ABSTRACT

Static Analysis-based Software Architecture Reconstruction And Its Applications in Microservices

Dipta Das, M.S.

Mentor: Tomas Cerny, Ph.D.

Microservice architecture (MSA) is the predominant building block of modern cloud-based enterprise applications. MSA has several advantages over monolithic applications like scalability and maintainability, but it comes with some downsides. Modern enterprise applications consist of hundreds of individual microservices and lack a unified view. Due to this lack of unified view and distributed nature, security and anomaly assessments are harder to automate for MSA. Software Architecture Reconstruction (SAR) can be used to construct a centralized perspective for MSA. This thesis proposes an approach to automate the process of SAR using static code analysis. Also, we extend SAR for containerized microservices which are typically deployed and managed using dedicated orchestration tools like Kubernetes. In addition, we demonstrate two applications of SAR in MSA: Role-Based Access Control (RBAC) inconsistency detection and code smell detection. Finally, we verify our approach through case studies on two real-world benchmark projects.

Static Analysis-based Software Architecture Reconstruction And Its Applications in Microservices

by

Dipta Das, B.S.

A Thesis

Approved by the Department of Computer Science

____

Erich Baker, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science

Approved by the Thesis Committee

____

Tomas Cerny, Ph.D., Chairperson

____

Eunjee Song, Ph.D.

____

Enrique Blair, Ph.D.

Accepted by the Graduate School
August 2021

____

J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

Copyright © 2021 by Dipta Das

All rights reserved

TABLE OF CONTENTS

LIST OF FIGURES vii

LIST OF TABLES viii

ATTRIBUTION ix

ACKNOWLEDGMENTS xi

DEDICATION xii

1 Introduction 1

2 Background and Related Works 4

2.1 Microservice Architecture 4

2.2 Containers and Their Orchestration 5

2.3 Static Code Analysis 6

2.4 Software Architecture Reconstruction 7

2.5 Role-Based Access Control 9

2.6 Code Smells 11

3 On Automated RBAC Assessment By Constructing Centralized Perspective For Microservice Mesh 13

3.1 Abstract 13

3.2 Introduction 13

3.3 Related Work 16

3.3.1 Role-Based Access Control 16

3.3.2 Software Architecture Reconstruction 20
3.4 Proposed Method 22
3.5 Case Study 32
3.6 Threats to Validity 38
3.6.1 Internal Threats 39
3.6.2 External Threats 40
3.7 Conclusion 40

4 Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study 42
4.1 Abstract 42
4.2 Introduction 42
4.3 Related Work 44
4.4 Microservice Code Smell Catalogue 47
4.5 Code Analysis and Extension for Enterprise Architectures 50
4.6 Proposed Solution to Detect Code Smells 52
4.6.1 ESB Usage 54
4.6.2 Too Many Standards 54
4.6.3 Wrong Cuts 55
4.6.4 Not Having an API Gateway 55
4.6.5 Shared Persistency 56
4.6.6 Inappropriate Service Intimacy 56
4.6.7 Shared Libraries 56
4.6.8 Cyclic Dependency 57
4.6.9 Hard-Coded Endpoints 57
4.6.10 API Versioning 58

4.6.11 Microservice Greedy 59
4.7 Case Study 59
4.7.1 Train Ticket 60
4.7.2 Teacher Management System 64
4.7.3 Validity Threats 67
4.8 Future Trends 69
4.9 Conclusions 70

5 Software Architecture Reconstruction for Containerized Microservices . . . 71
5.1 Proposed Method 71
5.2 Case Study 72
5.3 Threats to Validity 75

6 Conclusion And Future Work 77

BIBLIOGRAPHY 79

LIST OF FIGURES

3.1 A sample user-defined role hierarchy tree. 28

3.2 Construction of combined method-call graphs. 30

3.3 Reduction and aggregation of RBAC roles. 32

3.4 Role hierarchy tree of the TMS application. 33

3.5 Inter microservice REST communications in TMS. 35

3.6 Conflicting hierarchy violation among CMS and UMS. 36

3.7 Conflicting hierarchy violation within QMS. 36

3.8 RBAC assessment pseudocode. 39

4.1 Example ESB Usage 47

4.2 Shared Persistency 49

4.3 Inappropriate Service Intimacy 49

4.4 MSANose architecture diagram. 54

4.5 Train Ticket testbed architecture diagram. 63

4.6 Inter microservice REST communications of TMS application. 65

5.1 Inter microservice REST communications of TMS containers. 76

LIST OF TABLES

1 Contributions of authors for an article included in Chapter 3 ix

2 Contributions of authors for an article included in Chapter 4 x

3.1 JAVA EE security annotations 25

3.2 Annotations used in TMS project 33

3.3 Runtime against TMS testbed 38

4.1 Comparison of architectural code smell detection tools. 50

4.2 Case study on Train Ticket benchmark. 61

4.3 Case study on TMS benchmark. 64

ATTRIBUTION

Chapter three of this paper is duplicated from the published journal article "On Automated RBAC Assessment By Constructing Centralized Perspective For Microservice Mesh" (Das et al. 2021). Contributions of each author are listed bellow and summarized in Tables 1.

(1) *Dipta Das* analyzed the existing works, defined the problem, designed the solution, performed experiments, prepared the contents (text/figures/tables), performed major revisions, and approved the final draft.

(2) *Andrew Walker* proposed the inclusion of Software Architecture Reconstruction (SAR), reviewed the paper, and approved the final draft.

(3) *Vincent Bushong* performed minor revisions, reviewed the paper, and approved the final draft.

(4) *Jan Svacina* reviewed the paper and approved the final draft.

(5) *Tomas Cerny* administrated the research, reviewed the paper, and approved the final draft.

(6) *Vashek Matyas* reviewed the paper and approved the final draft.

Table 1. Contributions of authors for an article included in Chapter 3

Contribution	Dipta Das	Andrew Walker	Vincent Bushong	Jan Svacina	Tomas Cerny	Vashek Matyas
Design	X					
Experiment	X					
Content	X	X				
Revision	Major		Minor			
Review	X	X	X	X	X	X
Administration					X	

Chapter four of this paper is duplicated from the published journal article "Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study" (Walker et al. 2020). Contributions of each author are listed bellow and summarized in Tables 2.

(1) *Andrew Walker* analyzed the existing works, defined the problem, designed the solution, performed experiments, prepared the contents (text/figures/tables) for initial submission, and approved the final draft.

(2) *Dipta Das* analyzed the existing works, designed the solution for SAR, reworked the experiments for new submission after initial rejection, prepared the contents (text/figures/tables), performed major revisions, and approved the final draft.

(3) *Tomas Cerny* administrated the research, reviewed the paper, and approved the final draft.

Table 2. Contributions of authors for an article included in Chapter 4

Contribution	Andrew Walker	Dipta Das	Tomas Cerny
Design	X	X	
Experiment	X	X	
Content	X	X	
Revision		Major	
Review	X	X	X
Administration			X

ACKNOWLEDGMENTS

First and foremost, I have to thank my parents for their endless support and encouragement. I could never have made it to this point without my parents. I must express my immense gratitude to Dr. Tomas Cerny for his guidance through this entire process. He has been incredibly generous with his time and offered invaluable advice and encouragement. I have learned so much through his knowledgeable insights, not only about research but also about time and responsibility management. I am truly honored to have worked with such a remarkable person. I like to thank the entire faculty of the Department of Computer Science for their support and valuable time. I learned a lot from the courses I took at Baylor. In particular, I want to thank Dr. Greg Hamerly for his effort in effectively designing course materials, which significantly improved my brainstorming capability. I like to thank Andrew Walker and other coauthors of my journal articles for their valuable contribution. Finally, I am grateful to God for everything in my life.

This thesis is dedicated to my parents.

CHAPTER ONE

Introduction

In modern enterprise applications, Microservice Architecture (MSA) is the dominant approach (NGINX, Inc. 2015). There are numerous benefits to using this architecture, which has contributed to its popularity (Cerny et al. 2018). Microservices are self-contained modules that aim to reduce coupling among the services. The distributed nature of a microservice-based application allows for greater flexibility for the developer. For example, separate teams can work on separate modules independently. It allows developers to chose different languages and frameworks without thinking about the compatibility issue with other modules. Also, since microservices are self-contained, it is possible to deploy each microservice independently. Similarly, it allows developers to fix bugs and release new versions just for a single module. It improves the scalability and maintainability of the applications. As microservices are loosely coupled, if one module fails, others can still operate which ensures increased availability. For example, in a e-commerce system, if the *search* module crashes, *payment* module can still operate. It was not possible in traditional monolithic deployment.

Large enterprise applications highly depend on automated tools for security and anomaly assessments. However, in MSA, the security aspects gradually become more complex as the number of microservices grows. Modern microservice-based applications have a large number of distinct microservices created individually by separate teams due to their high feature set and operational complexity. These microservices typically communicate with each other using Representational State Transfer (REST) calls (Vural et al. 2017; Salah et al. 2016). Enforcing a strong security solution for such systems is time-consuming for developers and may result

in security disagreements among microservices. This is because individual developers are only aware of a subset of the microservices they manage but have no grasp of the bigger picture. Even system architects may not have a comprehensive view of the application since many of those microservices may not have been included in the initial plan of the application but were added later.

Thus, we need an automated approach to reconstruct the architecture of the whole application to apply robust security solutions and run security assessments. This process is known as Software Architecture Reconstruction (SAR). In this paper, we present an approach to automate SAR by identifying inter-microservice REST calls. We utilized static code analysis in our proposed solution to extract the required information from the applications' codebase. Static code analysis can be done either through source code analysis or bytecode analysis. While bytecode analysis is simpler than source code analysis, thanks to canonicalization done by compilers, not all languages support bytecode. Our approach is generalized for both bytecode and source code which allows us to apply it for both interpreted and compiled languages. Unlike runtime analysis like penetration testing, the static analysis does not require an application to be deployed and hence is more cost-effective.

SAR can be leveraged to answer numerous research questions. In this paper, we demonstrate two use cases to verify the capability of SAR generated by our proposed method. One identifies Role-Based Access Control (RBAC) inconsistencies among microservices and another detects MSA-specific code smells. Both cases are solely based on static analysis. Furthermore, we extend our proposed method of SAR for containerized microservices which shows it is possible to apply static analysis on containers.

The primary security requirement of an application is to guarantee that it can only be accessed by authorized users. Role-Based Access Control (RBAC) is a

common technique of protecting REST services in which each application user is allocated to a set of roles that grant access to various parts of the system. Due to varying levels of abstraction, poor coding practices, and interaction with third-party services, finding discrepancies across RBAC rules in a large system is a time-consuming and complex process. The ability to automatically identify potential security flaws can greatly minimize the chance of such events.

Code smells are irregularities within codebases (Fowler 2018). They have no bearing on an application's performance or correct functionality. They can, however, have an impact on a wide variety of program quality traits such as reusability, testability, and maintainability. Thus, it is critical that code smells are properly recognized and handled in an application (Fowler 2018; Yamashita and Moonen 2013a). Because of the distributed structure of the application, microservices present a unique scenario in terms of code smells. Inter-module concerns, rather than intra-module ones, are frequently the subject of microservice-specific code smells. Due to this, traditional code smell detection tools fail to operate on MSA and so it is required to revisit them for MSA.

We verify both use cases through rigorous case studies where we used two real-world benchmark applications. Our results show that it is possible to automate the security and anomaly assessments for MSA by augmenting SAR with static analysis. The rest of the thesis is organized as follows. Chapter 2 discusses the background and related works for MSA, containers, SAR, static analysis, RBAC, and code smells. Chapter 3 presents our proposed solution for SAR and how it can be utilized to detect RBAC inconsistencies. Chapter 4 describes how we identified eleven MSA-specific code smells using SAR. Chapter 5 represents the extension of our proposed method of SAR for containerized microservices. Lastly, Chapter 6 concludes the work and highlights future perspectives.

CHAPTER TWO

Background and Related Works

2.1 Microservice Architecture

As the recent business trend pushes towards mass size, cloud computing, and distribution, microservices are becoming the leading design in modern enterprise systems (NGINX, Inc. 2015). A few years ago, the trend in distributed system integration was to modularize functionality into services and utilize Service-Oriented Architecture (SOA) (Buelow et al. 2009) with centralized Enterprise Service Bus (ESB) (Cerny et al. 2018) to route messages and interconnect services. ESB can be used to centralize processes and apply restrictions where new services can be easily integrated into the existing process flow. Although designed for distributed systems, such architecture of SOA leads to a monolithic deployment and often uses a single schema for data modeling. It limits developers' ability to independently evolve certain modules (Cerny et al. 2018) that further impedes the cloud-friendliness of the application (Kratzke and Quint 2017).

Because of the above limitations, the software industry has pushed for innovative distributed and cloud-friendly Microservice Architecture (MSA) (Finnigan 2018; Cerny et al. 2018). MSA is built on three principles (Wolff 2016): "A program should fulfill only one task and do it well, programs should be able to work together, and programs should use a universal interface." The primary distinction between SOA and MSA is that MSA uses a *share-as-little-as-possible* design concept that strongly focuses on the idea of bounded context. SOA, on the other hand, is a *share-as-much-as-possible* design paradigm that emphasizes the reuse of abstraction and business features (Cerny et al. 2018).

The entire MSA system is separated into several heterogeneous, self-contained, self-deployable modules that communicate via remote calls or messages. This construction has several advantages. Individual modules can be developed by multiple teams, each of which can use a different language and framework. Modules can be deployed, maintained, and evolved independently since they are self-contained. While this provides greater flexibility for faster delivery, improved scalability (Walker and Cerny 2020), it makes understanding the overall structure of the application harder as software expands over time. It is possible to manually reconstruct the architecture for a smaller number of microservices. However, modern enterprise applications tend to have thousands of microservices. Thus it is required to have an automated approach to reconstruct the architecture for MSA. Software Architecture Reconstruction (SAR) can be used to solve this problem.

2.2 Containers and Their Orchestration

Virtual machines (VMs) have been used to achieve isolation in cloud-based software deployment. In recent years, containerization has become popular as an alternative to VMs. Containers provide lightweight virtualization and consume fewer resources compared to VMs. Unlike VMs, containers share the host OS's kernel where each container runs as an isolated process in userspace (Singh and Singh 2016). There exist several containerization tools among which Docker is the most popular one. Docker relies on AuFS (Advanced Multi-Layered Unification Filesystem) that can maintain a `diff` of changes in filesystem (Scheepers 2014). This layering approach allows us to extract files from the container filesystem using the appropriate client tool.

As modern enterprise applications tend to have hundreds of microservices, it is important to orchestrate them using a separate tool. Kubernetes (k8s) is the first choice among developers for container orchestration (Cerny et al. 2020) which provides a wide range of features including automatic scaling, load balancing, and

failure handling. Kubernetes also provides in-cluster service discovery where one container can be accessed from another container using a service name instead of a hardcoded IP address.

2.3 Static Code Analysis

Static code analysis (Cerny et al. 2020) creates a representation of the application by recognizing components like classes, methods, fields, and annotations. These representations include Abstract Syntax Trees (AST), Control-Flow Graphs (CFG), or Program Dependency Graphs (PDG). Unlike runtime analysis, such as penetration testing or log analysis, the static analysis does not need the deployment of an application, making it more cost-effective. Also, developers can apply static analysis during the development phase of the application which mitigates the risk of inconsistencies in production deployments.

There are two common approaches for static code analysis. *Bytecode analysis* (Albert et al. 2007) uses the application's compiled code while in *source code analysis* we parse through the source code of the application without having to compile it into an immediate representation. Source code parsing can be tricky due to different coding conventions while compilers typically normalize bytecode. Besides, bytecode analysis can be utilized as an alternative to source code analysis when we do not have access to the application's source code. Although bytecode analysis can be computationally easier and more accessible, not all languages support bytecode. Bytecode is only available for interpreted languages like JAVA, Python, PHP, etc. This paper uses static code analysis to extract REST API specifications, which are then combined into a REST interaction graph. The applications of SAR that we demonstrated i.e. RBAC inconsistency and code smells detection are also based on static code analysis. Our approach can utilize both bytecode and source code analysis and hence is generalized for both complied and interpreted languages.

2.4 Software Architecture Reconstruction

SAR utilizes an iterative reverse engineering process to extract a representation of software architecture from source code, or documentation (Bass et al. 2003). Historically it is defined with following four phases (Bass et al. 2003):

(1) *Extraction:* All necessary artifacts are collected in the extraction phase where each set of related artifacts is relevant to a *view*.

(2) *Construction:* The representations of the views are canonicalized in the construction phase.

