Prevalence of Security Vulnerabilities in C++ Projects

Thiago Gadelha, Wallisson Freitas, Eduardo Rodrigues,
José Maria Monteiro and Javam Machado
Computer Science Department, Federal University of Ceará, Brazil

Keywords: Security Vulnerabilities, Secure Software Development, Static Code Analysis.

Abstract: One of the most critical tasks for organizations nowadays is to maintain the security of their software products. Common software vulnerabilities can result in severe security breaches, financial losses, and reputation deterioration. A software security vulnerability can be defined as a flaw in the source code that can be exploited by an attacker to gain unauthorized access to the software, thereby compromising its behavior and functionality. Then, detecting and fixing security vulnerabilities in the source code of software systems is one of the most significant challenges in the field of information security. The Static Application Security Testing (SAST) tools are capable of statically analyzing the source code, without executing it, to identify security vulnerabilities, bugs, and code smells during the coding phase, when it is relatively inexpensive to detect and resolve security issues. In this context, this paper proposes an exploratory study of security vulnerabilities in C++ code from very large projects. We analyzed twenty-six worldwide C++ projects and empirically studied the prevalence of security vulnerabilities. Our results showed that some vulnerabilities occur together. Besides, some vulnerabilities are more frequent than others. Based on these findings, this paper has the potential to aid software developers in avoiding future problems during the development of a C++ project.

1 INTRODUCTION

Our daily lives rely on software that performs everyday tasks, from digital commerce to smart homes and autonomous driving. It is evident that the increase in the volume of source code will lead to more security requirements in programming. In this context, a software security vulnerability can be defined as a coding flaw present in the source code that can be exploited by an attacker to gain unauthorized access to the software, thereby disrupting its behavior and functionality.

Security vulnerabilities are causing significant financial losses to businesses and threatening critical security infrastructures. Therefore, an effective solution is needed to discover and fix vulnerabilities before private and valuable information is compromised. Consequently, maintaining the security of software products is one of the most critical tasks for organizations today, making vulnerability detection fundamental (Islam et al., 2024).

Detecting software defects after a product has been deployed forces the company to bear the cost of repairs, weakens the company's reputation, and sometimes involves legal expenses. So, this approach is costly, challenging, and time-consuming. On the other hand, detecting and fixing vulnerabilities during the development phase, before the introduction of a product to the market, can save time, effort, and money (Hussain et al., 2024). Still, manual software inspection is unfeasible because it is a tedious process and may not yield the desired results. To make the vulnerability detection process more efficient in terms of time and coverage (number of vulnerabilities detected), automated methods have been proposed (Moyo and Mnkandla, 2020).

Static code analysis scans the entire source code of a system seeking potential security vulnerabilities. Thus, it is a way to infer the behavior of a program without executing it. Additionally, vulnerability detection is performed while the software is still in the development phase. So, it leads to the detection of vulnerabilities in the early stages of the software development lifecycle. This makes the process of correcting these vulnerabilities easier for the developer, while also reducing the cost and time required by the organization to address them.

Static Application Security Testing (SAST) tools

Gadelha, T., Freitas, W., Rodrigues, E., Monteiro, J. M., Machado and J.
Prevalence of Security Vulnerabilities in C++ Projects.
DOI: 10.5220/0013570700003967
In *Proceedings of the 14th International Conference on Data Science, Technology and Applications (DATA 2025)*, pages 567-574
ISBN: 978-989-758-758-0; ISSN: 2184-285X

analyze a piece of code thereof to identify security issues, including potential flaws, suspicious constructs, insecure use of APIs, dangerous runtime errors, coding errors, bad smells and adherence to coding standards. They automatically scan codebases for common security issues like injection vulnerabilities, authentication flaws, and insecure data handling practices (Alqaradaghi and Kozsik, 2024). SAST tools assist developers in detecting security vulnerabilities during the coding phase, where it is relatively cheaper to identify and fix security issues in the source code. For this reason, SAST tools are becoming increasingly crucial in the software development lifecycle (Novak et al., 2010).

