Microservice Security Metrics for Secure Communication, Identity Management, and Observability

UWE ZDUN*, PIERRE-JEAN QUEVAL, and GEORG SIMHANDL, University of Vienna, Faculty of Computer Science, Research Group Software Architecture, Austria
RICCARDO SCANDARIATO, Hamburg University of Technology (TUHH), Germany
SOMIK CHAKRAVARTY, MARJAN JELIC, and ALEKSANDAR JOVANOVIC, European Risk and Resilience Institute (EU-VRi), Germany

Microservice architectures are increasingly being used to develop application systems. Despite many guidelines and best practices being published, architecting microservice systems for security is challenging. Reasons are the size and complexity of microservice systems, their polyglot nature, and the demand for the continuous evolution of these systems. In this context, to manually validate that security architecture tactics are employed as intended throughout the system is a time-consuming and error-prone task. In this article, we present an approach to avoid such manual validation before each continuous evolution step in a microservice system, which we demonstrate using three widely used categories of security tactics: secure communication, identity management, and observability. Our approach is based on a review of existing security guidelines, the gray literature, and the scientific literature, from which we derived Architectural Design Decisions (ADDs) with the found security tactics as decision options. In our approach, we propose novel detectors to detect these decision options automatically and formally defined metrics to measure the conformance of a system to the different options of the ADDs. We apply the approach to a case study data set of 10 open source microservice systems, plus another 20 variants of these systems, for which we manually inspected the source code for security tactics. We demonstrate and assess the validity and appropriateness of our metrics by performing an assessment of their conformance to the ADDs in our systems' dataset through statistical methods.

CCS Concepts: • **Software and its engineering → Software architectures**; **Distributed systems organizing principles**; • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Microservice Architecture, Microservice Security, Software Architecture Metrics, Software Architecture Detectors

ACM Reference Format:
Uwe Zdun, Pierre-Jean Queval, Georg Simhandl, Riccardo Scandariato, Somik Chakravarty, Marjan Jelic, and Aleksandar Jovanovic. 2022. Microservice Security Metrics for Secure Communication, Identity Management, and Observability. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2022), 34 pages. https://doi.org/10.1145/3532183

[*] Corresponding Author

Authors' addresses: Uwe Zdun, uwe.zdun@univie.ac.at; Pierre-Jean Queval, pierre-jean.queval@univie.ac.at; Georg Simhandl, georg.simhandl@univie.ac.at, University of Vienna, Faculty of Computer Science, Research Group Software Architecture, Währingerstraße 29, 1090, Vienna, Austria; Riccardo Scandariato, riccardo.scandariato@tuhh.de, Hamburg University of Technology (TUHH), Blohmstraße 15, 21079, Hamburg, Germany; Somik Chakravarty, schakravarty@risk-technologies.com; Marjan Jelic, mjelic@risk-technologies.com; Aleksandar Jovanovic, jovanovic@risk-technologies.com, European Risk and Resilience Institute (EU-VRi), Filderhauptstr. 142, 70599, Stuttgart, Germany.

1 INTRODUCTION

Microservice architectures [29, 33, 48] structure an application as a collection of autonomous services. They have a set of important tenets such as development in independent teams, polyglot technology stacks including polyglot persistence and programming languages, cloud-native technologies and architectures, use of lightweight containers, loosely coupled service dependencies, high releasability, end-to-end tracing and monitoring, and continuous delivery [29, 33, 62]. This article focuses on the security aspects of microservice architectures.

Despite numerous published guidelines and best practices [8, 34, 37], the architecture of microservice systems is challenging with regard to security. This is due to the size and complexity of microservice systems, their polyglot nature, and the need for continuous evolution and frequent release of these systems. In this context, manually validating whether security features are used as intended throughout the system is a time-consuming and error-prone task. For architecturally relevant security features, architectural abstraction can help focus only on the relevant aspects, but still substantial effort is required e.g. to check a large-scale system's architecture for conformance to security tactics.

In this article, we present an approach to avoid such manual validation before each continuous evolution step in a microservice system, which we demonstrate using three widely used categories of security tactics: secure communication, identity management, and observability. Our approach is based on a review of existing security guidelines, gray literature, and academic literature, from which we derived Architectural Design Decisions (ADDs) with security tactics as decision options. We focus on establishing a conformance relation between a microservice system model, derived from the system's source code, and our ADD model on microservice security tactics. In general, the conformance relation is defined as the consistency between models [45]. To guarantee the correctness of this consistency relation, an assessment is needed. Conformance assessment is challenging because it concerns the relation between a software system's architecture and its intended architecture [9]. This article aims to study the following research questions:

- **RQ1** How can we automatically assess conformance to ADDs on security tactics for secure communication, identity management, and observability in the context of microservice systems?
- **RQ2** How well do measures for assessing such security tactics as ADD options perform?
- **RQ3** What is a set of minimal elements needed in a microservice architecture model to compute such measures?

Our approach to addressing these challenges is to define a set of metrics for each ADD associated with the decision's options, i.e. at least one metric per major decision option. Based on a manual assessment of a small set of models and model variants that is representative of the possible decision options and option combinations of the studied decisions, we derive a ground truth. The ground truth is established from a thorough analysis of microservice security guidelines and the literature, as well as multiple rounds of review by five industrial security experts. By combining the outcome of all options of a decision, we can then derive an ordinal assessment of how well the decision is supported in each model. We then use the ground truth data to assess how well the hypothesized metrics predict the ground truth data by first performing a correlation analysis and then an ordinal regression analysis.

In this article, we propose an architectural component model based approach which uses only modeling elements that can be derived from the system's source code. For this reason, it is important to be able to work with a minimal set of modeling elements, else it might be difficult to continuously parse them from the source code. We do not focus on the extraction of models from the source code in this article, however, but use our existing static code analysis based architecture reconstruction approach [36] (which is only summarized in this article).

The core contributions of this article are: (1) a novel approach for the automated, model-based, and metrics-based assessment of conformance to ADDs on security tactics in microservice-based systems; (2) a detailed study of microservice security tactics and related ADDs in three areas (communication, identity management, and observability); (3) a multi-case study-based analysis and evaluation of our approach for these ADDs.

This article is structured as follows: First, in Section 2 we discuss the research methods used in this article in depth, thereby also providing an overview of the research study. In Section 3 we discuss the inspection and preparation of the 10 open source systems and 20 variant models we used as cases in a multi-case study. Then, Section 4 presents the ADDs for microservice security tactics which are considered in this article. Section 5 summarizes the ground truth assessment, whereas Section 6 formally specifies the metrics we propose. Next, Section 7 presents the statistical analysis of the obtained data. Integration in industrial tools is discussed in Section 8. Section 9 discusses our findings and potential threats to validity. After that, we compare to related work in Section 10. Finally, in Section 11 we conclude.

2 RESEARCH METHODS

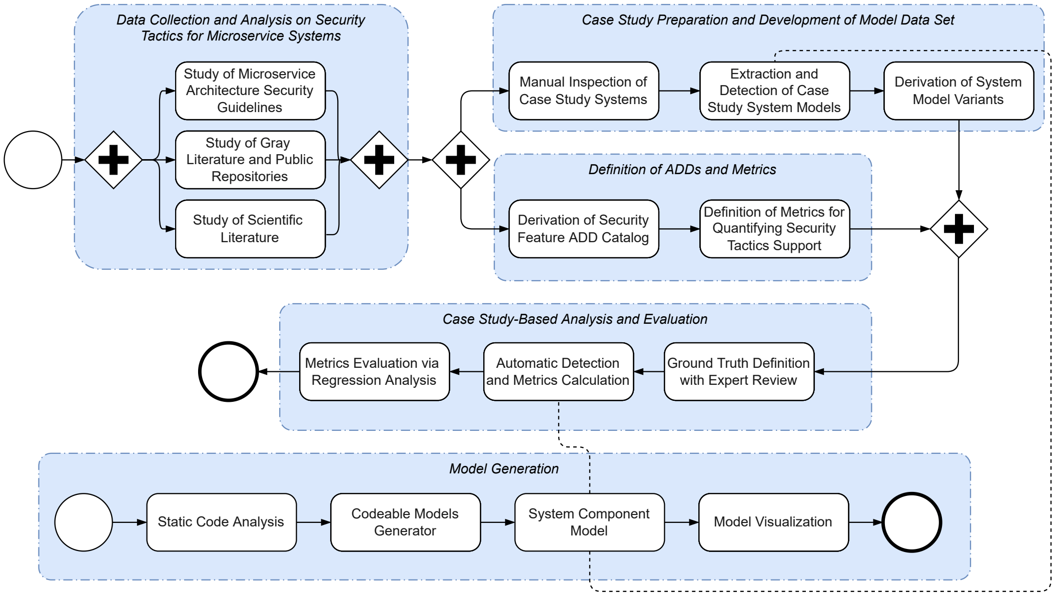
Data Collection and Analysis on Security Tactics for Microservice Systems
Study of Microservice Architecture Security Guidelines
Study of Gray Literature and Public Repositories
Study of Scientific Literature
Case Study Preparation and Development of Model Data Set
Manual Inspection of Case Study Systems
Extraction and Detection of Case Study System Models
Derivation of System Model Variants
Definition of ADDs and Metrics
Derivation of Security Feature ADD Catalog
Definition of Metrics for Quantifying Security Tactics Support
Case Study-Based Analysis and Evaluation
Metrics Evaluation via Regression Analysis
Automatic Detection and Metrics Calculation
Ground Truth Definition with Expert Review
Model Generation
Static Code Analysis
Codeable Models Generator
System Component Model
Model Visualization

Fig. 1. Overview diagram of the research methods followed in this study

2.1 Data Collection and Analysis of Security Tactics for Microservices

Figure 1 shows the research steps of our study. For the initial data collection and analysis of microservice security tactics, we first consulted existing microservice-specific recommendations by industry organizations such as those of NIST [34], OWASP [37], or the Cloud Security Alliance [8], which represent aggregations of existing industry best practices on a broad level. These were independently analyzed by a team of industrial security experts, including the last three authors of this study before they got involved in this article. They also related the standards to their experiences from their industry projects. We confirmed and augmented these recommendations with the results of a multi-vocal literature study [15] which the author team conducted within the context of the AssureMOSS EU project[1]. Among other things, in this prior work, we reviewed the scientific literature summarized in Section 10 for microservice-related security practices and tactics. In addition, we are currently conducting a gray literature study [16] (i.e. guidelines, public repositories, recorded interviews, podcasts, videos, practitioner articles, presentations, and blog posts) on microservice-related security practices and tactics. In it, we so far studied 30 practitioner sources in-depth. In this article, we will not report more details of these other literature works, as this article is mainly based on the practices recommended in the existing microservice-specific recommendations (with a focus on the NIST SP 800-204 special publication [34]). We mention this additional work here to explain that the practices, tactics, and corresponding ADDs, studied in this article, are derived based on substantial empirical data.

From the initially collected data, we aimed to address the security tactics with our approach that (1) are concerning the core software architecture in the sense that they can be modeled in software architecture decomposition view (i.e., in the component and connector view); and (2) are discernible from the source code of existing microservice systems. As stated in the research questions, our goal is only to detect conformance to ADD options, not interdependences between them. This limitation of scope was necessary as the guidelines and literature contain many practices that are only discernible at runtime (such as runtime warnings after a number of failed login attempts at a location), are not visible in the source code (such as organization measures or human review activities), or can only be modeled in other views such as deployment views. All those other measures are considered as out-of-scope of our article and will be covered in our future work.

2.2 Case Study Preparation and Development of Model Data Set

We studied 10 open source microservices systems as case studies line-by-line and manually annotated each security feature in their source code. Each is a system published by practitioners with microservice background (all are published on GitHub, see Tables 1 and 2). Next, we used our existing static code analysis approach for architecture reconstruction of polyglot microservice systems [36] (summarized in Section 2.6) as a foundation for the extraction of models from the source code of the 10 studied open source system cases, and the detection of security tactics in these models. For this approach and for the modeling of security tactics reported in this article, we began by performing an iterative study of a variety of microservice-related knowledge sources, and we gradually refined a meta-model that contains all the required elements to help us reconstruct existing microservice-based systems.

To increase our data set and explore the design space, we then created 20 additional models of system variants adapted from a published example according to discussions in the relevant literature in order to explore the possible decision space. Apart from the specific variations described in Tables 1 and 2, all other system aspects remained the same as in the base models. This resulted in a total of 30 models summarized in Tables 1 and 2. We assume that our evaluation systems are, or reflect, real-world practical examples of microservice architectures. As many of them are open source systems with the purpose of demonstrating practices or technologies, they are at most of the medium size and modest complexity, though. To ensure they represent industry practices, we have carefully compared them to commercial projects and industry guidelines. Those comparisons served as a guide to creating our 20 variants. We can assert that the found practices are representing industry practices, but in commercial systems, other combinations or interdependencies than in the studied open-source systems have been observed, too. To represent those combinations

[1] https://assuremoss.eu/en/

well in our data set, we created the 20 system variants following the practices observed in commercial systems and the literature/guidelines (see Section 2.1).

2.3 Definition of ADDs and Metrics

In parallel to and independent from the case study and model development, from our set of tactics, we selected a subset which the industrial security experts co-authoring our article, based on the collected data, judged as widely used security tactics for microservices. In particular, we selected the tactics on secure communication, identity management, and observability. We confirmed that these tactics are present or important concerns in the microservice open source systems studied. Based on both, the tactics and the open source code studies, we then defined a catalog of Architectural Design Decisions (ADDs) which use the found tactics as Design Options.

We then hypothesized a set of metrics that have the goal to automatically decide on each single decision point in our ADDs. These metrics are formally defined in Section 6. We developed and tested the metrics based on numerous small examples of our tactics and formalized them using a combination of set theory and first-order logic.

2.4 Case Study-Based Analysis and Evaluation

For case study-based analysis, we performed a systematic assessment on support or violation of the collected security tactics. Here, the three industrial security experts in the author team independently derived a recommendation based on the results of our tactics study which provides informal guidance for security experts to manually judge systems such as those in our models. Next, the other authors applied this recommendation as an ordinal rating scheme to each model variant summarized in Tables 1 and 2 to create a ground truth for our study. Then the three industrial security experts in the author team and two experts from another company reviewed the rating scheme and the ratings in the ground truth. Please note that the academic authors all have substantial experiences in industrial software development and software security, and all of the industrial authors have substantial experience in IT risk management and decision analysis in numerous industry projects (consulting for many different customers). In case of inconsistency of the votes, we performed a discussion among the involved experts to resolve the conflict. We would have applied a majority vote, if the discussion would not have yielded consistent votes; but in all cases consensus was reached after the discussion.

Finally, in a statistical analysis, we first inspected how well the independent variables correlate with the dependent variable using Spearman rank correlation and then assessed how well the hypothesized metrics can possibly predict the ground truth data by performing an ordinal regression analysis. Spearman rank correlation [32] is a widely used correlation analysis method suitable for continuous and discrete ordinal variables. We used R's *cor.test* function for this analysis. Ordinal regression is a widely used method for modeling an ordinal response's dependence on a set of independent predictors, which is applicable in a variety of domains. For the ordinal regression analysis we used the *lrm* function from the *rms* package in R [13].

2.5 Model Generation

For modeling microservice architectures we followed the method reported in our previous work [60]. We used our existing CodeableModels tool[2], a Python implementation for precisely specifying meta-models, models, and model instances in code. Based on CodeableModels, we realized automated code generators to generate graphical visualizations

[2] https://github.com/uzdun/CodeableModels

of all meta-models and models in PlantUML, detectors for detecting all relevant aspects of the metrics in the models, and generators for automatically calculating metrics.

