Predicting Security Weaknesses in Microservice Architectures using Structural Metrics

No Author Given

No Institute Given

Abstract. Correcting a system after it has been deployed is always more expensive than correcting it earlier in its lifecycle. When it comes to security weaknesses, this cost becomes even greater because the weakness can be exploited by an attacker to cause a major damage and fixes are disruptive. This work addresses the problem of security weaknesses in microservices architectures. We propose an approach for building a predictor of security weaknesses in this type of architecture. To achieve this, we used a comprehensive curated dataset of microservice architectures with annotated security weaknesses, and developed a machine learning model for early prediction of architectural security issues using design-level metrics. The work is grounded in the hypothesis that architectural structure patterns can serve as reliable predictors of security compliance in microservice systems, enabling early detection of security weaknesses before implementation. Our predictive model leverages architectural metrics to assess compliance with key security concerns defined by well known rules for secure microservice design. The analysis reveals specific architectural patterns as consistent indicators of security risks, providing interpretable insights that can guide architects in making informed design decisions.

Keywords: Microservice Architecture · Architectural Metrics · Security Weaknesses · Machine Learning

1 Introduction

In the realm of software development, addressing system flaws post-deployment invariably incurs higher costs compared to rectifying issues during earlier stages of the lifecycle. This financial burden is significantly amplified when dealing with security vulnerabilities, as these can be exploited by malicious actors to inflict substantial damage, and the necessary remedial measures often disrupt ongoing operations [1]. This research work focuses on the critical issue of security weaknesses within microservices architectures, a domain that has garnered considerable attention due to its inherent complexity and the high stakes involved in securing distributed systems.

Microservices architectures, characterized by their decentralized and independently deployable services, offer numerous advantages such as scalability, agility, and resilience. However, these benefits come at the cost of increased security challenges [12]. Each microservice, operating as an independent entity, presents a potential entry point for attackers, thereby expanding the attack surface and complicating the security landscape [12]. The dynamic nature of microservices, with frequent deployments and updates, necessitates continuous security monitoring and management to ensure that each service is adequately protected.

The importance of integrating security measures at multiple levels—including individual services, data exchanges, and the underlying infrastructure—cannot be overstated. Effective security strategies must encompass real-time threat detection, regular vulnerability scanning, and robust mechanisms for patching identified weaknesses [6]. Moreover, the adoption of practices such as threat modeling [15] during the design phase can help identify potential points of weakness and mitigate them early in the development process [6].

This work proposes an innovative approach to building a predictor for security weaknesses in microservices architectures. By leveraging a comprehensive curated dataset of microservice architectures with annotated security weaknesses, we developed a machine learning model capable of early prediction of architectural security issues using design-level metrics. Our hypothesis is that architectural structure patterns can serve as reliable indicators of security compliance, enabling the early detection of security weaknesses before implementation.

The predictive model we propose utilizes architectural metrics to assess compliance with key security concerns defined by established rules for secure microservice design. Our analysis reveals specific architectural patterns that consistently indicate security risks, providing interpretable insights that can guide architects in making informed design decisions. This approach not only enhances the security posture of microservices architectures but also supports regulatory compliance and fosters trust among users and stakeholders.

The remainder of the paper is organized as follows. Section 2 introduces the general approach. Section 3 describes the dataset preparation process, including rule labeling and architectural feature extraction. Section 4 details the construction of classifiers and the evaluation strategy for predicting security violations. Section 5 reports and analyzes the experimental results and discusses the findings then outlines threats to validity. Section 6 reviews related work. Finally, Section 7 concludes the paper and outlines future research directions.

2 General Approach

Our goal is to support secure-by-design development of microservice systems by predicting architectural security rule violations based on structural design information. To achieve this, we leverage architectural models and formulate the task as a multi-label classification problem, where each system may simultaneously violate multiple security rules.

Figure 1 presents an overview of our methodology. We begin with Data Flow Diagrams (DFDs) from the MicroSecEnD dataset [14], each representing a microservice-based system architecture. The dataset provides rule-related information from which we derived binary compliance labels for 17 architectural security rules [2], indicating the presence or absence of specific design-level violations.

