Exploratory Evaluation of Secure Design Methodologies Between Security and Code Quality

Arthur Alexi
Department of Software Engineering, Pontifical Catholic University of Minas Gerais (PUC Minas)
Belo Horizonte, Minas Gerais, Brazil
arthuralexi@hotmail.com

Yghor Ribas
Department of Software Engineering, Pontifical Catholic University of Minas Gerais (PUC Minas)
Belo Horizonte, Minas Gerais, Brazil
Yghorpb@gmail.com

Hugo Bastos
Department of Software Engineering, Pontifical Catholic University of Minas Gerais (PUC Minas)
Belo Horizonte, Minas Gerais, Brazil
hugo@pucminas.br

Aline Brito
Department of Computing, Federal University of Ouro Preto (UFOP)
Ouro Preto, Minas Gerais, Brazil
aline.brito@ufop.edu.br

Leonardo Cardoso
Department of Software Engineering, Pontifical Catholic University of Minas Gerais (PUC Minas)
Belo Horizonte, Minas Gerais, Brazil
leonardocardoso@pucminas.br

Cleiton Tavares
Department of Software Engineering, Pontifical Catholic University of Minas Gerais (PUC Minas)
Belo Horizonte, Minas Gerais, Brazil
cleitontavares@pucminas.br

ABSTRACT

This paper investigates the impacts of secure design methodologies on the structural quality and presence of vulnerabilities in systems developed in Java. We analyzed 333 popular GitHub repositories, classified according to four security approaches: Security by Design, Defense in Depth, Runtime Security and Zero Trust Architecture. Using static analysis techniques, we evaluated vulnerabilities based on the CWE taxonomy, in addition to code quality metrics such as coupling (CBO), cohesion (LCOM) and cyclomatic complexity. The results indicate that the adoption of secure design practices, although relevant for risk mitigation, can negatively impact the modularity of the system. In particular, approaches such as Security by Design presented higher levels of coupling and lower cohesion. We also observed that the combination of multiple security practices did not necessarily result in a lower number of vulnerabilities, suggesting an increase in complexity without proportional benefit. These findings highlight the importance of balanced architectural decisions that consider the commitments between security and internal software quality.

KEYWORDS

Secure Design, Secure Design Methodologies, Code Quality

1 Introduction

System design involves a series of strategic decisions that aim to balance multiple, and often conflicting, requirements such as performance, scalability, cost, maintainability, and security. Among these, security has emerged as both a priority and a source of tension, particularly as systems grow increasingly complex and interconnected. While secure design methodologies — such as *Threat Modeling*, the *Principle of Least Privilege*, and *Security by Design* — are essential for mitigating vulnerabilities, they frequently introduce trade-offs that affect other critical quality attributes. For instance, certain security mechanisms may increase system complexity, reduce modularity, or degrade performance and scalability [4].

The central research question of this study concerns the challenge of balancing security requirements with other software quality attributes. In microservices architectures, Ponce et al. [10] show that the implementation of security strategies, such as monitoring for *security smells*, contributes to the mitigation of vulnerabilities, although it may compromise the structural integrity of the software. These findings highlight the core issue addressed in this work: *How do secure design methodologies affect the structural quality of software?* Structural quality, in this context, refers to the degree to which the source code and system architecture satisfy requirements such as modularity, independently of the system's external behavior.

Understanding how security methodologies impact system design requirements is increasingly relevant as society becomes more dependent on software systems across all sectors [4, 8]. Failure to implement adequate security measures can lead to severe consequences such as data loss, privacy breaches, and even large-scale cyberattacks, ultimately affecting the trust and integrity of critical systems [9, 13]. However, solutions that prioritize security alone may negatively impact the software's internal quality, potentially limiting the adoption of technologies or introducing unintended vulnerabilities due to increased complexity [11]. Therefore, addressing this problem not only contributes to more secure systems but also ensures that design trade-offs are sustainable and appropriate for different usage contexts [6].

The goal of this study is identify how secure design methodologies influence structural quality metrics. To achieve this, the following specific objectives were defined: (i) categorically identify projects that adopt secure design methodologies; (ii) conduct a quantitative analysis of security metrics based on the Common Weakness Enumeration (CWE) standard and internal quality indicators of those projects; and (iii) investigate whether the adoption of one or more secure design methodologies correlates with vulnerability risk. Accordingly, the study sought to answer the following research questions:

RQ1 . What are the most frequent CWE in Java repositories, and how are they correlated with the adopted security methodologies?

RQ2 . Is there a relationship between security methodology and internal software quality?

RQ3 . Do repositories with a greater adoption of secure design practices present fewer CWE?

This study contributes to understanding the relationship between the adoption of secure design methodologies and internal software quality. The analysis revealed that certain approaches, such as Security by Design, were associated with structural code properties, such as increased coupling and decreased cohesion. Additionally, it was observed that the combination of multiple security practices did not necessarily lead to a reduction in vulnerabilities, suggesting an increase in system complexity without proportional benefits in risk mitigation. These findings may support decision-making processes when balancing security requirements and software maintainability.

This paper is structured as follows: Section 2 presents the theoretical background, addressing key concepts and secure design methodologies; Section 3 discusses related work; Section 4 details the study design used; Section 5 presents the results obtained from metric collection; Section 6 discusses the findings; Section 7 presents threats to validity; and finally, Section 8 concludes with final remarks and suggestions for future research.

2 Background

To gain a deeper understanding of the challenges related to system design with a focus on security requirements, this section presents a theoretical discussion of the key concepts underpinning this study.