2.6 Extraction of Architecture Models from Microservice Code

As explained above, this article uses an approach developed in our prior work for automatic extraction of architecture models from the code [36]. In this section, we briefly explain this static code analysis approach, to make this article self-contained. Numerous architecture reconstruction approaches have been proposed to automatically or semi-automatically produce architecture models from the source code [10, 30, 31]. Unfortunately, these approaches usually involve a substantial effort to either manually maintain the reconstructed architecture model or repeat the reconstruction after the system has evolved (see [21]), meaning that they are not well suited for supporting the continuous evolution of systems. In addition, automated approaches have low accuracy (see [14]), and much additional, manual effort is needed for correcting and augmenting their results. Finally, most reconstruction approaches focus on a very limited number of programming languages and technologies (see [10]), meaning they are hard to use with modern systems, such as microservice-based systems, which use typically polyglot programming, persistence, and technologies, often in their latest iterations.

Instead of aiming at developing a one-size-fits-all, generic reconstruction method as many other reconstruction approaches do, our approach leverages the fact that system experts usually know a lot about their projects and thus a generic, fully automated reconstruction may not be necessary. In particular, our work is an extension of the approach taken by Haitzer et al. [21]. We designed a detector-based approach that is capable of providing support for polyglot, continuously evolving systems, and can possibly use reusable detectors. Here, detectors are software components that continuously parse relevant parts of the source code and create model abstractions from the code. Reusable detectors are detectors that can be reused across different model abstraction tasks and projects.

Our approach presupposes that a system expert has identified the high-level architecture and the architectural security features of the system — with which the expert should be familiar either way — and modeled it in an execution script that iterates the detectors over each system element. In [36] we were able to show that this involves a modest initial effort per project, and then no or a relatively small per-release effort (such as for removal and addition of services and links between releases) is required.

3 CASE STUDY INSPECTION AND PREPARATION

This section explains the case studies' preparation and inspection. Tables 1 and 2 summarizes the 10 case studies we have manually modeled based on line-by-line inspection of their source from the linked Github sources. In addition, as explained above, we have derived 20 variant models to explore the decision space, by modeling possible other uses of the relevant security tactics, but keeping the rest of the system identical to the original. Each of those represents either an architectural refactoring step or a possible deterioration during system evolution. In our point of view, it is important to study such variations, as it is the goal to show that our approach helps to find issues that occur during refactoring or evolution steps. That is, our approach should be applicable in the context of a continuous delivery pipeline.

As shown in Figure 1, it is our goal to derive the models such as those in Tables 1 and 2 via static code analysis from the source code. This can be done using the approach from our previous work for architecture reconstruction of polyglot microservice systems [36]. In our previous work, we have extracted the components and connector model of the system RS0 in a fully automated fashion (and one other system). All other open source systems studied in this work, rely solely on the line-by-line manual analysis, and will in our future work be extracted automatically, too. Please

ID	*Model Size*	*Description/Sources*
AC0	7 components, 10 connectors	A system managing customer accounts, with OAuth2 support and Eureka service discovery. Source: https://github.com/piomin/sample-spring-oauth2-microservices/tree/with_database.
AC1	8 components, 11 connectors	Variant of AC0 that adds direct clients to service connections, some secure connections to OAuth server, plaintext authentication in all backend/client-service links, plaintext authorization on all backend/client-service links (one only with some requests scope), and zipkin tracing for account service (with plaintext sensitive data).
AC2	8 components, 14 connectors	Variant of AC0 that adds direct clients to services connections and link via API gateway, +secure connections in all links, SSL authentication to all backend/client-service links, token-based authorization on all backend/client-service links, and zipkin tracing for both services linked to clients/gateways.
BA0	11 components, 17 connectors	A REST API for creating and viewing bank accounts and transferring money between them; uses CQRS and event-driven communication via Kafka in the backend. Source: https://github.com/cer/event-sourcing-examples.
BA1	12 components, 21 connectors	Variant of BA0 that adds secure connections to Event Store, API Keys for API Gateway and services communication (for one services only for some requests), plaintext authentication for backendlinks, one view service accessed by clients directly, and Jaeger tracing for 4 services.
BA2	12 components, 24 connectors	Variant of BA0 that adds secure connections to event store and clients, API Keys for gateway/services communication, plaintext authentication in the backend over secure connections, encrypted authorization on all client-service paths, plaintext authorization with secure connections to event store,one service can be accessed by clients or by gateway, and Jaeger tracing for all services.
CI0	8 components, 12 connectors	NodeJS-based cinema API using interacting microservices shielded by an API Gateway, using encrypted communication, and a replicated Mongo DB. Source: https://github.com/Crizstian/cinema-microservice.
CI1	12 components, 21 connectors	Variant of CI0 that adds a Web UI client with HTTP connection, service authentication with API keys for all links except one, DB authentication using tokens, authorization with plaintext over secure connections in most links, authorization with encrypted information for database links, UI to gateway communication with HTTPS, a frontend facade service for catalog service (with some sensitive data), and Zipkin tracing for services except notification and frontend service.
CI2	12 components, 23 connectors	Variant of CI0 that adds a Web UI client with HTTPS connection, service/DB authentication with authentication tokens, authorization with plaintext over secure connections for services, authorization with encrypted information for database links, a frontend facade service and API gateway link for catalog and movies services, some sensitive data in component and connector code, and Zipkin tracing for all services.
CO0	13 components, 16 connectors	The Common Component Modelling Example (CoCoMe) realized as a microservice application using directly connected Web UIs. Source: https://github.com/cocome-community-case-study/cocome-cloud-jee-microservices-rest.
CO1	12 components, 19 connectors	Variant of CO0 that uses the services from the UI, and adds a frontend facade to the reports services, encrypted connections on all client and UI links, inter-service authentication via API keys (for two links only with some requests scope),DBs authenticated via SSL, restful client authenticated with API keys, inter-service plaintext authorization, no authorization for DBs, Jaeger tracing for 3 services and the facade, and 2 more services contain sensitive data in their code.
CO2	12 components, 16 connectors	Variant of CO0 that uses the services from the UI, and adds encrypted connections everywhere, service authentication is done with plaintext credentials for all requests,all DBs authenticated via SSL, inter-service plaintext authorization, all DB links authorized with encrypted information, Jaeger tracing for 3 services, and 3 more services contain sensitive data.
EP0	11 components, 11 connectors	A generic enterprise planner application following a microservices architecture shielded by an API gateway, using databases per service, and messaging-based communication. Source: https://github.com/gfawcett22/EnterprisePlanner.
EP1	16 components, 22 connectors	Variant of EP0 that adds a Web app & SPA Web client,encrypted messaging, HTTP basic authentication everywhere (for 1 link only for some requests), authorization with plaintext over secure links in the backend, authorization with encrypted information or tokens elsewhere, SPA web app to frontend communication with HTTPS, a frontend for web apps (to which all but one service are connected), direct connection of Web UIs to shipment service, sensitive data on service code.
EP2	16 components, 23 connectors	Variant of EP0 that adds a Web app & SPA Web client,encrypted messaging, all service client connections with HTTPs, API to services communication with HTTPs, SSL-based authentication, authorization with tokens on all distributed links, and a frontend for web apps to which all services are connected.

Table 1. Overview of modeled systems (size, details, and sources) – Part 1

note that this automation of the first step of our work is required for the continuous delivery-like application of our approach (which we can demonstrate so far only for RS0). All other steps of our approach, such as detectors, metrics calculation, visualization generations, and so on, have been fully automated already. The code and model data set are provided as an open access artifact on Zenodo to enable reproducibility of our study[3].

As an illustrative example of the modeling used in this paper, please consider the relatively simple Model AC0 in Figure 2. As can be seen, the system is accessed via an *API Gateway* by only one type of client, a RESTful HTTP client. All HTTP-based communication is plain HTTP, i.e. unencrypted. The Account service is access-controlled by plaintext[4] form-based login authentication (for all requests), offers single sign-on and sessions, and uses token-based authorization (for all requests). Communication to the Customer service happens via a Feign client which uses plaintext authentication (with sensitive data stored in the connector code). It uses the same token-based authorization as before. All authorization is handled via the OAuth2 service, which is also accessed using plaintext authentication, again with sensitive data stored in the connector code. The OAuth2 server also contains sensitive data in its code and in the code to access the AuthDB MySQL database (via unencrypted JDBC). All marked-up sensitive data in this system are

[3] https://doi.org/10.5281/zenodo.6424722

[4] Please note: Whenever ‘plaintext’ encoding is used, there is a risk of a Shared Secret. Sometimes plaintext encoding might be used and no secret is actually shared, e.g. consider a plaintext communication over an encrypted channel.

ID	*Model Size*	*Description/Sources*
ES0	18 components, 29 connectors	.NET Microservices Reference Application realizing an e-shop, with multiple clients and UIs, Backends-for-Frontends gateways, event-driven communication, GRPC and RESTful communication, and ELK monitoring. Source: https://github.com/dotnet-architecture/eShopOnContainers.
ES1	18 components, 37 connectors	Variant of ES0 that adds encrypted browser access, HTTPS for links to identity service/ELK, authentication with API keys for ID service, SSL-/token-based authentication to DBs, ELK services and event bus links, token-based authorization on grpc links, plaintext authorization over secure links to ELK, encrypted authorization on other links, identity service not authorized, ELK monitors for services but 1 service/1 BFF, and additional direct links to identity/catalog services.
ES2	18 components, 35 connectors	Variant of ES0 that adds encrypted browser access, HTTPS for links to identity service/ELK, encrypted grpc service links, authentication with SSL authentication for ID service, DBs, ELK services and event bus links, token-based authorization on all grpc links, plaintext authorization over secure links to ELK, encrypted authorization on other links,ELK monitors for all services but 1 service/1 BFF, and direct links to identity service.
OB0	13 components, 25 connectors	Online Boutique application that servers a sample application for the Google Could Platform using GRPC communication, stackdriver based monitoring and tracing, a frontend service, and a cart database. Source: https://github.com/GoogleCloudPlatform/microservices-demo.
OB1	13 components, 23 connectors	Variant of OB0 that adds encrypted database connections, connections to frontend with HTTPS, all links authenticated with plaintext, all links from gateways, to gcp stackdriver, and databases securely authorized, all links from checkout plaintext authorized, recommendation service links not authorized, 4 services directly accessible (rest behind frontend service), one service and the frontend not monitored, and 3 services contain sensitive data in their code.
OB2	13 components, 30 connectors	Variant of OB0 that adds encrypted client access, encrypted database connections, secure SSL authentication on all links, all links from gateways securely authorized, all links from checkout and recommendation services, and to gcp stackdriver and databases plaintext authorized, all services directly accessible and via frontend service, 2 services not monitored, and cart service contains sensitive data.
PM0	16 components, 37 connectors	Piggy Metrics, a microservice system for demonstrating collecting application metrics in a microservice architecture using a Zuul API Gateway, OAuth2, Sleuth tracing, Histrix monitoring, a Eureka registry, and various per-service DBs. Source: https://github.com/sqshq/piggymetrics.
PM1	18 components, 40 connectors	Variant of PM0 that adds additional an HTTPS client, oauth2 connections encrypted, no authentication used, all links not authenticated in PM0 now authenticated with SSL authentication, all links authorized with plaintext authorization (but most not secure connections), HTTPS client communication via a frontend facade, and service client via ZUUL gateway, additional backend HTTPS connections, and statistics service not traced or monitored.
PM2	17 components, 34 connectors	Variant of PM0 that adds encrypted connections, authentication with secure means except for one plaintext authentication, authorization with plaintext information (and use secure connections), the account service and oauth connected via a frontend service facade (not the Zuul gateway), andthe statistics and notification services not traced or monitored.
RS0	19 components, 30 connectors	Robot Shop, an application for demonstration Instana technology in highly polyglot microservice environment with services in different languages, different per-service DBs, various communication protocols, Instana tracing, and an NGINX API Gateway. Source: https://github.com/instana/robot-shop.
RS1	19 components, 32 connectors	Variant of RS0 that adds inter-service communication with HTTPS, authentication with API keys on gateway-service and inter-service communication, secure communication everywhere else with secure communication tokens, Rest client can bypass gateway to access catalogue, Cart service contains sensitive data in the code, andRatings service additional connected to Instana (traced).
RS2	19 components, 34 connectors	Variant of RS0 that adds HTTPS communication, authentication with API keys on gateway-service, SSL-based authentication on inter-service communication, secure communication everywhere else with secure communication tokens/SSL, ratings and catalog services directly accessed by UIs and clients, shipping and ratings services additional connected to Instana (traced), and cart, catalogue, ratings, payment, and shipping services contain sensitive data in their code.
TE0	17 components, 26 connectors	Tap-And-Eat application based on microservices using direct UI connection of various services, with per-service DBs, Hystrix monitoring, an Eureka discovery service, and a configuration service. Source: https://github.com/jferrater/Tap-And-Eat-MicroServices.
TE1	17 components, 28 connectors	Variant of TE0 that adds encrypted communication with Hystrix, some plaintext based authentication on path from client to services, some plaintext authentication between services, secure plaintext for authentication at discovery service, authorization with plaintext information everywhere, and two more services are monitored.
TE2	17 components, 30 connectors	Variant of TE0 that adds encrypted communication with Hystrix, HTTPS-based browser access, some plaintext-based authentication on path from client to services, plaintext based authentication between services, secure authenticationtoken used for authentication at config service, authorization with plaintext information everywhere except some inter-service encrypted authorization, two services are access from UIs without authorization, and all services but one monitored.

Table 2. Overview of modeled systems (size, details, and sources) – Part 2

credentials to access system services. Further, the system uses a Eureka discovery service which is accessed without any authentication or authorization.

Please note that the components and connectors are all of the types *Component* or *Connector*, respectively. Those types are extended by stereotypes for microservices architectures such as *API Gateway* or *Service* for components, or *RESTful HTTP* or *JDBC* for connectors. In addition, we offer such types for security annotations to mark up security features, such as *Authentication with Plaintext Credenditals* or *Token-based Authorization*. The metrics definitions in Section 6 define those types (modeled here as stereotypes in the model) that they need to detect the relevant metrics. Please note that our models also contain other security-related types than just those used in the metrics below; e.g. the *Single Sign-On* type in Figure 2 is not yet used in our metrics.

In a likewise manner we have modeled all 30 system models enlisted in Tables 1 and 2. Please note that some of the models contain a substantially higher number of components and connectors than the model in Figure 2 which is among the simplest models in our model data set. All models can be found in our replication data set.