This paper proposes an exploratory study of security vulnerabilities in C++ source code from very large projects. We analyzed twenty-six worldwide C++ projects and empirically studied the prevalence of security vulnerabilities. Vulnerability issues were identified through a static analysis conducted by Checkmarx[1] on the selected projects. The Checkmarx tool stands out as a leading provider of static application security testing solutions (Li et al., 2021). Their platform offers robust tools for identifying security vulnerabilities in software code through static code analysis techniques. We examined the occurrence patterns of the security issues and assessed the degree of co-occurrence among these issues. Our findings revealed that certain vulnerabilities tend to occur together, with some being more prevalent than others. Leveraging these insights, this paper has the potential to assist software developers and security experts in mitigating future problems during the development of C++ projects.

The remainder of this paper is structured as follows: Section 2 discusses the main related works. Section 3 presents the methodology used in this research. In Section 4, the obtained results are analyzed. Lastly, Section 5 presents the conclusions and outlines directions for future work.

2 RELATED WORK

The study presented in (Do et al., 2022) explored the challenges developers face when using SAST tools, focusing particularly on usability and integration into development pipelines. This paper underscores that beyond the technical capacity of these tools, their acceptance by developers is crucial.

The work in (Nguyen-Duc et al., 2021) explores the challenges of integrating SAST tools into real-world e-government software development. It introduces a novel approach where multiple SAST tools are combined to enhance the detection and analysis of security vulnerabilities. The authors in (Mehrpour and LaToza, 2023) focus on the potential of SAST tools to detect more defects typically identified in code reviews. The paper reformulates these issues as rule violations, which allows them to evaluate whether these violations could be detected by existing or future SAST tools. Besides, this paper suggests that SAST tools could detect more defects by supporting the creation of project-specific rules and enhancing their ability to simulate some aspects of human judgment.

The paper in (Ma et al., 2022) presents a newly developed Python auditing tool that leverages Abstract Syntax Tree analysis combined with data flow and control flow information. A significant contribution is the design of a plugin architecture that allows easy redevelopment or rewriting of specific rules. This flexibility enhances the tool's ability to adapt to different coding scenarios and security requirements. It processes Python code to identify security vulnerabilities efficiently. The tool includes a set of 70 detection rules covering common security issues in Python code. This extensive rule set helps identify a wide range of vulnerabilities, providing a robust framework for security auditing.

The article (Fan et al., 2020) presents Big-Vul, a comprehensive dataset that maps vulnerabilities in C/C++ code from open-source projects hosted on GitHub. This dataset was developed to support vulnerability detection and remediation research, providing a solid foundation for training machine learning models and conducting code analysis. It represents a significant contribution to the software security community, offering a structured and information-rich resource for developing tools and techniques for analyzing vulnerabilities in C/C++ code.

This paper instead of trying to detect security vulnerabilities, propose a new SAST tool, or analyze and compare SAST tools, presents an empirical study of the prevalence of security vulnerabilities in C++ projects. The idea of this research is to obtain useful information, such as correlations and association rules between security vulnerabilities and clustering these issues, to aid software developers in avoiding future problems while developing a C++ project.

3 METHODOLOGY

The methodology used in this research is composed of four steps. First, we select a set of worldwide C++

[1] https://checkmarx.com/

projects, published on public repositories on GitHub. So, twenty-six worldwide C++ projects were chosen for further analysis. Next, we collected a set of reports generated by a SAST tool called Checkmarx on the selected projects. In each execution, Checkmarx generates a PDF file, containing a report detailing the found security vulnerabilities. After this, a Python script was used to process these PDF files (Checkmarx reports) and generate a Security Vulnerabilities Dataset, which is a spreadsheet with summarized and aggregated information about the security vulnerabilities found in the selected projects. Finally, we applied a set of data analysis techniques to the Security Vulnerabilities Dataset. In order to favor the use and reproduction of these data analysis techniques, we used Python scripts and Jupyter Notebooks.