2.1 Secure Design Methodologies

Secure Design Methodologies are structured approaches that integrate security principles and practices across all phases of the system development lifecycle, from initial conception to implementation and maintenance. The main goal of these methodologies is to ensure that security is proactively considered and directly incorporated into the system's design, rather than being treated as a corrective step or an afterthought after development [2].

Secure design methodologies aim to minimize security risks without compromising the system's efficiency, scalability, or usability, as detailed in the following approaches:

- *Security by Design*: This approach prioritizes security from the early stages of the development lifecycle, making it a core element of system design. It includes practices such as access controls, encryption, and continuous security audits, helping to reduce the cost of later corrections and prevent vulnerabilities [6].
- *Defense in Depth*: This methodology applies multiple layers of security to hinder an attacker's progress, thereby making the system more resilient. Barriers such as authentication, encryption, and continuous monitoring significantly strengthen protection against unauthorized access [2].
- *Runtime Security*: Runtime security monitors and detects suspicious behavior while the system is in operation. Specialized tools analyze real-time activity and respond effectively to emerging threats, as emphasized in modern system architectures [13].
- *Zero Trust Architecture*: This approach follows the principle of "never trust, always verify," requiring strict authentication for every request, regardless of origin. It is particularly effective at mitigating insider threats and assumes that no connection is secure by default [1].

2.2 Common Weakness Enumeration

The *Common Weakness Enumeration* (CWE) is a taxonomy maintained by the *MITRE Corporation* that categorizes and documents common software vulnerabilities. CWE are generic patterns of weaknesses that can lead to security issues across various development contexts. This classification is widely used by security tools to identify, measure, and mitigate vulnerabilities in the codebase.

The presence of these vulnerabilities can compromise the overall security of a project by affecting its reliability, integrity, and availability. Moreover, inadequate security practices may negatively impact system performance, as corrective mechanisms such as additional checks or emergency patches can introduce computational overhead.

3 Related Work

The secure design of systems is a widely studied topic due to the growing need to balance security with performance and usability across various architectures. This section explores studies that address related problems and offer complementary methodologies to the present research, discussing the contributions and relevance of each work.

Fahmideh et al. [4] conducted an extensive review on the engineering of blockchain-based software systems, covering foundational aspects and key security challenges in distributed systems. The study examines the application of cryptographic methods and secure consensus mechanisms to ensure data protection and prevent fraud in blockchain systems. The authors found that while robust security mechanisms are essential, their implementation may impact system scalability and usability. This work is relevant to the proposed study because it emphasizes the inherent trade-offs between security and performance, a theme also explored in this paper, though in the context of microservices rather than blockchain. While their analysis is more theoretical and centered on architecture-level concerns, this work complements it by empirically assessing how security design choices affect code-level quality in real-world Java systems.

Ponce et al. [10] introduced the *TriSS* tool, designed to identify and prioritize security smells in microservices architectures. The tool assists developers in detecting and categorizing vulnerabilities by applying a triage algorithm that prioritizes issues with greater security impact. Results demonstrate that *TriSS* effectively highlights critical problems such as over-permissive access and insecure communication, which often compromise system integrity. While their study focuses on detection tools and prioritization of known vulnerabilities, our work investigates how different secure design methodologies, potentially including the practices targeted by TriSS,

influence overall code structure and vulnerability incidence. Together, these works provide a broader view of both detection and design implications in secure systems.

Mohammed and Mohammed [8] proposed a security architecture aimed at protecting sensitive data in cloud computing environments. The paper explores approaches such as advanced encryption, access control, and identity management to enhance data security in the cloud, discussing the associated challenges and benefits. The authors revealed that strong encryption and data segmentation are effective strategies for improving security, although they can introduce latency and system complexity. While focused on cloud environments, this work reinforces the broader pattern observed in our results, that increased security measures can come at the cost of complexity and performance. This study builds on this insight by quantifying such trade-offs across multiple methodologies in Java projects.

In the field of mobile payment security, Alamleh et al. [1] developed a secure payment architecture incorporating multi-factor authentication to reduce fraud and strengthen transaction security. The authors detail the development of a system combining biometric authentication with traditional methods, evaluated in terms of both effectiveness and usability. The results show that multi-factor authentication effectively reduces fraud risk but introduces complexity in the authentication process, potentially impacting user experience. This study supports the idea that security mechanisms often introduce usability or maintenance costs—an observation aligned with findings from this paper, which show that secure design methodologies can increase code complexity without proportionally reducing vulnerabilities.

Chengjie et al. [3] proposed a cloud-based security monitoring platform for the salt and alkali industry. The authors explored the use of cloud infrastructure to manage security data and support real-time emergency responses. The study found the system to be effective in ensuring data security and integrity, especially in high-risk situations, underscoring the importance of secure communication in critical systems. Although focused on a specific industrial context, this study illustrates the value of proactive security design. This research complements this by evaluating whether such proactive designs, when applied broadly, correlate with actual reductions in vulnerability occurrence or code degradation.

Mechri et al. [7] addresses the growing challenge of identifying vulnerabilities in large Python codebases, particularly given the diversity and complexity of security flaws. To address this challenge, the study leverages the *CWE* taxonomy as a foundation for categorizing and classifying detected vulnerabilities, enabling the large language model (LLM) to learn specific patterns of software weaknesses. This study directly informs our methodology by validating the use of the CWE taxonomy for vulnerability detection, which was adopt through the Bearer tool. While their work uses LLMs to assist in this task, our contribution lies in linking such detection results to the broader context of design methodology adoption and code quality.