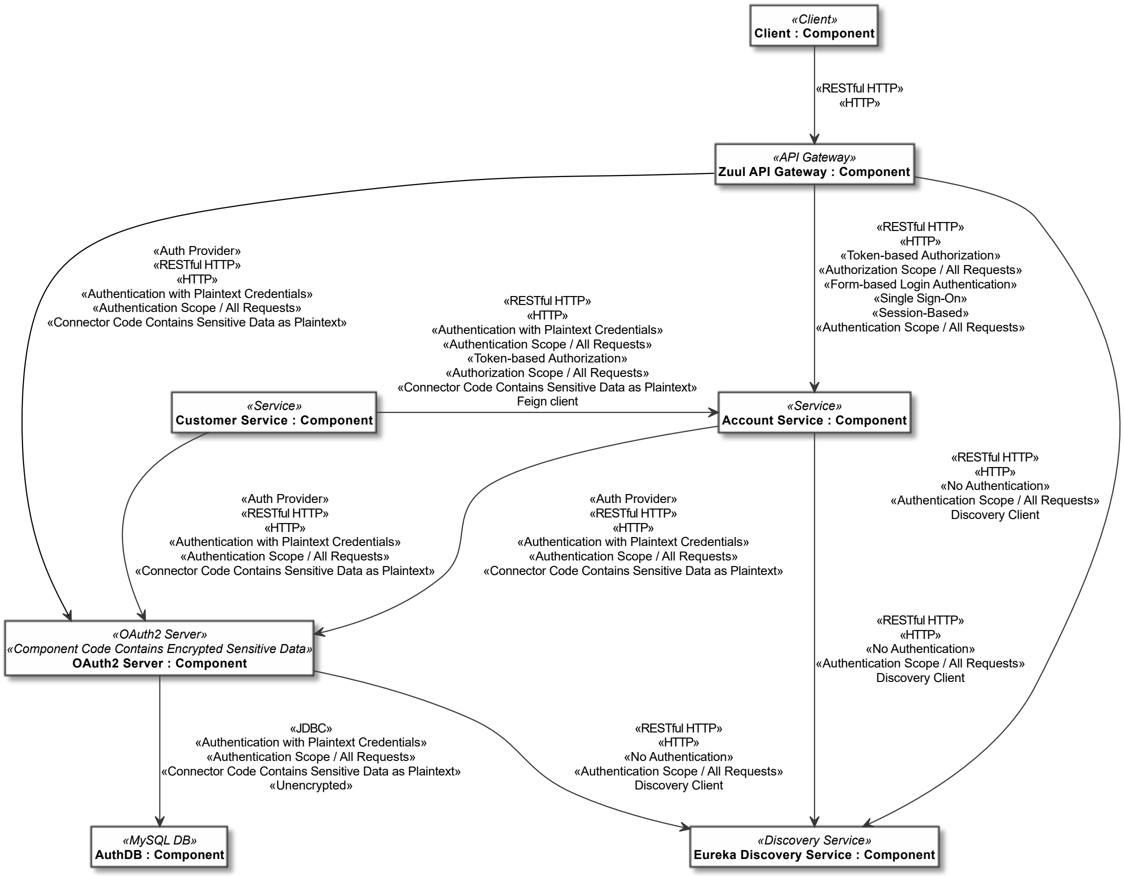
«Client»
Client : Component
«RESTful HTTP»
«HTTP»
«API Gateway»
Zuul API Gateway : Component
«Auth Provider»
«RESTful HTTP»
«HTTP»
«Authentication with Plaintext Credentials»
«Authentication Scope / All Requests»
«Connector Code Contains Sensitive Data as Plaintext»
«RESTful HTTP»
«HTTP»
«Token-based Authorization»
«Authorization Scope / All Requests»
«Form-based Login Authentication»
«Single Sign-On»
«Session-Based»
«Authentication Scope / All Requests»
«RESTful HTTP»
«HTTP»
«Authentication with Plaintext Credentials»
«Authentication Scope / All Requests»
«Token-based Authorization»
«Authorization Scope / All Requests»
«Connector Code Contains Sensitive Data as Plaintext»
Feign client
«Service»
Customer Service : Component
«Service»
Account Service : Component
«RESTful HTTP»
«HTTP»
«No Authentication»
«Authentication Scope / All Requests»
Discovery Client
«Auth Provider»
«RESTful HTTP»
«HTTP»
«Authentication with Plaintext Credentials»
«Authentication Scope / All Requests»
«Connector Code Contains Sensitive Data as Plaintext»
«Auth Provider»
«RESTful HTTP»
«HTTP»
«Authentication with Plaintext Credentials»
«Authentication Scope / All Requests»
«Connector Code Contains Sensitive Data as Plaintext»
«RESTful HTTP»
«HTTP»
«No Authentication»
«Authentication Scope / All Requests»
Discovery Client
«OAuth2 Server»
«Component Code Contains Encrypted Sensitive Data»
OAuth2 Server : Component
«JDBC»
«Authentication with Plaintext Credentials»
«Authentication Scope / All Requests»
«Connector Code Contains Sensitive Data as Plaintext»
«Unencrypted»
«RESTful HTTP»
«HTTP»
«No Authentication»
«Authentication Scope / All Requests»
Discovery Client
«MySQL DB»
AuthDB : Component
«Discovery Service»
Eureka Discovery Service : Component

Fig. 2. Illustrative Example: Overview of the AC0 Model

4 ADDS FOR MICROSERVICE SECURITY TACTICS

In this section, we present the ADDs we have derived from the selected security tactics. As mentioned in Section 2, our collected catalog of security tactics is much broader than the ones used here. For example, just the security strategies in the NIST SP 800-204 special publication [34] cover the categories identity management, access management, service registry configuration, secure communication, security monitoring/observability, availability and resilience improvement, induction of new versions of microservices, preventing credential abuse and stuffing attacks, and API gateway implementation, and service mesh implementation. Thus, we have selected three representative and widely used categories of microservice security tactics, namely secure communication, identity management, and observability for development, demonstration, and evaluation of our approach. Those were selected based on an assessment by the industrial experts in the author team, who confirmed that those are among the most widely used security tactics for microservices at the moment.

4.1 Tactics and ADD on Secure Communication

When considering the Secure Communication category and our scope of architecture decomposition models, we mainly found various tactics related to the use of encrypted protocols. We need to consider if a connector is a backend connector or a connector on the paths from clients or UIs to system services. Here, the backend refers to those connectors inside of the system that are not linked to clients or UIs. As the connectors on the paths from clients or UIs to system services can be seen as more vulnerable, we treat them as a special case in our ADD on Secure Communication. Both cases are treated in one ADD, as the ideal option in this category is that all communication is encrypted. The resulting ADD is:

ADD: *Secure Communication (SC)*

Decision Scope: Each connector between two components in the system (backend connectors) and each connector on paths from clients or UIs to system services, but not connectors to external services. To be decided for each connector.

Decision Options (Security Tactics):

- **Encrypted Communication:** Encrypted protocols such as SSL/TLS or HTTPS are used for communication on a connector. We can further distinguish:
  - **Encrypted Backend Communication**
  - **Encrypted Communication on Paths from Client/UIs to System Services**
- **Unencrypted Communication:** Unencrypted protocols such as HTTP are used for communication on a connector. We can further distinguish:
  - **Unencrypted Backend Communication**
  - **Unencrypted Communication on Paths from Client/UIs to System Services**

4.2 Tactics and ADDs on Identity Management

When considering the Identity Management category and our scope of architecture decomposition models, we mainly found various authentication-related tactics. In this context, it is important to note that authentication is crucial for all parts of the microservice architecture, but especially for services reachable directly or indirectly from the clients. As it makes in some case sense to only authenticate at the entry points of a system, we model the two resulting cases, backend authentication and authentication on paths from clients/UIs to services, using two separate ADDs with similar tactics, but a different decision scope:

ADD: *Backend Authentication (BE_AE)*

Decision Scope: Each connector between two components in the system (such as system services, databases, infrastructure components, discovery services, or access management servers), but not connections to clients and UIs (or between them) and/or external services. To be decided for each connector.

Decision Options (Security Tactics):

- **Token-based Authentication:** Authentication is performed using cryptographic identity tokens such as SAML, OpenID, JWT tokens.
- **Protocol-based Authentication:** Authentication is performed based on the features of an encrypted communication protocol, such as SSL- or SASL-based authentication.

- **API Keys:** Authentication is performed based on API Keys [64], i.e. authentication with a unique token per client that the client can present to the API endpoint for identification purposes.
- **Plaintext-based Authentication:** Authentication information is transferred as plaintext, such as in HTTP Basic Authentication, form-based authentication, or home-grown implementations.
- **Plaintext-based Authentication over an Encrypted Protocol:** Authentication information is transferred as plaintext over a secure (i.e. encrypted) communication protocol such as TLS/SSL.
- **No Authentication:** No authentication method is used.
- **Authentication Not Required:** The connector does not need any form of authentication. This is for instance the case, if a public API is offered that can be accessed by any client without restrictions.

ADD: *Authentication on Paths from Clients or UIs to System Services (CP_AE)*
Decision Scope: Each direct or transitive connector between a client or UI to a system service. In this context, transitive means that the connector can cross API Gateways and similar frontend components first, and then other system services, but no other kinds of components (such as infrastructure services, databases, and so on). To be decided for each connector.
Decision Options (Security Tactics): Exactly the same options are used as in the *Backend Authentication* ADD, but instead of the scope of distributed backend connectors, in the scope of paths from clients or UIs to system services.

4.3 Tactics and ADD on Observability

Observability in microservices means ensuring that access to the data required to identify problems and detect defects is provided. This is critical to "detect attacks and identify factors for degradation of services (which may impact availability)" [34]. Please note that service-level degradation or availability issues may be caused by an attack, e.g. a denial-of-service attack. Monitoring for security should be performed at both the system level (e.g. gateways) and service level to detect, report, and respond to inappropriate behavior [34]. Whereas the previous tactics provide "protective" security controls, observability measures provide "detective" controls in the sense that they are used "to ensure that the system is operating as designed and the data is secure" [8]. It is usually performed through infrastructure services for performing monitoring, logging, and/or tracing [37]. Thus, when considering component-level observability, we can distinguish between ordinary components of the system and components facing clients or UIs, such as *API Gateways* [48], *Backends for Frontends* [48], or frontend services. The latter can be summarized as *Facade* components.

ADD: *Microservice Observability (OBS)*
Decision Scope: (1) Each component that is a system service; that is, it is a service, but not an external one or an infrastructure component and not a Facade component. (2) Each component that is a Facade component, i.e. either an *API Gateway*, a *Backend for Frontends*, or a frontend service.
Decision Options (Security Tactics):

- **Observing Facade Components:** Facade components are observed (i.e., logged, monitored, or traced) by one or more dedicated components.
- **Observing System Services:** System services are observed (i.e., logged, monitored, or traced) by one or more dedicated components.

- **No Observation:** No observation is used.

Depending on which aspects require observation in a microservice system, three types of observability support can be distinguished, which are often combined in existing tools:

- **Logging:** Logs are records of events related to the system's state. Logging refers to the management of logs. It is used e.g. to help in diagnosing defects or providing auditing capabilities per component (microservice).
- **Monitoring:** Monitoring describes the use of metrics to observe how a service is handling its requests. The goal is to achieve a global view of the system that is being monitored.
- **Tracing:** Tracing (or distributed tracing or distributed request tracing) aims to observe the different conversations in a service-based system separately. A trace describes the complete processing of a request through different parts of the distributed system. Each trace is comprised of a number of spans. A span describes the operations happening in a single service of the distributed system. Tracing is typically focusing on performance or other optimizations across services.

5 GROUND TRUTH

To create a ground truth for the assessment of conformance to the ADDs described in the previous section, first the three industrial security experts in the author team together with other experts in their company created recommendations based on the results of our tactics study (i.e., from security guidelines, gray literature, and scientific literature studies). These recommendations provide informal guidance for security experts to manually judge systems such as those in our models. The other authors then analyzed these recommendations, compared them to the actual implementations found in the case study systems, and selected the recommendations in focus of our ADDs. The results are the recommendation per ADD in Table 3 in which more or less preferred ADD options (tactics) are mapped to a 5-point ordinal scale:

- ++: very well supported;
- +: well supported, but aspects of the solution could be improved;
- ~: serious flaws in the security design, but substantial support can already be found in the system;
- −: serious flaws in the security design, but initial support can already be found in the system;
- −−: no support for the security tactic can be found in the system.

This assessment scheme we then discussed again with the three industrial security experts until consensus was reached. The other authors then assessed the 10 case study systems and the 20 case study variants for conformance to each of the ADDs. The assessments are again reviewed by the three security assessment experts on the author team, as explained in Section 2. In addition, two industrial security experts from another company have reviewed our models, metrics, and code, and assessed their feasibility and applicability in the context of a continuous certification scheme (see Section 8.2 for details). Please note that some parts of the recommendation in Table 3 lead to a crisp assessment, especially the extreme cases (++, −−) are often referring to clear cut sets such as *all connectors* or *no connectors*. However, some of the other values contain fuzzy statements such as *the large majority of the connectors*, where human judgement per system model is required, to decide how big the set must be in order to be acceptable in this particular model. For example, system context, system size, and domain of the system are factors that can lead to individually different judgement on different models.

For each ground truth assessment, first, the architectural abstraction from source code to models had to be made, which requires locating architectural security concerns which are often scattered across multiple source code artifacts. In the next step, it is required to judge the UML models (annotated with the additional types explained in Section 6

ADD	*Assessment*	*Conditions/Explanation*
Secure Communication (SC)	++	All distributed connectors use **Encrypted Communication**.
	+	**Encrypted Communication** is supported for all connectors on paths from clients or UIs to system services (i.e., the most vulnerable connectors).
	~	Either the large majority of connectors on paths from clients or UIs to system services (i.e., the most vulnerable connectors) uses **Encrypted Communication**; or the majority of all distributed connectors plus the majority of connectors on paths from clients or UIs to system services use **Encrypted Communication**.
	−	At least some distributed connectors use **Encrypted Communication**, but less than the large majority of connectors on paths from clients or UIs to system services; or less than the majority of all distributed connectors plus less than the majority of connectors on paths from clients or UIs to system services use **Encrypted Communication**.
	−−	No distributed connectors are use **Encrypted Communication**.
Backend Authentication (BE_AE)	++	All distributed backend connectors are authenticated either with **Token-based Authentication** or **Protocol-based Authentication**.
	+	All distributed backend connectors are authenticated either with **Token-based Authentication**, **Protocol-based Authentication**, or **Plaintext-based Authentication over an Encrypted Protocol**.
	~	Either the large majority of distributed backend connectors is authenticated with **Token-based Authentication**, **Protocol-based Authentication**, or **Plaintext-based Authentication over an Encrypted Protocol**; or all distributed backend connectors are authenticated, but some or all of those are authenticated with **Plaintext-based Authentication** or **API Keys**.
	−	At least some distributed backend connectors are authenticated, but either **Plaintext-based Authentication** or **API Keys** are used and not all connectors are authenticated; or, if no **Plaintext-based Authentication** or **API Keys** is used, less then the large majority of distributed backend connectors is authenticated with **Token-based Authentication**, **Protocol-based Authentication**, or **Plaintext-based Authentication over an Encrypted Protocol**.
	−−	No distributed backend connectors are authenticated.
	Notes	If the *Authentication Not Required* option is selected, the connector should not be further analyzed with regard to identity management aspects.
Authentication on Paths from Clients/UIs to Services (CP_AE)	++	All connectors on paths from clients or UIs to system services are authenticated either with **Token-based Authentication** or **Protocol-based Authentication**.
	+	All connectors on paths from clients or UIs to system services are authenticated either with **Token-based Authentication**, **Protocol-based Authentication**, **API Keys**, or **Plaintext-based Authentication over an Encrypted Protocol**.
	~	Either the large majority of connectors on paths from clients or UIs to system services is authenticated with **Token-based Authentication**, **Protocol-based Authentication**, **API Keys**, or **Plaintext-based Authentication over an Encrypted Protocol**; or all these connectors are authenticated, but some or all of those are authenticated with **Plaintext-based Authentication**.
	−	At least some connectors on paths from clients or UIs to system services are authenticated, but either **Plaintext-based Authentication** is used and not all connectors are authenticated; or, if no **Plaintext-based Authentication** is used, less then the large majority of these connectors are authenticated with **Token-based Authentication**, **Protocol-based Authentication**, **API Keys**, or **Plaintext-based Authentication over an Encrypted Protocol**.
	−−	No connectors on paths from clients or UIs to system services are authenticated.
	Notes	This scheme is almost identical to the one for BE_AE with the exception that the **API Keys** option is viewed on paths from clients or UIs to system services as an inferior, but still acceptable option, whereas its use in the backend is not recommended (see [34, 64]).
Observability (OBS)	++	All **System Services** and all **Facade Components are Observed** (logged, monitored, or traced) by a dedicated component.
	+	All **System Services** or all **Facade Components are Observed** (logged, monitored, or traced) by a dedicated component.
	~	The large majority of **System Services** and **Facade Components are Observed** (logged, monitored, or traced) by a dedicated component.
	−	At least some **System Services** and **Facade Components are Observed** (logged, monitored, or traced) by a dedicated component.
	−−	**No System Services or Facades are Observed** (logged, monitored, or traced) by a dedicated component.