3.1 Selecting C++ Projects

First, we searched for large and significant C++ projects, published on public repositories on github. Our projects were selected based on 3 criteria: i) the period of time in which the project is under development; ii) the projects should have at least 70% of their code written in C/C++; and the number of commits, which actually are real code enhancement like new features or code refactoring, on the projects repository. Following the previously defined criteria, twenty-six public C++ projects were chosen for further analysis. The links to all these projects are available in the GitHub repository of this work[2].

3.2 Collecting Checkmarx Reports

In this step, we collect a set of reports from Checkmarx for the selected projects. Checkmarx vulnerability reports include a detailed and categorized series of security flaws identified during static code analysis. The report extensively describes these vulnerabilities, typically encompassing details on their identification, examples of problematic code, potential impact on application security, and specific recommendations for correction. In our analysis, we considered 26 Checkmarx reports. The Checkmarx's documentation identifies 367 types of security vulnerabilities. However, in the analyzed project's reports, only 79 vulnerabilities were automatically identified, which are presented in Table 1.

[2]https://github.com/jmmfilho/psv

3.3 Building the C++ Security Vulnerabilities Dataset

Each PDF file generated previously contains a very large number of security vulnerabilities. For example, one of these files has 6,844 security vulnerabilities. So, it is not suitable to extract the data in each PDF file in a manual way. For this reason, we developed a Python script that processes these PDF files (containing Checkmarx reports) and generates a Security Vulnerabilities Dataset, which is a spreadsheet with summarized and aggregated information about the security vulnerabilities found in the selected projects in an automated way. This dataset was used in the data analysis methods.

For each of the projects, the commits corresponding to the quartiles were selected so that there is coverage of the code development, and a more detailed analysis can be carried out. The records were grouped based on the project and commit hash so that vulnerabilities were counted on the commit hash. As a software development scenario is being analyzed, there may be rows whose vulnerability count may be the same, as projects may not have had new security issues between commits. Thus, these lines can be configured as redundancy since the count of these lines can be based on code snippets that have not been changed. To solve this problem in our analysis, neighboring lines whose count is entirely equal to the original line were removed for each project.

3.4 Analyzing the Security Vulnerabilities Dataset

Data analysis is conducted for both quantitative and qualitative data. This is done to extract relevant information and knowledge regarding the data's context. This exploratory case study employed various data analysis techniques, such as descriptive statistics, correlation analysis, association rule learning, and clustering algorithms. Thus, in this step, we applied a set of data analysis techniques to the Security Vulnerabilities Dataset. To favor the use and reproduction of these data analysis techniques, we used Python libraries that allow reproducibility, like mlxtend (Raschka, 2018) and scikit-learn (Pedregosa et al., 2011). This step aimed to understand the prevalence of vulnerabilities and the existing relationships among them.

3.4.1 Quantitative Analysis

Quantitative analysis used descriptive statistics methods to explore the prevalence of security vulnerabili-

Table 1: Checkmarx Vulnerabilities Identified in the Selected Projects.