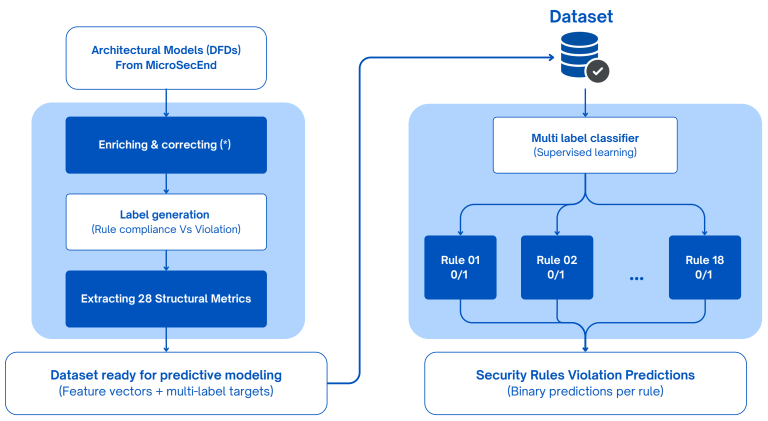
Architectural Models (DFDs)
From MicroSecEnd
Enriching & correcting (*)
Label generation
(Rule compliance Vs Violation)
Extracting 28 Structural Metrics
Dataset ready for predictive modeling
(Feature vectors + multi-label targets)
Dataset
Multi label classifier
(Supervised learning)
Rule 01
0/1
Rule 02
0/1
...
Rule 18
0/1
Security Rules Violation Predictions
(Binary predictions per rule)

Fig. 1. Overview of the proposed approach for predicting architectural security rule violations using structural metrics. (*) indicates that enrichment was performed after contacting the first author of the MicroSecEnd dataset. Binary predictions per rule are generated where 0 denotes compliance and 1 indicates a violation.

From each DFD, we extract a 28-dimensional feature vector composed of architectural metrics that capture structural and design-level properties of the system. These features include system size, service coupling, communication flow, best-practice adherence, and observability coverage. They are designed to reflect architectural characteristics that may influence security-relevant properties.

The resulting feature vectors are used to train supervised machine learning classifiers, detailed later, that learn to associate architectural configurations with rule violations. We adopt a binary relevance strategy for multi-label classification: one independent binary classifier is trained per rule, allowing for tailored tuning, individual performance analysis, and interpretable predictions. Each classifier produces a binary output where 0 denotes compliance and 1 indicates a violation of the corresponding rule.

3 Dataset Preparation for Predictive Modeling

This work investigates whether architectural metrics can be used to predict security weaknesses in microservice systems. Therefore, the dataset needed must contain structural metrics and corresponding security weakness information.

To the best of our knowledge, there are two prominent datasets that address microservice architectures and security:

– `Dataset 1`: Authors in [22] created this dataset by manually modeling 10 open-source microservice systems as component diagrams, each with secure and insecure variants (30 models in total). The diagrams include components, connectors, and microservice-specific roles (e.g., API Gateway), annotated with security mechanisms (e.g., token-based authorization). Security metrics were computed on the extracted models using custom scripts.

- `MicroSecEnD` [14]: Contains 17 Java-Spring microservice applications manually modeled as DFDs. These DFDs were assessed for compliance with a set of architectural security rules, derived from a benchmark proposed in [2]. A secure variant was created for each violation. The dataset provides 226 models (original and secure) in PlantUML, PNG, and JSON formats, with traceability to the source code.

Given the limited size of the first dataset, MicroSecEnD best fits our needs: its DFDs represent services, data stores, external entities, and dataflows, enriched with security annotations and linked to source code to ensure traceability between the model and the implementation.

3.1 Security Rules

To assess architectural security in MicroSecEnD, the authors applied 17 rules derived from the microservices architectural security benchmark [2]. This benchmark originally defined 18 rules based on widely recognized best-practice sources, including OWASP, NIST, and the Cloud Security Alliance. One rule (R15) was excluded due to its inapplicability to the selected systems.

In the dataset, each system is annotated per rule with one of three labels: *follow*, *partially follow*, or *violate*, reflecting the degree of compliance.

Below is a grouped summary of the rule categories as defined by the authors:

`Authentication and Authorization (R1-R6):` These rules emphasize the importance of routing all external requests through a central access point (typically an API gateway), responsible for enforcing authentication and authorization. Responsibilities should be separated from other logic and handled by dedicated services. Internal services must also authenticate and authorize each other to block unauthorized internal calls. External credentials should be transformed into internal representations that avoid exposing sensitive identity information. Additionally, mechanisms must be in place to detect and limit repeated failed login attempts.

`Encryption (R7-R8):` All communication involving sensitive data or credentials must be encrypted using secure protocols such as HTTPS. This applies to both inter-service and service-to-user communication, aiming to prevent tampering and man-in-the-middle attacks.

`Logging (R9-R12):` Promote centralized logging and monitoring to detect abnormal behavior. Logs should be collected by local agents on the same host as each microservice. These agents must sanitize entries (e.g., remove passwords, tokens), then forward them via a message broker, ensuring only authenticated services can exchange logs and preventing spoofing or traffic injection.

`Availability (R13-R15):` Aim to maintain system responsiveness and resilience through API gateway-level load balancing, circuit breakers to halt calls to failing services, and usage limits to avoid service overload. R15 was excluded due to its inapplicability to the selected systems.