4 Study Design

This study is classified as exploratory, quantitative, and descriptive research, with a focus on analyzing different secure design approaches. The choice of this research type is grounded in the need to collect measurable data in order to assess the impact of security strategies on the system. This facilitates comparisons and supports the identification of recurring patterns among repositories that adopt secure design methodologies. The experiment followed a linear approach, as illustrated in Figure 1 and detailed as follows.

This study focuses on popular Java repositories due to the language's strong ecosystem, mature tooling, and widespread use in security-critical systems. However, this choice limits the generalizability of the results to Java-based contexts. Language-specific design patterns, dependency structures, and development practices may influence the findings and should be taken into account when applying them to other ecosystems. These implications are further discussed in Section 7.

4.1 Steps Description

The steps carried out throughout the project are detailed as follows:

- **Classification of *Maven* artifacts:** In this step, the Maven Repository artifacts were ranked according to the security design methodology they implement. To achieve this, a Python script was developed to interact with the Maven Repository API using the keyword security and ordered by score, based on relevance, popularity, and general usage within the community, in descending order. After collection, the Google Search API was used to locate the corresponding GitHub repositories for each artifact. For methodology classification, a scoring scheme was adopted that considered keywords. In the case of tied scores, the artifact was replicated across each of the tied methodologies.
- **Repository selection:** This step involved the development of a Python script to collect repositories from GitHub via the GraphQL API. Repository selection was based on two main criteria: primary language (*Java*) and popularity (number of stars). From the top 1,000 repositories, each was classified into one or more secure design methodologies. The script scanned the project description, README.md contents, and relevant configuration files (e.g., *pom.xml*, *build.gradle*), identifying keywords and dependencies previously classified in the prior step. In case of ties, the repository was replicated for each tied methodology.
- **Static code analysis:** At this stage, a script was used to apply the *Bearer*[1] tool to the repositories. *Bearer* is an automated security analysis tool capable of identifying weaknesses in source code. It uses the Common Weakness Enumeration (CWE) taxonomy, a widely adopted catalog for classifying and describing common software security flaws.
- **Data visualization:** The dataset collected in the previous steps was used to generate charts and statistical analyses. The statistical analysis was performed in Google Colab, and the *SciPy* and *Matplotlib* libraries were used for graph generation.
- **Results:**The final stage consisted of a thorough analysis of the collected data, as well as a critical examination of the questions outlined in Section 1.

[1] https://github.com/Bearer/bearer

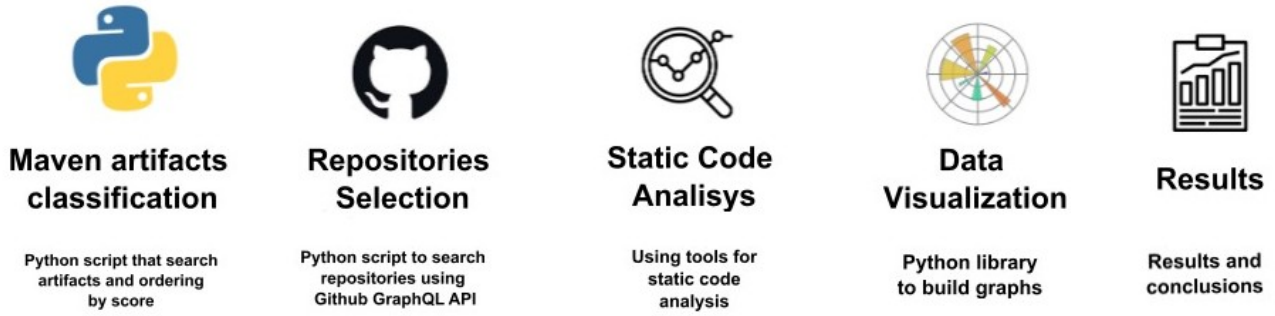

Figure 1: Study Steps

4.2 Evaluation Metrics

Evaluation metrics play a key role in objectively quantifying the structural quality and security of source code. These indicators provide empirical evidence to assess the maintainability, robustness, and reliability of a system. Analyzing these indicators allows early identification of design flaws and vulnerabilities, supporting software refactoring and enhancement processes [5].

- **Vulnerability Detection:** This metric assessed the number of security weaknesses identified in the collected repositories, based on the analysis performed by the *Bearer* tool. It was essential for evaluating the effectiveness of applied security practices, considering the vulnerabilities cataloged in the CWE.
- **Structural Quality Metrics:** Three code quality metrics were extracted using the *CK*[2] and *PMD*[3] tools: Coupling Between Objects (CBO), Lack of Cohesion in Methods (LCOM), and Cyclomatic Complexity. These metrics were computed from the source code of the repositories using static analysis tools compatible with Java projects. Each metric was normalized by the total Lines of Code (LOC) of each repository to enable comparisons between projects of different sizes.

5 Results

This section presents the results obtained in each stage of the study.

5.1 Dataset Characterization

From the analysis of the top 1,000 Java repositories on GitHub, 333 unique repositories were identified that could be classified according to different secure design methodologies. Figure 2 shows the distribution of these classifications. It is important to note that some repositories were associated with more than one methodology.

Among the analyzed categories, *Defense in Depth* stands out as the most frequent, being identified in approximately 190 repositories. *Zero Trust Architecture* appears in about 160 cases. On the other hand, categories such as *Runtime Security* and *Security by Design* are significantly less frequent, with about 75 and 55 occurrences, respectively.