ID	Vulnerability	ID	Vulnerability
1	Arithmetic Operation On Boolean	41	Incorrect Permission Assignment For Critical Resources
2	Buffer Improper Index Access	42	Information Exposure Through Comments
3	Buffer Overflow AddressOfLocalVarReturned	43	Insecure Temporary File
4	Buffer Overflow LongString	44	Insufficiently Protected Credentials
5	Buffer Overflow Unbounded Buffer	45	Integer Overflow
6	Buffer Overflow Unbounded Format	46	Leaving Temporary Files
7	Buffer Overflow Wrong Buffer Size	47	Memory Leak
8	Buffer Overflow cin	48	MemoryFree on StackVariable
9	CGI Reflected XSS	49	NULL Pointer Dereference
10	CGI Stored XSS	50	Off by One Error
11	Cleartext Transmission Of Sensitive Information	51	PBKDF2 Insufficient Iteration Count
12	Command Injection	52	Path Traversal
13	Comparison Timing Attack	53	Personal Information Without Encryption
14	Creation of chroot Jail without Changing Working Directory	54	Plaintext Storage Of A Password
15	Dangerous Functions	55	Privacy Violation
16	Divide By Zero	56	Process Control
17	DoS by Sleep	57	Reliance on DNS Lookups in a Decision
18	Double Free	58	Resource Injection
19	Encoding Used Instead of Encryption	59	Stored Command Injection
20	Environment Injection	60	Symmetric Encryption Insecure Predictable Key
21	Exposure of System Data to Unauthorized Control Sphere	61	Symmetric Encryption Insecure Static Key
22	Float Overflow	62	TOCTOU
23	Format String Attack	63	Type Conversion Error
24	Hardcoded Absolute Path	64	Unchecked Array Index
25	Hardcoded password in Connection String	65	Unchecked Return Value
26	Hashing Length Extension Attack	66	Uncontrolled Recursion
27	Heap Inspection	67	Unreleased Resource Leak
28	Heuristic 2nd Order Buffer Overflow malloc	68	Use After Free
29	Heuristic 2nd Order Buffer Overflow read	69	Use Of Deprecated Class
30	Heuristic 2nd Order SQL Injection	70	Use Of Hardcoded Password
31	Heuristic Buffer Improper Index Access	71	Use Of Weak Hashing Primitive
32	Heuristic Buffer Overflow malloc	72	Use of Hard coded Cryptographic Key
33	Heuristic Buffer Overflow read	73	Use of Insufficiently Random Values
34	Improper Exception Handling	74	Use of Obsolete Functions
35	Improper Null Termination	75	Use of Sizeof On a Pointer Type
36	Improper Resource Access Authorization	76	Use of Uninitialized Variable
37	Improper Resource Shutdown or Release	77	Use of a One Way Hash without a Salt
38	Inadequate Encryption Strength	78	Weak Randomness Biased Random Sample
39	Inadequate Pointer Validation	79	Wrong Memory Allocation
40	Inconsistent Implementations		

ties. The focus was on identifying the most common vulnerabilities and their categories.

3.4.2 Correlation Between Vulnerabilities

A correlation coefficient quantifies the degree to which two variables vary together, indicating their relationship's strength and direction. Spearman's rank correlation coefficient assesses the monotonic relationship between two continuous or ordinal variables. This non-parametric test does not assume a specific data distribution (McCrum-Gardner, 2008). So, this study employed Spearman's rank correlation coefficient at a significance level of 0.05 to assess the statistical correlation between each pair of security vulnerabilities. Our correlation analysis incorporates both qualitative and quantitative approaches. Initially, we filtered out correlation coefficient estimates that lacked statistical significance (ρ-value ≥ 0.05). Subsequently, we examined strong correlations based on the scale proposed in (Berger and Guo, 2014) and sought explanations based on our domain knowledge.

3.4.3 Association Rules Learning

Association rule learning seeks to discover relationships among data records (items) that demonstrate statistical correlations. The main goal is to identify item subsets whose occurrence is associated with the presence of another item within the same transaction. Apriori (Agrawal and Srikant, 1994) is a prominent technique in association rule learning widely applied in market basket analysis, cross-marketing, and understanding customer buying patterns. This study employed association rule learning to uncover associations among vulnerabilities.

Support and confidence are essential metrics in association rule learning. Support measures the relative frequency with which a particular itemset appears in the dataset. Higher support values indicate more frequent occurrences, thereby suggesting that the itemset may represent a meaningful association. Confidence, on the other hand, assesses the reliability of an association rule by estimating the conditional probability of item Y appearing in a transaction, given that item X is already present. A high confidence value suggests a strong likelihood that item Y co-occurs with item X, thus indicating the rule's predictive effectiveness.