Table 3. Summary of the Recommendations for the ADDs

Decision	*AC0*	*AC1*	*AC2*	*BA0*	*BA1*	*BA2*	*CI0*	*CI1*	*CI2*	*CO0*	*CO1*	*CO2*	*EP0*	*EP1*	*EP2*	*ES0*	*ES1*	*ES2*	*OB0*	*OB1*	*OB2*	*PM0*	*PM1*	*PM2*	*RS0*	*RS1*	*RS2*	*TE0*	*TE1*	*TE2*
SC	--	-	++	--	-	+	+	~	+	--	+	++	--	~	++	--	~	++	--	-	++	-	~	++	--	-	~	--	-	-
BE_AE	-	~	++	-	~	~	-	~	++	-	~	+	-	-	++	-	~	++	--	~	++	-	--	+	--	~	~	--	-	-
CP_AE	~	~	++	~	~	+	--	+	++	--	~	+	--	-	++	~	+	++	--	~	++	~	--	++	--	+	+	--	-	-
OBS	--	-	++	--	-	++	--	~	+	--	+	-	--	-	++	+	~	++	+	~	+	++	-	-	-	~	+	-	-	~

Table 4. Ground Truth Assessment for the Case Study Systems

as stereotypes, such as those in the example in Figure 2). In this judgment, there is always a subjective element of human judgment. Please note that some of the example systems are way more complex than the example in Figure 2 and real-life commercial systems are often even substantially larger and more complex. Thus, without knowledge of systematic assessment such as our metrics and their automated assessment outcomes, explained below, an objective decision is in many cases hard to reach and non-obvious.

Table 4 enlists the ground truth assessments for each decision and for each the case study systems from Tables 1 and 2.

6 METRICS

In this section, we describe the metrics we have hypothesized for each of the decisions described in Section 4. All metrics, unless otherwise noted, are continuous values ranging from 0 to 1, with 1 representing the optimal case where the respective option is fully supported, and 0 the worst-case scenario where it is completely absent. Our metrics are based on a microservices-based architecture decomposition model. We use formal definitions adapted from our prior work [60] to define the metrics below. We understand the microservices-based architecture model as a decomposition into a directed components and connectors graph with a set of component types for each component and a set of connector types for each connector, formally:

ff f f

An architecture decomposition model M is a tuple (CP_M, CN_M, CPT_M, CNT_M, cn_source, cn_target, $cp\_directtype$, $cn\_directtype$, $cp\_supertype$, $cn\_supertype$, cp_type, cn_type) where:

- CP_M is a finite set of **component nodes** in Model M.
- $CN_M \subseteq CP_M \times CP_M$ is an ordered finite set of **connector edges**.
- CPT_M is a set of **component types**.
- CNT_M is a set of **connector types**.
- $cn\_source : CN_M \rightarrow CP_M$ is a function returning the component that is the **source** of a link between two components.
- $cn\_target : CN_M \rightarrow CP_M$ is a function returning the component that is the **target** of a link between two components.
- $cp\_conn : CP_M \rightarrow \mathbb{P}(CN_M)$ is a function returning the connectors that are connected to a particular component, i.e. $\forall cp \in CP_M, cn \in cp\_conn(cp) : cn\_source(cn) = cp \lor cn\_target(cn) = cp$.
- $cp\_directtype : CP_M \rightarrow \mathbb{P}(CPT_M)$ is a function that maps each component node cp to its set of **direct component types**,
- $cp\_supertype : CPT_M \rightarrow \mathbb{P}(CPT_M)$ is a function called **component type hierarchy**. $cp\_supertype_M(cpt)$ is the set of direct supertypes of cpt; cpt is called the subtype of those supertypes. The transitive closure[5] $cp\_supertype^*$ defines the inheritance in the hierarchy such that $cp\_supertype^*(cpt)$ contains the **direct and indirect (aka transitive) supertypes** of cpt. The inheritance hierarchy is cycle free, i.e. $\forall cpt \in CPT_M : cp\_supertype^*(cpt) \cap \{cpt\} = \emptyset$.
- $cp\_type : CP_M \rightarrow \mathbb{P}(CPT_M)$ is a function that maps each component to its set of **direct and transitive component types**, i.e., $\forall cp \in CP_M, dt \in CPT_M : dt = cp\_directtype(cp) \Rightarrow cp\_type(cp) = dt \cup cp\_supertype^*(dt)$.
- $cn\_directtype : CN_M \rightarrow \mathbb{P}(CNT_M)$ is a function that maps each connector cn to its set of **direct connector types**.
- $cn\_supertype : CNT_M \rightarrow \mathbb{P}(CNT_M)$ is a function called **connector type hierarchy**. $cn\_supertype(cnt)$ is the set of direct supertypes of cnt; cnt is called the subtype of those supertypes. The transitive closure $cn\_supertype^*$ defines the inheritance in the hierarchy such that $cn\_supertype^*(cnt)$ contains the **direct and indirect (aka transitive) supertypes** of cnt. The inheritance hierarchy is cycle free, i.e. $\forall cnt \in CNT : cn\_supertype^*(cnt) \cap \{cnt\} = \emptyset$.
- $cn\_type : CN_M \rightarrow \mathbb{P}(CNT_M)$ is a function that maps each connector to its set of **direct and transitive connector types**, i.e., $\forall cn \in CN_M, dt \in CNT_M : dt = cn\_directtype(cn) \Rightarrow cn\_type(cn) = dt \cup cn\_supertype^*(dt)$.

[5] All transitive closures in this article are assumed to be calculated with a standard algorithm for transitive closures like Warshall's algorithm.

Please note that below, to simplify the metrics definition texts, when we simply say Component cp or Connector cn is of type t, we refer to the use of the function call $cp.cp\_type(t)$ or $cn.cn\_type(t)$, respectively. We introduce the component and connector types used in this article in the text below when they are first needed. The full type hierarchies are modeled in the CodeableModels distribution[6].

Detectors are functions that calculate a **Detector Result** DR, such as $detector : \mathbb{P}(ME_M) \longrightarrow DR$, where ME_M are model elements of a model M, with: $\forall CN_M, CP_M \in M : ME_M \supset CP_M \wedge ME_M \supset CN_M$. DR is a tuple $<$ *successful, undefined, failed* $>$. $dr.successful$ returns the successful detection results (with $\forall dr \in DR : dr.successful \in \mathbb{P}(ME_M)$), $dr.failed$ returns the failed detection results (with $\forall dr \in DR : dr.failed \in \mathbb{P}(ME_M)$), and $dr.undefined$ returns the detection results for which success or failure cannot be decided (with $\forall dr \in DR : dr.undefined \in \mathbb{P}(ME_M)$).

6.1 Metrics for Secure Communication (SC) Decision

The metrics for the SC decision are shown in Figure 3. The first metric **Secure Distributed Connectors** $SCO : \mathbb{P}(CN_M) \longrightarrow [0, 1]$ sets the secure distributed connectors in relation to all distributed connectors. Both numerator and denominator of the ratio rely on the $distributed\_connectors : \mathbb{P}(CN_M) \longrightarrow \mathbb{P}(CN_M)$ function. A connector is a **distributed connector** if it is not of type $InMemoryConnector$. The detector $secure\_connectors$ is $successful$, if a connector is of type $EncryptedCommuniction$ (or $HTTPS$). The next metric **Secure Client Connectors** $SCC : \mathbb{P}(CN_M) \longrightarrow [0, 1]$ is similar, but it uses the $client\_connectors : \mathbb{P}(CN_M) \longrightarrow \mathbb{P}(CN_M)$ function as its basis. This function selects the connectors with a cn_source of type $Client$, or a cn_source of type UI but only if this UI does not connect to another UI as target (i.e. it selects also system clients in UIs such as AJAX calls). **Secure UI Connectors** $SUC : \mathbb{P}(CN_M) \longrightarrow [0, 1]$, in turn, uses $ui\_connectors : \mathbb{P}(CN_M) \longrightarrow \mathbb{P}(CN_M)$ as its and selects only those connectors that have a cn_source of type UI.

In addition to these atomic metrics, we defined two metrics aggregating features from the others. Both again have the same form of metric definition as the previous ones (i.e. using $secure\_connectors$ with varying basis sets). **Secure External Client/UI Connectors (SEC)** $SEC : \mathbb{P}(CN_M) \longrightarrow [0, 1]$ calculates the external connectors to clients/UIs simply as the set union of the client and UI connectors, and uses this set as the basis for its definition. Finally, **Secure Internal Distributed Connectors (SIC)** is based on the internal distributed connectors. We call these in the following backend connectors: A connector is a **backend connector** if it is not connected to a component of type $Client$ or UI. We are interested in the distributed backend connectors which the function $distributed\_backend\_connectors : CN_M \longrightarrow \mathbb{P}(CN_M)$ selects by calculating the complement of the distributed connector set and the union of the client and UI connector sets.

6.2 Metrics for Backend Authentication (BE_AE) Decision

The metrics for the BE_AE decision are shown in Figure 4. They are all addressing distributed backend connectors that require authentication. To define this set of connectors formally, we first define a function $distributed\_backend\_connectors\_requiring\_authentication : \mathbb{P}(CN_M) \longrightarrow \mathbb{P}(CN_M)$ for constructing this connector set. The basis for this definition is the function $distributed\_backend\_connectors$ defined in Section 6.1. By placing the type $AuthenticationNotRequired$ onto a connector, an architect can indicate that a **connector does not require to be authenticated** (e.g. consider a connector to a Public API); we use the function

[6] The component type hierarchy can be found at: https://github.com/uzdun/CodeableModels/blob/master/docs/_images/Component_Stereotypes.png, and the connector type hierarchy is at: https://github.com/uzdun/CodeableModels/blob/master/docs/_images/Connector_Stereotypes.png. Both models are explained in the documentation of CodeableModels: https://uzdun.github.io/CodeableModels/07_meta_model_with_stereotypes.html.

Secure Distributed Connector (SCO)

$$SCO(cn) = \frac{|\ \mathit{secure\_connectors(distributed\_connectors(cn)).successful}\ |}{|\ \mathit{distributed\_connectors(cn))}\ |}$$

Secure Client Connectors (SCC)

$$SCC(cn) = \frac{|\ \mathit{secure\_connectors(client\_connectors(cn)).successful}\ |}{|\ \mathit{client\_connectors(cn))}\ |}$$

Secure UI Connectors (SUC)

$$SCC(cn) = \frac{|\ \mathit{secure\_connectors(ui\_connectors(cn)).successful}\ |}{|\ \mathit{ui\_connectors(cn))}\ |}$$

Secure External Client/UI Connectors (SEC)

$$SEC(cn) = \frac{|\ \mathit{secure\_connectors(client\_connectors(cn)} \cup \mathit{ui\_connectors(cn)).successful}\ |}{|\ \mathit{client\_connectors(cn)} \cup \mathit{ui\_connectors(cn))}\ |}$$

Common Function For Distributed Backend Connectors

$$\mathit{distributed\_backend\_connectors} = \mathit{distributed\_connectors(cn)} \setminus (\mathit{client\_connectors(cn)} \cup \mathit{ui\_connectors(cn)})$$

Secure Internal Distributed Connectors (SIC)

$$SIC(cn) = \frac{|\ \mathit{secure\_connectors(distributed\_backend\_connectors(cn)).successful}\ |}{|\ \mathit{distributed\_backend\_connectors(cn)}\ |}$$

Fig. 3. Metrics Definitions for Secure Communication

$connectors\_that\_require\_authentication : \mathbb{P}(CN_M) \rightarrow \mathbb{P}(CN_M)$ to selects the connectors from of a model M's connectors which are not of type $AuthenticationNotRequired$.

Based on the $distributed\_backend\_connectors\_requiring\_authentication$ function, we define the metrics in the remainder of Figure 4. We first define the **Authenticated Backend Connectors** $AEI : \mathbb{P}(CN_M) \rightarrow [0, 1]$ metric which returns the number of authenticated distributed backend connectors requiring authentication in relation to the total number of distributed backend connectors requiring authentication. This way, we can measure the total level of authentication in a system without considering the specific authentication method used. Here, $authenticated\_connectors : \mathbb{P}(CN_M) \rightarrow DR$ is a detector that detects the authenticated distributed backend connectors.

Next, we define the **Securely Authenticated Backend Connectors** $AEI\_S : \mathbb{P}(CN_M) \rightarrow [0, 1]$ which selects only the securely authenticated distributed backend connectors requiring authentication in relation to the total number of distributed backend connectors requiring authentication. Here, "securely" is measured through the detector $securely\_authenticated\_connectors : \mathbb{P}(CN_M) \rightarrow DR$ which detects the authenticated connectors which are either of type $ProtocolBasedSecureAuthentication$ or $SecureAuthenticationToken$. The **Backend Connectors Authenticated with API Keys** $AEI\_K : \mathbb{P}(CN_M) \rightarrow [0, 1]$ metric selects only the distributed backend connectors authenticated with API Keys, in relation to the total number of distributed backend connectors. This is done using the $connectors\_authenticated\_with\_api\_keys : \mathbb{P}(CN_M) \rightarrow DR$ detector, which detects the distributed backend connectors of type $AuthenticatedWithAPIKeys$. The **Backend Connectors Authenticated with Plaintext Credentials** $AEI\_P : \mathbb{P}(CN_M) \rightarrow [0, 1]$ metric selects only the distributed backend connectors authenticated with plaintext credentials, in relation to the total number of distributed backend connectors. This is done using the

Common Function For Distributed Backend Connectors Requiring Authentication

$$\mathit{distributed\_backend\_connectors\_requiring\_authentication(cn) = connectors\_that\_require\_authentication(distributed\_backend\_connectors(cn))}$$

Authenticated Backend Connectors (AEI)

$$AEI(cn) = \frac{|\ \mathit{authenticated\_connectors(distributed\_backend\_connectors\_requiring\_authentication(cn)).successful}\ |}{|\ \mathit{distributed\_backend\_connectors\_requiring\_authentication(cn))}\ |}$$

Securely Authenticated Backend Connectors (AEI_S)

$$AEI\_S(cn) = \frac{|\ \mathit{securely\_authenticated\_connectors(distributed\_backend\_connectors\_requiring\_authentication(cn)).successful}\ |}{|\ \mathit{distributed\_backend\_connectors\_requiring\_authentication(cn)}\ |}$$

Backend Connectors Authenticated with API Keys (AEI_K)

$$AEI\_K(cn) = \frac{|\ \mathit{connectors\_authenticated\_with\_api\_keys(distributed\_backend\_connectors\_requiring\_authentication(cn)).successful}\ |}{|\ \mathit{distributed\_backend\_connectors\_requiring\_authentication(cn)}\ |}$$

Backend Connectors Authenticated with Plaintext Credentials (AEI_P)

$$AEI\_P(cn) = \frac{|\ \mathit{connectors\_authenticated\_with\_plaintext\_credentials(distributed\_backend\_connectors\_requiring\_authentication(cn)).successful}\ |}{|\ \mathit{distributed\_backend\_connectors\_requiring\_authentication(cn)}\ |}$$

Authenticated Backend Connectors Over a Secure Connection (AEI_C)

$$AEI\_C(cn) = \frac{|\ \mathit{authenticated\_connectors(secure\_connectors(distributed\_backend\_connectors\_requiring\_authentication(cn).successful)).successful}\ |}{|\ \mathit{distributed\_backend\_connectors\_requiring\_authentication(cn)}\ |}$$

Authenticated Backend Connectors Using a Secure Method or Transferred Over a Secure Connection (AEI_A)

$$\begin{aligned}&\mathit{authenticated\_backend\_connectors\_using\_secure\_method\_or\_secure\_communication(cn)} = \{acn \in cn :\\ &\quad \mathit{securely\_authenticated\_connectors(distributed\_backend\_connectors\_requiring\_authentication(cn)).successful} \vee\\ &\quad \mathit{authenticated\_connectors(secure\_connectors(distributed\_backend\_connectors\_requiring\_authentication(cn)).successful).successful}\}\end{aligned}$$

$$AEI\_A(cn) = \frac{|\ \mathit{authenticated\_backend\_connectors\_using\_secure\_method\_or\_secure\_communication(cn).successful}\ |}{|\ \mathit{distributed\_backend\_connectors\_requiring\_authentication(cn)}\ |}$$

Fig. 4. Metrics Definitions for Backend Authentication

$connectors\_authenticated\_with\_plaintext\_credentials : CN_M \rightarrow DR$ detector, which detects the distributed backend connectors of type $AuthenticatedWithPlaintextCredentials$.