(3) *Manipulation:* The views are combined to create a more compact representation in the manipulation phase.

(4) *Analysis:* Specific research questions like security vulnerabilities, architectural fault, etc. are addressed from the reconstructed views in the analysis phase.

The creation of effective views of a system's architecture is at the heart of successful SAR. It is critical to select perspectives that are appropriate to the questions being addressed about a system. Rademacher et al. (Rademacher et al. 2020a) described SAR from following four different views:

(1) *Domain view* is based on the bounded context of entity objects.

(2) *Technology view* emphasises on the technology stack of each microservice.

(3) *Services view* concentrates on how services are connected i.e. inter microservice REST calls.

(4) *Operation view* focuses on the deployment and infrastructure aspects of the system like containerization, service discovery, and monitoring.

The Model-Driven Engineering (MDE) approach is widely used in SAR. In MDE, models are recognized as first-class entities to construct an efficient representation of the software architecture. Alshuqayran et al. (Alshuqayran et al. 2018a) described an MDE approach for SAR that maps a metamodel to the architecture

using mapping rules. The authors initially created the metamodel and mapping rules for one system and then validated them using seven additional systems.

Mayer et al. (Mayer and Weinreich 2018) described an automatic method for extracting the domain, service, and operation information of an application. It uses a combination of static analysis and runtime analysis to construct a language-independent representation of each service and its interaction with other services. The advantage of this approach is that it does not require separate parsers for the different languages of a heterogeneous system. However, it can generate an incomplete view since communication paths that are not traversed during the extraction phase are absent from the final view.

MicroART (Granchelli et al. 2017a) tool automates the service and operations views of SAR. It applies source code analysis to extract information about the service concerns like service names, ports, etc. Besides, it employs log analysis during runtime to discover containers, network interfaces, and service interaction.

SAR has been used to answer a wide range of research questions, many of which are related to security analysis and code anomalies. Walker et al. (Walker et al. 2020) described a method to identify microservice-specific code smells (Taibi and Lenarduzzi 2018) by reconstructing a REST communication graph. Ibrahim et al. (Ibrahim et al. 2019) propose an approach to extract a *attack graph* to identify attack paths that lead to vulnerability exploitation. They analyzed container-based deployment configuration files, more specifically, Docker Compose files to extract MSA module topology. Their implementation is based on Clair (Quay 2020) which is an open-source vulnerability scanner for Docker containers.

In this thesis, we utilized static analysis to accomplish SAR by reconstructing the inter-microservice REST communication graph. However, the preliminary results of our ongoing research indicate that SAR can also be achieved by dynamic analysis

like log tracing, making it language agnostic. We listed this approach as our future work in Chapter 6.

2.5 Role-Based Access Control

In microservice-based applications, each microservice implements a subset of functionality. These functionalities can be accessed by end-users or other microservices via an Application Programming Interface (API). There are generally two primary API development options: REST and SOAP (Simple Object Access Protocol) (Tihomirovs and Grabis 2016). While REST is an API development framework that uses the standard HTTP protocol, SOAP is just a protocol. SOAP was the de facto standard for web service interfaces for many years. However, in recent years, REST has ruled. According to Stormpath, REST is used in the design of more than 70% of public APIs (Hunsaker 2015). REST's major benefit over SOAP is its simplicity and ease of understanding. REST is lightweight, making it suitable for a broad range of devices, including mobile devices (Wagh and Thool 2012). In addition, REST employs the JavaScript Object Notation (JSON) format, which is faster to parse than the Extensible Markup Language (XML) standard used in SOAP (Tihomirovs and Grabis 2016; Castillo et al. 2011).

Securing REST API endpoints is generally easy when existing HTTP security approaches are leveraged instead of implementing a new security model (Sudhakar 2011). Securing REST endpoints involves both authentication and authorization (Brachmann et al. 2012). Authentication is the process of verifying the credentials associated with a particular request. To authenticate incoming requests, many enterprise apps employ various techniques such as basic authentication, token-based authentication, hash-based digest authentication (HMAC), OAuth, and so on (Lee et al. 2015). Authorization, on the other hand, entails determining if a request connection is authorized to execute a certain activity via a REST API.

Securing REST API endpoints is generally easy when existing HTTP security approaches are leveraged instead of implementing a new security model (Sudhakar 2011). Securing REST endpoints involves both authentication and authorization (Brachmann et al. 2012). Authentication is the process of verifying the credentials associated with a particular request. Different enterprise applications use different strategies to authenticate incoming requests, such as basic authentication, token-based authentication, hash-based digest authentication (HMAC), OAuth, etc. (Lee et al. 2015). On the other hand, authorization involves verifying whether a request connection is allowed to perform a particular action through a REST endpoint. Mandatory access control (MAC), discretionary access control (DAC), attribute-based access control (ABAC), and role-based access control (RBAC) are popular approaches for enforcing authorization (Sandhu and Samarati 1994).

Because of its simplicity and flexibility, RBAC has been extensively embraced as an alternative to traditional discretionary and obligatory access controls (Sandhu and Samarati 1994; Sandhu et al. 1996). It restricts users' access to resources based on the roles and permissions (Ahn and Sandhu 2000). System administrators can statically or dynamically control user's access by defining roles, role hierarchies, relationships, and constraints (Ferraiolo et al. 1995). In distributed systems, RBAC controls can be split into central and local domains (Ferraiolo et al. 1995). In the case of MSA, RBAC can be implemented centrally using a separate identity management tool, such as Red Hat's Keycloak (Red Hat Inc 2020a). It is also possible for each microservices to enact RBAC locally by utilizing security features of underlying frameworks, such as spring-security for spring-based applications (Scarioni and Nardone 2019).

Because of the importance of security-related concerns, significant research and development have gone into resolving role violations. One similar study focused on finding security vulnerabilities of API implementations among different libraries

based on security-sensitive events using a flow graph (Srivastava et al. 2011). Another research study described a model-based approach for testing access control rules based on consistency, completeness, and redundancy (Xu et al. 2012). The tool FixMeUp (Son et al. 2013) proposed an automated way to fix access control issues in PHP applications using static code analysis. It automatically edits the source code to resolve access control issues. Walker et. al (Walker et al. 2020a) described a similar static analysis on enterprise JAVA applications to find issues in RBAC rules defined using security annotations. However, it only considers intra-microservice RBAC issues while in this paper we demonstrated both inter and intra-microservice issues by using SAR. Distributed RBAC (dRBAC) (Freudenthal et al. 2002) and Separation of Duties (SoD) (Basin et al. 2009) has also been widely studied in the context of RBAC. A more detailed analysis of RBAC-related studies is discussed in Chapter 3.

2.6 Code Smells

Code smell was first defined by Fowler (Fowler 2018) as problems in code caused by poor design decisions. It grew in the field of modern software engineering as the characteristics of the software that indicate a code or design problem and make software hard to evolve and maintain (Fontana and Zanoni 2011).

Code smells are not always an issue, but they are a sign of one. They are code structures that suggest a breach of fundamental design principles and have a detrimental influence on design quality (Suryanarayana et al. 2014). Urgent maintenance efforts that prioritize feature delivery over code quality frequently result in code smells (Tufano et al. 2015). Code smells may have an impact on several aspects of software, including reusability, testability, and maintainability.

Emden and Mooden (Van Emden and Moonen 2002) created an automated code-smell detection tool for Java which was one of the earliest efforts at automatic code-smell detection. Since then, the field of code-smell detection has expanded. Code smell tools have been developed for *high level design* (Alikacem and Sahraoui

2009; Marinescu and Ratiu 2004; Rao and Reddy 2008), *architectural smells* (Moha 2007; Moha et al. 2010; Moha et al. 2008), and *language-specific code smells* (Moha et al. 2010; Khomh et al. 2009; Moha et al. 2006), measuring not just code smells but also the quality (Marinescu 2005; Gupta et al. 2016) of the application.

In monolithic systems code smells are frequently identified using static code-analysis. For instance, tools such as SpotBugs (SpotBugs 2019), FindBugs (Pugh 2015), CheckStyle (CheckStyle 2019), or PMD (PMD 2019) can detect code patterns that resemble a code smell. Anil et al. (Mathew and Capela 2019) recently analyzed 24 code smells detection tools.

While significant research has been conducted to identify and detect code smells in monolithic applications, less study has been conducted for distributed systems and microservices (Azadi et al. 2019). Developers could conduct code-smell detection on each of the individual modules, but this would not address any code smells unique to microservice architecture. Because of the distributed structure of the application, microservices create a unique issue when it comes to code smells. Microservice-specific code smells are frequently associated with inter-module concerns rather than intra-module issues. Taibi et al. (Taibi and Lenarduzzi 2018) proposed a catalog of eleven microservice-specific code smells. A detailed discussion about general code smells and MSA-specific code smells are presented in Chapter 4.

CHAPTER THREE

On Automated RBAC Assessment By Constructing Centralized Perspective For Microservice Mesh

This chapter is published as: Das D, Walker A, Bushong V, Svacina J, Cerny T, Matyas V. On automated RBAC assessment by constructing a centralized perspective for microservice mesh. 2021. PeerJ Computer Science 7:e376. https://doi.org/10.7717/peerj-cs.376.

3.1 Abstract

It is important in software development to enforce proper restrictions on protected services and resources. Typically software services can be accessed through REST API endpoints where restrictions can be applied using the Role-Based Access Control (RBAC) model. However, RBAC policies can be inconsistent across services, and they require proper assessment. Currently, developers use penetration testing, which is a costly and cumbersome process for a large number of APIs. In addition, modern applications are split into individual microservices and lack a unified view in order to carry out automated RBAC assessment. Often, the process of constructing a centralized perspective of an application is done using Systematic Architecture Reconstruction (SAR). This paper presents a novel approach to automated SAR to construct a centralized perspective for a microservice mesh based on their REST communication pattern. We utilize the generated views from SAR to propose an automated way to find RBAC inconsistencies.

3.2 Introduction

With the software industry's growth, the complexity of security administration is becoming more and more challenging. As the current software development trend is moving rapidly from monolithic to microservices architecture (MSA), we must address communication patterns not only for the simple client to server scenarios but

also for service to service scenarios. Since the client-server communication pattern has existed for many years, its security implications have already been well addressed. In contrast, not much has been studied for service-to-service communication patterns.

Currently, the most popular way to establish communication between services is to use Representational State Transfer (REST) (Vural et al. 2017; Salah et al. 2016). Developing a secured REST-based infrastructure is relatively easy for a smaller number of microservices. However, the security aspects gradually become more complex as the number of microservices grows. Due to their high feature set and operational complexity, modern microservice-based applications tend to have a large number of individual microservices developed separately by separate teams. Enforcing a robust security solution for such applications is tedious for developers and might lead to security disagreement among microservices. This is because individual developers only have an idea of a subset of microservices they maintain but lack an understanding of the overall picture. Even system architects may not understand the complete picture of the application since many of those microservices may not be in the initial blueprint of the application but rather were added later.

Thus, we need to establish an automatic way to generate the overall communication pattern for the whole application before diving into the security aspects. This is done through a process of Systematic Architecture Reconstruction (SAR) in which overall views are constructed from existing application artifacts. SAR is divided into four phases: extraction, construction, manipulation, and analysis.

In this paper, we first introduce a solution for automatic SAR of a microservice application, which generates a view of the microservices' REST communication pattern. By automating the first three phases of SAR and utilizing the constructed views, we can focus on the analysis phase and present an approach to enumerate possible security loopholes in the application. More specifically, we focused on finding Role-Based Access Control (RBAC) inconsistencies among microservices using

static code analysis. We present a case study on a single enterprise application called Teacher Management System (TMS) consisting of four individual microservices. This application was developed separately but re-purposed here as a testbed for performing static code analysis. Our work focuses on intra- and inter-microservice inconsistencies highlighting all possible role-based access control issues.

An application's core security requirement is to ensure that it can only be used by legitimate users (Mohanty et al. 2016). Role-Based Access Control (RBAC) is one of the popular methods of securing REST services where each user of the application is assigned to a set of roles that grant access to different parts of the system. In microservice-based applications, there can be two different abstractions to enforce RBAC rules. First, centralized among all the microservices and, second, per microservice-based.

Thus, next in this paper, we focus on the centralized approach. Finding inconsistencies among RBAC rules in a large system is a cumbersome and difficult task due to different levels of abstractions, poor coding practices, and coupling with third-party services. According to a survey conducted in 2014 by the International Data Group (IDG) (Mohanty et al. 2016), about 63% of applications have not been tested for security vulnerabilities. This can be easily mitigated by enforcing standard security features during the regular software development process (McGraw 2004). Ignoring such security vulnerabilities is expensive. Security breaches can cost companies billions and require significant time and effort to resolve. For example, the 2014 eBay hack, which was caused by improper access control restrictions, impacted over 145 million users (Swinhoe 2020). Being able to list possible security vulnerabilities automatically can significantly reduce the likelihood of such incidents.

System administrators should wisely choose the approaches to test the security vulnerability of the system. The most accurate outcome from such a test can be obtained via rigorous penetration testing. However, such an approach needs the

application to be fully deployed, and running penetration tests against a production deployment could lead to disruption for end users. Also, it is difficult to emulate all possible scenarios for penetration testing. In contrast, static code analysis can be a much easier alternative that does not require an application to be deployed and hence is more cost-effective. Although static code analysis is no panacea, when carefully implemented, it can detect many vulnerabilities. It is for these reasons we use static code analysis for our automated SAR process.

The paper is organized as follows. Section two discusses the related work and state of the art. Section three describes our proposed method in detail, and section four explores a case study. Finally, we conclude our paper by summarizing our work outcomes, describing our future goals, and listing the references. Throughout the paper, the terms "inconsistency", "violation" and "issue" are used interchangeably to indicate a potential flaw.

3.3 Related Work

In this section, we present related work from the two different perspectives considered in this paper. First, we assess the limitations of RBAC analysis in the context of enterprise systems. Next, we assess existing approaches for the SAR.

3.3.1 Role-Based Access Control

In microservice-based applications, each microservice implements a subset of features. End-users or other microservices can access these features through an application programming interface (API). There are typically two main API development choices: REST and SOAP (Simple Object Access Protocol) (Tihomirovs and Grabis 2016). While REST is an architecture for API development that works over standard HTTP protocol, SOAP is just a protocol. For many years, SOAP was a standard approach for web service interfaces. However, it has been dominated by REST in recent years. According to Stormpath, over 70% of public APIs are designed using REST

(Hunsaker 2015). The main advantage of REST compared to SOAP is its simplicity and ease of learning. REST is lightweight and hence better suited for a wide range of devices, including mobile devices (Wagh and Thool 2012). Apart from that, REST uses JavaScript Object Notation (JSON) format which is faster to parse compare to Extensible Markup Language (XML) used in SOAP (Tihomirovs and Grabis 2016; Castillo et al. 2011).

Securing REST API endpoints is generally easy when existing HTTP security approaches are leveraged instead of implementing a new security model (Sudhakar 2011). Securing REST endpoints involves both authentication and authorization (Brachmann et al. 2012). Authentication is the process of verifying the credentials associated with a particular request. Different enterprise applications use different strategies to authenticate incoming requests, such as basic authentication, token-based authentication, hash-based digest authentication (HMAC), OAuth, etc. (Lee et al. 2015). On the other hand, authorization involves verifying whether a request connection is allowed to perform a particular action through a REST endpoint. Mandatory access control (MAC), discretionary access control (DAC), attribute-based access control (ABAC), and role-based access control (RBAC) are popular approaches for enforcing authorization (Sandhu and Samarati 1994). In this paper, instead of authentication breaches, we focus on exploring and detecting possible authorization inconsistencies, specifically role-based authorization inconsistencies.

RBAC has been widely adopted as an alternative to classical discretionary and mandatory access controls because of its advancement in flexibility and detail of control (Sandhu and Samarati 1994; Sandhu et al. 1996). It regulates users' access to information and system resources based on activities that users need to execute in the system and requires the identification of roles in the system (Ahn and Sandhu 2000). RBAC's administrative capabilities have made it stand out from the alternative approaches because system administrators can statically or dynamically regulate user's

access by defining roles, role hierarchies, relationships, and constraints (Ferraiolo et al. 1995). For distributed systems, another advantage is that RBAC administrative responsibilities can be divided into central and local protection domains (Ferraiolo et al. 1995). In the case of microservice-based applications, these can be translated into central policies for all associated microservices and per microservice-based policies. Central RBAC policies can be enforced by delegating authentication and authorization tasks to a separate identity management tool, such as Red Hat's Keycloak (Red Hat Inc 2020a). On the other hand, individual microservices can carry out such policies using security features of underlying frameworks, such as spring-security for spring-based applications (Scarioni and Nardone 2019).