Figure 3 shows the distribution of repositories based on the number of secure design methodologies they employ. The majority

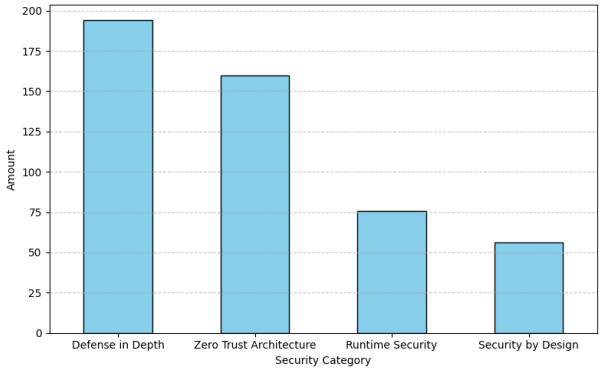

Figure 2: Count of security category in Java repositories

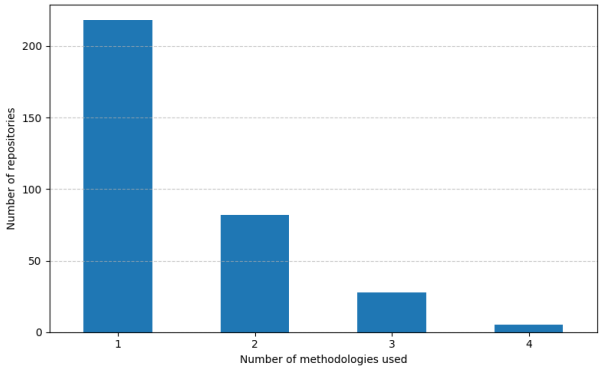

Figure 3: Distribution of repositories by number of methodologies

of repositories adopt only one methodology, suggesting a limited integration of diverse security strategies. As the number of methodologies increases, the number of repositories decreases sharply, indicating that multi-methodology adoption is relatively rare. This distribution highlights a potential area for improvement in secure system design practices, encouraging broader incorporation of complementary methodologies.

[2]https://github.com/mauricioaniche/ck
[3]https://github.com/pmd/pmd

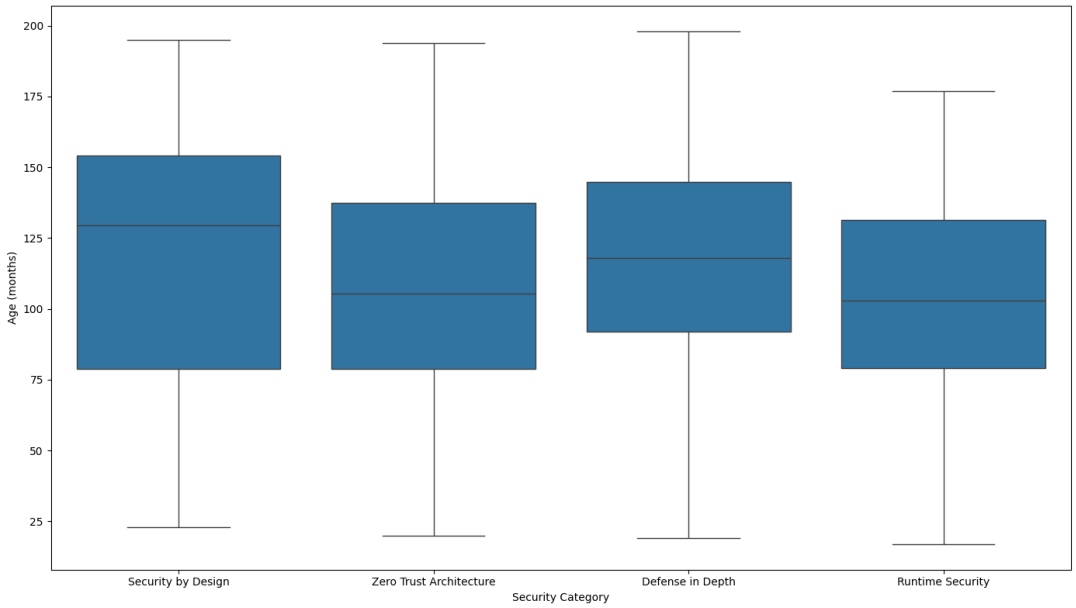

Figure 4: Distribution of repository age (in months) by security category

Table 1: Code quality metrics by security category

Category	CBO			LCOM			Cyclomatic Complexity		
	Mean	Std Dev	Median	Mean	Std Dev	Median	Mean	Std Dev	Median
Defense in Depth	5.6885	1.5624	5.4706	541.7429	4907.1321	28.4984	37.8470	27.9710	31.5294
Runtime Security	5.8585	1.9545	5.8155	60.7092	86.4668	27.7700	35.4457	18.3320	29.8545
Security by Design	6.3556	1.1144	6.3546	78.4182	140.0899	30.6891	29.3886	13.7678	28.6588
Zero Trust Architecture	5.8465	1.7375	5.7540	106.9017	231.3824	25.5964	38.5003	29.9745	30.9276

Figure 4 presents the age in months of the selected repositories. Understanding this data may be useful for assessing any correlation between the methodology used and the age or maturity of the repository. It is observed that repositories classified as *Security by Design* are generally older and show greater age variability, with a median near 125 months. *Defense in Depth* also shows large dispersion, but with a slightly lower median, around 115 months. Meanwhile, categories like *Zero Trust Architecture* and *Runtime Security* are concentrated in smaller ranges, with *Runtime Security* showing the lowest dispersion.