`Service Registry (R16-R17):` Require that registry components be isolated and validate that only legitimate services can register, update, or query. This prevents unauthorized components from accessing or manipulating service metadata.

`Secret Management (R18):` Recommends centralized secret handling (e.g., API keys, passwords) using a "Secrets as a Service" model. Credentials should be created on demand, rotated and revoked after a lease period to reduce the risk of exposure.

These rules form the basis for assigning per-system multi-label annotations, which serve as ground truth for training and evaluating our predictive models.

3.2 Preprocess of Labeling

To support the prediction task, we prepared the labels by converting them into a binary format. For each architectural model, this results in a vector of 17 binary values indicating whether the architecture model violates each security rule. We also addressed observed inconsistencies between the labels and models and completed missing variants to maintain alignment across the dataset. This was done after contacting the dataset's first author.

The label *partially follow* appeared only for Rule R1, which requires a single API Gateway that handles both authentication and authorization. When only one condition was met, we treated it as non-compliant (value 1). Since no variants were provided for these cases, we created new ones by adding missing elements at the model level.

After finalizing the base rule table—containing labels for the base DFDs as published in [14]—and correcting the variants, we assigned binary labels to all entries using the following procedure:

- For each variant, we began by copying the labels from its corresponding base system.
- The rule for which the variant was created was labeled as 0 (compliant).
- We also accounted for rule dependencies. For example, Rule R12 requires the use of a message broker to securely transfer logs from a local logging agent (Rule R10) to a central logging subsystem (Rule R9). Therefore, when a variant is created to comply with Rule R12 (i.e., R12 is labeled as 0), we also labeled Rules R10 and R9 as 0, since both are necessary conditions for satisfying the requirements of Rule R12.

In total, this process produced binary labels for 237 architectural models, including base systems and their variants.

3.3 Architectural Metrics as Features

We use architectural features from the DFD models in the form of quantitative structural metrics. These metrics describe how components are organized and interact, serving as input to supervised models that learn patterns linked to rule compliance.

To ensure both practicality and relevance, we followed the criteria established in [10]. Specifically, selected metrics had to be clearly defined, automatically derivable from DFDs, and produce consistent, objective results across systems.

They also needed to capture meaningful aspects of microservice architecture and remain applicable regardless of the programming language used.

Based on these criteria, we selected 28 structural metrics, supported by the architectural metrics framework from [16], which aligns metrics with ISO/IEC 25010:2011 quality attributes. Each metric was mapped to this framework and adapted to microservice context.

The metrics fall into the following categories:

- `Size and Composition Metrics:` Quantify the number and proportion of architectural elements, such as services and infrastructure components (e.g., `num_services`, `num_components`). These offer a high-level view of system scale and structure.
- `Coupling and Degree Metrics:` Measure inter-service connectivity, focusing on direct dependencies and fan-in/fan-out (e.g., `avg_connections_per_service`, `max_service_fan_in`). They reveal interaction intensity and component interdependence.
- `Centrality and Flow Complexity Metrics:` Capture structural patterns like communication paths, cyclic dependencies, and central routing roles (e.g., `longest_shortest_path`, `num_cycles`), reflecting architectural complexity and control flow.
- `Interface and Exposure Metrics:` Reflect external exposure by quantifying entry points and endpoint densities (e.g., `num_entry_points`, `avg_endpoints_per_service`), relevant to attack surface analysis.
- `Design Practice and Pattern Metrics:` Indicate the presence of recommended architectural patterns for security and maintainability, such as API gateways, config servers, and per-service databases (e.g., `api_gateway_presence`, `config_centralization`).
- `Observability and Monitoring Metrics:` Assess instrumentation for monitoring and tracing, supporting runtime visibility and operational security (e.g., `monitoring_coverage`, `tracing_server_presence`).
- `External Dependency Metrics:` Capture interactions with external systems and third-party providers (e.g., `num_external_entities`), reflecting reliance on elements outside the trust boundary.

The complete list of metrics, including formulas, descriptions, and the architectural metric from the framework [16] that most closely correspond to each are available in the supplementary Git repository [1].

To generate input features for predictive modeling, each system's DFD model was transformed into a 28-dimensional numerical vector, where each dimension represents a structural metric. This involved parsing JSON models to classify components by stereotype, identifying and labeling dataflows by communication type, constructing a directed graph with components as nodes and flows as edges, and computing structural metrics on the graph. The resulting vectors, paired with multi-label annotations of security rule violations, constitute the dataset used for training and evaluation.

[1] `https://anonymous.4open.science/status/archmetrics-sec-predictor-msa-176C`

4 Prediction Modeling of Security Rules Violation

The dataset of architectural metrics and rule violation labels described above are used in our approach to automatically identifying security weaknesses from design-time representations of microservice systems. Specifically, each violation of an architectural security rule is treated as a potential security weakness, and our objective is to predict which rules are violated in a given architecture.