3.4.4 Clustering

Clustering techniques were employed to identify groups of security issues based on their occurrence frequencies across multiple projects, aiming to explore potential relationships among different types of vulnerabilities. This study applied three clustering algorithms: K-Means, Agglomerative Clustering, and Affinity Propagation. The use of multiple techniques enabled comparative analysis of the results, considering the distinct characteristics of each method in handling unlabeled datasets. Furthermore, a domain expert was consulted to qualitatively validate the clustering outcomes.

4 RESULTS AND DISCUSSION

We explored the nature of C++ security vulnerabilities in very large software projects through the following research questions (RQs):

- RQ1: How often are security vulnerabilities in real C++ projects?
- RQ2: Is there evidence of a significant correlation between different vulnerabilities identified in the analysis?
- RQ3: Are there security vulnerabilities that occur together in real C++ projects?
- RQ4: Is it possible to find groups of vulnerabilities that occur together in real C++ projects?

We focus on real-world applications according to the criteria presented in section 3.1. Each project has a specific purpose and/or complexity, meaning that the size and potential errors that may occur are inherent to the project. Moreover, bug resolution will depend on the community engaged in the project and the difficulty of addressing them. This choice was made to encompass different projects utilizing C++ to provide more comprehensive coverage.

A Jupyter Notebook was created to present the results and allow reproduction of the experiments carried out. Next, we will present the experiments performed to evaluate the research questions.

4.1 Quantitative Analysis

Figure 1 presents the total number of vulnerabilities identified in all projects analyzed. As can be seen, Memory Leak, Dangerous Functions, Unchecked Return Value, Use of Uninitialized Variables, and Unchecked Array Index are the vulnerabilities with the most occurrences reported by the Checkmarx tool.

The large number of memory leaks present in the C++ projects evaluated occurred due to the pointer features offered by the language. These features allow the developer to allocate memory manually and manage that memory using pointers, which are variables that hold memory addresses. In this scenario, a memory leak can occur if a pointer is not correctly freed at the right moment or if an exception occurs between a memory allocation and deallocation.

The second most prevalent vulnerability, dangerous functions, relates to functions that require caution when used, especially in source codes that also present the third, fourth, and fifth most common vulnerabilities: Unchecked Return Value, Use of Uninitialized Variable, and Unchecked Array Index. These dangerous functions often lack proper bounds checking and error handling, making them susceptible to flaws such as buffer overflows, memory corruption, and undefined behavior. Additionally, the large presence of these five vulnerabilities suggests that the code review process is weak and does not adhere to standard security patterns.

4.2 Correlation Between Vulnerabilities

In exploring the research question concerning correlated vulnerabilities in software development, it is hypothesized that a significant correlation exists between pairs of vulnerabilities within the Dataset (H1). Conversely, the null hypothesis (H0) suggests no correlation between these vulnerabilities. Due to the non-normal distribution of variables in the Dataset, Pearson's coefficient is unsuitable for testing hypothesis H1. Therefore, an alternative correlation test approach recommended in (Bagheri and Gasevic, 2011) was chosen instead.

An analysis was conducted to examine strongly correlated features to determine potential relationships among them. Given the abundance of strongly correlated Security Vulnerabilities, the decision was made only to list those with correlation coefficients exceeding 0.75. Table 2 presents these correlations.

The correlation coefficient between vulnerabilities 15 and 35 is 0.83. Dangerous functions may not properly handle strings that lack correct termination, potentially leading to unexpected behavior and security vulnerabilities through memory exploitation. Besides, vulnerability 15 is strongly correlated with vulnerability 74, with a correlation coefficient of 0.90. Using dangerous functions combined with deprecated functions can create unpredictable scenarios and security loopholes, as deprecated functions may have known vulnerabilities that are not adequately addressed by dangerous functions.