5.2 Analysis of Quality Metrics by Security Category

Table 1 presents the means, standard deviations, and medians of the code quality metrics CBO, LCOM, and Cyclomatic Complexity, by secure design methodology. It is noted that the *Security by Design* category has the highest average CBO (6.36) and LCOM (78.42) values, suggesting higher coupling and lower cohesion between classes. In contrast, *Defense in Depth* stands out with the highest average LCOM (541.74), although with high variability (standard deviation of 4907.13), indicating the presence of outliers. Regarding cyclomatic complexity, the average values among categories are similar, ranging from 29.39 to 38.50, suggesting that regardless of the adopted security approach, repositories show comparable levels of method complexity.

5.3 Most Frequent *CWE* in Repositories

From the static code analysis, several security weaknesses were identified across the analyzed repositories. This section presents the most frequently observed weaknesses in the sample. Figure 5 displays the top 10 most common *CWE* among the repositories, highlighting issues such as sensitive data leakage, use of insecure random values, and hardcoded credentials.

Complementarily, Figure 6 shows the distribution of unique *CWE* per repository in each security category. It is noted that categories such as *Security by Design* and *Zero Trust Architecture* display a higher median of distinct weaknesses per repository, which may reflect a greater diversity or complexity in the attack surfaces of those systems. To mitigate biases due to imbalance in the number of repositories per category, the *undersampling* technique was applied, which involved matching the sample size of each category based on the smallest available count. This normalization aims to ensure a fairer comparison between distributions.

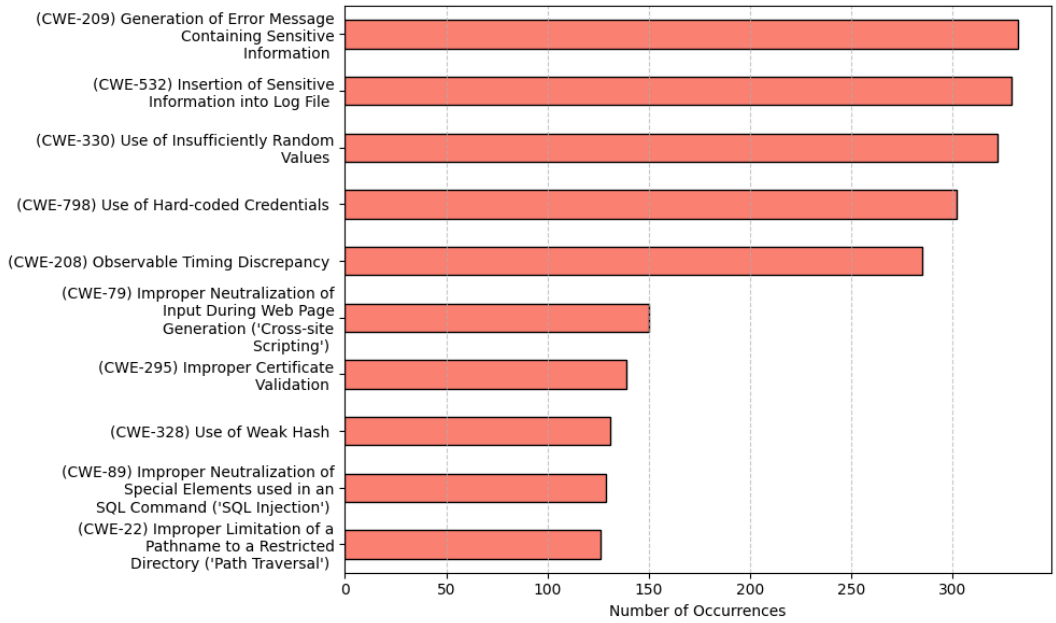

Figure 5: Top 10 *CWE* in repositories

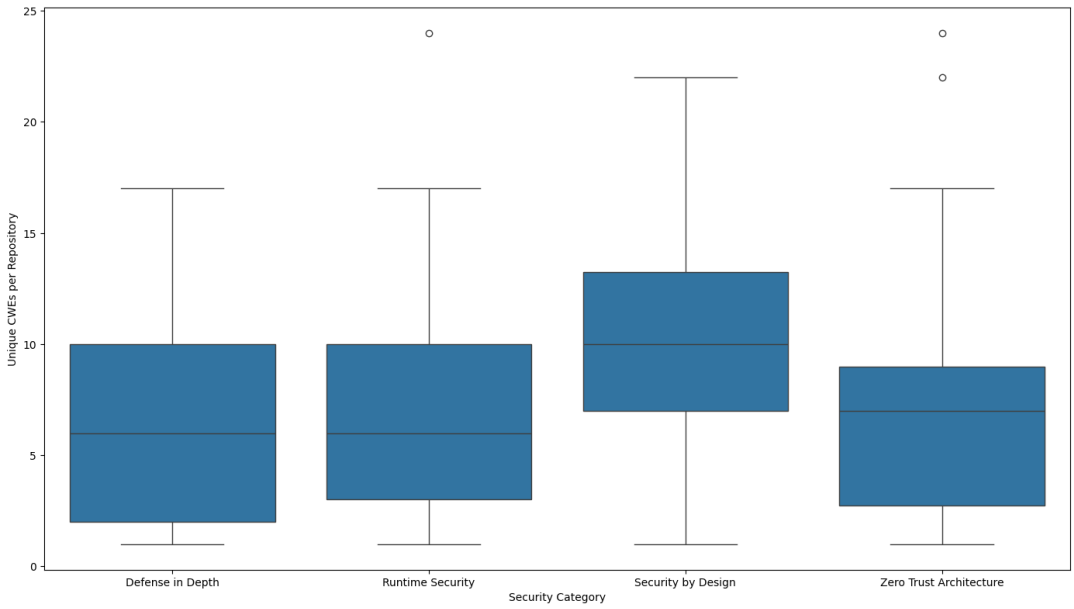

Figure 6: Distribution of unique *CWE* per repository in each security category (normalized)