4.1 Problem Formulation and Modeling Strategy

We formulate the task as a multi-label classification problem: a given system may simultaneously violate multiple security rules. To support model transparency and diagnosis, we adopt a Binary Relevance (BR) transformation [13], which decomposes the task into 17 independent binary classification problems. This approach enables per-rule interpretability, allows tailored handling of class imbalance, and remains computationally scalable. Each binary classifier takes as input a 28-dimensional feature vector derived from the system's DFD model. The output is a binary label indicating whether the specific rule is violated.

4.2 Model Selection and Class Imbalance Handling

We selected three widely used classifiers for our rule-specific prediction tasks: Decision Trees (DT), Support Vector Machines (SVM), and Random Forests (RF). These models were chosen for their complementary strengths, namely interpretability, robustness to nonlinear boundaries, and ensemble-based performance, making them suitable for analyzing architectural rule violations under varying data conditions [7].

The dataset exhibits significant class imbalance across different architectural rules, as illustrated in Figure 2. The distribution reveals that most rules (R1-R12, R17, R18) are heavily skewed toward rule violations (Class 1). Conversely, rules R13, R14, and R16 demonstrate the opposite pattern, with compliance instances (Class 0) dominating the dataset.

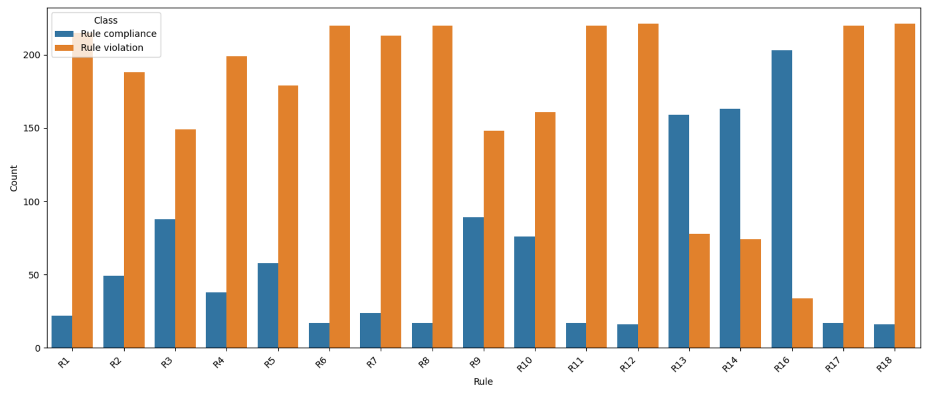

Fig. 2. Per-rule class distribution showing the number of violating and compliant samples for each security rule.

This pronounced imbalance poses challenges for traditional machine learning approaches, which tend to favor majority classes. To address this issue, we avoid data-level resampling techniques, which can distort the underlying distributions and reduce model interpretability [19]. Instead, we employ cost-sensitive learning strategies [3]. For Decision Trees (DT), class weights are integrated into the impurity measure to penalize errors on minority instances. Support Vector Machines (SVM) use class-weighted objectives to scale penalties for misclassifying underrepresented classes. Random Forests (RF) aggregate cost-sensitive trees using ensemble voting that emphasizes correct minority class predictions.

4.3 Evaluation Metrics

To assess classifier performance under class imbalance, we adopt metrics that reflect both majority and minority class behavior [21, 3]. As recommended in [21], we considered:

- `True Positive Rate (TPR)` and `True Negative Rate (TNR)`: Also known as sensitivity and specificity metrics:

$$\text{TPR} = \frac{TP}{TP + FN}, \quad \text{TNR} = \frac{TN}{TN + FP}$$

- `Geometric Mean (G-Mean)`: Captures balanced performance across classes and penalizes poor performance:

$$\text{G-Mean} = \sqrt{\text{TPR} \cdot \text{TNR}}$$

Final model selection prioritizes G-Mean, as it emphasizes balanced sensitivity and specificity and demonstrates robustness to skewed class distributions effectively [21].

5 Experimentation and Results

We evaluate our prediction approach through a structured experimentation process designed to assess per-rule classifier performance, compare algorithms, and address challenges such as class imbalance and model instability.

The dataset is split into 80% training and 20% testing. All training results are averaged using 5-fold stratified cross-validation, with classifiers tuned to maximize the G-Mean metric. To evaluate model stability, we compute the standard deviation of G-Mean across folds (CV Std). Hyperparameters are selected via grid search using a custom G-Mean-based scoring function, ensuring fair and balanced model comparisons.