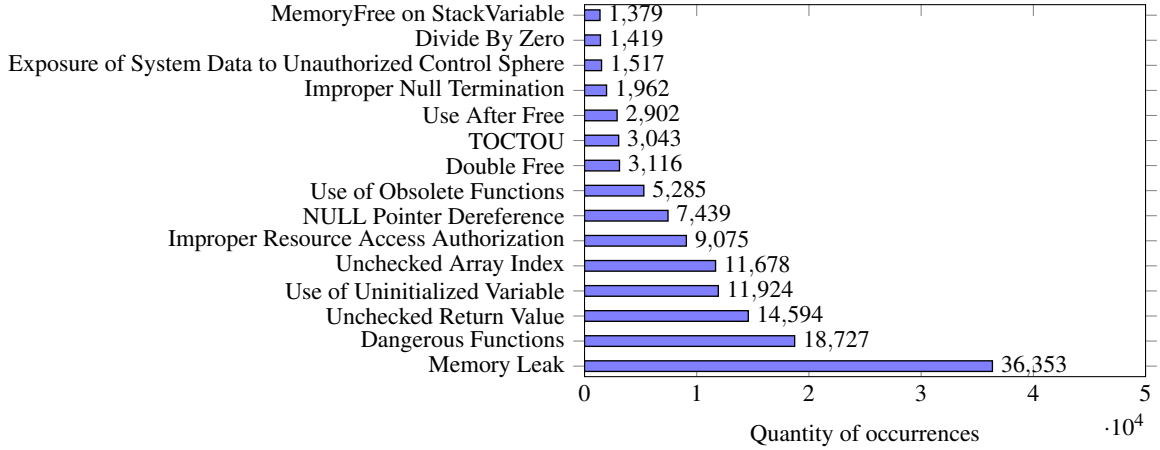

Figure 1: Top 15 Most Frequently Identified Security Vulnerabilities.

Table 2: Vulnerabilities with Correlation Coefficient Greater than 0.75.

Vulnerability ID	Corr. Coef. >0.75
15	35, 74
26	42
30	32
35	74
36	41, 62, 74
38	42, 71
39	59
41	62
60	61
62	65
64	76
73	78

It can be inferred from examining the correlation between vulnerabilities 26 and 42 that information left in comments within the source code can provide clues that may enable hash length extension attacks, compromising data integrity.

Examining the correlation between vulnerabilities 30 and 32, this can be explained as follows: second-order SQL injections can be used to exploit buffer overflow, which could compromise data integrity or availability.

The correlation coefficient between vulnerabilities 35 and 74 is 0.84. Deprecated functions may lack the necessary controls and validations to handle and/or return strings with incorrect termination, creating security vulnerabilities that malicious agents can exploit.

Vulnerabilities 36 and 41 can be correlated because they are related to access critical resources. In this scenario, inadequate authorization for resource access and improper permission assignment can allow unauthorized users to access critical resources.

4.3 Association Rules Learning

The correlation analysis revealed the statistical relationships among vulnerabilities, but whether certain security issues co-occur frequently in real-world projects remains to be seen. To answer this question, the Apriori algorithm implemented in the *mlxtend* Python library was employed to discover association rules. Setting a support threshold of 80% the analysis identified pairs of frequently co-occurring vulnerabilities. Additionally, it pinpointed those most commonly reported across projects, highlighting their direct association with prevalent vulnerabilities across all projects evaluated. These findings are detailed in Table 3. Based on these results, we concluded that Vulnerabilities 15, 62, 64, 65, 74 tend to occur in most projects.

Table 3: Association Rules Results - Support 80%.

Security Issues	Support
(15)	0.88
(62)	0.84
(64)	0.87
(65)	0.94
(74)	0.82
(15, 64)	0.84
(15, 65)	0.87
(62, 65)	0.82
(64, 65)	0.87
(65, 74)	0.81
(15, 64, 65)	0.84

The subsequent step involves deriving trust-based association rules. The same library mentioned for the previous step was used to achieve this, establishing a confidence limit set at 90%. The results of this analysis are presented in Table 4.

Table 4: Association Rules - Confidence 90 %.