5.4 Relationship Between Maturity and Weakness Variability

The Spearman correlation analysis between the age of the security approaches and the number of unique *CWE* aimed to investigate whether the adoption time of such strategies could be associated with quality issues in the projects. This finding is relevant because it allows us to rule out age as a central factor in explaining quality issues, reinforcing the importance of analyzing the security strategies themselves as possible determinants of project robustness.

Table 2: Relationship between repository age and occurrence of unique *CWE*

Category	Age (months)		Unique *CWEs*		Spearman	p-value
	Mean	Median	Mean	Median		
Defense in Depth	115.28	116.5	6.30	5.0	-0.02	0.772
Runtime Security	103.07	101.5	7.45	6.5	-0.03	0.823
Security by Design	118.14	128.0	10.05	10.0	-0.16	0.244
Zero Trust Architecture	106.66	103.5	7.59	7.0	-0.05	0.527

Table 2 presents the mean and median of repository age and unique *CWE*, along with the Spearman correlation coefficient and its respective p-value. The Spearman correlation analysis between repository age and the number of unique *CWEs* revealed no significant correlation between these variables in any category. Correlation coefficients ranged from -0.16 to -0.02, with p-values well above the conventional threshold for statistical significance ($p < 0.05$).

The *Security by Design* category presented the most negative coefficient (ρ = -0.16; p = 0.244), suggesting a slight tendency for the number of unique *CWE* to decrease as age increases. However, this relationship is not statistically significant and therefore does not allow us to confidently affirm a pattern. The other categories exhibited coefficients even closer to zero (ranging from -0.05 to -0.02), reinforcing the absence of correlation.

6 Discussion of Results

RQ1 . What are the most frequent CWE in Java repositories, and how are they correlated with the adopted security methodologies?

Figure 7 presents the relative frequency of CWE-IDs by security category, considering the analyzed repositories that were grouped according to a secure design methodology. The analysis of the relative frequency of CWE-IDs in Java repositories reveals that certain vulnerabilities occur more frequently across the different secure design methodologies. CWE-IDs 209 (*Generation of Error Message containing Sensitive Information*), 532 (*Insertion of Sensitive Information into Log File*), 330 (*Use of Insufficiently Random Values*), and 798 (*Hardcoded Credentials*) stand out as the most frequent in the systems analyzed. The heatmap shows that these vulnerabilities present high relative frequency across all four evaluated categories: *Defense in Depth*, *Runtime Security*, *Security by Design*, and *Zero Trust Architecture*.

Normality tests, conducted using the *Shapiro-Wilk* test, indicated that the distribution of frequencies by category does not follow a normal distribution ($p < 0.001$ in all cases), which justifies the adoption of non-parametric statistical methods in the analyses. Applying the *Kruskal-Wallis* test, no statistically significant differences were found between secure design methodologies regarding the distribution of relative CWE frequencies ($H = 3.4253,\ p = 0.3306$).

RQ2 . Is there a relationship between security methodology and internal software quality?

Table 1, presented the mean, standard deviation, and median values of internal software quality metrics by security category, highlighting objective numerical differences between the groups. To assess the statistical significance of these differences, non-parametric tests were conducted, which indicated significance in the metrics of coupling (CBO), cohesion (LCOM), and cyclomatic complexity.

The relationship between secure design methodologies and internal software quality was investigated using non-parametric tests, as the data did not meet the assumptions of normality. The *Kruskal-Wallis* tests revealed statistically significant differences among the groups for all quality metrics analyzed: coupling (CBO, $H = 9.5950,\ p = 0.0223$), cohesion (LCOM, $H = 14.0132, p = 0.0029$), and cyclomatic complexity ($H = 16.9450, p = 0.0007$).

Post-hoc analyses using Dunn's test with Bonferroni correction indicated that the *Security by Design* methodology showed significant differences compared to the other groups in all three metrics ($p < 0.05$ in all cases). Statistically significant results were also observed for the *Defense in Depth* methodology in cohesion and complexity, and for *Runtime Security* in LCOM. Table 3 presents the consolidated test results.

Based on these findings, it is observed that variation in internal software quality is more strongly associated with certain secure design methodologies than with others. The *Security by Design* methodology, in particular, presented significant differences in all three analyzed metrics. Nonetheless, the interpretation of these results should consider possible contextual factors and dataset limitations.

RQ3 . Do repositories with a greater adoption of secure design practices present fewer CWE?

The relationship between the adoption of secure design practices and the occurrence of critical vulnerabilities was assessed through the correlation between the number of security methods identified in the repositories and the number of distinct CWE-IDs. The distribution of both datasets was analyzed using the *Shapiro-Wilk* test. Subsequently, Spearman's correlation coefficient was applied, resulting in a value of 0.318. This coefficient indicates a weak to moderate positive correlation between the number of security methods and the number of CWE vulnerabilities, suggesting that repositories implementing more security practices do not necessarily exhibit fewer vulnerabilities. In fact, this may reflect increased complexity in handling security concerns.