The experimentation follows five phases:

1. `Baseline Evaluation:` Cost-sensitive Decision Trees trained for all 17 rules.
2. `Algorithm Comparison:` Random Forest and SVM models evaluated for comparative analysis.
3. `Rule Categorization:` Rules grouped based on model performance and stability.
4. `Refinement:` Additional tuning for unstable or underperforming classifiers.

5. `Model Selection:` Final models chosen per rule based on cross-validation G-Mean.

Detailed hyperparameter grids are available in the project's Git repository mentioned before.

5.1 Algorithm Performance

We evaluated three classifiers—cost-sensitive Decision Trees, Random Forests, and Support Vector Machines (SVMs)—on 17 architectural rules. The results, summarized in Figure 3, report the mean G-Mean scores along with standard deviation across cross-validation folds. Performance varies significantly across rules and algorithms. SVMs often yielded the highest and most stable G-Mean scores (e.g., reaching 0.994 on R16), while Random Forests and Decision Trees performed comparably well on many rules. However, all classifiers struggled on some rules such as R8, R11, and R17, where G-Mean scores dropped below 0.3 and stability was low.

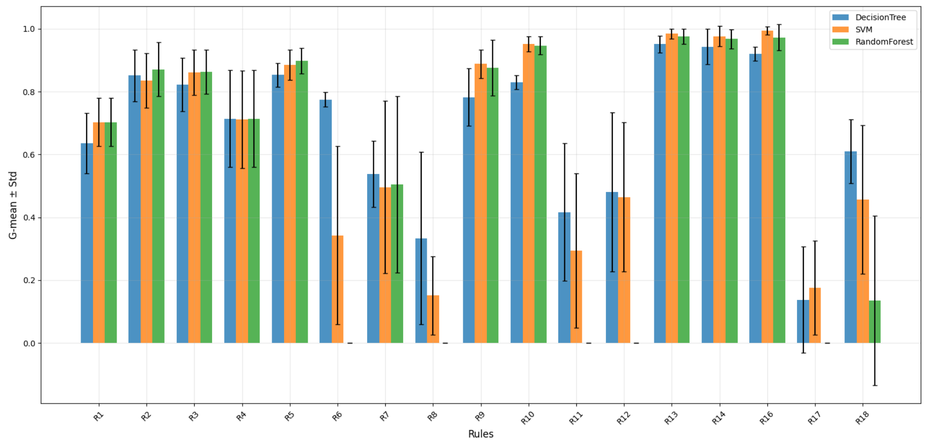

Fig. 3. Cross-validation performance comparison of three machine learning algorithms across the whole rule sets.

Based on classifier performance and stability, we grouped the rules into three categories. `Category A (Excellent)` includes six rules (R5, R9, R10, R13, R14, R16) with G-Mean above 0.8 and consistent performance across classifiers. `Category B (Good)` covers four rules (R1–R4) with G-Mean exceeding 0.7 but exhibiting higher variance. `Category C (Poor)` comprises seven rules (R6–R8, R11, R12, R17, R18) with low performance and instability across classifiers.

5.2 Refinement Experiments

To improve Category B classifiers, which showed good G-Mean but poor stability, we applied a refinement pipeline combining feature selection and Random Forest tuning. We hypothesized that irrelevant or redundant dimensions in the

28-feature space could destabilize the models. Using `SelectKBest` (ANOVA F-test) and grid search over model depth, split size, and feature count, we achieved mixed results. R1 and R2 did not yield significant improvements. However, R3 and R4 saw substantial gains, improving from 0.862±0.070 to 0.8956±0.0514 and 0.714±0.154 to 0.7739± 0.0461 respectively, suggesting limited generalizability of tuning strategies across rule contexts.

For Category C, persistent poor performance across classifiers prompted a deeper data quality analysis. Specifically, we analyzed class imbalance and class overlap as potential causes of model failure. The level of imbalance is measured using the imbalance ratio (IR), defined in [21] as:

$$\mathrm{IR} = \frac{\text{Number of majority class instances}}{\text{Number of minority class instances}}$$

Class overlap was assessed using the Kolmogorov–Smirnov (KS) test, comparing class-conditional distributions. For each feature i:

$$D_i = \sup_x |F_1^{(i)}(x) - F_2^{(i)}(x)|$$

Where $F_1^{(i)}(x)$ and $F_2^{(i)}(x)$ are the empirical cumulative distribution functions for the positive and negative classes, respectively. We define the overlap score as:

$$\text{Overlap}_i = 1 - D_i, \quad \text{Overlap Score} = \frac{1}{n}\sum_{i=1}^{n}(1 - D_i)$$

This provides a model-agnostic, interpretable measure of class similarity in feature space [8].