Authenticated Backend Connectors Over a Secure Connection $AEI\_C : \mathbb{P}(CN_M) \rightarrow [0, 1]$ selects only the authenticated distributed backend connectors which are sent over a secure connection, in relation to the total number of distributed backend connectors. It uses the $secure\_connectors : \mathbb{P}(CN_M) \rightarrow DR$ detector defined above, and uses the successful results of this detector as input for the $authenticated\_connectors : \mathbb{P}(CN_M) \rightarrow DR$ detector, defined at the beginning of this section. Finally, the **Authenticated Backend Connectors Using a Secure Method or Transferred Over a Secure Connection** $AEI\_A : \mathbb{P}(CN_M) \rightarrow [0, 1]$ metric selects only the distributed backend connectors which either use a secure authentication method, as defined above via the $securely\_authenticated\_connectors : \mathbb{P}(CN_M) \rightarrow DR$ detector or are authenticated and use a secure connection.

6.3 Metrics for Authentication on Paths from Clients or UIs to System Services (CP_AE) Decision

The metrics for the CP_AE decision are all addressing detectors on the path from clients or UIs to system services. They are defined in Figure 5. In them, we focus on the connectors between clients/UIs and system services that require authentication. This is calculated by the function $client\_service\_path\_connectors\_requiring\_authentication : \mathbb{P}(CP\_M) \rightarrow \mathbb{P}(CN\_M)$, which has the following ingredients:

- A basic notion in this function is a **path**: A path p is a sequence of components $p = (c_1, \ldots, c_n)$ with $c_1 \ldots c_n \in CP\_M$ which are all connected via connectors such that $\forall c_n, c_{n+1} \in P\, \exists\, cn \in CN\_M : cn\_source(cn) = c_n \wedge cn\_target(cn) = c_{n+1}$. Let P_M denote the set of all paths in model M.
- The function $all\_paths\_from\_clients\_or\_uis\_to\_system\_services : \mathbb{P}(CP\_M) \rightarrow \mathbb{P}(P_M)$ selects all paths from clients or UIs to system services. For this, the function first selects all components of type *Client* or *UI* as clients, and all components of type *Service* as system services. A service is a **system service**, if it is not of the type *ExternalComponent* (i.e., no external services are system services), *MiddlewareService* (i.e., no middleware infrastructure services such as a Discovery Service), or *Facade* (i.e., no frontend services or gateways with the sole purpose of shielding the system from clients). Then the function uses a simple Depth-First Search algorithm to calculate all paths from clients to services. From those paths, we select only the ones that are well-formed in the sense that first *Clients* or *UIs* are on the paths, then zero, one, or more *Facades* (e.g. *APIGateways* or frontend services are of type *Facade*), and finally one or more system services (in the sense defined above). That is, paths going across other components such as *Databases* or *MiddlewareServices* are excluded; paths going into the system, then out of the system, and back into the system are excluded, too.
- Let the function $connectors\_on\_client\_service\_paths : \mathbb{P}(P_M) \rightarrow \mathbb{P}(CN\_M)$ return the set of all connectors on a set of paths (without connectors having *Facades* or *ExternalComponents* as targets or that are of type *InMemoryConnector*).
- Finally, it selects the connectors that require authentication using the function $connectors\_that\_require\_authentication : \mathbb{P}(CN\_M) \rightarrow \mathbb{P}(CN\_M)$, defined above.

Based on the *client_service_path_connectors_requiring_authentication* function all metrics for measuring authentication on the paths from clients or UIs to system services are defined in much the same way as the metrics on backend authentication, defined in Section 6.2. The differences are that the components CP_M are used as inputs (i.e. they are all of the form $AEC : \mathbb{P}(CP\_M) \rightarrow [0, 1]$), the *client_service_path_connectors_requiring_authentication* function is used instead of the *distributed_backend_connectors_requiring_authentication* function, and the AEC_A metric considers API Keys as well, as API Keys coming from clients are deemed an acceptable practice (see explanation in Table 3). To avoid repetition, we do not explain each metric in Figure 5 in detail again.

6.4 Metrics for Observability (OBS) Decision

The metrics for the *OBS* decision are shown in Figure 6. The first metric **Observed System Services** $OSS : \mathbb{P}(CP_M) \rightarrow [0, 1]$ sets the observed system services in relation to all system services. Both numerator and denominator of the ratio rely on the $system\_services : \mathbb{P}(CP_M) \rightarrow \mathbb{P}(CP_M)$ function. A component is a **system service**, if it is not of the type *ExternalComponent* (i.e., no external services are system services), *MiddlewareService* (i.e., no middleware infrastructure services such as a Discovery Service), or *Facade* (i.e., no frontend services or gateways with the sole purpose of shielding the system from clients).

The detector *observed_by_dedicated_component* is *successful* for a component c, if a dedicated observer component is connected to c, or formally: $c \in \{t \in CP_M : (\exists\, o \in CP_M, l \in cp\_conn(t) : o \in observer(CP\_M) \vee (cn\_source(l) = t \wedge cn\_target(l) = o) \vee (cn\_source(l) = o \wedge cn\_target(l) = t))\ \}$. In this formula, the function $observer : \mathbb{P}(CP_M) \rightarrow \mathbb{P}(CP_M)$ determines the dedicated observer components in a model, i.e. components of the types *Logging*, *Monitoring*, or *Tracing*.

Common Function for Calculating the Connectors on Client/UI To Service Path Requiring Authentication

$$\begin{aligned}&\mathit{client\_service\_path\_connectors\_requiring\_authentication(cp)} = \\ &\quad \mathit{connectors\_that\_require\_authentication(connectors\_on\_client\_service\_paths(all\_paths\_from\_clients\_or\_uis\_to\_system\_services(cp)))}\end{aligned}$$

Authenticated Connectors on Client/UI to Service Path (AEC)

$$AEC(cp) = \frac{|\ \mathit{authenticated\_connectors(client\_service\_path\_connectors\_requiring\_authentication(cp)).successful}\ |}{|\ \mathit{client\_service\_path\_connectors\_requiring\_authentication(cp))}\ |}$$

Securely Authenticated Connectors on Client/UI to Service Path (AEC_S)

$$AEC\_S(cp) = \frac{|\ \mathit{securely\_authenticated\_connectors(client\_service\_path\_connectors\_requiring\_authentication(cp)).successful}\ |}{|\ \mathit{client\_service\_path\_connectors\_requiring\_authentication(cp))}\ |}$$

Connectors on Client/UI to Service Path Authenticated with API Keys (AEC_K)

$$AEC\_K(cp) = \frac{|\ \mathit{connectors\_authenticated\_with\_api\_keys(client\_service\_path\_connectors\_requiring\_authentication(cp)).successful}\ |}{|\ \mathit{client\_service\_path\_connectors\_requiring\_authentication(cp)}\ |}$$

Connectors on Client/UI to Service Path Authenticated with Plaintext Credentials (AEC_P)

$$AEC\_P(cp) = \frac{|\ \mathit{connectors\_authenticated\_with\_plaintext\_credentials(client\_service\_path\_connectors\_requiring\_authentication(cp)).successful}\ |}{|\ \mathit{client\_service\_path\_connectors\_requiring\_authentication(cp)}\ |}$$

Connectors on Client/UI to Service Path Authenticated Over a Secure Connection (AEC_C)

$$AEC\_C(cp) = \frac{|\ \mathit{authenticated\_connectors(secure\_connectors(client\_service\_path\_connectors\_requiring\_authentication(cp)).successful).successful}\ |}{|\ \mathit{client\_service\_path\_connectors\_requiring\_authentication(cp)}\ |}$$

Connectors on Client/UI to Service Path Authenticated Using a Secure Method, API Keys, or Transferred Over a Secure Connection (AEC_A)

$$\begin{aligned}&\mathit{authenticated\_connectors\_on\_client\_service\_path\_using\_secure\_method\_api\_keys\_or\_secure\_communication} = \{acp \in cp : \\ &\quad \mathit{securely\_authenticated\_connectors(client\_service\_path\_connectors\_requiring\_authentication(cp)).successful} \vee \\ &\quad \mathit{connectors\_authenticated\_with\_api\_keys(client\_service\_path\_connectors\_requiring\_authentication(cp)).successful} \vee \\ &\quad \mathit{authenticated\_connectors(secure\_connectors(client\_service\_path\_connectors\_requiring\_authentication(cp)).successful).successful}\}\end{aligned}$$

$$AEC\_A(cp) = \frac{|\ \mathit{authenticated\_connectors\_on\_client\_service\_path\_using\_secure\_method\_api\_keys\_or\_secure\_communication(cp).successful}\ |}{|\ \mathit{client\_service\_path\_connectors\_requiring\_authentication(cp)}\ |}$$

Fig. 5. Metrics Definitions for Authentication on Client/UI to Service Paths

The next metric **Observed Facade Components** $OFA : \mathbb{P}(CP_M) \rightarrow [0, 1]$ is similar, but it uses the $facades : \mathbb{P}(CP_M) \rightarrow \mathbb{P}(CP_M)$ function as its basis. A component is a **facade**, if it is of the type $Facade$, which is used directly for simple frontend services, or one of its subclasses such as $APIGateway$ or $BackendsforFrontendsGateway$

Finally, **Observed System Services and Facades** $OSF : \mathbb{P}(CP_M) \rightarrow [0, 1]$ is a simple integration of the two prior metrics by just using the union of the component sets returned by $system\_services$ and $facades$ as its basis.

7 ANALYSIS

In this section, we first report the results of our correlation analysis and then our ordinal logistic regression analysis. Table 5 shows an overview of the metric calculation results for all metrics defined in the previous section, computed with our detectors from the models automatically.

Observed System Services (OSS)

$$OSS(cp) = \frac{|\ \mathit{observed\_by\_dedicated\_component(system\_services(cp)).successful}\ |}{|\ \mathit{system\_services(cp)}\ |}$$

Observed Facade Components (OFA)

$$OFA(cp) = \frac{|\ \mathit{observed\_by\_dedicated\_component(facades(cp)).successful}\ |}{|\ \mathit{facades(cp)}\ |}$$

Observed System Services and Facades (OSF)

$$OSF(cp) = \frac{|\ \mathit{observed\_by\_dedicated\_component(system\_services(cp) \cup facades(cp)).successful}\ |}{|\ \mathit{system\_services(cp) \cup facades(cp)}\ |}$$

Fig. 6. Metrics Definitions for Observability

Metric	*AC0*	*AC1*	*AC2*	*BA0*	*BA1*	*BA2*	*CI0*	*CI1*	*CI2*	*CO0*	*CO1*	*CO2*	*EP0*	*EP1*	*EP2*	*ES0*	*ES1*	*ES2*	*OB0*	*OB1*	*OB2*	*PM0*	*PM1*	*PM2*	*RS0*	*RS1*	*RS2*	*TE0*	*TE1*	*TE2*
SCO		0.27	1.0		0.62	0.92	0.67	0.76	0.83		0.47	1.0		0.5	1.0		0.54	1.0		0.04	1.0	0.03	0.6	1.0		0.2	0.5		0.14	0.23
SCC			1.0			1.0	1.0	1.0	1.0		1.0	1.0		0.25	1.0		0.73	1.0			1.0		0.5	1.0			1.0			
SUC	NA	NA	NA	NA	NA	NA	NA		1.0		1.0	1.0	NA	1.0	1.0		1.0	1.0	NA	NA	NA	NA	NA	NA						1.0
SEC			1.0			1.0	1.0	0.8	1.0		1.0	1.0		0.5	1.0		0.79	1.0			1.0		0.5	1.0			0.86			0.2
SIC		0.33	1.0		0.68	0.91	0.64	0.75	0.78			1.0		0.5	1.0		0.39	1.0		0.06	1.0	0.03	0.61	1.0		0.23	0.4		0.17	0.24
AEI	0.67	1.0	1.0	0.88	0.95	1.0	0.36	1.0	1.0	0.67	0.9	1.0	0.5	0.83	1.0	0.74	1.0	1.0		1.0	1.0	0.53		1.0		1.0	1.0		0.35	0.32
AEI_S			1.0			0.09		0.62	1.0		0.7	0.55			1.0	0.26	0.83	1.0			1.0	0.06		0.97		0.54	0.84			0.24
AEI_K					0.16			0.38			0.2						0.17									0.46	0.16			
AEI_P	0.67	1.0		0.88	0.79	0.91	0.36			0.67		0.45	0.5	0.83		0.48				1.0		0.47		0.03					0.35	0.08
AEI_C		0.33	1.0		0.68	0.91		0.75	0.78			1.0		0.5	1.0		0.39	1.0		0.06	1.0			1.0		0.23	0.4			
AEI_A		0.33	1.0		0.68	1.0		1.0	1.0		0.7	1.0		0.5	1.0	0.26	1.0	1.0		0.06	1.0	0.06		1.0		0.77	1.0			0.24
AEC	1.0	1.0	1.0	1.0	0.8	1.0		1.0	1.0		0.8	1.0		0.67	1.0	1.0	1.0	1.0		1.0	1.0	1.0		1.0		1.0	1.0		0.83	0.67
AEC_S			1.0					0.14	1.0						1.0	0.6	0.78	1.0			1.0	0.4		1.0			0.38			
AEC_K					0.8			0.86			0.8						0.22									1.0	0.62			
AEC_P	1.0	1.0		1.0		1.0						1.0		0.67		0.4				1.0		0.6							0.83	0.67
AEC_C			1.0			1.0		1.0	1.0		0.6	1.0			1.0		0.67	1.0			1.0			1.0		0.42	1.0			
AEC_A			1.0		0.8	1.0		1.0	1.0		0.8	1.0			1.0	0.6	1.0	1.0			1.0	0.4		1.0		1.0	1.0			
OSS		0.5	1.0		0.8	1.0		0.8	1.0		0.75	0.75		0.67	1.0	1.0	0.8	1.0	0.89	0.78	0.67	1.0	0.67	0.33	0.71	0.86	1.0	0.17	0.5	0.83
OFA			1.0			1.0		0.5		NA	1.0			0.5	1.0		0.5	1.0	1.0		1.0	1.0	0.5	0.5				NA	NA	NA
OSF		0.33	1.0		0.67	1.0		0.71	0.71		0.8	0.6		0.6	1.0	0.71	0.71	1.0	0.9	0.7	0.7	1.0	0.6	0.4	0.62	0.75	0.88	0.17	0.5	0.83

Table 5. Metric Calculation Results for the Case Study Systems (empty cells indicate a value of 0.0)

In the regression analysis, the dependent outcome variables are the ground truth assessments for each decision, as described in Section 5 and summarized in Table 4. The metrics defined in Section 6 and summarized in Table 5 are used as the independent variables. The ground truth assessments are ordinal variables. All the independent variables are measured on a scale from 0.0 to 1.0.