Rule	Precedent Support	Consequent Support	Rule Support	Confidence	Lift	Conviction
(64) ⇒ (15)	0.87	0.88	0.84	0.96	1.08	2.62
(15) ⇒ (64)	0.88	0.87	0.84	0.94	1.08	2.3
(65) ⇒ (15)	0.94	0.88	0.87	0.93	1.05	1.61
(15) ⇒ (65)	0.88	0.94	0.87	0.99	1.05	5.31
(62) ⇒ (65)	0.84	0.94	0.82	0.98	1.04	2.51
(65) ⇒ (64)	0.94	0.87	0.87	0.93	1.06	1.75
(64) ⇒ (65)	0.87	0.94	0.87	1.00	1.06	inf
(74) ⇒ (65)	0.82	0.94	0.81	0.99	1.05	4.90
(15, 65) ⇒ (64)	0.87	0.87	0.84	0.96	1.09	2.84
(64, 65) ⇒ (15)	0.87	0.88	0.84	0.96	1.08	2.62
(15, 64) ⇒ (65)	0.84	0.94	0.84	1.00	1.06	inf
(15) ⇒ (64, 65)	0.88	0.87	0.84	0.95	1.08	2.30
(64) ⇒ (15, 65)	0.87	0.87	0.84	0.96	1.09	2.84

4.4 Clustering

This study investigated the potential for identifying groups of security vulnerabilities that occur together. Given the unknown number of underlying clusters, Affinity Propagation was employed as an initial clustering method. This technique automatically determines the optimal number of clusters, which in this case was eighteen. Next, K-means and Hierarchical Agglomerative Clustering were applied to validate the initial clustering and assess consistency. The number of clusters in these methods was set to eighteen to align with the Affinity Propagation results. The clusters generated by the 3 algorithms differ slightly from each other. For analysis purposes, Hierarchical Agglomerative Clustering clusters will be used. Following the clustering process, a Cybersecurity specialist analyzed the resulting groupings to identify any relationships with Vulnerabilities within each cluster.

This analysis revealed the following about Cluster 1: the lack of index boundary checks for an array and the use of uninitialized variables can result in unexpected behavior and the possibility of unintended memory read-and-write operations, compromising the integrity of a system's data.

Examining Cluster 2, this association can be explained as follows: the use of dangerous functions combined with deprecated functions can create unpredictable scenarios and security vulnerabilities, as deprecated functions may have known vulnerabilities that are not properly addressed by dangerous functions.

Cluster 5 can indicate that both vulnerabilities occur when there are issues with memory deallocation. A Use After Free may occur when attempting to use a stack memory area that a memory function has freed. The Use After Free would occur in this scenario due to a MemoryFree on a StackVariable.

Examining Cluster 6, it can be seen that the combination of these vulnerabilities can allow a malicious agent to access sensitive files, manipulate system memory, and execute malicious commands.

Table 5: Hierarchical Agglomerative Clustering Results.

Cluster ID	Vulnerabilities ID
1	64, 76
2	15, 74
3	1, 16, 63, 66
4	2, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 17, 19, 20, 22, 23, 24, 25, 26, 28, 29, 30, 32, 33, 38, 39, 40, 42, 43, 44, 46, 50, 51, 53, 54, 55, 56, 57, 59, 60, 61, 67, 69, 70, 71, 72, 77, 79
5	48, 68
6	3, 52, 58
7	41, 62
8	6, 31
9	27, 34, 37
10	49
11	65
12	21
13	73, 78
14	35, 75
15	36
16	45
17	47
18	18

Examining Cluster 7, this association can be explained as follows: incorrect permission assignment for access to critical resources in a system with TOCTOU vulnerabilities can allow attackers to access or modify these resources in an unauthorized manner between the check and use.

Furthermore, some clusters contained a single Vulnerability, implying a higher degree of specificity for these issues and a lower likelihood of co-occurrence with other Vulnerabilities.