All results for IR and overlap scores are available in the supplementary Git repository. They reveal significant data quality issues across all Category C rules, with imbalance ratios ranging from 8.88 to 13.81 and overlap scores consistently exceeding 0.75. These values indicate substantial class overlap in the feature space, confirming that the limitations observed in Category C are primarily due to structural data quality problems. Consequently, our final multi-label model includes only classifiers from Categories A and B.

5.3 Model Selection and Interpretations

The final classifier for each rule was selected from Categories A and B based on cross-validated G-Mean and model stability. SVMs were chosen for the rules R9, R10, R13, R14 and R16, while Random Forests were selected for rules R1 through R5.

In terms of generalization, the average performance on the held-out test set across these 10 selected rules demonstrates strong results. The `True Positive Rate (TPR)` averaged `97.7%`, indicating that the models are highly effective at detecting security rule violations, which is a critical objective in proactive security assessment. `G-Mean` reached `89.7%`, confirming balanced performance across both classes despite class imbalance. While the `True Negative Rate (TNR)` was slightly lower at `83.2%`, it remains acceptable given the priority of capturing true violations in security-critical systems.

To ensure interpretability, we conducted feature importance analysis using Random Forest models, which offer built-in mechanisms for ranking feature relevance. Although RFs were not the final classifier for all rules, they achieved competitive performance and thus served as a common basis for interpretation. The most influential features reflect architectural properties such as structural complexity, exposure surface, and modularity. High-importance features like `cycle_ratio_and_count` and `longest_shortest_path` capture tightly coupled components and deep communication chains. These patterns are often associated with weak boundary enforcement and increased attack surfaces. The feature `num_external_entities` also ranked highly, suggesting that the number of system entry points is a strong structural cue for differentiating secure architectures. Features such as `api_gateway_dependency` and `system_interaction_density` help distinguish between centralized and distributed control structures. Finally, a group of moderately weighted features, including `ratio_max_fan_in`, `database_per_service_ratio`, and `ratio_infrastructural_services`, highlights the importance of balanced service responsibilities and resource isolation. Overall, the models leverage these structural signals without manual thresholding, adapting their interpretation to the context of each rule.

Nonetheless, the exclusion of Category C rules from the final multi-label classifier underscores the limitations of the current structural feature set in capturing certain security concerns. Authentication and encryption policies (R6, R7, R8) require detecting rate-limiting mechanisms and protocol-level configurations such as TLS/SSL usage. Data handling and sanitization rules (R11, R12) involve internal processing and specific security configurations that are not structurally encoded. Similarly, service validation and secret management rules (R17, R18) rely on validation logic and configuration details that exceed the granularity of structural representations. While our features effectively capture topological and complexity-related traits, they fall short in reflecting protocol-level configurations. Moreover, high overlap scores suggest that structural similarity does not necessarily imply similarity in security implementation. This highlights the need for multi-layered feature engineering to fully represent a system's security posture, as some aspects remain underrepresented in the current approach.

5.4 Discussion

This study explored whether architectural security rule violations in microservice systems can be predicted from structural design metrics. The results provide strong empirical support for the hypothesis that architectural structure reflects security-relevant signals. This reinforces the view that security, like maintainability or performance, can be assessed early in the software lifecycle through architectural analysis.

Our findings show that structural features—such as service connectivity, communication patterns, and infrastructural component use—correlate with specific types of violations, supporting the notion of security as a measurable architectural quality. This challenges the traditional treatment of security as a late-stage concern, highlighting its roots in system design.

However, variation in performance across rules reveals that structural models are more predictive for some security aspects rather than others. Rules involving architectural anti-patterns or exposure boundaries were effectively predicted, while those related to encryption or authentication were less so. This indicates the need to integrate richer features in future work.

A key strength of our approach is that it enables early, cost-effective detection of design-level security issues without requiring system implementation. It is thus suitable for integration into continuous architectural evaluation and DevSecOps workflows. Moreover, using interpretable classifiers allows for actionable insights, as feature importance reveals the structural anti-patterns driving each violation prediction.

5.5 Threats to validity

While the results are promising, several threats to validity must be acknowledged. Construct validity refers to how accurately the study captures the intended concepts [18]. A primary threat in this context is the gap between abstract security concepts and what structural metrics can capture.

– *Limited metric expressiveness:* The structural metrics used in this study cannot fully express security concerns related to encryption configurations, authentication policies, or runtime data flows. To mitigate this limitation, only architectural rules whose violations can be meaningfully inferred from structural design were included. Future work should integrate additional feature types to extend coverage and improve semantic fidelity.