7.1 Correlation Analysis

To compare the independent variables in their association to their dependent variables, we first computed the Spearman rank correlations [32] in Table 6. In the context of this analysis, we first inspected the plots of each independent vs. its dependent variable for inspecting their monotonic relationship. As can be seen in Table 6, most of the independent variables alone have already a very strong correlation ($\rho > 0.8$) or strong correlation ($\rho > 0.6$) with the dependent

Decision	Metric	Spearman's ρ	p-value
Secure Communication (SC)	*SCO*	0.9618466	2.2e-16
	SCC	0.9098859	3.264e-12
	SUC	0.7482699	0.000856
	SIC	0.898277	1.654e-11
	SEC	0.9345801	4.305e-14
Backend Authentication (BE_AE)	*AEI*	0.9143827	1.64e-12
	AEI_S	0.8130569	4.774e-08
	AEI_K	0.2080114	0.27
	AEI_P	-0.1830621	0.3329
	AEI_C	0.880331	1.435e-10
	AEI_A	0.8848047	8.66e-11
Authentication on Paths from Clients or UIs to System Services (CP_AE)	*AEC*	0.8624187	9.012e-10
	AEC_S	0.7665777	7.836e-07
	AEC_K	0.215354	0.2531
	AEC_P	-0.07136348	0.7079
	AEC_C	0.8735141	2.983e-10
	AEC_A	0.8981329	1.686e-11
Observability (OBS)	*OSS*	0.9057307	5.976e-12
	OFA	0.6852825	0.000112
	OSF	0.952798	4.997e-16

Table 6. Spearman Rank Correlation Analysis Results

variable, with very low p-values (< 0.05), signaling high statistical significance. This indicates that those independent variables are well chosen, in the sense that they are highly associated with the dependent variable.

There are some notable discussion points in the correlation analysis. Firstly, the variables on *API Keys*-based authentication (AEI_K and AEC_K) and *Plaintext-based Authentication* (AEI_P and AEC_P) are neither highly correlated nor are the results statistically significant. Please note that this is not surprising: All other hypothesized variables in their ADDs have a plausible relation to the dependent variable that implies a high association with the dependent variable; this is not the case for these variables only measuring the fraction of one specific authentication method (API Keys or plaintext). But as the disambiguation of API Key or plaintext based methods is plausibly important to detect all ADD options fully and those aspects are not covered well in any of the other variables, we decided to include these variables nonetheless in our regression analysis, in the hope that they could introduce aspects in the models not covered well by the other variables. Indeed, in the regression analysis below, all the best models found for the two respective ADDs (BE_AE, CP_AE) use at least one of these variables. However, we also decided to never use them alone in regression models, but only with at least one other variable with a strong correlation.

Secondly, as many of our variables have a strong or very strong correlation with high significance, there was no clear indication of which variables to use in the regression analysis below. We thus decided to try all reasonable combinations of our dependent variables to find the models that best predict the dependent variable.

7.2 Ordinal Logistic Regression Analysis

The objective of the ordinal logistic regression analysis is to predict the likelihood of the dependent outcome variable for each of the decisions by using the relevant metrics for each decision. Ordinal logistic regression is the recommended regression model in the case of ordinal response variables [13]. To enable comparison of the resulting regression models, we calculated many possible models for each ADD, and report the three best-performing models for each (see Columns 3-5 of Table 7).

Each resulting regression model consists of a *baseline intercept* and the independent variables multiplied by *coefficients*. There are different intercepts for each of the value transitions of the dependent variable ($\geq$ $[-]$: *Insufficient* *support in a few places*; ≥ [~]: *Insufficient support in many places*, ≥ [+]: *Acceptable support*, ≥ [++]: *Very good support*), while the coefficients reflect the impact of each independent variable on the outcome. For example, a positive coefficient, such as +5, indicates a corresponding five-fold increase in the dependent variable for each unit of increase in the independent variable; conversely, a coefficient of -30 would indicate a thirty-fold decrease.

The statistical significance of each regression model is assessed by the p-value; the smaller the p-value, the stronger the model is. A p-value smaller than 0.05 is generally considered statistically significant. In Table 7, we report the p-values for the resulting models, which in all cases are very low, indicating that the sets of metrics we have defined are able to predict the ground truth assessment with high statistical significance.

Frequently, the C-index (which is also called concordance index and is equivalent to the area under the Receiver Operating Characteristic (ROC) curve) is reported in the statistical literature as a measure of the predictive power of ordinal regression models [1]. A C-index of 0.5 indicates random splitting, whereas a C-index of 1 indicates perfect prediction. Harrel [13] suggests bootstrapping as a method for obtaining nearly unbiased estimates of a model's future performance based on re-sampling. It is a powerful model validation technique that supports estimating predictive accuracy without holding back data from the model development process [13]. This feature is important for our study, as our data set contains only 30 models. We used lrm's *validate* function to perform bootstrapping and calculated the bias-corrected C-index in addition to the original C-index. The C-indexes, reported in Table 7, are all larger than 0.9, which indicates that the models are good enough for predicting the outcomes of individuals.

The Brier score is another commonly reported accuracy measure [17]. Even though somewhat less interpretable than the C-index, the Brier score overcomes some issues of the C-index, as it reflects both calibration and discrimination of a model [26]. In this sense, it can be better suited to assess overall model performance and compare models [52]. The value of the Brier score is always between 0.0 and 1.0. A score of 0 represents perfect accuracy, and a score of 1 represents perfect inaccuracy. Again, we do not only present the original Brier scores but also the bias-corrected scores after bootstrapping. Except for one model of CP_AE, all bias-corrected scores are below 0.1 (and the one model above 0.1 has also a very low Brier score of 0.1177); thus for each ADD, we were able to find models that are comparatively good with a very low Brier score.

We used lrm's function *pentrace* to assist in the selection of penalty factors for fitting regression models using penalized maximum likelihood estimation (see [13]). In the reported models, we generally used a simple penalty of 1 and a non-linear penalty of 5. Please note that the penalized regression models are offering slightly improved performance compared to non-penalized models.

The reported models in Table 7 do not always use the full set of our hypothesized metrics. We tried many combinations of independent variables and penalties. In the table, for each decision, we show the three models with the highest bias-corrected C-index values, which we found. It can be observed that we were able to reach for both original and bootstrapped C-indexes values above or very close to 0.9, and at the same time very low Brier scores. We can also observe that all our hypothesized independent variables are used in at least one of the models that are reported. That is, all hypothesized variables are relevant in our predictions, but in no decision all of them are needed to make a prediction with a high C-index, low Brier score, and low p-value.

8 INTEGRATION IN INDUSTRIAL TOOLS

The work presented in this article has so far found adoption in two industrial tools. In this section, we illustrate these early industry adoptions, which we plan to extend in our future work.

Decision	**Model Parameter/Measure**	**Regression Model 1**	**Regression Model 2**	**Regression Model 3**
Secure Communication (SC)	*Intercept: ≥ [−]*	-0.7157	-0.7437	-0.7590
	Intercept: ≥ [∼]	-3.4324	-4.8399	-4.6476
	Intercept: ≥ [+]	-6.5069	-8.7636	-7.9085
	Intercept: ≥ [++]	-8.2981	-11.1942	-10.2721
	Metric Coefficient: SCO	—	4.1212	4.3808
	Metric Coefficient: SCC	3.3725	2.0746	—
	Metric Coefficient: SUC	2.3068	—	—
	Metric Coefficient: SIC	3.3617	2.4653	2.4620
	Metric Coefficient: SEC	—	3.5020	4.4002
	Model p-value:	1.196800e-07	7.549517e-15	7.438494e-15
	C-Index (original):	0.9801980	0.9943662	0.9915493
	C-Index (bootstrapped, bias-corrected):	0.8757673	0.9239085	0.9270775
	Brier-Score (original):	0.0184	0.0089	0.0116
	Brier-Score (bootstrapped, bias-corrected):	0.0258	0.0132	0.0172
Backend Authentication (BE_AE)	*Intercept: ≥ [−]*	-0.3781	-0.8102	-0.1915
	Intercept: ≥ [∼]	-5.4616	-5.5538	-3.8680
	Intercept: ≥ [+]	-8.1466	-9.4906	-8.0389
	Intercept: ≥ [++]	-9.1059	-10.5578	-9.2688
	Metric Coefficient: AEI	—	7.5049	—
	Metric Coefficient: AEI_S	5.7363	—	6.1902
	Metric Coefficient: AEI_K	-1.8124	-3.3980	1.3726
	Metric Coefficient: AEI_P	3.4443	-2.2369	3.0572
	Metric Coefficient: AEI_C	—	3.5866	3.8202
	Metric Coefficient: AEI_A	3.6733	—	—
	Model p-value:	6.091905e-12	5.488943e-13	2.343792e-12
	C-Index (original):	0.9910979	0.9910979	0.9910979
	C-Index (bootstrapped, bias-corrected):	0.9175964	0.9231602	0.9375445
	Brier-Score (original):	0.0507	0.0392	0.0479
	Brier-Score (bootstrapped, bias-corrected):	0.0618	0.0475	0.0596
Authentication on Paths from Clients or UIs to System Services (CP_AE)	*Intercept: ≥ [−]*	-0.4692	0.1161	-0.4065
	Intercept: ≥ [∼]	-1.4394	-0.6236	-1.4118
	Intercept: ≥ [+]	-4.3053	-3.0822	-4.8040
	Intercept: ≥ [++]	-8.0709	-5.4270	-7.7217
	Metric Coefficient: AEC	—	—	—
	Metric Coefficient: AEC_S	4.6584	—	—
	Metric Coefficient: AEC_K	—	0.2793	-1.1230
	Metric Coefficient: AEC_P	1.9127	0.8766	1.3876
	Metric Coefficient: AEC_C	4.9044	5.2875	3.3351
	Metric Coefficient: AEC_A	—	—	4.4675
	Model p-value:	1.052047e-12	1.740458e-08	9.255219e-11
	C-Index (original):	0.9730878	0.9036827	0.9660057
	C-Index (bootstrapped, bias-corrected):	0.930085	0.9053824	0.9435552
	Brier-Score (original):	0.0804	0.1118	0.0632
	Brier-Score (bootstrapped, bias-corrected):	0.0862	0.1177	0.0679
Observability (OBS)	*Intercept: ≥ [−]*	-1.9703	-2.2049	-1.8885
	Intercept: ≥ [∼]	-8.4478	-6.6498	-6.0532
	Intercept: ≥ [+]	-10.0269	-8.0875	-7.4003
	Intercept: ≥ [++]	-12.1232	-10.5628	-9.9765
	Metric Coefficient: OSS	4.9827	8.2015	—
	Metric Coefficient: OFA	—	3.1110	1.8509
	Metric Coefficient: OSF	7.5947	—	8.9371
	Model p-value:	1.709743e-14	1.870948e-12	5.463852e-12
	C-Index (original):	0.9717514	0.9887640	0.9700375
	C-Index (bootstrapped, bias-corrected):	0.9449153	0.9514326	0.9451966
	Brier-Score (original):	0.0568	0.0713	0.0749
	Brier-Score (bootstrapped, bias-corrected):	0.0610	0.0800	0.0822

Table 7. Regression Analysis Results ("—" means: not used in the model)

8.1 Integration into a Cyber-Threat Resilience Assessment Tool

The company EU-VRi, represented by three co-authors of this article, is currently developing an extension of its existing cyber-threat resilience assessment tool. The tool aims to enable developers, architects, or assessors to assess and optimize the security threat resilience (risk analysis, preparedness, absorption, recovery, adaption to adverse conditions, stresses, attacks, or compromises) of a system. To reach this goal the tool calculates a cyber-threat resilience level index for a system, as a composite, multi-level indicator. It is supposed to be based on existing industry guidelines, such as the ones summarized in Section 2.1. The analysis of these guidelines performed by the last three co-authors of this article, explained in Section 2.1, was performed initially with the purpose to create manually verifiable metrics and indicators for this tool.

Our approach can be used as a building block for the cyber-threat resilience assessment tool to automatically calculate metrics for the software architectural parts of the composite resilience level index. The tool provides an API to integrate building blocks such as the metrics provided by our approach and use the composite indicator-based approach to calculate an aggregated score for resilience assessments. Further, the tool supports before/after analysis, multi-assessment monitoring over time, and decision support based on sensitivity analysis. As the tool is expected to perform automated assessments, we have designed our metrics and models to be integrated within this commercial tool as an automated component.

8.2 Integration into an Incremental and Continuous Security Certification Tool

The company SEARCH-LAB[7] is currently working on a new industrial tool for incremental and continuous security certification. This tool addresses the following problem: The rapid development in today's microservice systems, with continuous integration and delivery being the expected norm rather than the exception, poses a substantial challenge to the certification of software security. Existing security certification schemes focus on certifying that software projects are following certain security best practices, and they largely focus on the software development process. Security concerns, such as the ADDs covered in this article, are usually checked manually, just as in our ground truth analysis, typically even without the support of (generated) architectural models, but instead directly in the source code.

In microservice projects that require certification before a release, this is highly problematic. In such projects, development processes are continuously adapted by developers, and thus process-based schemes are hard to apply. Architectural concerns are scattered throughout the source code and hard to find just by inspecting the source code. Changes and releases are expected to be happening with a high frequency, meaning that organizations might not have the necessary resources to perform certifications for each release or it would result in significant costs to maintain the certified status by paying independent evaluators due to the rapid release cycles.

The incremental and continuous security certification tool (called DeltAICert), being developed at SEARCH-LAB, takes a different view by focusing on artifacts rather than processes. It aims to certify only the required delta, by comparing evidence (source, code, logs, models, test results) and indicators (metrics from other analysis tools) from previous evaluations. For changed artifacts and their dependencies, lightweight and largely automated techniques for the continuous and incremental security evaluation shall be applied to enable the continuous re-certification of the developed software. The DeltAICert tool is able to make automated evaluations for the AssureMOSS certification scheme, which was also developed in the AssureMOSS project[8]. In the context of this tool development, our approach is used as a building block to automate the (re)certification of the practices embodied in our ADDs based on the

[7] https://www.search-lab.hu/
[8] https://assuremoss.eu/

architectural models and the derived metrics. For this purpose, two industrial security experts from SEARCH-LAB have reviewed our models, metrics, and code, and designed an integration into SEARCH-LAB's novel continuous certification scheme.

9 DISCUSSION

In this section, we first discuss our lessons learned with regard to the posed research questions and then discuss potential threats to validity.

9.1 Discussion of Research Questions

We initially performed a number of data collection and analysis steps, including a study of microservice architecture security guidelines, the gray literature, and the scientific literature, plus a manual inspection of the code of 10 open source system and in-depth reviews by five industrial security experts in order to find four industrial ADDs, each covering multiple security tactics. This enabled us to design our ADD model, reported in Section 4, which is a prerequisite to study our research questions. To answer **RQ1** and **RQ2**, we proposed a set of generic, technology-independent metrics for each of the ADDs. Each of the decision options (aka security tactics) corresponds at least one metric, but some aspects were covered by multiple metrics which we hypothesized could provide good predictions later on. We aimed to objectively assess for each model how well the security tactics are supported for establishing the ground truth. For this, the three industrial security experts on the author team and the team in their company created (independently of the work reported in this article, before they got involved in this article) a recommendation which serves as informal guidance for security experts to manually judge systems such as those in our models. The academic authors used this recommendation to asses each model manually, and then another independent review by the five industrial security experts was performed. We believe that these steps make it highly likely that our ADDs represent well-established practices in the industry, and that our ground truth assessment is very likely within the range of manual assessments to be expected from industrial security experts.

We formulated the metrics to numerically assess a security tactic's implementation in each model, as well as automated detectors enabling the automatic computation of the models. In our previous work [36], we have developed an approach for automatically extracting the needed models from a system's source code with modest manual specification effort. As this approach was applied to one of the larger models in our model data set which contained a highly polyglot microservice architecture (the Case RS0), we can assess that it is highly likely possible to extract all our models automatically from the source code with a small to medium specification effort in the same way. As a consequence, all stages in our work can be fully automated, once such an extraction specification has been realized. This means, one can apply our approach e.g. in the context of a continuous delivery pipeline on each single code commit, which was one of the original motivations for our work (see Section 1). That is, our approach reported in Sections 4-6, together with the source code extraction technique reported in our earlier work [36] if manual modeling should be avoided, provide our answer to **RQ1**.