5 CONCLUSION

This paper presents an exploratory study of security vulnerabilities in C++ source code from very large projects. We analyzed twenty-six worldwide C++ projects and empirically studied the prevalence of security vulnerabilities. From this exploratory study we have answered four research questions (RQs). RQ1 - How often are code vulnerabilities in real C++ projects? Some vulnerabilities are more frequent than others. Besides, some vulnerabilities were not found in the selected projects. RQ2 - Are there significant correlations between pairs of vulnerabilities? Our results showed that many pairs of vulnerabilities have high correlation coefficients, over 0.6. RQ3 - Are there code vulnerabilities that occur together in real C++ projects? We have explored the Apriori algorithm and found some interesting association rules, which indicate that there are code vulnerabilities that occur together. RQ4 - Is possible to find groups of vulnerabilities that occur together in real C++ projects? We found two 13 clusters of code vulnerabilities. Leveraging these insights, this paper has the potential to assist software developers and security experts in mitigating future problems during the development of C++ projects. In future work, we intend to investigate the occurrence of false positives and negatives in security reports. Moreover, we will use machine learning techniques to predict code vulnerabilities.

ACKNOWLEDGMENTS

This work was partially funded by Lenovo as part of its R&D investment under the Information Technology Law. The authors would like to thank LSBD/UFC for the partial funding of this research.

REFERENCES

Agrawal, R. S. and Srikant, R. (1994). R. fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB*, pages 487–499.

Alqaradaghi, M. and Kozsik, T. (2024). Comprehensive evaluation of static analysis tools for their performance in finding vulnerabilities in java code. *IEEE Access*.

Bagheri, E. and Gasevic, D. (2011). Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal*, 19:579–612.

Berger, T. and Guo, J. (2014). Towards system analysis with variability model metrics. In *Proceedings of the 8th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 1–8.

Do, L. N. Q., Wright, J. R., and Ali, K. (2022). Why do software developers use static analysis tools? A user-centered study of developer needs and motivations. *IEEE Trans. Software Eng.*, 48(3):835–847.

Fan, J., Li, Y., Wang, S., and Nguyen, T. N. (2020). Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th international conference on mining software repositories*, pages 508–512.

Hussain, S., Anwaar, H., Sultan, K., Mahmud, U., Farooqui, S., Karamat, T., and Toure, I. K. (2024). Mitigating software vulnerabilities through secure software development with a policy-driven waterfall model. *Journal of Engineering*, 2024(1):9962691.

Islam, N. T., Karkevandi, M. B., and Najafirad, P. (2024). Code security vulnerability repair using reinforcement learning with large language models. *arXiv preprint arXiv:2401.07031*.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., and Chen, Z. (2021). Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258.

Ma, L., Yang, H., Xu, J., Yang, Z., Lao, Q., and Yuan, D. (2022). Code analysis with static application security testing for python program. *J. Signal Process. Syst.*, 94(11):1169–1182.

McCrum-Gardner, E. (2008). Which is the correct statistical test to use? *British Journal of Oral and Maxillofacial Surgery*, 46(1):38–41.

Mehrpour, S. and LaToza, T. D. (2023). Can static analysis tools find more defects? *Empir. Softw. Eng.*, 28(1):5.

Moyo, S. and Mnkandla, E. (2020). A novel lightweight solo software development methodology with optimum security practices. *IEEE Access*, 8:33735–33747.

Nguyen-Duc, A., Do, M. V., Hong, Q. L., Khac, K. N., and Quang, A. N. (2021). On the adoption of static analysis for software security assessment-a case study of an open-source e-government project. *Comput. Secur.*, 111:102470.

Novak, J., Krajnc, A., et al. (2010). Taxonomy of static code analysis tools. In *The 33rd international convention MIPRO*, pages 418–422. IEEE.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Raschka, S. (2018). Mlxtend: Providing machine learning and data science utilities and extensions to python's scientific computing stack. *Journal of open source software*, 3(24):638.