Internal validity relates to whether observed results are attributable to the proposed approach rather than external biases. This experiment faces several threats:

– *Reliability of Architectural Models:* The MicroSecEnD dataset provides Data Flow Diagrams (DFDs) manually constructed by the original authors through semi-structured analysis of open-source microservice systems. While this ensures abstraction consistency and domain relevance, it also introduces human subjectivity in identifying components and data flows. To increase reliability, the original authors applied a validation process involving five researchers, providing reasonable assurance of the correctness of the models. In our study, we did not perform the architecture extraction ourselves, relying instead on this independently produced dataset created by researchers who were unaware of our study's objectives. This separation reduces the potential bias in our experiments. Future work could explore automated or tool-assisted extraction to further enhance consistency and reproducibility.
– *Labeling reliability:* Security violation labels were originally assigned through manual inspection based on 17 architectural security rules. Although our team conducted a thorough review and correction process, some residual noise may persist due to the inherently interpretive nature of design-level security rules. To reduce the influence of subjective bias, labeling in our

study was performed by multiple authors and cross-validated through independent reviews. Support tools or semi-automated labeling frameworks may help reduce this risk in future studies.

External validity concerns whether the findings generalize beyond the studied context:

- *Dataset diversity and representativeness:* The dataset includes 237 architectural models from Java-based microservices using the Spring framework. While it is one of the largest datasets of its kind, it may not capture the full diversity of architectural practices, technologies, or security concerns found in real-world systems. The findings may therefore be biased toward Spring-style architectures. To reduce this risk, the architectural metrics used in this study were carefully defined at an abstract level, independently of any specific technology stack, in order to reflect structural properties that generalize across platforms and development environments.
- *Scope of security rules:* This study focuses on 17 architectural security rules selected from a broader set of 18. While not covering all possible security threats – such as deployment-specific issues – the selected rules target design-time architectural weaknesses that are amenable to automation. They were derived through a systematic analysis of relevant standards and literature, covering diverse areas such as authentication, encryption, availability, logging, and secret management. Although deployment concerns represent a complementary viewpoint, the chosen rule set offers broad and representative coverage of architectural-level security issues, supporting the validity of our findings within this scope.

6 Related Work

Architecture-level security assessment in microservice systems has gained attention through the use of metrics and formal models. Zdun et al. [22] assess conformance to secure design decisions using microservice-specific metrics, though their approach targets post-implementation systems. Design-time methods based on enriched Data Flow Diagrams (DFDs) and formal rules have been proposed to detect security flaws in software architectures more generally [17, 20]. Chondamrongkul et al. [5] specifically apply ontology reasoning to identify security vulnerabilities in microservice architectures through component-and-connector models. While these approaches rely on manually defined rules and formal verification, our method complements them by learning predictive patterns from labeled architectural data, allowing broader generalization to unseen designs and reducing the manual effort required for rule specification.

Beyond security-specific efforts, studies such as [9, 11] show that the analysis of recovered architectures can reveal issues like unnecessary service interactions or dependency cycles. Brogi et al. [4] propose a systematic method for detecting and resolving such architectural smells that may violate microservice design principles. Although not explicitly security-focused, these works highlight the relevance of structural features and support automated, early-stage analysis for quality assurance, including security.

7 Conclusion

This paper proposes an approach to predict architectural security rule violations in microservice systems from early design. We proposed a set of 28 architectural metrics computed from Data Flow Diagrams, specifically designed to capture structural characteristics relevant to secure design. These metrics were used to train predictive models that assess compliance with predefined security rules before implementation. The approach was evaluated using a curated dataset of annotated microservice architectures. The results show that several categories of rule violations, particularly those related to exposure, service connectivity, and logging infrastructure, can be effectively predicted from structural features. These findings support the use of architecture-level metrics for early security validation and contribute to secure-by-design development practices.

As future work, we plan to expand the metric set by incorporating semantic and configuration-level aspects, such as authentication mechanisms, communication protocols, and deployment properties, in order to better capture security rules that are not adequately reflected in structural representations. We also aim to assess the scalability and generalizability of the approach on larger and more heterogeneous microservice systems. Ultimately, our goal is to integrate this predictive capability into architecture modeling environments to enable continuous and automated security feedback throughout the design process.

