York, NY, USA: Association for Computing Machinery, 2013. 518 pp. ISBN: 978-1-4503-2309-3.

[12] *NIST Software Assurance Reference Dataset*. NIST Software Assurance Reference Dataset. URL: https://samate.nist.gov/SARD (visited on 02/22/2023).

[13] *NVD - Home*. URL: https://nvd.nist.gov/ (visited on 02/22/2023).

[14] SonarSource. *Issues*. URL: https://docs.sonarsource.com/sonarqube/latest/user-guide/issues/ (visited on 07/12/2023).

[15] Tolga Yilmaz and Özgür Ulusoy. "Understanding Security Vulnerabilities in Student Code: A Case Study in a Non-Security Course". In: *Journal of Systems and Software* 185 (Mar. 1, 2022), p. 111150. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111150. URL: https://www.sciencedirect.com/science/article/pii/S0164121221002430 (visited on 09/18/2022).