Due to the high impact of security-related issues, much research and development have been done addressing role violations. (Ciuciu et al. 2012) described one such strategy where appropriate security annotations are recommended for developers based on the ontology extracted from the business information. However, since this recommendation strategy works only based on business information irrespective of source code, if the business information provided is flawed, then the recommendation will also be faulty.

One similar study focused on finding security vulnerabilities of API implementations among different libraries based on security-sensitive events (Srivastava et al. 2011). It finds discrepancies among security policies associated with the same API using a flow graph. The inherent drawback of this approach is that it requires multiple independent implementations of the same API, and it can not find which ones of whose multiple implementations are faulty. Another research study described a model-based approach for testing access control rules based on consistency, completeness, and redundancy (Xu et al. 2012). It checks whether access control rules are consistent across the methods, whether they are unnecessarily repeated, and whether they covered all subset of permissions. However, the coverage of access control rules

over a set of methods does not necessarily relate to security issues. In (Xu et al. 2012), the system under study does not allow a user to rent a book on maintenance due to the incompleteness of access control rules, which is more of a system flaw rather than a security issue. In contrast, our proposed method finds whether a user can access a resource-restricted by one RBAC rule through an alternative path that has less restriction.

The tool FixMeUp (Son et al. 2013) proposed an automated way to fix access control issues in PHP applications using static code analysis. It automatically edits the source code to resolve access control issues. Although it seems compelling to automate the task, it might lead to syntax errors and might result in unintended consequences in case of false positives. On the other hand, our RBAC tool pinpoints the location of possible inconsistencies in the source code without adversely affecting the codebase since it does not modify the source code while performing analysis.

The most similar analysis to our proposed method has been described by (Walker et al. 2020a). That tool performs static code analysis on enterprise JAVA applications to find issues in RBAC rules defined using security annotations. The key difference is that it only considers intra-microservice issues, while our method works for both intra- and inter-microservice issues, taking into account all the microservices that constitute the application.

(Freudenthal et al. 2002) proposed a distributed RBAC (dRBAC) mechanism that decentralizes the trust-management across multiple administrative domains. Due to its distributive nature dRBAC is highly scalable for a large number of mutually anonymous users. It features third-party delegation that enables one authorized entity to entrust roles created by another entity. Besides, it controls the access levels for the same resource by valued attributes. Also, dRBAC presents continuous monitoring by utilizing a pub-sub model to ensure the validity of trust relationships for extended interactions. In this paper we do not assess such decentralized RBAC techniques,

rather we assume that the user authentication and role mapping are handled through a centralized service while individual microservices are responsible for the imposition of those roles on API endpoints.

Separation of Duties (SoD) has been widely studied in the context of RBAC. It ensures data integrity and fraud prevention by distributing critical tasks among multiple users (Basin et al. 2009). It enforces that no single user can execute all actions and thus any kind of fraudulent activity will cause collision among at least two users (Habib et al. 2014). In RBAC, SoD can either static or dynamic (Sandhu 1990). In the static separation of duties (SSD) constraints are enforced during the assignment of users to roles. On the other hand, in dynamic separation of duties (DSD) constraints are activated on the roles within a user session (Omicini et al. 2005). In this paper, we are not considering the user assignments and user sessions. Instead, we performed static code analysis that solely focused on a subset of SSD including statically defined roles and role hierarchies.

3.3.2 Software Architecture Reconstruction

Although many studies address access control issues, most of them are focused on single microservice or monolith applications. However, modern cloud-based applications are commonly designed as a set of microservices for better flexibility and scalability (Salah et al. 2016). The key challenge to perform a holistic analysis across multiple microservices is the automated construction of the application's centralized perspective. SAR extracts a representation of software architecture from source code or documentation through an iterative reverse engineering process (Bass et al. 2003). It is historically defined with four phases: extraction, construction, manipulation, and analysis (Bass et al. 2003). In the extraction phase, all necessary artifacts are collected. Each set of related artifacts is relevant to a *view* that represents relations among certain elements of the software architecture (Bass et al. 2003). The

construction phase creates canonical representations of the views. Then the manipulation phase combines the views to create a more compact representation to answer specific questions in the analysis phase. Lastly, the analysis phase answers specific research questions from the reconstructed views. In this paper, the analysis phase addresses the detection of possible RBAC inconsistencies. Also, to the best of our knowledge SAR has not been used to detect RBAC inconsistencies in MSA.

One approach of SAR of microservice-based systems is described by (Rademacher et al. 2020a). This method describes different modeling based on different viewpoints (Rademacher et al. 2020) where domain modeling is based on bounded context, services modeling is based on REST calls, and operation modeling is based on deployment specifications, e.g., Dockerfiles.

The Model-Driven Engineering (MDE) approach is commonly used in SAR. In MDE, models are used as first-class entities to depict an efficient representation of the software architecture (Cicchetti et al. 2013). (Alshuqayran et al. 2018a) described a manual analysis through the MDE approach to reconstruct the architecture of microservice-based open-source projects. They defined a metamodel which is then mapped to the architecture using mapping rules. The metamodel and mapping rules are initially created for one system and then refined and validated using seven additional systems. However, the authors did not apply their reconstruction strategy to answer specific questions.

(Ibrahim et al. 2019) proposes an approach to derive MSA module topology from container-based deployment configuration files, more specifically, from Docker Compose files. In addition to topology, they extracted the *attack graph*, a directed acyclic graph, to identify attack paths that lead to vulnerability exploitation. Their implementation is based on a open-source vulnerability scanner for Docker containers named Clair (Quay 2020).

The MicroART tool described by (Granchelli et al. 2017b) extracts the deployment architecture of a microservice-based system from the source code repository. It utilizes a domain-specific language to represent key elements of the architecture by using the MDE approach. It employs runtime log analysis to discover containers, network interfaces, and service interactions. However, users need to provide a reference to the container engine since MicroART does not automatically detect it from deployment configuration files.

Our proposed method reconstructs MSA architecture based on the REST communication pattern, similar to the service modeling described by (Rademacher et al. 2020a). However, unlike that system, which depends on a Service Modeling Language (Rademacher et al. 2020), our reconstruction is solely based on static code analysis and works independently.

3.4 Proposed Method

Enterprise applications are typically organized into a three-layer structure: controller layer, service layer, and repository layer. It is also common to organize microservices into the presentation layer, business layer, persistence layer, and database layer (Richards 2015). These two commonly used structures essentially indicate the same strategy. The presentation layer maps to the controller layer, which defines API endpoints, and the business layer maps to the service layer, which contains business logic. The persistence layer maintains data access objects (DAO) to interact with the database layer (Alur et al. 2003). These two layers (persistence and database) are consolidated into the repository layer in the three-layer architecture (Richards 2015; Steinegger et al. 2017).

Microservices typically communicate over REST APIs (Salah et al. 2016). Each microservice's controller layer defines the REST endpoints that serve as request entry points for that particular microservice. Requests are delegated from the controller layer to the service layer. The service layer typically implements business logic.

It processes the request and generates an appropriate response. The service layer can also incorporate with the persistence layer to store and retrieve data relevant to a specific request. However, sometimes the service layer depends on other microservices to process the request. In that case, it creates REST calls to other microservices and implements business logic based on the response. This describes a typical REST communication scenario among microservices. In particular, the service layer of one microservice makes REST calls to other microservice's controller layers to implement its business logic. Thus, the REST endpoints of each microservice can be either accessed by end-users or other microservices.

Enterprise frameworks adopted annotation-based configuration to define REST endpoints, for example, `@RestController` annotation in Spring-based JAVA applications and `@app.route` annotation in Flask based Python applications (VMware Inc 2020; Pallets Projects 2020). Since the REST endpoints are the entry points to the microservice, securing them is the single most important task for the developers. While there are several ways to enforce role-based authorization, the most widely adopted method in enterprise applications is to define authorization realms through the application server (Oberle et al. 2004) or through separate identity management tools like Keycloak (Red Hat Inc 2020b). A realm is a security policy domain defined in the application server that contains a collection of users (Jendrock et al. 2014). These users might be further organized into several groups (Jendrock et al. 2014). Centralized authorization systems like Keycloak handles user authentication and role mapping. But such systems do not verify whether developers properly enforced RBAC policies during API implementation or not. For example, some API endpoints might have missing RBAC roles. In that case, any authenticated user can access those endpoints. Similarly, two API endpoints with different roles might eventually access the same entity which might be unintentional and left unnoticed. These inconsistencies are not

flagged by the centralized authorization system and thus defining authorization policies are not enough to secure the endpoints. Developers need to enforce those policies within the application's source code that runs in that application server. Designing proper authorization policies are just one part of ensuring robust RBAC enforcement, we need to consider coding problems that might lead to security loopholes. In this paper, we focused on detecting such coding problems through static code analysis. Also, it is important to classify these problems to understand the severity and origin of them. We have defined five types of possible inconsistencies or violations:

(1) Missing role violations: This type of violation occurs when an API endpoint does not have any roles associated with it. In this case, all authenticated users can access the endpoint. Such violation typically happens when developers forget to enforce authorization roles on an API endpoint. However, it could be false-positive, for example, some API endpoint might be intentionally left open for all users.

(2) Unknown access violations: If an API endpoint contains an authorization role that is not present in the user-defined role hierarchy, then we define it as an unknown access violation. Usually this type of violation results from typographical errors and in most cases, such typos are left unnoticed since they do not cause any compilation errors. As a result, legitimate users with proper access are denied from accessing the endpoint.

(3) Entity access violations: If input and output i.e. request and response types of two API endpoints are similar but they have different authorization roles, then we classify it as an entity access violation. This kind of violation indicates that the same entity is being accessed by users with different access roles.

(4) Conflicting hierarchy violations: This type of violation happens when an intermediate method in the service layer or repository layer contains two different roles that are ancestor of each other's in the role hierarchy. This violation

signifies that users with a junior role are accessing some functionalities that might be intended for users with a senior role (Walker et al. 2020a).

(5) Unrelated access violations: Similar to conflicting hierarchy these violations focus on intermediate methods instead of endpoint methods. When an intermediate method contains two multiple roles that are located in different subtrees of the role hierarchy, we classify it as an unrelated access violation. This type of violation indicates poorly separated concerns while distributing access roles across different functionalities of the application (Walker et al. 2020a).

Like REST configurations, authorization policies are typically applied by annotating methods or functions with appropriate security annotations. These annotations can differ based on the framework used to develop the application; for example, JAX-RS security annotations are used with JAVA EE based application (Oracle 2020). A similar approach to enforce RBAC using annotation can also be found in Python applications based on the Flask framework (Thio 2020). These security annotations define the level of restrictions applied to the associated methods or functions. Table 3.1 highlights the most commonly used security annotations in JAVA EE applications supported by JSR 250 (Mordani 2016; Oracle 2020).

Table 3.1. JAVA EE security annotations

Annotation	Description
`@permitAll`	All security roles are permitted
`@denyAll`	No security roles are permitted
`@rolesAllowed`	List of permitted security roles

For example, if we add a `@rolesAllowed(ADMIN)` annotation to a controller endpoint method, only the users that have the "ADMIN" role (defined in the realms) can access the endpoint. However, since the number of such endpoint methods can be significant and can grow over each iteration of the development cycle, it is possible

to introduce inconsistencies among the allowed roles or even missing roles. Moreover, since these inconsistencies and missing roles do not cause any compilation or run time error, it is tempting for developers to overlook them, and that might result in potential security loopholes.

Our proposed method analyzes a set of microservice artifacts that communicate with each other through REST calls. It finds potential RBAC violations for the whole microservice mesh by scrapping security metadata of individual microservices and by combining them based on their REST communication flow. We divided the analyzer into three modules: a discovery module, a flow-matcher module, and an analysis module. The discovery module implements the extraction phase of SAR. It collects endpoint specification and security metadata. Next, the flow-matcher module performs the construction and manipulation phases of SAR by resolving the interaction among microservices. Finally, the analysis module completes the analysis phase of SAR and detects potential RBAC violations based on the other two modules' output.

The discovery module performs static code analysis on individual microservice artifacts. It detects the REST endpoints and security roles attached to those endpoints. Apart from that, it also lists the REST calls to other microservices, which are typically implemented in the service layer. The discovery module works for both source code artifacts and bytecode artifacts (e.g., JAR file, Python bytecode) and thus provides generalization for both interpreted languages (e.g., JAVA, Python) and compiled languages (e.g., C++, Go). The source code version of the discovery module takes a microservice artifact as input and parse class definitions while the bytecode version does the same using bytecode analysis. As discussed above, both REST endpoints and security policies are typically defined using the annotation-based configuration in enterprise applications. The descriptions of these annotations are well

structured and preserved in the source code and in the bytecode. The discovery module scans each class to find REST annotations and security annotations that define REST endpoints and security roles, respectively. It aggregates class-level annotations with method-level annotations to derive the complete definition of each REST endpoints. It collects the allowed roles, port, path, HTTP type, type of request object, and type of response object for each endpoint. It takes account of all standard HTTP types, with the most commonly used ones being *GET, POST, PUT,* and *DELETE.* The discovery module then further analyzes service layer classes to detect REST calls to other microservices. For each REST call, it detects the URL, HTTP type, type of request object, and type of response object. It parses REST client definitions to gather those attributes.

However, detecting the URL string involves further intensive analysis since the URL string is usually constructed by performing consecutive append operations in different parts of the source code. For this, our discovery module applies a backward recursive data flow analysis from the point where the URL is used to the point where the URL was initialized. In each intermediate step of the data flow where the URL was referenced, it scans for any append operations and resolves them to restore the final URL. Parts of the URL may also be constructed using values defined in the configuration files instead of hardcoded strings within the source code. Our module also scans configuration files of the project to resolve those values. Finally, the discovery module generates method-call graphs for individual microservices. It takes each controller method as the root node and populates child nodes by traversing subsequent method calls to the service layer and repository layer methods. For each microservice, the discovery module organizes all the scrapped information described above into a usable structure and passes them to the flow matcher module and analysis module.

As discussed by (Walker et al. 2020a), RBAC security analysis for individual microservices is insufficient. It fails to acknowledge violations when an end-user gains

access to a normally restricted resource by creating a proxy request through another microservice mediating the resource access through service layer REST calls. To detect such violations, we need to consider the whole MSA mesh instead of a single MSA, and we need to resolve REST communications between them to construct a complete centralized perspective. In our proposed model, the flow matcher module constructs the centralized communication graph for the whole MSA mesh. It takes descriptions of REST-endpoints and REST-calls for each microservice prepared by the discovery module. It combines all the REST endpoints into a list and all the REST calls into another list. Then it performs a brute force matching between those two lists to resolve all REST communications among the microservices. This involves matching the URL (including port and path), HTTP method, request type, and response type.

However, it is common for modern microservices to use service discovery and service registry instead of a hardcoded IP address in the URL (Montesi and Weber 2016). To resolve this, our flow matcher module matches both the IP address and service name and checks if one of them matches. The service name is usually defined in each microservices' configuration files and scrapped during the discovery phase. The flow matcher module also generates a diagram of REST communication for the whole microservice mesh for better visualization.

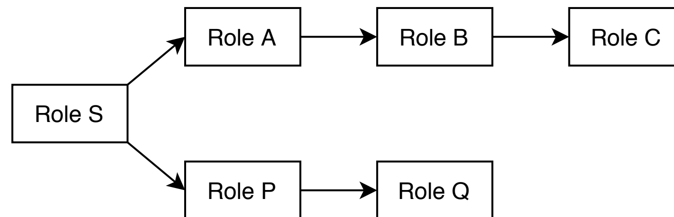

Figure 3.1. A sample user-defined role hierarchy tree.

The analysis module takes descriptions of method-call graphs and allowed roles from the discovery module and REST communication descriptions from the flow matcher module. Additionally, it takes the role hierarchy tree from the user. Figure 3.1 shows a user-defined role tree passed to the analysis module as input. Roles higher in the hierarchy tree are senior to the roles below in the tree; senior roles should have all the access rights junior roles have, plus additional rights the junior roles do not have. Roles in separate paths of the hierarchy are not related to each other. The analysis module combines method-call graphs of different microservices based on their REST communication. Figure 3.2 depicts a typical scenario of how combined method-call graphs are constructed. Each node of the combined graphs can be a controller node, service node, or repository node. Typically, only the controller nodes contain RBAC information, i.e., a list of allowed roles; however, the service layer and repository layer nodes can also have RBAC information. To find potential RBAC violations in those layers, the analysis module loops through all the nodes and analyzes the roles associated with them. The first three types of violations are only related to controller nodes. If any controller node does not have any roles associated with it, we detect it as a missing role violation. This is the most common type of violation that might occur since missing roles on controller methods do not cause any compilation errors. If a node contains a role that is not defined in the user-provided role hierarchy, we detect it as an unknown access violation. This type of violations typically results from typographical errors. If request types, response types, and HTTP types of two controller methods are equal, but they have different RBAC roles, we detect it as an entity access violation. This violation implies similar access to a particular entity with different roles.