References

1. Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems, third edition*. John Wiley & Sons Inc, January 2021.
2. Anusha Bambhore Tukaram, Simon Schneider, Nicolás E. Díaz Ferreyra, Georg Simhandl, Uwe Zdun, and Riccardo Scandariato. Towards a security benchmark for the architectural design of microservice applications. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ARES '22, New York, NY, USA, 2022. Association for Computing Machinery.
3. Paula Branco, Luís Torgo, and Rita P. Ribeiro. A survey of predictive modeling on imbalanced domains. *ACM Comput. Surv.*, 49(2), August 2016.
4. Antonio Brogi, Davide Neri, and Jacopo Soldani. Freshening the air in microservices: Resolving architectural smells via refactoring. In Sami Yangui, Athman Bouguettaya, Xiao Xue, Noura Faci, Walid Gaaloul, Qi Yu, Zhangbing Zhou, Nathalie Hernandez, and Elisa Yumi Nakagawa, editors, *Service-Oriented Computing - ICSOC 2019 Workshops - WESOACS, ASOCA, ISYCC, TBCE, and STRAPS, Toulouse, France, October 28-31, 2019, Revised Selected Papers*, volume 12019 of *Lecture Notes in Computer Science*, pages 17–29. Springer, 2019.
5. N. Chondamrongkul, J. Sun, and I. Warren. Automated security analysis for microservice architecture. In *2020 IEEE ICSA Companion 2020, Salvador, Brazil, March 16-20, 2020*, pages 79–82. IEEE, 2020.
6. Daniel Deogun, Dan Bergh Johnsson, and Daniel Sawano. *Secure By Design*. Manning, September 2019.
7. Pedro M. Domingos. A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87, 2012.
8. J. Gauss, F. Scheipl, and M. Herrmann. Dcsi - an improved measure of cluster separability based on separation and connectedness. *ArXiv*, abs/2310.12806, 2023.
9. Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. Microart: A software architecture recovery tool for

maintaining microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017*, pages 298–302. IEEE Computer Society, 2017.
10. Vira Liubchenko. Evaluating software architecture: A systematic mapping study on design metrics and their applications. In Igor Sinitsyn and Philip Andon, editors, *Proceedings of the 14th International Scientific and Practical Programming Conference (UkrPROG 2024), Kyiv, Ukraine, May 14-15, 2024*, volume 3806 of *CEUR Workshop Proceedings*, pages 100–111. CEUR-WS.org, 2024.
11. Shang-Pin Ma, Chen-Yuan Fan, Yen Chuang, Wen-Tin Lee, Shin-Jie Lee, and Nien-Lin Hsueh. Using service dependency graph to analyze and test microservices. In Sorel Reisman, Sheikh Iqbal Ahamed, Claudio Demartini, Thomas M. Conte, Ling Liu, William R. Claycomb, Motonori Nakamura, Edmundo Tovar, Stelvio Cimato, Chung-Horng Lung, Hiroki Takakura, Ji-Jiang Yang, Toyokazu Akiyama, Zhiyong Zhang, and Md. Kamrul Hasan, editors, *2018 IEEE 42nd Annual Computer Software and Applications Conference, COMPSAC 2018, Tokyo, Japan, 23-27 July 2018, Volume 2*, pages 81–86. IEEE Computer Society, 2018.
12. Sam Newman. *Building Microservices: Designing Fine-Grained Systems.* O'Reilly Media, Inc., 1st edition, 2015.
13. Pooja Pant, A Sai Sabitha, Tanupriya Choudhury, and Prince Dhingra. Multi-label classification trending challenges and approaches. *Emerging Trends in Expert Applications and Security: Proceedings of ICETEAS 2018*, pages 433–444, 2019.
14. Simon Schneider, Tufan Özen, Michael Chen, and Riccardo Scandariato. microsecend: A dataset of security-enriched dataflow diagrams for microservice applications. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 125–129, 2023.
15. Adam Shostack. *Threat Modeling: Designing for Security.* Wiley, February 2014.
16. Samira Silva, Adiel Tuyishime, Tiziano Santilli, Patrizio Pelliccione, and Ludovico Iovino. Quality metrics in software architecture. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, pages 58–69, 2023.
17. L. Sion, K. Tuma, R. Scandariato, K. Yskout, and W. Joosen. Towards automated security design flaw detection. In *34th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASE Workshops 2019, San Diego, CA, USA, November 11-15, 2019*, pages 49–56. IEEE, 2019.
18. Dag IK Sjøberg and Gunnar R Bergersen. Improving the reporting of threats to construct validity. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 205–209, 2023.
19. C. Tantithamthavorn, A. E. Hassan, and Kenichi M. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*, 46(11):1200–1219, 2020.
20. K. Tuma, L. Sion, R. Scandariato, and K. Yskout. Automating the early detection of security design flaws. In Eugene Syriani, Houari A. Sahraoui, Juan de Lara, and Silvia Abrahão, editors, *MoDELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020*, pages 332–342. ACM, 2020.
21. Pattaramon Vuttipittayamongkol, Eyad Elyan, and Andrei Petrovski. On the class overlap problem in imbalanced data classification. *Knowledge-Based Systems*, 212:106631, 2021.
22. Uwe Zdun, Pierre-Jean Queval, Georg Simhandl, Riccardo Scandariato, Somik Chakravarty, Marjan Jelic, and Aleksandar S. Jovanovic. Microservice security metrics for secure communication, identity management, and observability. *ACM Trans. Softw. Eng. Methodol.*, 32(1):16:1–16:34, 2023.