Contrary to expectations, repositories implementing more security practices did not necessarily exhibit fewer vulnerabilities. Our analysis revealed a weak to moderate positive correlation (Spearman's $p = 0.318$) between the number of security methods and CWEs, suggesting that adding security layers may increase complexity without a proportional reduction in risk.

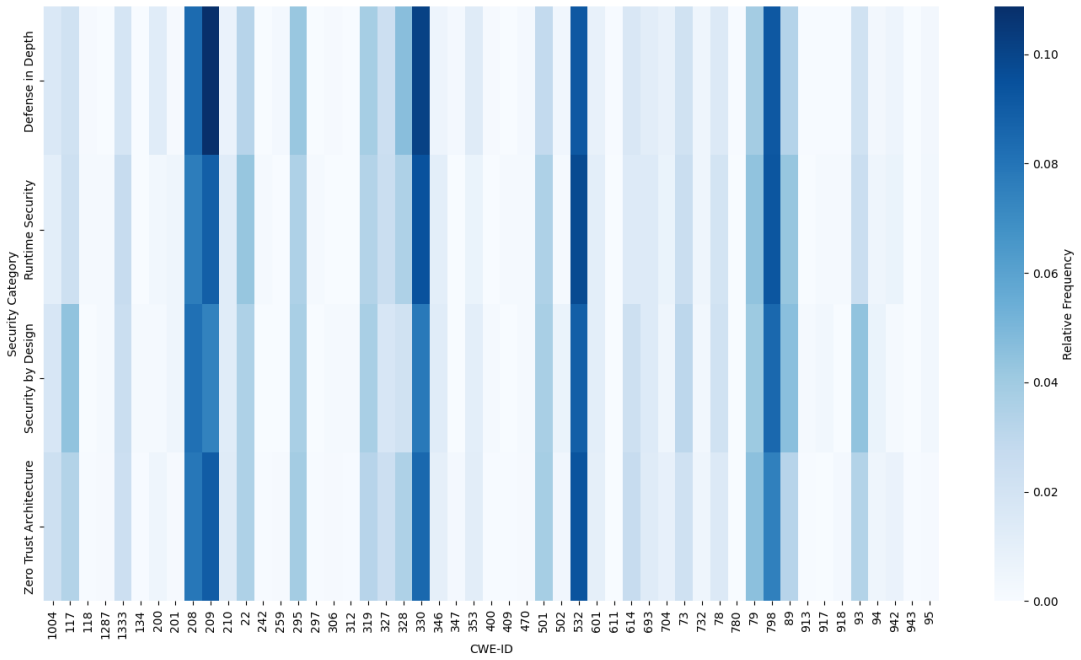

Figure 7: Relative frequency of CWE in repositories grouped by secure design methodology

Table 3: Kruskal-Wallis test results by category (p-values)

Category	CBO (p-value)	LCOM (p-value)	COMPLEXITY (p-value)
Defense in Depth	0.2065	**0.0167**	**0.0369**
Runtime Security	0.1249	**0.0466**	0.0791
Security by Design	**0.0065**	**0.0044**	**0.0003**
Zero Trust Architecture	0.4985	0.5427	0.4712

7 Threats to Validity

Every empirical study in software engineering is subject to different threats to validity, which must be addressed to strengthen the reliability of the conclusions [12]. This section discusses the main threats to the validity of this study's results and the strategies adopted to mitigate them. The threats are organized into the following categories: *internal*, *external*, *construct*, and *conclusion* validity.

Internal Validity. Internal validity concerns the ability to establish a causal relationship between the adopted secure design methodologies and the internal quality of the code. A relevant threat is the potential bias in classifying repositories into secure design methodologies (*Security by Design*, *Defense in Depth*, *Runtime Security*, *Zero Trust Architecture*), which may impact the results of static analysis. To mitigate this threat, we employed an automated process based on keywords extracted from technical documentation (*README*, `pom.xml`, `build.gradle`, etc.), as well as from previously classified dependencies. Nonetheless, we acknowledge that this classification may contain noise due to linguistic ambiguity or incomplete documentation.

External Validity. External validity refers to the generalization of results to other contexts or populations. In this study, the analyzed repositories were extracted from GitHub and filtered by popularity (stars) and primary language (Java), which may introduce a bias regarding lesser-known projects or those hosted on other platforms. To reduce this impact, we chose to use Maven Central as the starting point, as it represents a widely used and relevant subset within the Java ecosystem, increasing the likelihood of representativeness. However, extrapolation to other development environments should be approached with caution.

Construct Validity. Construct validity relates to how well the metrics and instruments used represent the concepts intended to be measured. In this case, we used the Bearer tool to identify security weaknesses based on the CWE. A possible threat lies in the partial coverage of CWE by the tool or the occurrence of false positives/negatives. The Bearer tool was selected for its standardized taxonomy and automated analysis capabilities, providing a replicable approach. Still, we recognize limitations in the accuracy of automated tools and that not all relevant CWEs may be detected.

Conclusion Validity. Conclusion validity concerns the robustness of statistical inferences. Since the collected data naturally varies across

projects (e.g., code size, number of dependencies, development practices), there is a risk of spurious correlations or misinterpretations. To mitigate this risk, we adopted statistical analysis using the *SciPy* library, along with careful graphical representation using *Matplotlib*. We also avoided making direct causal inferences, emphasizing the exploratory nature of the study.