The unrelated access and conflicting hierarchy violations occur when a node contains multiple roles after performing the reduction and aggregation. In the reduction phase, the analysis module goes through each node and keeps the lowest role

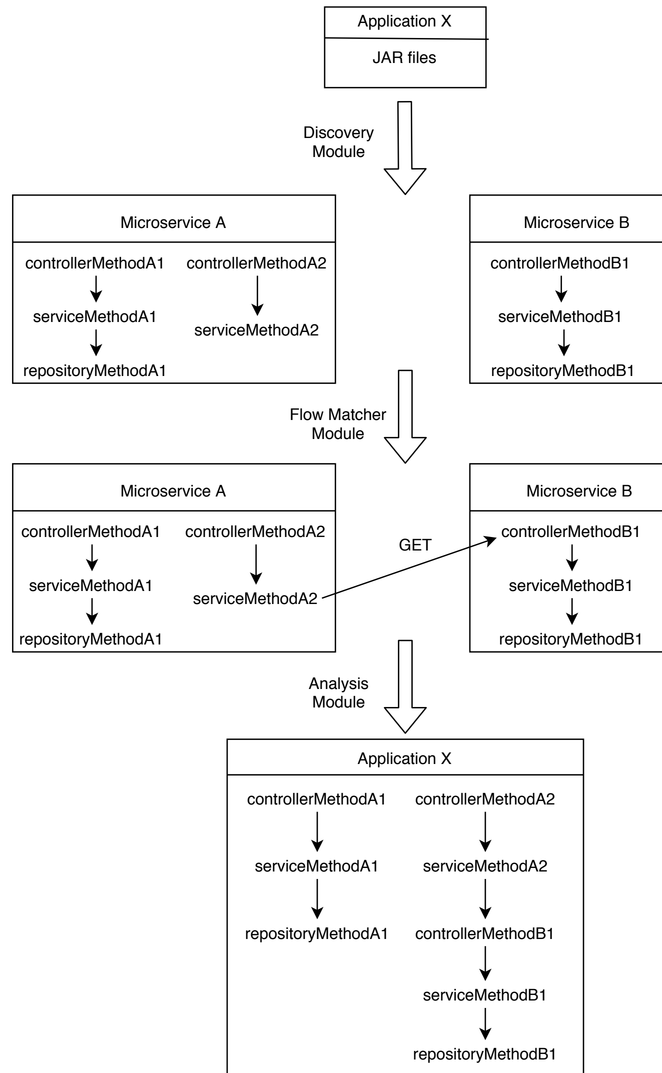

Figure 3.2. Construction of combined method-call graphs.

defined in the user-provided role hierarchy. The significance of this reduction is that it defines the minimum role required to access a specific part of the application. After reduction, in the aggregation phase, the analyzer traverses each graph and copies the allowed role from the parent node to the child node. If a child node contains an RBAC role or a child node has multiple parents with different roles, then it aggregates the roles for that particular child node. Figure 3.3 shows how the analysis module labels each child node using the RBAC roles of its parent nodes according to the role hierarchy shown in Figure 3.1. Senior roles are higher in the tree; in this example,

Role S is the most senior role, Role A is senior to B, which is senior to C. Role P is senior to Q.

The conflicting hierarchy violation occurs when a node code contains two different roles where one role is an ancestor of another role in the user-defined role hierarchy. This violation indicates a place where a junior role potentially accesses an area reserved for a more senior role. It is only a potential violation because it is ambiguous whether a junior role is accessing an area reserved for a senior role or the senior role is accessing something allowed for the junior role (Walker et al. 2020a). The unrelated access violation is the opposite of the conflicting role violation. It happens when a node contains two roles located in a different subtree of the user-defined role tree, i.e., one role is not an ancestor of another role. This violation indicates areas where unrelated roles are accessing the same application area, which may indicate poorly separated concerns that could be refactored (Walker et al. 2020a). For example, considering the role hierarchy shown in Figure 1, if a node has roles {A, C} then it is detected as a conflicting hierarchy violation, and if a node has roles {A, P} then it is marked as an unrelated access violation. The categorization of violations defined in our proposed method is mostly similar to the ones discussed by (Walker et al. 2020a). However, (Walker et al. 2020a) only considered only a single microservice at a time, whereas we also analyze inconsistencies across microservices.

Our system finds potential RBAC violations based on a user-defined role hierarchy for the whole microservice mesh (a set of microservices). It warns the developer about potential violations by providing a report of specific locations where the violations are detected and the categories, as discussed above. While some of the detected violations may be false-positive and intentional, our proposed method provides an overall idea of all possible RBAC violations for a large and complex system. The categorization of the violations helps the developer understand each violation's severity, while the specific locations of the violations help to find and fix them easily.

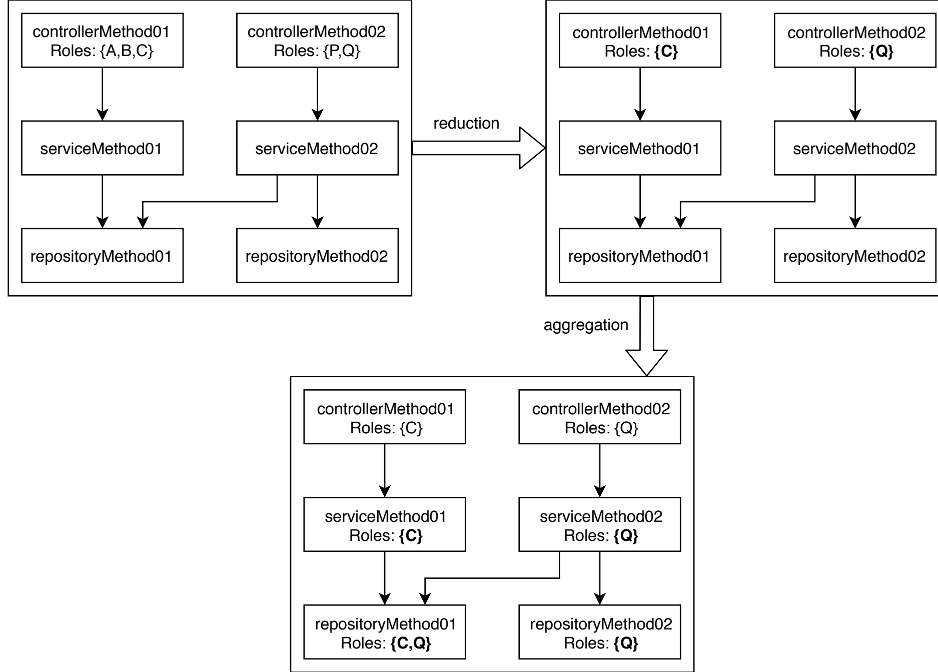

Figure 3.3. Reduction and aggregation of RBAC roles.

3.5 Case Study

The Teacher Management System (TMS)[1] is an enterprise application developed at Baylor University for Central Texas Computational Thinking, Coding, and Tinkering to facilitate the Texas Educator Certification training program. The whole TMS system consists of four individual microservices: user management system (UMS), question management system (QMS), exam management system (EMS), and configuration management system (CMS). All of the microservices are developed using the Spring Boot framework (Walls 2016) and structured into the controller, service, and repository layers. The RBAC authorization is enforced using annotations on each controller method for the individual microservices, while the central authentication and authorization policies are defined using Keycloak (Red Hat Inc 2020a). Figure 3.4 shows the role hierarchy tree for the TMS application. For our case study, we added mutants (Jia and Harman 2011) for each type of violations that

[1]https://github.com/cloudhubs/tms2020

resulted in a total of seven RBAC violations. Our system successfully detected all those violations and provided a report with specific locations of the violations. In this section, we will discuss how our analysis process works in detail for the mutated application.

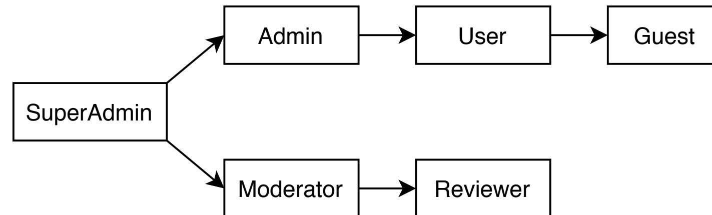

Figure 3.4. Role hierarchy tree of the TMS application.

The TMS project utilizes an annotation-based configuration technique to define application layers. REST API configurations and RBAC restrictions are also applied through annotations, which are common practice for enterprise applications. Table 3.2 lists frequently used annotations throughout the TMS project.

Table 3.2. Annotations used in TMS project

Annotation	Target	Description

`@Controller`
`@Service`
`@Repository`
	Class	Indicates controller, service, and repository layers

`@RestController`	Class	Sub type of `@Controller` to activate REST APIs
`@RequestMapping`	Class and Method	Defines HTTP types and paths for REST endpoints

`@GetMapping`
`@PostMapping`
`@DeleteMapping`
	Method	Sub types of `@RequestMapping` for specific HTTP types

`@RolesAllowed`	Method	Lists a set of allowed roles

For our purpose, we only looked for the `@RestController` annotation in the discovery module. The HTTP and paths type were extracted from the parameters of `@RequestMapping` annotation or subtype annotations. Paths can be defined at both class level or method level. We aggregated the class level paths with method-level paths to get the final path for each endpoint. The endpoints' request and response types are resolved by detecting parameters and return types of respective methods where the endpoints are defined. Finally, the RBAC roles are listed by detecting the parameters of the `@RolesAllowed` annotation applied to each endpoint method.

The `RestTemplate` class is usually used for making REST calls in the Spring Boot applications where the methods `exchange`, `getForObject`, `postForObject`, `deleteForObject`, etc. are used for performing REST calls with specific HTTP type. Each of those methods takes a URL parameter and a request object and returns a response object. We scan classes annotated with `@Service` annotation and filter them if they contain `RestTemplate` in their import statements to detect service layer REST calls. We then look for the methods described above and detect request and response types by checking the parameter type and return type. The URLs are detected by performing a backward data flow analysis recursively, as described in the proposed method section. The method calls graph is constructed by traversing each endpoint method to the service layer and repository layer methods.

After the discovery module completes gathering metadata for each MSA, the flow matcher module combines them, and the analysis module performs the final analysis. The flow matcher module also generates a visual graph of the REST communications among the microservices using Graphviz library (Ellson et al. 2002). Figure 3.5 shows the generated graph for the TMS application.

While matching the request and response types, we only considered the super-type of the generic types. For example, `List<AClass>` and `ArrayList<AClass>` are considered equal during matching.

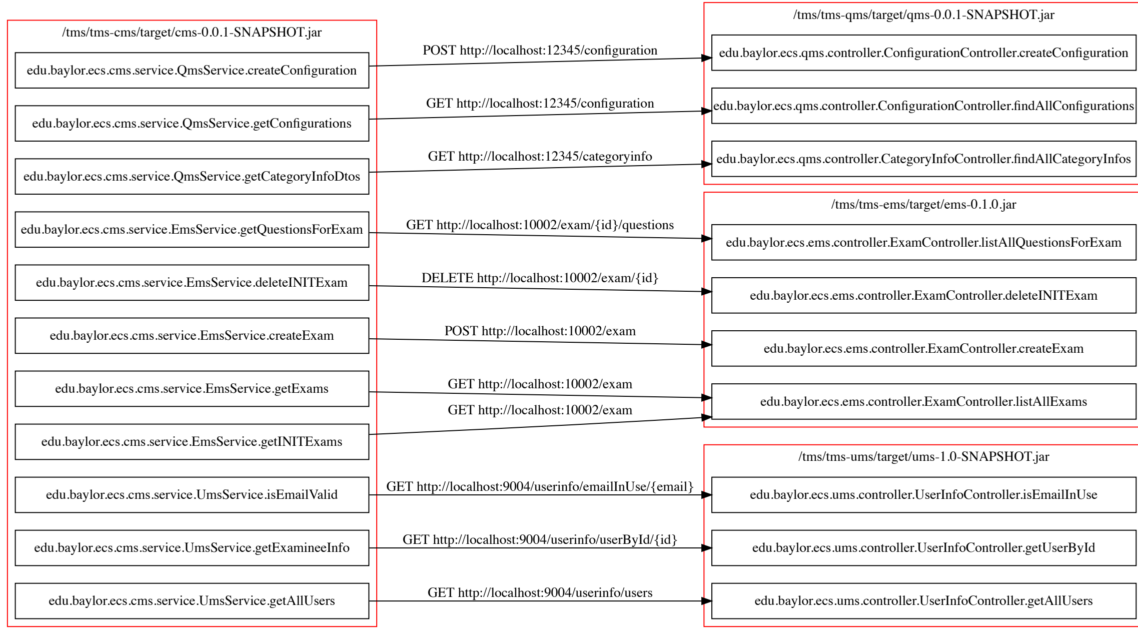

Figure 3.5. Inter microservice REST communications in TMS.

Our analyzer reported two missing-role violations for the mutated applications by specifying the fully qualified name (MSA name + package name + class name + method name) of the endpoint methods that are defined without specifying any RBAC roles. It detected two unknown-role violations along with their locations. These two violations have resulted from data entry errors where “user” and “admin” roles are mistakenly typed as “usre” and “adnin” respectively, which are not present in the role hierarchy shown in Figure 3.4. Our analyzer flagged one entity access violation by pointing out a pair of fully qualified method names. Methods `getExams` and `getINITExams` in CMS have the same return type `List<Exam>` and the same HTTP type `GET` but they have different RBAC roles: “user” and “moderator” respectively.

We found two conflicting hierarchy violations in the mutated TMS application. One of them occurred in inter microservice communication, shown in Figure 3.6, where the CMS module calls the UMS module to retrieve examinee info. The `getExaminee` endpoint method in CMS can be accessed with a “user” role which calls the `getUserById` endpoint method of EMS via service layer REST call. However, the

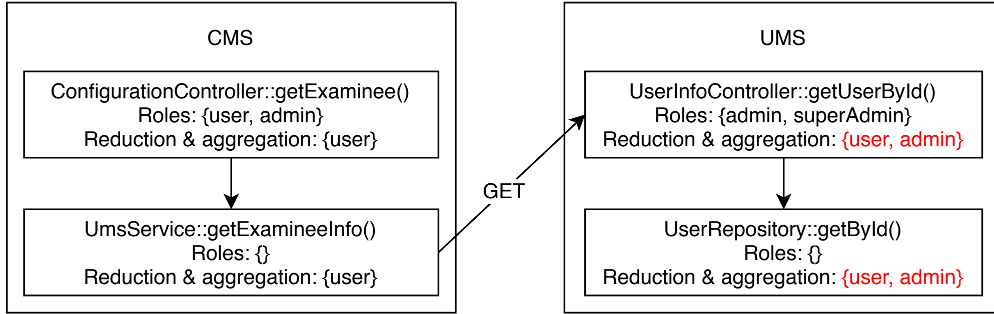

Figure 3.6. Conflicting hierarchy violation among CMS and UMS.

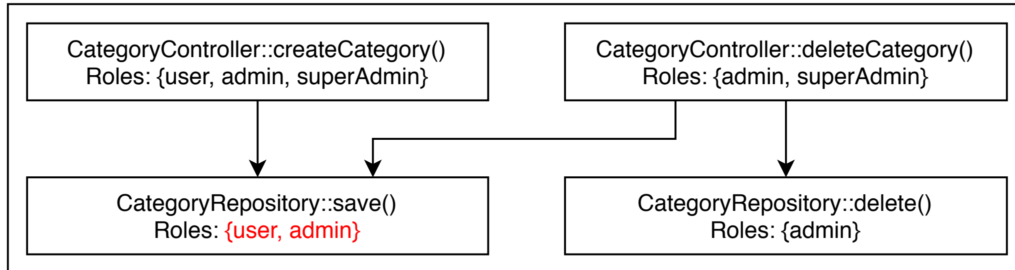

Figure 3.7. Conflicting hierarchy violation within QMS.