We performed an ordinal regression analysis using our metrics as independent variables to predict the ground truth assessment. Our results show that every set of decision-related metrics can predict with high statistical significance and high accuracy our objectively evaluated assessment, using a number of regression models (three regression models per ADD were reported). We also assessed the correlation of the individual independent variables to the dependent variable, and selected on models in which at least one variable with a strong or very strong correlation was present. This suggests that automatic metrics-based assessment of a system's conformance to security tactics used as options in

ADDs is possible with a high degree of confidence. The p-values, C-index values, and Brier scores provided in Section 7 provide concrete measurements showing that the metrics are accurate measures for assessing ADD conformance in our model data set, which answers **RQ2**.

Regarding **RQ3**, we consider that existing modeling practices can be easily mapped to our microservice meta-model (derived from [61]). Rather lightweight extensions for security tactics markup have been made. Both have been formally specified in Section 6. More specifically, for completing the modeling of our model data set, we needed to introduce 16 component types and 28 connector types, such as the *OAuth2 Server* component stereotype or *Token-based Authorization* connector stereotype (both shown as examples in Figure 2). In addition, we needed to introduce types for basic microservice modeling. In particular 25 component types and 38 connector types are offered, ranging from general notions such as the *Service* component type, to very technology-specific classes such as the *RESTful HTTP* connector, which is a subclass of *Service Connector*. Our study shows that in each ADD context and the proposed metrics, only a small subset of the meta-model is required. We confirmed for each of the stereotypes that they can be extracted from the source code with the methods reported in our earlier work (i.e. [36]).

9.2 Threats to Validity

We deliberately relied on third-party systems as the basis for our study to increase internal validity, thus avoiding bias in system composition and structure. It is possible that our search procedures introduced some kind of unconscious exclusion of certain sources; we mitigated this by assembling an author team with many years of experience in the field (including the substantial industry experiences of three industrial experts as co-authors), and performing very general and broad searches. Given that our search was not exhaustive, and that most of the systems we found were made for demonstration purposes, i.e. relatively modestly sized, this means that some potential architecture elements were not included in our meta-model. In addition, this raises a possible threat to external validity of generalization to other, more complex, systems. We nevertheless feel confident that the systems documented are a representative cross-cut of current practices in the field, as the points of variance between them were limited and well attested in the literature.

It might however be the case that for substantially different kinds of systems, e.g. systems from other domains or substantially larger systems, the expert judgement for the ground truth would differ. Then it would be necessary to re-run our regression analysis with data from a few such systems in order to calibrate the prediction models to the changed circumstances.

To avoid threats with regard to the generalizability of our approach, we limited our scope to microservice-based systems, even though some aspects of our approach are likely applicable to other kinds of distributed systems than microservice-based systems as well. For instance, the security tactics and ADDs, as well as some of the related metrics, are likely applicable for many other kinds of systems as well. Reasons for the limitation to microservices are: Firstly, many aspects of the practices we investigated are specific to microservices and security recommendations for them (such as [34]). For instance, our investigation of observability practices is based on the concrete technologies currently being employed in the microservice field. Secondly, our evaluation is purely based on microservice-based systems. Thirdly, our meta-model and reconstruction approach focuses on microservice-specific abstractions, as explained in Section 2.6.

Another potential threat is the fact that the variant systems were derived by the author team. This is mitigated by the independent review of the variants by the industrial reviewers. Also, this was done according to existing practices documented in literature. We carefully made sure only to change specific aspects in a variant and keep all other aspects stable. That is, while the variants do not represent actual systems, they are reasonable evolutions of the original designs (e.g., representing possible architectural refactoring steps or deteriorations during system evolution). As a major goal of our approach was to be able to find issues during system evolution (e.g. in the context of continuous delivery), such as a mistake made in a refactoring step or deteriorations during system evolution, the modeling of variants actually introduces this aspect into our approach and our statistical analysis.

An aspect that might limit the validity of the reviews by our industrial co-authors is the shared context they have, as they work in the same company. We mitigated this threat firstly by involving two other industrial reviewers from another company. Secondly, we mitigated it by extensive reviews by the academic authors of this article, which all have substantial experiences in industrial software development and software security. Fourthly, please note that the company of the industrial co-authors consults other companies; that is, each of those industrial co-authors has experience from many different industry projects (across sizes, domains, software development methods). Yet another mitigation of the threat is that all practices and ADDs are directly derived from widely-used industry guidelines, which substantially limits the chance that only practices relevant for a particular company have been considered.

The modeling process is also considered as a source of an internal validity threat. The models of the systems were repeatedly and independently cross-checked by the author team that has considerable experience in similar methods, but the possibility of some interpretative bias remains: other researchers might have coded or modeled differently, leading to different models. As a mitigation, we also offer the whole models and the code as open access artifacts for review and reproducibility. Since we aimed only to find one model that is able to specify all observed phenomena, and this was achieved, we consider this threat not to be a major issue for our study. As mentioned, it is easily possible to change the modeling slightly and re-run our regression analysis to calibrate the prediction models to a different modeling approach.

The ground truth assessment might also be subject to different interpretations by different practitioners. For this purpose, we deliberately chose only a five-step ordinal scale, and given that the ground truth evaluation for each decision is fairly straightforward and based on industrial best practices and recommendations, we do not consider our interpretation controversial. Likewise, the individual metrics used to evaluate the presence of each security tactic were deliberately kept as simple as possible, so as to avoid false positives and enable a technology-independent assessment. As stated previously, generalization to more complex systems might not be possible without modification. But we consider that the basic approach taken when defining the metrics is validated by the success of the regression models.

Please note that we do not claim completeness of the metrics we present in this article. They are only complete in the sense that they cover all options of the ADDs they address. As discussed in Section 8, they are meant to be used as automatable building blocks in tools that cover a broader range of metrics (including e.g. organizational metrics that cannot be automatically extracted and calculated in the same way).

10 RELATED WORK

In this section, we compare our approach to related works. First, we analyzed other works on best practices and patterns in related areas, as our approach is based on them – as options in the ADDs. Our approach is a microservice architecture modeling and reconstruction-based approach, and thus we compare to existing works in this field secondly. Thirdly, we focus on security features in this field, for which we discuss related reconstruction and analysis approaches next. Finally, our analysis is based on metrics and indicators, and thus we also compare to other approaches providing security metrics and indicators.

10.1 Related Works on Best Practices and Patterns

Much research has been conducted in collecting and systematizing microservice patterns. For instance, Richardson [48] collected microservice patterns related to major design and architectural practices. Zimmermann et al. [65] introduced microservice API related patterns. Skowronski [50] collected best practices for event-driven microservice architectures. Microservice fundamentals and best practices are also discussed by Fowler and Lewis [29], and are summarized in a mapping study by Pahl and Jamshidi [38]. Taibi and Lenarduzzi [53] studied microservice bad smells, i.e. practices that should be avoided, which would correspond to metrics violations in our work. In this article, we use such guidance as the foundation for microservice architecture modeling.

Likewise, attempts to define security patterns have been made [20, 49]. In the security field, recommendations by industry or standards organizations (such as the OWASP Top 10[9] or the ISO 27002 standard[10]) are widely used. For microservices, microservice-specific recommendations by industry organizations such as those of NIST [34], OWASP [37], or the Cloud Security Alliance [8] are proposed, which represent aggregations of existing industry best practices on a broad level. In our work, we used such guidance for selecting the security practices we investigate.

10.2 Related Works on Microservice Architecture Modeling and Reconstruction

Microservices [29, 33, 48] are, among many other things, a way to decompose an architecture based on services [62]. This is an area which has been studied intensively in recent years (see e.g. [38, 40, 63]). An important part of our approach is to model architecture component models and reconstruct such models via static code analysis, and analyze the resulting models later on. This part of our approach is related to architecture reconstruction and related modeling approaches. Architecture reconstruction focuses on automatically or semi-automatically producing architecture abstractions from the source code [10, 30, 31]. While such generic approaches fall short in being able to address the specific characteristics of microservice systems, a number of more specific modeling and reconstruction approaches have been proposed to address this gap.

Granchelli et al. [18] provide one of the few existing microservice-specific architecture reconstruction approaches. It statically analyses Docker and Docker Compose files for names and ports, and then the Docker containers and network bridges dynamically, to reconstruct the deployed microservices from the system's communication logs. Alshuqayran et al. [2] present an approach that is intended as a groundwork for architecture reconstruction of microservices. From the analysis of 8 open source projects, the approach derived a meta-model and possible mapping rules for microservices.

Vianden et al. [57] report on a study of a microservice-based reference architecture as a starting point for enterprise measurement infrastructures. This can be seen as an alternative to reconstruction, but it requires manual maintenance of the architecture in relation to the reference architecture. Rademacher et al. [46] suggest addressing the polyglot nature of microservices using an aspect-oriented modeling approach.

Our own prior work [36], used for reconstruction purposes in this article, aims to address issues of reconstructing microservice systems when facing polyglot programming, persistence, and technologies. As a consequence, our microservice architecture model explained in Section 6 is richer in this regard than the models used in the other related works. In this article, we use this approach as a foundation for extracting security tactics related models from the source code and perform automated metrics-based analysis of the resulting models. To the best of our knowledge, our approach is the first to consider microservice system architectures and their security aspects in a modeling-/reconstruction-based approach.

[9] https://owasp.org/Top10/

[10] https://www.iso27001security.com/html/27002.html

10.3 Related Works on Reconstruction and Analysis of Security Features

In addition to microservice architecture reconstruction, a second direction in which this article is related to architecture reconstruction approaches is the reconstruction and analysis of security features. This has been addressed before in a number of existing approaches. Sohr and Berger [51] use the Bauhaus tool to build security views. In particular, they extract a resource flow graph from Java code (J2EE or Android). The authors build security views that represent the RBAC permissions in the application. In a later work, Bunke and Sohr [6] extend the previous approach based on RFGs to extract security patterns from the code and check their correct implementation.

Concerning Android Java applications, Hamad et al. [22] proposed an approach to automatically extract the architectural design of an application starting from the APK (bytecode). The DelDroid tool is able to extract the application components (activities, services, broadcast receivers, content providers), the explicit and implicit communication flows (via intents), as well the granted permissions to each component. The tool also analyzes the code and computes the permissions that are actually used.

Vanciu and Abi-Antoun [56] proposed the Scoria approach. The approach is based on the extraction from Java code of an Ownership Object Graph (OOG). Objects are hierarchically organized in a tree according to ownership domains. Dataflow edges are automatically added on top of the hierarchy and represent information flows (i.e., read and write), as well as point-to and inheritance relationships among the objects. The resulting graph is called a Sec Graph and can be further enriched by software architects with security-related properties. The architect can run queries on the graph to identify security issues, such as insecure communication.

Peldszus et al. [42] defines an approach for the model-based security analysis of Java applications. The analysis is performed on a SecPL model, which is an extension of UMLsec. The SecPL model can be reverse engineered from the code, provided that the developers have enriched the code with SecPL annotations, which carry security information and properties.

Jasser [25] categorizes a number of ADDs for security. These security constraints are formalized as LTL formulas, which predicate over architectural concepts (e.g., components). The approach entails that the architect provides a manual mapping between the architectural concepts and the code (a DSL is provided to this aim). Finally, the tool checks the compliance between the security rules and the code.

Bauer et al. [4] focus on formally validating the security specification (as UMLsec diagrams) and then monitoring the system at run-time in order to check the compliance of the implementation with the security specification. To enable the latter step, the architect has to manually provide a mapping between code elements and model elements.

Peldszus et al. [43] present a semi-automated approach to map model elements from a SecDFD diagram to code elements (Java methods and types). The tool provides some suggestions that the architect can approve or discard. The architect can also suggest their own mappings. In an iterative way, the tool suggests new mappings by prioritizing the heuristics that have been 'approved' by the human. The heuristics work on an intermediate representation of the code called Program Model, which fuses concepts from the class diagram, the call graph, and the dependency graph.

The approaches discussed above have in common with our approach that they build a security-specific model from the source code as a foundation for later analysis. In contrast to our approach, they do not support typical microservice architecture characteristics in their models, such as various possible distributed systems invocations used throughout microservice systems, the link to continuous delivery approaches, or the use of polyglot programming, persistence, and technologies. This adds a considerable layer of complexity to the system models under investigation in this article, compared to those earlier works. In consequence, our proposed metrics and detectors need to consider the extra elements in the software architecture, too.

10.4 Related Works on Security Metrics and Indicators

As microservice architectures rely on a level of assurance between the actors involved for engaging in interactions, the prospect of quantifying that level of assurance using security-relevant indicators is attractive. These indicators can enable organizations to assess architecture security [27]. Security metrics are then needed to understand the current state of security, and possibly improve that state [41]. While several organizations including Microsoft [23] and OWASP [59] propose processes and checklists for building secure architectures, there are very few tools that can automate these processes for tailor-made solutions due to the dynamics in and heterogeneity of the systems [39]. Our approach is to the best of our knowledge the first that supports automated security assessment based on an empirical investigation of existing (best) practices in the field of microservice architectures.

Ramos et al. [47] conducted a detailed review of the main existing model-based quantitative security metrics with a focus on network security metrics, analyzing their advantages and disadvantages in their use. Model-based security metrics use techniques to describe a system in terms of an abstract model that captures the necessary attributes of interest, based on the assumptions of the attacker and the system behavior. For example, Attack Graphs (AG) are a model commonly used to quantify network security. It uses the causal relationships between vulnerabilities which allows quantification of the likelihood of potential multi-step attacks that combine multiple vulnerabilities [58]. Noel et al. [35] describe a set of metrics for measuring network-wide cybersecurity risk based on a model of multi-step attack vulnerability with AGs. Their system for computing security metrics from vulnerability-based network AGs used the data imported from sources that are commonly deployed within enterprise networks, such as vulnerability scanners and firewall configuration files. They used a Topological Vulnerability Analysis (TVA) tool [24], Cauldron, which analyses network attack vulnerability from scanning tools and other data sources. The tool correlates cyber-vulnerabilities and environmental metadata and applies network access policy (firewall) rules.

Such general security metrics and indicators are a foundation for our work. But as none of them considers the specifics of microservice architectures yet, they cannot at all or without substantial adaptation be applied to our research problems. For this reason, we decided to design metrics based on existing recommendations (such as NIST [34], OWASP [37], or the Cloud Security Alliance [8]) specifically for microservice-based systems.

10.5 Related Works on Microservice Metrics and Indicators

Many of the works on service metrics today are focused on runtime properties (see e.g. [44]). A number of studies have used metrics to assess microservice-based software architectures, e.g. [5, 40, 60], but each is focused on narrow sets of architecture-relevant tenets (e.g. loose coupling), and no general approach for an assessment across different microservice tenets exists. Pautasso and Wilde [40] propose a composite, facet-based metric for the assessment of loose coupling in service-oriented systems. Zdun et al. [60] study the independent deployment of microservices by defining metrics to assess architecture conformance to microservice patterns, focused on two aspects: independent deployment and shared dependencies of services. Bogner et al. [5] propose a maintainability quality model which combines eleven easily extracted code metrics into a broader quality assessment. Engel et al. [11] also propose a method of using real-time system communication traces to extract metrics on conformance to recommended microservice design principles such as loose coupling and small service size. Our work broadly follows the same approach, but does not focus on specific tenets only and adds notions for measuring the use of security tactics in security-related ADDs.

Once metrics can be checked automatically, our approach can be classified as a metrics-based, microservice-specific approach for software architecture conformance checking. In general, approaches for architecture conformance checking are often based on automated extraction techniques [19, 55]. Conformance to architecture patterns [19, 21] or other kinds of architectural rules [55] can often be checked by such approaches. Techniques that are based on a broad set of microservice-related metrics to cover multiple microservice tenets and security do not yet exist.