8 Conclusions

This study evaluated the impact of four secure design methodologies, *Security by Design, Defense in Depth, Runtime Security, and Zero Trust Architecture*, on vulnerabilities (CWE) and internal software quality across 333 Java repositories, with *Defense in Depth* emerging as the most frequently adopted category. These projects demonstrated similar ages, as shown in Table 2, suggesting comparable maturity levels.

The most frequent vulnerabilities identified were *CWE-209* (Generation of Error Message Containing Sensitive Information), *CWE-532* (Insertion of Sensitive Information into Log File), *CWE-330* (Use of Insufficiently Random Values), and *CWE-798* (Hardcoded Credentials), common across all security categories. This indicates that the adoption of an isolated strategy does not guarantee vulnerability reduction. Moreover, repositories combining multiple secure design methodologies did not necessarily present fewer weaknesses, suggesting that additional defense layers may increase code complexity without proportionally mitigating risks.

Regarding structural software quality, the *Security by Design* category exhibited the highest average CBO (6.36) and the highest median LCOM (30.69), suggesting higher coupling and lower cohesion compared to other methodologies. In contrast, *Defense in Depth*, *Runtime Security*, and *Zero Trust Architecture* maintained more homogeneous CBO, LCOM, and complexity values, indicating potentially lower structural overhead. These findings are aligned with Table 3, in which *Security by Design* differs significantly from the other categories in all metrics ($p < 0.05$), while *Defense in Depth* and *Runtime Security* showed significant differences only in LCOM and complexity, and *Zero Trust Architecture* showed no statistically relevant variation.

As further work, we recommend expanding the sample to include projects in other programming languages in order to verify whether the identified patterns hold across different development ecosystems. Qualitative investigations, such as interviews with developers and commit history analysis, could shed light on the motivations behind the introduction of recurring vulnerabilities. Another promising direction would be to integrate performance metrics, such as latency and throughput, with security and structural quality indicators, enabling a more concrete evaluation of the trade-offs between efficiency and robustness.

ARTIFACT AVAILABILITY

The *scripts* and *dataset* are available at: *https://github.com/CleitonSilvaT/secure-design-sbqs-2025*

REFERENCES

[1] H. Alamleh, A. A. S. AlQahtani, and B. Al Smadi. 2023. Secure Mobile Payment Architecture Enabling Multi-factor Authentication. In *2023 Systems and Information Engineering Design Symposium (SIEDS)*. 19–24. doi:10.1109/SIEDS58326.2023.10137778

[2] Len Bass, Paul Clements, and Rick Kazman. 2021. *Software Architecture in Practice* (4 ed.). Addison-Wesley Professional.

[3] X. Chengjie, W. Guojun, W. Honghua, J. Yinjie, and M. Dai. 2015. Design of Cloud Safety Monitoring Management Platform of Saline Alkali Industry. In *2015 International Conference on Intelligent Transportation, Big Data and Smart City*. 294–297. doi:10.1109/ICITBS.2015.79

[4] Mahdi Fahmideh, John Grundy, Aakash Ahmad, Jun Shen, Jun Yan, Davoud Mougouei, Peng Wang, Aditya Ghose, Anuradha Gunawardana, Uwe Aickelin, and Babak Abedin. 2023. Engineering Blockchain-based Software Systems: Foundations, Survey, and Future Directions. *ACM Comput. Surv.* 55, 6 (2023), Article 110. doi:10.1145/3530813

[5] Marco Antônio Filó, Mariza Bigonha, and Wellington Ferreira. 2024. Evaluating Thresholds for Object-Oriented Software Metrics. *Journal of the Brazilian Computer Society* 30, 1 (2024), 1–25.

[6] Jenny T. Liang et al. 2023. A Qualitative Study on the Implementation Design Decisions of Developers. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 435–447.

[7] Abdechakour Mechri, Mohamed Amine Ferrag, and Merouane Debbah. 2025. SecureQwen: Leveraging LLMs for vulnerability detection in python codebases. *Computers & Security* 148 (2025), 104151. doi:10.1016/j.cose.2024.104151

[8] Tareq Abed Mohammed and Ahmed Burhan Mohammed. 2020. Security Architectures for Sensitive Data in Cloud Computing. In *Proceedings of the 6th International Conference on Engineering & MIS 2020 (ICEMIS'20)*. doi:10.1145/3410352.3410828

[9] Z. Peng, T. Liu, and L. Mai. 2020. Design and Implementation of Dormitory Management System based on SSM framework. In *2020 International Conference on Information Science, Parallel and Distributed Systems (ISPDS)*. 321–325. doi:10.1109/ISPDS51347.2020.00074

[10] Francisco Ponce, Jacopo Soldani, Carla Taramasco, Hernan Astudillo, and Antonio Brogi. 2024. Triaging Microservice Security Smells, with TriSS. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE '24)*. Association for Computing Machinery, New York, NY, USA, 698–706. doi:10.1145/3661167.3661282

[11] Nathan Semertzidis, Fabio Zambetta, and Florian Mueller. 2023. Brain-Computer Integration: A Framework for the Design of Brain-Computer Interfaces from an Integrations Perspective. *ACM Trans. Comput.-Hum. Interact.* 30, 6 (2023), Article 86. doi:10.1145/3603621

[12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Vol. 236. Springer, Berlin.

[13] H. Zhang, S. Li, Z. Jia, C. Zhong, and C. Zhang. 2019. Microservice Architecture in Reality: An Industrial Inquiry. In *2019 IEEE International Conference on Software Architecture (ICSA)*. 51–60. doi:10.1109/ICSA.2019.00014