`getUserById` method in EMS has annotated with the "admin" role, which is a direct ancestor of the "user" role. The second conflicting hierarchy violation, shown in Figure 3.7, occurred entirely within the QMS module where both `createCategory` and `deleteCategory` endpoint methods call the `save` method of `CategoryRepository` with conflicting roles. Finally, we detected one unrelated access violation between CMS and EMS, where the method `getQuestions` in CMS has transitive access to the method `listAllQuestionsForExam` in EMS via service layer REST call. They are annotated with "user" and "moderator" roles, respectively defined in separate subtrees of the role hierarchy.

We tested both source code and bytecode version of our discovery module, which utilizes the JavaParser library (Bruggen 2020) to parse the source code and

JavaAssist library (Chiba 1998) to perform bytecode analysis to extract class definitions. We published our implementation as an open-source tool[2][3][4]. We ran it against the TMS project for benchmarking our analyzer and separately measured the runtime for each discovery, flow matcher, and analysis modules. For the discovery module, we break down our measurements for each microservice (CMS, QMS, EMS, and UMS) and count the number of classes it scanned. Note that the discovery module performs a deep scanning for the controller layer classes that are annotated with `@RestController` annotation and service layer classes that have `RestTemplate` import to detect REST endpoints, security roles, and REST calls. For other classes, it performs just a shallow scan to construct the method call graphs.

Table 3.3 shows the total runtime[5] for each module and the breakdown for the discovery module for static bytecode analysis. We can immediately see that the discovery module takes the most significant time since it performs scanning of all class files to extract metadata. In contrast, the flow-matcher and the analysis module, operating on the extracted metadata, take comparatively less time. For the discovery module, runtime depends on the number of class files in each microservices. The runtime of the flow-matcher module depends on the number of REST endpoints and the number of REST calls, while the runtime of the analysis module depends on the number of inter-microservice REST connections and the depth of the function call graph.

---

[2]SAR from bytecode: https://github.com/cloudhubs/rad

[3]SAR from source code: https://github.com/cloudhubs/rad-source

[4]RBAC analysis: https://github.com/cloudhubs/rad-analysis

[5]The benchmark is run on a Mac OS computer with a 2.9 GHz 6-core Intel Core i9 processor and 32 GB RAM

Table 3.3. Runtime against TMS testbed

Module Name	Total Runtime (sec)	Time Breakdown MSA	Time (sec)
Discovery	1.04	CMS	0.43
		EMS	0.18
		QMS	0.31
		UMS	0.12
Flow Matcher	0.13		-
Analysis	0.29		-

Our experiment exhibits a reasonable runtime to perform the static code analysis for enterprise applications. In total, it took 1.43 seconds against the TMS application, which consists of four microservices, a total of 102 classes, and 11 inter-microservice REST connections. For enterprise applications with many microservices, it is possible to run the discovery module in parallel for multiple microservices, which will significantly reduce the runtime.

To show the performance of our method on larger systems, the pseudocode for our algorithm is given in Figure 3.8. The amount of work necessary scales linearly with both the number of methods in the system and with the product of the REST calls and endpoints within the system, meaning our algorithm runs in $O(M + E * C)$, where M is the number of methods, E is the number of REST endpoints, and C is the number of REST calls. Since the number of methods in a system is usually much larger than the number of REST calls and endpoints, our algorithm will usually run in $O(M)$. This is in line with the results of our experiment; the discovery module, which searches every method for the needed metadata, was responsible for the majority of the time taken.

3.6 Threats to Validity

There are several threats to the validity of our work to address. Some of these arise from our experiment and some from how generalizable our approach is.

```
RBACAssessment(pathToMicroservices, roleHierarchy) {
    // extract metadata
    for each path in pathToMicroservices {
        analyze project property files to get service-name,
            port, hard-coded string values, etc.

        extract class definition using static code analysis

        // populate serverList and clientList
        for each class {
            for each method {
                if the method annotated with REST annotations {
                    extract API endpoint definition metadata

                    add extracted metadata to serverList

                    follow each method call to create a method call graph

                    extract RBAC security roles associated with those methods

                    add the graph to methodCallGraph as a subgraph
                }
                if the method contains REST API calls {
                    extract API call descriptions e.g. HTTP method, URL, etc.

                    add extracted metadata to clientList
                }
            }
        }
    }
    // resolve inter-microservice REST connections
    for each server in serverList {
        for each client in clientList {
            if URL, port, HTTP method matches for server and client {
                add (server, client) pair to restConnections
            }
        }
    }
    // update method call graph
    for each connection in restConnections {
        add an edge from client to server in methodCallGraph
    }
    // reduction
    for each method in methodCallGraph {
        keep only the lowest role in roleHierarchy and discard others
    }
    // aggregation
    for each disjoint subgraph in methodCallGraph {
        traverse all paths and merge the roles from parent to child
    }
    // find inconsistencies
    for each method in methodCallGraph {
        if the method has conflicting roles according to roleHierarchy {
            report inconsistency
        }
    }
}
```

Figure 3.8. RBAC assessment pseudocode.

3.6.1 Internal Threats

The primary threats to the validity of our experiment are the accuracy of the violations detected and the accuracy of the performance measures. Since we introduced known mutants for the errors, we know our tool accurately detected all of the issues. Performance-wise, we showed that our tool performed well on a small-sized application, and that the algorithm should scale up well with larger applications

since the most expensive portion of the analysis scales only linearly with the number of methods in the project.

3.6.2 External Threats

There are three external threats to our work's validity, which may affect how generalizable our results are. First, some of the detected inconsistencies might be false positives i.e. those might be intentionally left behind by the developers. Second, it depends on a user-defined role hierarchy that is assumed to contain roles universal to the application. This may not be true if users are defined in separate security realms; a role name in one realm may not be equivalent to the same role name in a different realm, either in its own access rights or in its relative position in the role hierarchy. In this case, a mapping would have to be supplied, showing which, if any, roles should be equivalent across the different realms. Another limitation is the use of security annotations. If security policies are implemented differently than through annotations, are defined in a language or a framework that does not support annotations, the current approach would not detect the roles. However, if another method was used to extract allowed roles, they could be used in the rest of the analysis process.

3.7 Conclusion

We introduced a novel solution to automatically detect authorization inconsistencies in the role-based access control (RBAC) implementation for enterprise applications using automated SAR. Our solution categorizes the violations into five types: missing-role violation, unknown access violation, entity access violation, unrelated access violation, and conflicting hierarchy violation. Our analyzer scans a set of microservice artifacts and provides a report listing all the possible violations by pinpointing their locations and types. While some of the detected violations may be false-positive, the violation type, along with a specific location, helps the developer

easily debug them, fix them, or discard them if they were intentional. Although our analyzer was developed for a JAVA enterprise application, our proposed approach is not restricted to any particular programming language or framework. It can easily be implemented for other languages and frameworks since all modern languages now have a well-structured abstraction for REST APIs and RBAC policies.

One major shortcoming of our method is that it assumes the role hierarchy and association of users with roles are defined centrally. However, individual microservices can have separate role hierarchies or even different user-role associations. Similarly, the trust management can be distributed across multiple domains like the dRBAC. In the future, we will extend our system to address these issues to allow multiple role hierarchies and multiple role mappings along with their decentralization. Besides, we like to experiment on role assignment within a user session to identify possible inconsistencies while enforcing DSD. Our long term goal is to perform such analysis within the cloud-native environment commonly used in production deployments, for example, analyzing Dockerfiles and Kubernetes artifacts.

CHAPTER FOUR

Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study

This chapter is published as: Walker A, Das D, Cerny T. Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study. 2020. Applied Sciences 10(21):7800. https://doi.org/10.3390/app10217800.

4.1 Abstract

Microservice Architecture (MSA) is becoming the predominant direction of new cloud-based applications. There are many advantages to using microservices, but also downsides to using a more complex architecture than a typical monolithic enterprise application. Beyond the normal poor coding practices and code smells of a typical application, microservice-specific code smells are difficult to discover within a distributed application setup. There are many static code analysis tools for monolithic applications, but tools to offer code-smell detection for microservice-based applications are lacking. This paper proposes a new approach to detect code smells in distributed applications based on microservices. We develop an MSANose tool to detect up to eleven different microservice specific code smells and share it as open-source. We demonstrate our tool through a case study on two robust benchmark microservice applications and verify its accuracy. Our results show that it is possible to detect code smells within microservice applications using bytecode and/or source code analysis throughout the development process or even before its deployment to production.

4.2 Introduction

Microservices are becoming the preeminent architecture in modern enterprise applications (NGINX, Inc. 2015). There are several advantages to utilizing this architecture, which have led to its rise in popularity (Cerny et al. 2018). The distributed

nature of a microservice-based application allows for greater autonomy of developer units. While this provides greater flexibility for faster delivery, improved scalability, and benefits in existing problem domains (Walker and Cerny 2020), it also presents the opportunity for code smells to be more readily created within the application. This is especially true since distinct teams manage different distributed modules of the overall system.

Code smells (Fowler 2018) are anomalies within codebases. They do not necessarily impact the performance or correct functionality of an application. They are patterns of poor programming practice and deteriorate program quality. They can affect a wide range of quality attributes in a program including reusability, testability, and maintainability. If code smells go unchecked in a microservices-based application, the benefits of using a distributed development process can be mitigated. It is therefore crucial that the code-smells in an application are appropriately detected and managed (Fowler 2018; Yamashita and Moonen 2013a).

Microservices present a unique situation when it comes to code-smells due to the distributed nature of the application. Microservice-specific code smells often focus on inter-module issues rather than an intra-module issue. Traditional code-smell detecting tools cannot detect code smells between discrete modules, so these issues go unchecked during the development process. This paper shows that when we augment static code analysis to recognize enterprise development constructs, then we can effectively detect code smells in distributed microservice applications.

We share a case study targeting eleven recently identified code smells for this architecture to demonstrate our approach. Furthermore, we develop a prototype code smell detector for microservices and share it with the community as open-source. Our prototype bases on code-analysis and recognizes Java code along with Enterprise Java platform constructs and standards (DeMichiel and Shannon 2016; DeMichiel

and Keith 2006; Bernard 2009; DeMichiel 2009; Hopkins 2009). Next, it identifies eleven microservice code smells targeted in this chapter.

The rest of the chapter is as follows. Section 3 assesses related work for code smells detection and the shortcomings for distributed systems. Section 4 introduces the code smells used in this paper. Section 5 describes the static code analysis of enterprise systems. Section 6 proposes our solution for automatic code-smell detection for microservices. Section 7 verifies our approach on two existing microservice benchmark applications. Lastly, Sections 8 and 9 conclude the work and highlight future perspectives.

4.3 Related Work

Although first defined by Fowler (Fowler 2018) as problems in code caused by poor design decisions, code smells have evolved in the world of modern software engineering to encompass much more. Code smells can be defined as "*characteristics of the software that may indicate a code or design problem that can make software hard to evolve and maintain*" (Fontana and Zanoni 2011).

Code smells are not necessarily a problem but rather an indicator of a problem. They can be seen as code structures that indicate a violation of fundamental design principles and negatively impact design quality (Suryanarayana et al. 2014). Urgent maintenance activities prioritizing feature delivery over code quality often lead to code smells (Tufano et al. 2015). Thus, code smells are codebase anomalies and do not necessarily impact the performance or correct functionality; they are poor programming practice patterns. Code smells can affect a wide range of areas in a program, including reusability, testability, and maintainability.

Gupta et al. (Gupta et al. 2016) underlined that it is essential to identify and control code smells during the design and development stages to achieve higher code maintainability and quality. However, even if developers are not invested in fixing them, code smells do matter to the overall software maintainability (Yamashita

and Moonen 2013b; Moonen and Yamashita 2012; Yamashita and Counsell 2013). Furthermore, if left unchecked, code smells can begin to impact the overall system's architecture (Macia et al. 2012). Code smells can be deceptive and hide the true extent of their 'smelliness' and even carry into further refactorings of the code (Counsell et al. 2010; Macia et al. 2012). Frequently code smells are also related to anti-patterns (Reeshti et al. 2019) in an application.

Code-smell correction is a necessary process for developers (Sae-Lim et al. 2017), but it is often pushed aside. A study by Sae-Lim et al. (Sae-Lim et al. 2017) found that the most prevalent factor towards developers addressing code smells is the importance of the issue and the relevance of the issue to the task they were working on. Another study by Peters et al. (Peters and Zaidman 2012) found that, while developers are frequently aware of the code smells in their application, they do not care about actively fixing them. Most of the time, the code smells are fixed accidentally through unrelated code refactoring (Fowler 2018). Much has been done in research to address the problem of code smells, and many studies have been performed, exploring how code smells are created (Counsell et al. 2010), managed (Oliveira 2016), and fixed in industry (Tufano et al. 2017).

Tahir et al. (Tahir et al. 2020) studied how developers discussed code smells in stack exchange sites and found that these sites work as an informal crowd-based code smell detector. Peers discuss the identification of smells and how to get rid of them in a specific given context. Thus, the question is how to detect and eliminate them in a given context. They found that the most popular smells discussed between developers are also shown to be most frequently covered by available code analysis tools. It is also noted that, while Java support is the broadest, other platforms, including C#, JavaScript, C++, Python, Ruby, and PHP, are lacking in support. Concerns were also raised that there is a missing classification for how harmful smells are on a given application.

Some researchers would argue that developers do not have the time to fix all smells. For instance, Gupta et al. (Gupta et al. 2016) identified 18 common code smells and the driving power of these code smells to improve the overall code maintainability. The effect is that developers could refactor one of the smells with higher driving power, rather than address all smells in an application, and still significantly improve code maintainability.

One of the first attempts at automatic code-smell detection came from Emden and Mooden (Van Emden and Moonen 2002), who defined an automated code-smell detection tool for Java. Since then, the field of code-smell detection has continued to grow. Code smell tools have been developed for *high level design* (Alikacem and Sahraoui 2009; Marinescu and Ratiu 2004; Rao and Reddy 2008), *architectural smells* (Moha 2007; Moha et al. 2010; Moha et al. 2008), and *language-specific code smells* (Moha et al. 2010; Khomh et al. 2009; Moha et al. 2006), measuring not just code smells but also the quality (Marinescu 2005; Gupta et al. 2016) of the application. The field of automatic code-smell detection continues to evolve with an ever-changing list of code smells and languages to cover.

It is common to identify code smells in monolithic systems using code-analysis. For instance, tools such as SpotBugs (SpotBugs 2019), FindBugs (Pugh 2015), CheckStyle (CheckStyle 2019), or PMD (PMD 2019) can detect code patterns that resemble a code smell. Anil et al. (Mathew and Capela 2019) recently analyzed 24 code smells detection tools. While the tools correctly mapped the code smells in an application, they are limited to a single codebase, and so they become antiquated as modern software development tends towards microservice architectures.

While extensive research has been done to define and detect code smells in a monolithic application, little has been done for distributed systems (Azadi et al. 2019). It would be possible for a developer to run code-smell detection on each of the

individual modules, but this does not address any code smells specific to microservice architecture.

In a distributed environment, in particular microservices, there have been multiple code smells identified. In one study (Taibi and Lenarduzzi 2018), these smells include improper module interaction, modules with too many responsibilities, or a misunderstanding of the microservice architecture. Code smells can be specific to a certain application perspective, including the *communication perspective*, or in the *development and design process* of the application. These smells can be detected manually, which usually requires assessing the application and a basic understanding of the system, but this demands considerable effort from the developers. With code analysis instruments, smells can be discovered almost instantly and automatically with no previous knowledge of the system required. However, we are aware that no tool can detect the code anomalies that can exist between discrete modules of a microservice application.

4.4 Microservice Code Smell Catalogue

For this paper's purposes, we reused the definition of eleven microservice specific code smells from a recent exploratory study by Taibi et al. (Taibi and Lenarduzzi 2018). It used existing literature and interviews with industry leaders to distill and rank these eleven code smells for microservices. The code smells are briefly summarized as follows:

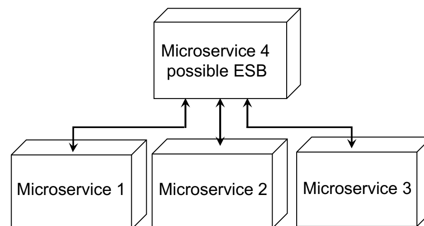

Figure 4.1. Example ESB Usage

- *ESB Usage (EU)*: An Enterprise Service Bus (ESB) (Cerny et al. 2018) is a way of message passing between modules of a distributed application in which one module acts as a service bus for all of the other modules to pass messages on. There are pros and cons to this approach. However, in microservices, it can become an issue of creating a single point of failure, and increasing coupling, so it should be avoided. An example is displayed in Fig. 4.1.
- *Too Many Standards (MS)*: Given the distributed nature of the microservices application, multiple discrete teams of developers often work on a given module, separate from the other teams. This can create a situation where multiple frameworks are used when a standard should be established for consistency across the modules.
- *Wrong Cuts (WC)*: This occurs when microservices are split into their technical layers (presentation, business, and data layers). Microservices are supposed to be split by features, and each fully contains their domain's presentation, business, and data layers.
- *Not Having an API Gateway (NAG)*: The API gateway pattern is a design pattern for managing the connections between microservices. In large, complex systems, this should be used to reduce the potential issues of direct communication.
- *Hard-Coded Endpoints (HCE)*: Hardcoded IP addresses and ports to communicate between services. By hardcoding the endpoints, the application becomes more brittle to change and reduces the application's scalability.
- *API Versioning (AV)*: All Application Programming Interfaces (API) should be versioned to keep track of changes properly.
- *Microservice Greedy (MG)*: This occurs when microservices are created for every new feature, and oftentimes, these new modules are too small and do not serve many purposes. This increases complexity and the overhead of the system. Smaller features should be wrapped into larger microservices if possible.