While a couple of works specifically focus on microservice security metrics, again many of those focus on runtime-related or non-architectural aspects [3, 12, 28, 54]. Chondamrongkul et al.[7] present an early approach for automatically investigating certain security flaws in a component and connector architecture, such as man-in-the-middle or denial-of-service attacks. As it is unclear why, if this information is modeled, the flaw is not fixed instead, in this article, we do rather not model potential flaws, but security tactics, which seem better suited for modeling and analysis at the architectural level. Also, our work is based on empirical data, whereas Chondamrongkul et al.'s work only uses modeling examples.

11 CONCLUSION

In this article we have presented a novel approach for measuring conformance to ADDs on microservice security via metrics. Our approach (to answer **RQ1**) was to first to systematically examine the ADDs and security tactics in the areas of interest, here: secure communication, identity management, and observability. Next, we proposed relatively simple metrics evaluating at least each major decision option. These can automatically be calculated using model-based detectors used as functions in the formally defined metrics. To answer **RQ2**, our statistical analysis in Section 7 shows that for each ADD multiple regression models can be found with high statistical significance and accuracy. For each dependent variable, we found a number of metrics that have a strong or very strong correlation to it. These metrics are used in each of the selected the regression models. Within the scope of the case study system models (i.e., similar modeling abstractions, size, complexity, etc.), the regression models should provide good indicators for the conformance to the investigated ADDs on microservice security. Please note that in other scopes (e.g., more complex or much larger system), likely a recalibration of the models is necessary. Regarding **RQ3**, we consider that existing modeling practices can be easily mapped to our microservice meta-model (derived from [61]). Rather lightweight extensions for security tactics markup have been made. Both have been formally specified in Section 6. We confirmed for each of the stereotypes that they can be extracted from the source code with the methods from our earlier work (i.e. [36]).

As future work, we plan to apply the research approach followed in this article to other views such as deployment or behavioral views. Further, we plan to analyze systems at runtime to augment the static analysis techniques developed in this article. We plan to extend and improve the integration of our approach into the two commercial tools, summarized in Section 8.

12 ACKNOWLEDGMENTS

Our work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952647 (AssureMOSS project). This work was supported by: FWF (Austrian Science Fund) project API-ACE: I 4268; FWF (Austrian Science Fund) project IAC^2: I 4731-N.

We thank the two experts Gergely Eberhardt and Ákos Milánkovich from SEARCH-LAB for their review of our models, metrics, and code, and work on the integration with the continuous certification scheme explained in Section 8.2.

REFERENCES

[1] Antti Airola, Tapio Pahikkala, Willem Waegeman, Bernard De Baets, and Tapio Salakoski. 2011. An experimental comparison of cross-validation techniques for estimating the area under the ROC curve. *Computational Statistics & Data Analysis* 55, 4 (2011), 1828–1844.

[2] N. Alshuqayran, N. Ali, and R. Evans. 2018. Towards Micro Service Architecture Recovery: An Empirical Study. In *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, Washington, DC, USA, 47–4709. https://doi.org/10.1109/ICSA.2018.00014

[3] Alberto Avritzer. 2020. Challenges and Approaches for the Assessment of Micro-Service Architecture Deployment Alternatives in DevOps: A tutorial presented at ICSA 2020. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, Washington, DC, USA, 1–2.

[4] Andreas Bauer, Jan Jürjens, and Yijun Yu. 2011. Run-Time Security Traceability for Evolving Systems. *Comput. J.* 54 (2011), 58–87. Issue 1.

[5] Justus Bogner, Stefan Wagner, and Alfred Zimmermann. 2017. Towards a Practical Maintainability Quality Model for Service-and Microservice-Based Systems. In *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings* (Canterbury, United Kingdom) *(ECSA '17)*. Association for Computing Machinery, New York, NY, USA, 195–198. https://doi.org/10.1145/3129790.3129816

[6] Michaela Bunke and Karsten Sohr. 2011. An Architecture-Centric Approach to Detecting Security Patterns in Software. In *Engineering Secure Software and Systems*, Úlfar Erlingsson, Roel Wieringa, and Nicola Zannone (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 156–166.

[7] Nacha Chondamrongkul, Jing Sun, and Ian Warren. 2020. Automated security analysis for microservice architecture. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, Washington, DC, USA, 79–82.

[8] Cloud Security Alliance. 2020. Best Practices in Implementing a Secure Microservices Architecture. https://cloudsecurityalliance.org/artifacts/best-practices-in-implementing-a-secure-microservices-architecture/.

[9] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens. 2010. Flexible architecture conformance assessment with ConQAT. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 2. IEEE, Washington, DC, USA, 247–250. https://doi.org/10.1145/1810295.1810343

[10] S. Ducasse and D. Pollet. 2009. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering* 35, 4 (Jul-Aug 2009), 573–591.

[11] Thomas Engel, Melanie Langermeier, Bernhard Bauer, and Alexander Hofmann. 2018. Evaluation of Microservice Architectures: A Metric and Tool-Based Approach. In *Information Systems in the Big Data Era*, Jan Mendling and Haralambos Mouratidis (Eds.). Springer International Publishing, Cham, 74–89.

[12] José Flora. 2020. Improving the Security of Microservice Systems by Detecting and Tolerating Intrusions. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, Washington, DC, USA, 131–134.

[13] Jr. Frank E. Harrell. 2015. *Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis* (2nd ed.). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-319-19425-7

[14] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. 2013. A Comparative Analysis of Software Architecture Recovery Techniques. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Silicon Valley, CA, USA) *(ASE'13)*. IEEE Press, Piscataway, NJ, USA, 486–496.

[15] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Inf. Softw. Technol.* 106 (2019), 101–121. https://doi.org/10.1016/j.infsof.2018.09.006

[16] Vahid Garousi, Michael Felderer, Mika V. Mäntylä, and Austen Rainer. 2020. *Benefitting from the Grey Literature in Software Engineering Research.* Springer International Publishing, Cham, 385–413.

[17] Thomas A Gerds and Martin Schumacher. 2006. Consistent estimation of the expected Brier score in general survival models with right-censored event times. *Biometrical Journal* 48, 6 (2006), 1029–1040.

[18] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle. 2017. Towards Recovering the Software Architecture of Microservice-Based Systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, Washington, DC, USA, 46–53.

[19] George Yanbing Guo, Joanne M Atlee, and Rick Kazman. 1999. A software architecture reconstruction method. In *Software Architecture*. Springer, Berlin, Heidelberg, 15–33.

[20] Munawar Hafiz, Paul Adamczyk, and Ralph E Johnson. 2012. Growing a pattern language (for security). In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. Association for Computing Machinery, New York, NY, USA, 139–158.

[21] Thomas Haitzer and Uwe Zdun. 2014. Semi-automated architectural abstraction specifications for supporting software evolution. *Science of Computer Programming* 90 (2014), 135–160.

[22] Mahmoud Hammad, Hamid Bagheri, and Sam Malek. 2019. DelDroid: an automated approach for determination and enforcement of least-privilege architecture in android. *Journal of Systems and Software* 149 (2019), 286–295.

[23] Joseph Ingeno. 2018. *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts.* Packt Publishing Ltd, Birmingham.

[24] Sushil Jajodia, Steven Noel, Pramod Kalapa, Massimiliano Albanese, and John Williams. 2011. Cauldron mission-centric cyber situational awareness with defense in depth. In *2011-MILCOM 2011 Military Communications Conference*. IEEE, IEEE, Washington, DC, USA, 1339–1344.

[25] Stefanie Jasser. 2020. Enforcing Architectural Security Decisions. In *International Conference on Software Architecture (ICSA)*. IEEE, Washington, DC, USA, 35–45. https://doi.org/10.1109/ICSA47634.2020.00012

[26] Michael W Kattan and Thomas A Gerds. 2018. The index of prediction accuracy: an intuitive measure useful for evaluating risk prediction models. *Diagnostic and prognostic research* 2, 1 (2018), 1–7.

[27] Ken Laskey, Jeff A Estefan, Francis G McCabe, and Danny Thornton. 2009. Reference architecture foundation for service oriented architecture version 1.0. *Oasis, Committee Draft* 2 (2009), 26.

[28] Joshua Levin and Theophilus A Benson. 2020. ViperProbe: Rethinking Microservice Observability with eBPF. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, Washington, DC, USA, 1–8.

[29] James Lewis and Martin Fowler. 2004. Microservices: a definition of this new architectural term. http://martinfowler.com/articles/microservices.html.

[30] Kim Mens, Tom Mens, and Michel Wermelinger. 2002. Maintaining software through intentional source-code views. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering* (Ischia, Italy) *(SEKE '02)*. ACM, New York, NY, USA, 289–296.

[31] Gail C. Murphy, David Notkin, and Kevin Sullivan. 1995. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering* (Washington, D.C., United States) *(SIGSOFT '95)*. ACM, New York, NY, USA, 18–28.

[32] Jerome L Myers, Arnold D Well, and Robert F Lorch Jr. 2013. *Research design and statistical analysis.* Routledge, Abingdon, UK.

[33] Sam Newman. 2015. *Building Microservices: Designing Fine-Grained Systems.* O'Reilly, Sebastopol, CA.

[34] NIST. 2019. NIST Special Publication (SP) 800-204, Security Strategies for Microservices-based Application Systems. https://www.nist.gov/news-events/news/2019/08/security-strategies-microservices-based-application-systems-nist-publishes.

[35] Steven Noel and Sushil Jajodia. 2017. A suite of metrics for network attack graph analytics. In *Network security metrics*. Springer, Berlin, Heidelberg, 141–176.

[36] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Patric Genfer, Sebastian Geiger, Sebastian Meixner, and Wilhelm Hasselbring. 2021. Detector-based component model abstraction for microservice-based systems. *Computing* 103 (2021), 2521–2551.

[37] OWASP. 2021. Microservices based Security Arch Doc Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Microservices_based_Security_Arch_Doc_Cheat_Sheet.html.

[38] Claus Pahl and Pooyan Jamshidi. 2016. Microservices: A systematic mapping study. In *6th International Conference on Cloud Computing and Services Science*. SCITEPRESS, Setubal, PRT, 137–146. https://doi.org/10.5220/0005785501370146

[39] Pierre Parrend, Timothé Mazzucotelli, and Florent Colin. 2017. *Using design structure matrices (dsm) as security controls for software architectures.* Technical Report. Tech. Rep. 1, Complex System Digital Campus, cS-DC Research Report, ARK ....

[40] Cesare Pautasso and Erik Wilde. 2009. Why is the web loosely coupled?: a multi-faceted metric for service design. In *18th Int. Conf. on World wide web*. Association for Computing Machinery, New York, NY, USA, 911–920.

[41] Shirley C Payne. 2006. A guide to security metrics. SANS Institute Information Security Reading Room.

[42] Sven Peldszus, Daniel Strüber, and Jan Jürjens. 2018. Model-Based Security Analysis of Feature-Oriented Software Product Lines. *SIGPLAN Not.* 53, 9 (Nov. 2018), 93–106. https://doi.org/10.1145/3393934.3278126

[43] Sven Peldszus, Katja Tuma, Daniel Strüber, Jan Jürjens, and Riccardo Scandariato. 2019. Secure Data-Flow Compliance Checks between Models and Code based on Automated Mappings. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer, Berlin, Heidelberg, 23–33.

[44] R. Pietrantuono, S. Russo, and A. Guerriero. 2018. Run-Time Reliability Estimation of Microservice Architectures. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Washington, DC, USA, 25–35. https://doi.org/10.1109/ISSRE.2018.00014

[45] D. Quartel and M. van Sinderen. 2007. On Interoperability and Conformance Assessment in Service Composition. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*. IEEE, Washington, DC, USA, 229–229. https://doi.org/10.1109/EDOC.2007.11

[46] F. Rademacher, S. Sachweh, and A. Zündorf. 2019. Aspect-Oriented Modeling of Technology Heterogeneity in Microservice Architecture. In *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, Washington, DC, USA, 21–30. https://doi.org/10.1109/ICSA.2019.00011

[47] Alex Ramos, Marcella Lazar, Raimir Holanda Filho, and Joel JPC Rodrigues. 2017. Model-based quantitative network security metrics: A survey. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2704–2734.

[48] Chris Richardson. 2017. A pattern language for microservices. http://microservices.io/patterns/index.html.

[49] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. 2013. *Security Patterns: Integrating security and systems engineering.* John Wiley & Sons, New York, NY, USA.

[50] Jason Skowronski. 2019. Best Practices for Event-Driven Microservice Architecture. https://hackernoon.com/best-practices-for-event-driven-microservice-architecture-e034p21lk.

[51] Karsten Sohr and Bernhard Berger. 2010. Idea: Towards Architecture-Centric Security Analysis of Software. In *Engineering Secure Software and Systems*, Fabio Massacci, Dan Wallach, and Nicola Zannone (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 70–78.

[52] Ewout W Steyerberg, Andrew J Vickers, Nancy R Cook, Thomas Gerds, Mithat Gonen, Nancy Obuchowski, Michael J Pencina, and Michael W Kattan. 2010. Assessing the performance of prediction models: a framework for some traditional and novel measures. *Epidemiology (Cambridge, Mass.)* 21, 1 (2010), 128.

[53] D. Taibi and V. Lenarduzzi. 2018. On the Definition of Microservice Bad Smells. *IEEE Software* 35, 3 (May 2018), 56–62. https://doi.org/10.1109/MS.2018.2141031

[54] Kennedy A Torkura, Muhammad IH Sukmana, Anne VDM Kayem, Feng Cheng, and Christoph Meinel. 2018. A cyber risk based moving target defense mechanism for microservice architectures. In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/-SocialCom/SustainCom)*. IEEE, Washington, DC, USA, 932–939.

[55] Arie Van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva. 2004. Symphony: View-driven software architecture reconstruction. In *4th Working IEEE/IFIP Conf. on Software Architecturen(WICSA 2004)*. IEEE, IEEE, Washington, DC, USA, 122–132.

[56] Radu Vanciu and Marwan Abi-Antoun. 2013. Finding architectural flaws using constraints. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Washington, DC, USA, 334–344. https://doi.org/10.1109/ASE.2013.6693092

[57] M. Vianden, H. Lichter, and A. Steffens. 2014. Experience on a Microservice-Based Reference Architecture for Measurement Systems. In *2014 21st Asia-Pacific Software Engineering Conference*, Vol. 1. IEEE, Washington, DC, USA, 183–190. https://doi.org/10.1109/APSEC.2014.37

[58] Lingyu Wang, Tania Islam, Tao Long, Anoop Singhal, and Sushil Jajodia. 2008. An Attack Graph-Based Probabilistic Security Metric. In *Data and Applications Security XXII*, Vijay Atluri (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–296.

[59] Martin Woschek. 2015. Owasp cheat sheets. *pp* 315 (2015), 4.

[60] Uwe Zdun, Elena Navarro, and Frank Leymann. 2017. Ensuring and Assessing Architecture Conformance to Microservice Decomposition Patterns. In *Service-Oriented Computing*, Michael Maximilien, Antonio Vallecillo, Jianmin Wang, and Marc Oriol (Eds.). Springer International Publishing, Cham, 411–429.

[61] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. 2018. Guiding Architectural Decision Making on Quality Aspects in Microservice APIs. In *Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018 (LNCS, Vol. 11236)*. Springer, Berlin, Heidelberg, 73–89.

[62] Olaf Zimmermann. 2017. Microservices Tenets. *Computer Science-Research and Development* 32, 3-4 (July 2017), 301–310. https://doi.org/10.1007/s00450-016-0337-0

[63] Olaf Zimmermann, Thomas Gschwind, Jochen Küster, Frank Leymann, and Nelly Schuster. 2007. Reusable architectural decision models for enterprise application development. In *Int. Conf. on the Quality of Software Architectures*. Springer, Berlin, Heidelberg, 15–32.

[64] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. 2021. Microservice API Patterns. https://microservice-api-patterns.org/.

[65] Olaf Zimmermann, Mirko Stocker, Uwe Zdun, Daniel Luebke, and Cesare Pautasso. 2019. Microservice API Patterns. https://microservice-api-patterns.org.