- *Shared Persistency (SP)*: When two microservice application modules access the same database. This breaks the microservice definition. Each microservice should have autonomy and control over its data and database. An example is provided in Fig. 4.2.
- *Inappropriate Service Intimacy (ISI)*: One module requesting private data from a separate module. This likewise breaks the microservice definition. Each microservice should have control over its private data. An example is given in Fig. 4.3.
- *Shared Libraries (SL)*: If microservices are coupled with a common library, that library should be refactored into a separate module. This reduces the fragility of the application by migrating the shared functionality behind a common, unchanging interface. This will make the system resistant to ripples from changes within the library.
- *Cyclic Dependency (CD)*: Cyclic connection between calls to different modules. This can cause repetitive calls and also increase the complexity of understanding call traces for developers. This is a poor architectural practice for microservices.

To highlight the gap in microservice code smells, we took a list of existing state-of-the-art architecture-specific code-smell detection tools from a previous and recent study (Azadi et al. 2019) and verified whether they detect any of the previously mentioned microservice-specific code smells. We chose these tools as they were

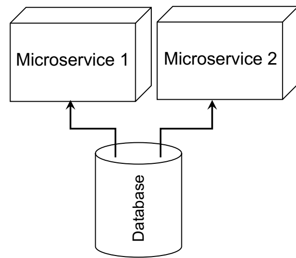

Figure 4.2. Shared Persistency

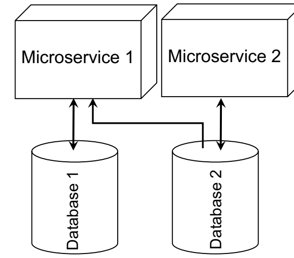

Figure 4.3. Inappropriate Service Intimacy

Table 4.1. Comparison of architectural code smell detection tools.

Tools	1 EU	2 MS	3 WC	4 NAG	5 HCE	6 AV	7 MG	8 SP	9 ISI	10 SL	11 CD
AI Reviewer (Logarix)											X
ARCADE (Le et al. 2015)											X
Arcan (Pigazzini et al. 2020)					X			X			X
Designite (Sharma 2016)											X
Hotspot Detector (Mo et al. 2015)											X
Massey Architecture Explorer (Dietrich 2012)											X
MSA Nose	X	X	X	X	X	X	X	X	X	X	X
Sonargraph (Von Zitzewitz 2019)											X
STAN (Bugan IT Consulting UG 2020)											X
Structure 101 (Headway Software Technologies Ltd)											X

compiled to study the existing state of the art of architecture smell detection tools and were shown to meet a minimum threshold of documentation and information about the tool. We compile our results in Table 4.1. The closest tool, called Arcan, was recently published (Pigazzini et al. 2020) and only detected three of the smells.

4.5 *Code Analysis and Extension for Enterprise Architectures*

Static code analysis (Cerny et al. 2020) is one of the most important software development topics, primarily its role in detecting bugs in a system. However, as with most other problem domains, there exist gaps around enterprise architectures. The two static code analysis processes, source code and bytecode analysis, ultimately create a representation of the application. This is done through several processes,

including recognizing components, classes, methods, fields, or annotations, tokenization, and parsing, which produce graph representations of the code. These include Abstract Syntax Trees (AST), Control-Flow Graphs (CFG) (Kumar and Malathi 2017; Ribeiro et al. 2007; Syaikhuddin et al. 2018), or Program Dependency Graphs (PDG) (Roy et al. 2009; Selim et al. 2010).

Bytecode analysis (Albert et al. 2007) uses the application's compiled code and is useful in uncovering endpoints, components, authorization policy enforcements, classes, and methods. It can augment or derive CFG or AST (Keivanloo et al. 2014; Keivanloo et al. 2012; Lau 2018). However, the disadvantage is that not all languages have a bytecode.

In *source code analysis* (Chatley et al. 2016), we parse through the source code of the application without having to compile it into an immediate representation. Many approaches exist to do this; however, most tools tokenize the code and construct trees, including AST (Roy et al. 2009; Selim et al. 2010), CFG (Kumar and Malathi 2017; Ribeiro et al. 2007; Syaikhuddin et al. 2018), or PDG (Gabel et al. 2008; Su et al. 2016).

However, limits exist with these representations in encapsulating the complexity of enterprise systems. To mitigate the shortcomings of existing static code analysis techniques on enterprise systems, we augment the current techniques to recognize enterprise standards (DeMichiel and Shannon 2016; Makai 2019). A more realistic representation of the enterprise application can be constructed with aid from either source code analysis or bytecode analysis. This includes a tree representation, detection of the system's endpoints, and a communication map's construction. These augmented representations along with metadata have been successful in other problem domains including security (Walker et al. 2020b), networking (Trnka et al. 2020) and semantic clone detection (Svacina et al. 2020).

The following section shows how we can use these representations and metadata for a more thorough analysis of code smells in microservice applications.

4.6 Proposed Solution to Detect Code Smells

Previous studies have shown that, without readily available information about the code smells and easy integration into the software development pipeline, the smells are often not addressed. Thus, our approach uses static-code analysis for fast and easily-integrated reports on the code smells in an application. To cover a wide variety of possible issues within a microservice application, as well as the different concerns (application, business, and data) that the identified smells cover, we must statically analyze a couple of different areas of an application. Our approach specifically involves the Java Enterprise Editions platform because of its rich standards for enterprise development. In fact, we include Spring Boot (https://spring.io/projects/spring-boot) and Java EE (https://docs.oracle.com/javaee). However, alternative standard adoptions exist also for other platforms. Next, these standards can promote to UML (Torres et al. 2009; Cerny et al. 2013), which shows platform-independence. Furthermore, extending our tool to another language would be trivial since we utilize an intermediate representation for analysis, as explained below.

The core of our solution is an automated derivation of *a centralized view of the application*, also sometimes referred to as Software Architecture Reconstruction (Rademacher et al. 2020b; Alshuqayran et al. 2018b; Granchelli et al. 2017a). To begin with, we individually analyze each microservice in the application. Once each module is fully analyzed, it can be aggregated into a larger service mesh. Then, the full detection can be done on the aggregated mesh.

Our analysis process's first step is to generate *a graph of interaction* between the different microservices. This involves exploring each microservice for a connection to another microservice, which is usually done through a REST API call. The inter-microservice communications are realized using a two-phase analysis: scanning and

matching. In the first phase, we scan each microservice to list all the REST endpoints and their specification metadata. This metadata contains the HTTP type, path, parameter, and return type of the endpoint. Additionally, the server IP addresses (or their placeholders) are resolved by analyzing application configuration files that accompany system modules. These IP addresses, together with the paths, define the fully-qualified URLs for each endpoint. We further analyze each microservice to enumerate all REST calls along with request URLs and similar metadata. We list these endpoints and REST calls based solely on static code analysis, where we leverage the annotation-based REST API configuration commonly used in enterprise frameworks. We match each endpoint with each REST call across different microservice modules based on the URL and metadata in the second phase. During matching, URLs are generalized to address different naming of path variables across different microservice modules. Each resultant matching pair indicates inter-microservice communication.

Afterward, the underlying *dependency management configuration* file is analyzed for each of the different microservices (e.g., pom.xml file for maven). This allows us to find the dependencies and libraries used by each of the applications. Lastly, the application configuration, where developers define information such as the port for the module, the databases it connects to, and other relevant environment variables for the application, is analyzed.

The overall architecture of our proposed solution is shown in Figure 4.4. The resource service module takes the path of the source files and extracts metadata from those files. These metadata are then fed into the entity service and API service modules, which produce descriptions of entities and definitions of API endpoints, respectively. The REST discovery service module takes the definitions of the API endpoints and resolves inter-microservice communications. Once the processing of each module is done, we begin the process of code-smell detection. In the following text, we provide details relevant to each particular smell and its detection.

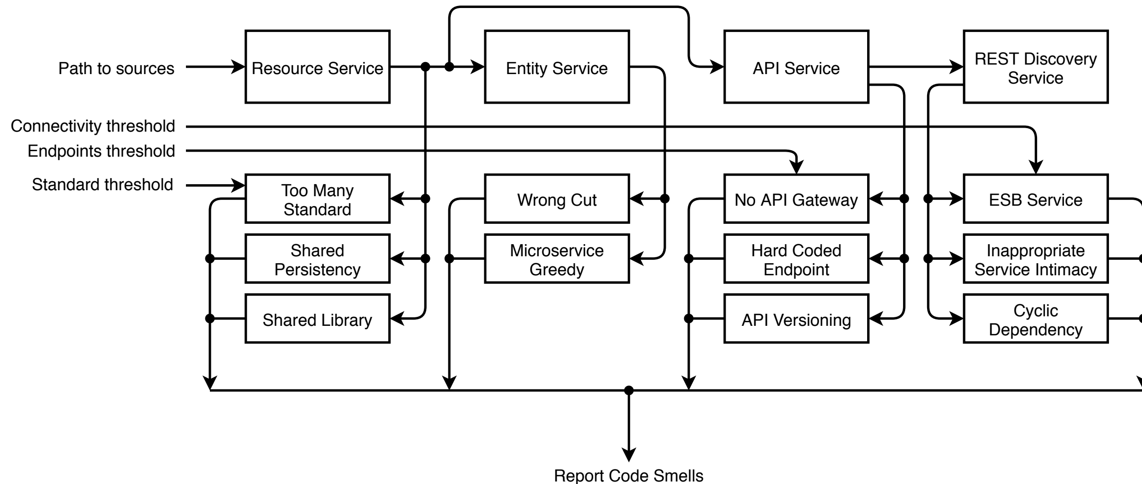

Figure 4.4. MSANose architecture diagram.

4.6.1 ESB Usage

To detect if an Enterprise Service Bus (ESB) is being used, we start by tallying up all of the incoming and outgoing connections within each module of the system. We define an ESB as a module with a high, almost outlier, number of connections and a relatively equal number of incoming and outgoing connections. Additionally, an ESB should connect to nearly all the modules.

4.6.2 Too Many Standards

Detecting if too many standards are used in an application is a tricky problem since it is entirely subjective on how many standards is "too many." Additionally, there are very good reasons developers would choose different standards for different system modules, including speed, available features, and security (Walker et al. 2020b). We tally the standards used for each of the layers of the application (presentation, business, and data). The user can configure how many standards is too many for each of the respective sections.

4.6.3 Wrong Cuts

Wrong cuts depend on the business logic and, therefore, nearly impossible to automatically detect without extrapolating a deep understanding of the business domain. However, we would expect to see an unbalanced distribution of artifacts within the microservices along with the different layers of the application (presentation, business, and data). To detect an unbalance presentation microservice, we look for an abnormally high number of front-end artifacts (such as HTML/XML documents for JSP). For the potentially wrong cut business microservices, we look for an unbalanced number of service objects, and, lastly, for wrongly cut data microservices, we look for an unbalanced number of entity objects. To find microservices with this smell, we look for outliers in the number of the specified artifacts within each microservice. Next, we report the possibility of a wrongly cut microservice to the user. We define an outlier count of greater than two times the standard deviation away from the average count of the artifacts in each microservice, which is seen in Equation (4.1).

$$2 * \sqrt{\frac{\sum_{i=0}^{n}(x_i - \bar{X})^2}{n-1}} \tag{4.1}$$

4.6.4 Not Having an API Gateway

Not having an API gateway is something that is not always possible to determine from code analysis alone. It is especially the case as cloud applications increasingly rely on routing frameworks such as AWS API Gateway (https://aws.amazon.com/api-gateway/), which uses an online configuration console and is not discoverable from code analysis, to handle routing API calls. In the study by Taibi et al. (Taibi and Lenarduzzi 2018), it was found that developers could adequately manage up to 50 distinct modules without needing to rely on an API gateway. For this reason, if the scanned application has more than 50 distinct modules, we include

a warning message in the final report that they will likely want to use an API gateway. This is not classified as an error but rather a suggestion for best practice.

4.6.5 Shared Persistency

Shared persistency happens when two or more modules of the application share the same relational database. An example of this can be seen in Figure 4.2. This is detected by parsing the application's configuration files and finding the submodules' persistence settings location. For example, in a Spring Boot application, the application YAML file is parsed for the datasource URL. Then, the persistence of each module is compared to the others to find shared datasources.

4.6.6 Inappropriate Service Intimacy

Inappropriate service intimacy can appear in a couple of different ways. It is defined as one microservice requesting the private data of another microservice. An example of this can be seen in Figure 4.3. One of the ways we detect this is as a variant of the shared persistency problem. Here, instead of sharing a datasource between two or more modules, a module is directly accessing another's datasource in addition to its own. This is detected in a similar way as shared persistency; however, once a duplicate datasource is found, if the module also has its own private datasource, then it is an instance of inappropriate service intimacy. Another way in which we search for inappropriate service intimacy is to look for two modules with the same entities. If one of those modules is only modifying/requesting the other's data, we define it as inappropriate service intimacy.

4.6.7 Shared Libraries

To detect shared libraries, the dependency management files are scanned for each module of the application to locate all shared libraries. Clearly, some shared outside libraries will exist among the microservices; however, the focus should be on

Listing 4.1. Find All Cycles.

```
boolean isCyclic() {

    // Mark all the vertices as not visited
    // and not part of recursion stack
    boolean[] visited = new boolean[V];
    boolean[] recStack = new boolean[V];

    // Call the recursive helper function to
    // detect cycle in different DFS trees
    for (int i = 0; i < V; i++)
        if (isCyclicUtil(i, visited, recStack))
            return true;

    return false;
}
```

any in-house libraries. Developers can then decide to extract into a separate module if necessary to bolster the application against changes in the libraries.

4.6.8 Cyclic Dependency

To find all cycles between modules, we use a modified depth first search (Tarjan 1971). First, we extract the REST communication graph for the microservice mesh. In the graph, each vertex represents a microservice, and each edge represents a REST API call. Then, we run our cyclic dependency detection algorithm on the graph. We maintain a recursive stack of vertices while traversing the graph. Since the graph is unidirectional (client to server), we mark it as a cycle if a vertex already exists in the stack. The algorithm is presented in Listing 4.1 and Listing 4.2.

4.6.9 Hard-Coded Endpoints

Hard-coded endpoints are found during the bytecode analysis phase of the application. Using the bytecode instructions, we can peek at the variable stack and see what parameters are passed into the function calls used to connect to other microservices. In the case of Spring Boot, for example, we look at any calls from RestTemplate. We then link the passed address back to any parameters passed to the function or

Listing 4.2. A Helper Function to Find All Cycles.

```
boolean isCyclicUtil(int i,
    boolean[] visited,
    boolean[] recStack) {

    // Mark the current node as visited
    // and part of recursion stack
    if (recStack[i])
        return true;

    if (visited[i])
        return false;

    visited[i] = true;

    recStack[i] = true;
    List<Integer> children = adjList.get(i);

    for (Integer c: children)
        if (isCyclicUtil(c,visited,recStack))
            return true;

    recStack[i] = false;

    return false;
}
```

any class fields to find the path parameters used. Our system tests for both hardcoded port numbers and hardcoded IP addresses. Both should be avoided to make scalability of the system easier in the future.

4.6.10 API Versioning

To find unversioned APIs in the application, we first find all fully qualified paths for the application. For example, the Spring Boot code in Listing 4.3 would produce a fully qualified API path of "/api/v1/users/login". Each API path is matched against a regular expression pattern `.*/v[0-9]+(.?[0-9]*).*` to locate the unversioned paths. All unversioned APIs are reported back to the user.

Listing 4.3. Example Spring Boot API.

```
@RestController
@RequestMapping("/api/v1/users")
public class UserController{

    @Autowired
    private UserService userService;

    @Autowired
    private TokenService tokenService;

    @PostMapping("/login")
    public ResponseEntity<?> getToken(...){
        return ResponseEntity.ok(...);
    }

    ...
}
```

4.6.11 *Microservice Greedy*

To find superfluous microservices, we find a couple of different metrics for each microservice. This includes front-end files (e.g., HTML, CSS, and javaScript), service objects, and entity objects in the application. Then, we find outliers, if any exist, as potential microservice greedy modules. We define outliers in the same way as when finding a wrongly cut microservice using Equation (4.1). However, we focus only on those that are outliers due to being undersized, as opposed to too large.

4.7 *Case Study*

Based on the described approach, we developed a prototype tool called MSANose (https://github.com/cloudhubs/msa-nose). This tool accepts Java-based microservice projects and performs static analysis of microservice modules. From the individual modules, it extracts the interaction patterns. It combines the partial results from each module to derive a single overall holistic view of the distributed system.

MSANose utilizes the system's derived centralized perspective to perform the eleven distinct detections mentioned above. The tool's outcome is a report containing a list of microservice code smell patterns with references to the offending modules and

code. In the next section, we describe a case study to demonstrate our approach and the developed prototype tool MSANose.

Recent efforts (Márquez and Astudillo 2019) to catalog microservice testbed applications have found a lack of applications that adhere to the guidelines for testbeds outlined by Aderaldo et al. (Aderaldo et al. 2017). We processed the benchmarks list and concluded that these are insufficient in size, nature, and state to study code smells. We introduce two testbed systems to verify the effectiveness of our application.

4.7.1 Train Ticket

To test our application, we chose to run it on an existing microservices benchmark, the Train Ticket Benchmark (Zhou et al. 2018). We chose this benchmark since it is a reasonable size for a microservice application and would provide a good test of all of our application conditions. This benchmark was designed as a model of real-world interaction between microservices in an industry environment. Next, it is one of the largest microservice benchmarks available. This benchmark consists of 41 microservices and contains over 60,000 lines of code. It uses either Docker (https://www.docker.com/) or Kubernetes (https://kubernetes.io/) for deployment which relies on either NGINX (https://www.nginx.com/) or Ingress (https://kubernetes.io/docs/concepts/services-networking/ingress/) for routing.

Before running our application on the testbed system, we manually analyzed the testbed for each of the eleven microservice code smells. This was performed as follows: first, through manual tracing of REST calls, and then through the cataloging of entities and endpoints within the system. We utilized two student researchers familiar with research into enterprise systems to ensure our manual assessment accuracy. We show the results of our manual assessment in Column 2 of Table 4.2.

After taking the results manually, we ran our application on the testbed system. Column 3 of Table 4.2 is a quick overview of the results from running our application on the testbed. The application took just 10 s to run on a system with

Table 4.2. Case study on Train Ticket benchmark.

Smell	Manual	MSANose	Time (ms)
ESB Usage	No	No	1
Too Many Standards	No	No	213
Wrong Cuts	0	2	1487
Not Having an API Gateway	No	No	1
Hard-Coded Endpoints	28	28	1
API Versioning	76	76	1981
Microservice Greedy	0	0	2093
Shared Persistency	0	0	123
Inappropriate Service Intimacy	1	1	1617
Shared Libraries	4	4	237
Cyclic Dependency	No	No	1
Total			7755

an Intel i7-4770k and 8 Gb of RAM. This includes the average time (taken over ten runs) to analyze the source code and compile bytecode of the testbed application. An individual breakdown of the times for each of the code smells available in Column 4 of Table 4.2.

We tested the testbed for potential ESBs with a connectivity threshold on the microservices of 80%, which could be adjusted by the user. Our application reported no potential ESBs, which matched our earlier manual assessment of the system.

Further, the testbed is built with Spring Boot, MongoDB, and uses static hosting for the front-end. We confirmed this through manual verification and the publicly available design documents (https://github.com/FudanSELab/train-ticket) for the system. Our application correctly identified these standards. We test for too many standards specifically within each layer (presentation, business, and data), and so no layer was beyond our threshold of two standards.

Wrong cuts were one of the most difficult to discern within the testbed. Since the testbed used static files for the front-end, we could only detect microservices that were wrongly split based on the business and data layers. We manually searched for wrongly cut microservices, but the testbed was designed well and did not have any

that we could manually determine were wrongly cut. However, our application found two potential wrongly cut microservices in the system. To find the wrong cuts on the business and data layers, we looked at the distribution of services and entities within the microservices. We know that any microservice that is wrongly cut with on the business layer would have excess amounts of services, and any cut wrongly on the data layer would have excess entities.

To discover the entity objects as part of the system's data model, we utilized standard annotations (DeMichiel and Shannon 2016) for entities and *@Document* annotation for MongoDB entities. We then matched any object we found in the system against the previously known entities and on name similarity match. To calculate name similarity, we used the WS4J project (https://github.com/Sciss/ws4j). Based on this, we could determine that object was an entity in its microservice.

The two potential wrong cuts we found were both microservices with an unusually high number of entity objects. Since the application only has 41 microservices, we did not report a need to use an API gateway; however, we can verify using the publicly available design document, as seen in Figure 4.5, that an API gateway is nonetheless used.

Of the 28 hard-coded endpoints we found, all of them were hardcoded port numbers. This is still an issue for the application, as it makes it significantly harder to scale and change the microservices later.

Of the 76 API endpoints that were unversioned, most were found in the admin modules, and some miscellaneous, non-data endpoints throughout the other modules.

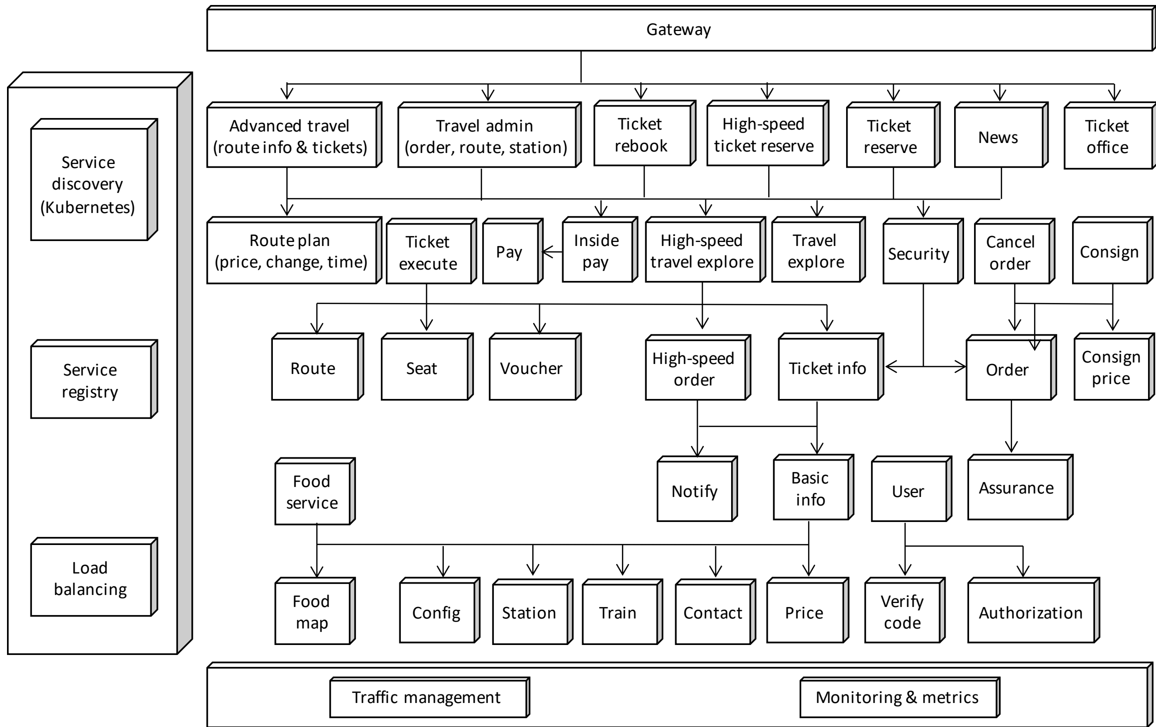

Figure 4.5. Train Ticket testbed architecture diagram.

There were no shared persistencies among the microservices, as each microservice had its own database. We could manually verify one inappropriate service intimacy in the system, which our tool correctly found. The modules *ts-admin-route-service* and *ts-route-service* both use the exact same entities, and *ts-admin-route-service* solely requests/modifies the private data of *ts-route-service* as opposed to using its own data.

Of the four shared libraries our application found, only one was widespread. The library was also not a public library, but an in-house library developed for the system. This is a perfect example of a library that is too coupled to the microservices and should be refactored. Lastly, no cyclic dependencies were found, which matches both what we found in our manual testing and the architecture diagram in Figure 4.5.

Table 4.3. Case study on TMS benchmark.

Smell	Manual	MSANose	Time (ms)
ESB Usage	No	No	1
Too Many Standards	No	No	66
Wrong Cuts	0	0	279
Not Having an API Gateway	Yes	No	1
Hard-Coded Endpoints	2	2	1
API Versioning	62	62	546
Microservice Greedy	0	0	271
Shared Persistency	0	0	60
Inappropriate Service Intimacy	0	0	1
Shared Libraries	2	2	47
Cyclic Dependency	No	No	1
Total			1074

4.7.2 Teacher Management System

The Teacher Management System (TMS) (https://github.com/cloudhubs/tms2020) is an enterprise application developed at Baylor University for Central Texas Computational Thinking, Coding, and Tinkering to facilitate the Texas Educator Certification training program. The TMS application consists of four microservices: user management system (UMS), question management system (QMS), exam management system (EMS), and configuration management system (CMS). All of the microservices are developed using the Spring Boot framework and structured into the controller, service, and repository layers. It uses Docker for application packaging, Docker-compose (https://docs.docker.com/compose/) for deployment, and NGINX for routing.

Similar to the first case study, we manually analyzed the testbed for each of the eleven microservice code smells. Then, we ran our MSANose application on the TMS testbed. It took around 2 s to run the benchmark on a 2.9 GHz Intel Core i9 computer with 32 GB of RAM. Table 4.3 shows the results of the manual assessment and results from MSANose side-by-side. An individual breakdown of the times for each of the code smells is listed in Column 4 of Table 4.3.

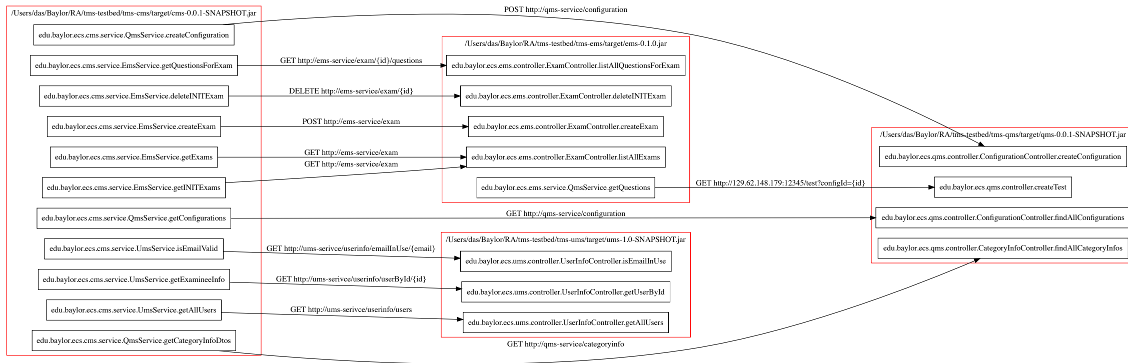

Figure 4.6. Inter microservice REST communications of TMS application.

We ran our MSANose tool on the TMS testbed for potential ESBs with a connectivity threshold on the microservices of 80%. Our tool reported no potential ESBs, which can be verified using the REST communication diagram shown in Figure 4.6. At first look, one might think of the CMS as an ESB since most of the outgoing connections are from CMS. However, ESBs are simply routers with some intelligence like data conversion and filtering. Thus, ESBs typically have a high number of incoming and outgoing connections compared to microservices. In Figure 4.6, we can see that CMS does not have any incoming connection. This indicates CMS is not an ESB and our tool produced the correct result for ESB detection.

Our manual assessment found that the testbed is built with Spring Boot, PostgreSQL with JPA is used for persistence, and static hosting is used for the front-end. Our application correctly identified these standards. We ran our tool for too many standards for each of the presentation, business, and data layers. No layer was beyond our threshold of two standards.

Since the application has only four microservices, we did not explicitly report for an API gateway. However, our manual assessment found that no API gateway was used. Our tool did not report any possible wrong cuts. We manually checked for access amount of services in the business layers and access amount of entities in the

data layers. Our manual assessment confirmed that the application was well designed, and there were no possible wrong cuts.

Our tool found two hard-coded endpoints, both of them were in EMS and pointing to QMS. Both of these endpoints were hardcoded IP, which makes it harder to scale the application. From our manual assessment, as shown in Figure 4.6, we found QMS is also accessed from CMS using a non-hard coded endpoint, which is considered to be the best practice. Thus, those hardcoded endpoints were probably mistakenly left unnoticed during the application development process.

The MSANose tool did not report any shared presidencies for the testbed, and we confirmed it by identifying that each microservices had its own database. There was no inappropriate service intimacy. We manually checked all entities and did not found any pair of entities that are exactly the same.

Our tool identified 62 unversioned API endpoints, which were verified by our manual assessment. The further assessment found that the application did not use any API versioning at all, which is critical for client-side code stability. Two shared libraries were identified, which matches the count of our manual assessment. However, both of those libraries are related to the Spring Boot framework. Thus, it is one of the false-positive warnings reported by our tool and can be ignored safely. Lastly, no cyclic dependency was found, which matches our manual assessment.

Our tool correctly analyzed both testbed systems and successfully identified the applications' microservice code smells. Code smells do not always break the system or cause system-crashing bugs, but they are problems nonetheless and are indicators of poor programming practice. As a system grows organically, as the testbed applications have done over the past couple of years, these smells can easily work their way into the system. Our tool can assist developers in locating code smells in their enterprise application, as well as providing a catalog of the smells and their common solutions as they attempt to fix them.

4.7.3 Validity Threats

One of the main validity threats is the three code smells microservice greedy, wrong cuts, and too many standards. While these code smells are specifically defined as to what they are, they do not have an established system for detection or solution (Taibi and Lenarduzzi 2018). We used our discretion, along with knowledge of enterprise architecture, to determine how our application would detect those, but it is ultimately up to interpretation. Below, we also address the internal and external threats to validity.

4.7.3.1 Internal Threats. We ran our application ten times to validate that the times we record in Tables 4.2b and 4.3b for our system's running time to avoid an unusual system deviation. Our application is tested against manually gathered results. To mitigate potential error when collecting the results, we had multiple researchers gather the results and matched them. We used these results to validate the correctness of our system.

Our application uses several thresholds for determining different code smells, which are documented with the results. These thresholds are required to estimate the severity of certain code smells and set our detection algorithms' tolerance. We used 80% connectivity threshold for detecting *ESB Usage*. A threshold of 50 microservices was used to report *not having an API gateway*. The default values of those thresholds were originally proposed by Taibi et al. (Pigazzini et al. 2020) through an extensive survey among industry specialist. However, these thresholds could be adjusted by the user, which would produce different results.

4.7.3.2 External Threats. Our application was run on two open-source benchmark applications that were similar to real-world conditions to make our test as applicable as possible. Our analysis utilizes established enterprise standards. Thus, if applications follow the best practice standards, they are analyzable by our system.

Both of the benchmark applications used in the case study are primarily written in Java. However, microservice architecture usually follows polyglot programming styles. We utilized bytecode and source code analysis in our tool to show that it can support both interpreted and compiled languages. We used Java Parser and Javassist for parsing Java source code and bytecode, respectively. Similar parsing tools are available for other modern languages; for example, Python and Golang have a built in parser package to obtain AST from the source code. Provided that a language has an appropriate parser, our tool can be extended to support a wide variety of languages used in MSA.

We designed generic interfaces to analyze and detect code smells. In our case study, we chose our first benchmark application (Train Ticket) randomly from a list of exclusively designed applications for benchmarking (Zhou et al. 2018). Then, we implemented those interfaces for Spring boot and enterprise Java since the chosen benchmark follows these standards. To verify our prototype is not application-specific, we chose our second benchmark application (Teacher Management System) that follows similar standards. However, there might exist different standards in other languages. To address this, we need to implement the interfaces for those specific standards.

Modern cloud-native microservices are usually packaged as containers and deployed using orchestration platforms such as Kubernetes, Cloud Foundry, Docker Swarm, etc. During containerization, source codes are not typically included; only the compiled artifacts such as JAR or EXE files are added into the containers. It is still possible to perform bytecode analysis for containerized microservices by extracting the bytecode artifacts (e.g., JAR file) from container layers (Cerny et al. 2020). For this approach, we also need to analyze the deployment configuration files to identify service names associated with each microservices (Cerny et al. 2020).

However, additional hard-coded dependencies in container images might require further research to identify them properly. In addition, for compiled languages, source code analysis is not possible within a containerized environment.

4.8 Future Trends

The area of microservice verification has only recently begun to be thoroughly explored. This means that an enterprise system's typical problem domains, such as security (Walker et al. 2020b), data constraints, and networking (Smid et al. 2019), have only a surface-level examination for verification. The problem domain for code smells is not different. Although this work is based on established code smells from industry advice and examination (Taibi and Lenarduzzi 2018), there exists the possibility to expand the pool of code smells for microservice-based applications. For monolithic systems, there exists hundreds of code smells in a multitude of languages. Definitions of those code smells can be adjusted to make them appropriate for MSA through an extensive survey among industry specialists. For example, the *Artificial Coupling* and *Hidden Dependencies* smells described in (Fowler 2018) can be interpreted for microservice level coupling instead of class-level coupling. In addition, similar to the study described in (Mantyla et al. 2003), a taxonomy of code smells can be done exclusively for MSA.

Our implementation has a clear separation between the metadata extraction and code-smell detection, where each detection algorithm is implemented in separate modules. Thus new detection mechanisms can be easily plugged in as a discreet module without affecting the existing metadata extraction and detection algorithms.

Similarly, this research could be expanded into other languages and enterprise standards. In addition, exploration for containerized microservices along with rigorous deployment configuration analysis can be done for cloud-native applications (Cerny et al. 2020).

Our open-source tool can be integrated into the software development lifecycle. For instance, it can be added to the CI/CD pipeline to run an automatic screening test before performing the deployment. Further, it can be used to accelerate the code review process. These adoptions will reduce the manual efforts and human errors of code reviewers and DevOps engineers resulting in a shortened release and update cycle of a microservice applications along with improved code quality.

4.9 Conclusions

In this paper, we discuss the nature of code smells in software applications. Code smells, which may not break the application in the immediate time-frame, can cause long-lasting problems for maintainability and efficiency later on. Many tools have been developed which automatically detect code smells in applications, including ones designed for architecture and overall design of a system. However, none of these tools adequately address a distributed application's needs, specifically a microservice-based application. To address these issues, we draw upon previous research into defining microservice specific code smells to build an application capable of detecting eleven unique microservice-based code smells. Our prototype application, MSANose, is open-source and available at https://github.com/cloudhubs/msa-nose. We ran our application on two established microservice benchmark applications and compared our results to manually gathered ones. We show that it is possible, through static code analysis, to analyze a microservice-based application and accurately derive microservice-specific code smells.

For future work, we plan to assess more application testbeds. Moreover, we plan to continue our work on integrating the Python platform to our approach since there are no platform-specific details, and most of the enterprise standards apply to across platforms. We also plan to detect code clones in distributed enterprise microservice applications in future work.

CHAPTER FIVE

Software Architecture Reconstruction for Containerized Microservices

Microservices are commonly packaged and delivered as containers that ensure seamless deployment in a cloud-cluster. Although several studies on Software Architecture Reconstruction (SAR) for MSA have been conducted, those methods require access to each module's source code. However, access to source code may not be readily available where containers are more easily accessible. Thus, current solutions for SAR need to be adjusted to operate on containers instead of source code. This chapter presents an effective method to perform SAR on containerized microservices. Our solution augments container instrumentation with bytecode analysis. We further analyzed deployment files to operate on container orchestration frameworks like Kubernetes.

5.1 Proposed Method

Our proposed solution reconstructs the REST interaction graph for a set of containerized microservices. We divided our approach into two phases. First, we analyzed deployment configuration files and instrument container images to extract executable bytecode artifacts. Second, the REST endpoint's specifications are collected using bytecode analysis, which is then merged to accomplish SAR. Our proposed method of SAR combines the service and operation views of the application architecture.

Our analysis process begins by examining a set of deployment configuration files. More specifically, we analyzed Kubernetes' deployment and service files. This analysis can be either static or dynamic. In the case of static analysis, deployment files are inputted as standalone JSON or YAML files. For dynamic analysis, we connect to a running Kubernetes cluster with user-provided credentials and enumerate

all currently deployed containers. Next, we examine each container image using a Docker client, which is the primary way to interact with Docker containers. For static analysis, we use a standalone Docker client. However, for dynamic analysis, we run the Docker client within a Docker container that links the underlying Docker host of a Kubernetes cluster. It can be achieved by mounting the host Docker socket path which is typically set to `/var/run/docker.sock`. This ensures that the same containers as the deployed ones are being analyzed. The remaining steps are similar for both static and dynamic analysis. The Docker client has the capability to instrument and decompose file system layers of a Docker container. We examine each container image to find the path to the executable bytecode file, i.e., JAR file and then extract that file from the file system layers of the container. Once all bytecode files are extracted, we analyzed the labels and selectors to find service names and ports associated with each container.

In the second phase, we used bytecode analysis to recognize REST API endpoints and API calls to reconstruct a REST interaction graph. This requires exploring each module's bytecode for a connection to another module, i.e., identifying inter-module REST API calls. The generation of the interaction graph involves two steps: scanning and matching. In the scanning step, we analyze each module to list all REST endpoints and REST calls and their specifications. This specification includes the IP address or service name, port, HTTP type, path, parameter, and return type. In the matching step, we compare REST endpoints with REST calls based on their specification. Each matching pair indicates an inter-module REST connection.

5.2 Case Study

This section evaluates the accuracy and performance of our proposed method through a case study on the Teacher Management System (TMS)[1] project. TMS is

[1] https://github.com/cloudhubs/tms2020

an enterprise application developed at Baylor University for Central Texas Computational Thinking, Coding, and Tinkering to facilitate the Texas Educator Certification training program. The TMS application comprises four individual microservices: user management system (UMS), question management system (QMS), exam management system (EMS), and configuration management system (CMS). All of the microservices are developed using the Spring Boot framework (Walls 2016) and structured into the controller, service, and repository layers. However, unlike the case studies described in Chapter 3 and 4, here all four of these microservices are packaged is packaged and delivered through Docker containers. The whole application can be deployed either using Docker Compose or Kubernetes.

We deployed the TMS project in a single node Kubernetes cluster using Minikube (The Kubernetes Authors 2021) and connected to the cluster using the Java client of kubernetes. Listings 5.1 and 5.2 are the Kubernetes service and deployment for the CMS microservice.

First, we list all the available k8s services excluding the built-in services. However, a Kubernetes cluster can contain multiple applications. Thus, we utilized the user-provided label `"project: tms"` to filter the services that are relevant to the TMS application. Services allow k8s containers to publish specific ports to receive incoming API calls. We parsed the name and port of the service which are `"cms"` and `"80"` respectively in Listing 5.1. Next, we parse the selectors of the services e.g. `"app: cms"`. These selectors are used to link a deployment pod with a service. Afterward, we list all the k8s deployments that match the selectors of the services. Once deployments are filtered out, we extract the container image names for each of them e.g. `"cloudhubs2/tms-cms"` in Listing 5.2. Then the list of all container images is passed to the Docker client. The Docker client analyzes each container image and extracts bytecode artifacts from their file system layers. These bytecode artifacts are then dumped into a separate location for further analysis.

Listing 5.1. CMS Service

```
apiVersion: v1
kind: Service
metadata:
  name: cms-service
  labels:
    project: tms
spec:
  selector:
    app: cms
  ports:
    - targetPort: 9081
      port: 80
      name: http
  type: NodePort
```

Once all the executable bytecode artifacts are extracted, we run the bytecode analysis for each of them. We analyzed the annotations associated with each class and methods to find REST endpoint descriptions. Typically, REST endpoints are defined in controller classes which are marked with a `@RestController` annotation in Spring Boot projects. Each of the endpoint methods is then analyzed based on the annotations associated with them. For example, `@GetMapping` annotation is used to define a GET API while `@PostMapping` annotation is used to define a POST API. Parameters of these annotations are also examined to find the API paths, query parameters, request type, and response type.

Unlike REST endpoints, REST API calls are tricky to identify as they are not defined using annotations. In a Spring Boot application, API calls are typically done through the `RestTemplate` class where the methods `getForObject`, `postForObject`, `deleteForObject`, etc. are used for performing REST calls with specific HTTP type. Each of those methods takes a URL parameter which can be either a hardcoded IP address or, service name.

For each of the REST endpoints and REST API calls, a Fully Qualified Domain Name (FQDN) is generated by concatenating the service name, port, and path. If

Listing 5.2. CMS Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cms
  labels:
    app: cms
spec:
  replicas: 1
  template:
    metadata:
      name: cms
      labels:
        app: cms
    spec:
      containers:
        - name: cms
          image: cloudhubs2/tms-cms
          ports:
            - containerPort: 9081
  selector:
    matchLabels:
      app: cms
```

the FQDN and HTTP type match for a pair of an API call and an API endpoint, then we identify it as a REST connection between two modules.

Figure 5.1 shows the REST interaction graph of the TMS application that consists of twelve inter-module REST connections.

Among the twelve inter-module REST connections shown in Figure 5.1, our tool detected eleven of them that use service name. It fails to detect the one that uses a hardcoded IP address. However, using a hardcoded IP address is not recommended as it reduces the scalability of the application and commonly recognized as a code smell (Walker, Das, and Cerny 2020).

5.3 Threats to Validity

Our proposed solution of SAR for containerized microservices relies on bytecode analysis. However, not all languages support intermediate bytecode representation. For compiled languages like C++ or Golang, it is possible to extract executable

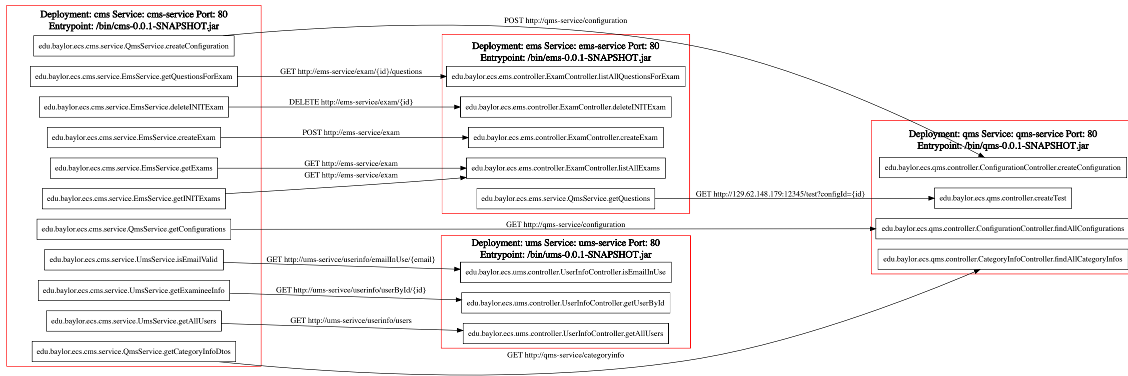

Figure 5.1. Inter microservice REST communications of TMS containers.

files from the Docker file system, but extracting REST API definitions from those executable files will require a complex reverse engineering process. Our current implementation can only analyze microservice bytecode that is developed using Java EE and Spring Boot frameworks. However, our proposed method uses generic standards of MSA. Thus it can be extended for other interpreted languages and frameworks.

CHAPTER SIX

Conclusion And Future Work

Compared to traditional monolithic applications, MSA provides greater flexibility for faster delivery and cloud deployment along with improved scalability and maintainability. However, these added benefits come with the cost of the increased complexity of application architecture. Developers need to have a firm understanding of the overall system to implement new features or trace errors. Besides, automating security assessments is harder in MSA due to the lack of a unified view. SAR can be used to mitigate this problem by reconstructing a centralized perspective of the application. In this thesis, we described an approach to automate the SAR process through static analysis. However, since microservices are commonly delivered as containers, it is important to revisit SAR approaches and include containerization. Thus, we extended our proposed method further for containerized microservices. This extension made our method stand out from traditional SAR strategies that operated only on the source code of the application. Based on the generated SAR, we demonstrated two use cases of our approach. One of them finds RBAC inconsistencies among microservices while another identifies MSA-specific code smells. Finally, we presented separate case studies for each of our proposed methods to verify them on two industry-standard benchmark projects.

In the future, we plan to extend our case study for a heterogeneous benchmark application. We are also working on an extension of the tool for Python and Golang frameworks. Apart from these, we are investigating on the following research questions:

(1) Is it possible to generalize SAR for all languages? Can we define an intermediate language-agnostic Microservice Descriptor (MSD) to represent all entities and services along with their interactions?

(2) Is it feasible to achieve SAR using dynamic analysis instead of static analysis? This approach will remove the language dependency associated with static code analysis. Our ongoing work indicates a positive outcome where we are utilizing the log tracing of service mesh (Li et al. 2019) deployed in a Kubernetes cluster.

(3) Similar to SAR, can we conduct business processes modeling (BPM) (Mayr et al. 2007) using static analysis? BPM creates a compact representation of the applications' critical paths that can significantly improve the onboarding experience of new developers on an existing project. However, current BPM approaches mostly work on a single module. We can augment them with SAR to make them suitable for MSA.

(4) Can we utilize SAR for estimating MSA-specific technical debt (Cunningham 1992) that traditional tools fail to approximate?

BIBLIOGRAPHY

Aderaldo, C. M., N. C. Mendonça, C. Pahl, and P. Jamshidi (2017). Benchmark requirements for microservices architecture research. In *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*, ECASE '17, pp. 8–13. IEEE Press.

Ahn, G.-J. and R. Sandhu (2000, November). Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Secur. 3*(4), 207–226. https://doi.org/10.1145/382912.382913.

Albert, E., M. Gómez-Zamalloa, L. Hubert, and G. Puebla (2007). Verification of java bytecode using analysis and transformation of logic programs. In M. Hanus (Ed.), *Practical Aspects of Declarative Languages*, Berlin, Heidelberg, pp. 124–139. Springer Berlin Heidelberg.

Alikacem, E. H. and H. A. Sahraoui (2009). A metric extraction framework based on a high-level description language. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 159–167.

Alshuqayran, N., N. Ali, and R. Evans (2018a). Towards micro service architecture recovery: An empirical study. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pp. 47–4709.

Alshuqayran, N., N. Ali, and R. Evans (2018b). Towards micro service architecture recovery: An empirical study. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pp. 47–4709.

Alur, D., D. Malks, J. Crupi, G. Booch, and M. Fowler (2003). *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies* (2 ed.). USA: Sun Microsystems, Inc.

Azadi, U., F. Arcelli Fontana, and D. Taibi (2019). Architectural smells detected by tools: a catalogue proposal. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pp. 88–97.

Basin, D., S. J. Burri, and G. Karjoth (2009). Dynamic enforcement of abstract separation of duty constraints. In M. Backes and P. Ning (Eds.), *Computer Security – ESORICS 2009*, Berlin, Heidelberg, pp. 250–267. Springer Berlin Heidelberg.

Bass, L., P. Clements, and R. Kazman (2003). *Software architecture in practice.* Addison-Wesley Professional.

Bernard, E. (2009, November). JSR 303: Bean validation. http://jcp.org/en/jsr/detail?id=303. Accessed 16 July 2020.

Brachmann, E., G. Dittmann, and K.-D. Schubert (2012). Simplified authentication and authorization for restful services in trusted environments. In F. De Paoli, E. Pimentel, and G. Zavattaro (Eds.), *Service-Oriented and Cloud Computing*, Berlin, Heidelberg, pp. 244–258. Springer Berlin Heidelberg.

Bruggen, D. V. (2020). JavaParser : Analyse, transform and generate your Java codebase. https://javaparser.org. Accessed 14 August 2020.

Buelow, H., M. Deb, J. Kasi, D. LHer, and P. Palvankar (2009). *Getting Started With Oracle SOA Suite 11G R1 A Hands-On Tutorial*. Packt Publishing.

Bugan IT Consulting UG (2020). STAN: Structure Analysis for Java. http://stan4j.com. Accessed 21 September 2020.

Castillo, P., J. Bernier, M. Arenas, J. Merelo Guervós, and P. García-Sánchez (2011, 01). Soap vs rest: Comparing a master-slave ga implementation. *CoRR abs/1105.4978*.

Cerny, T., K. Cemus, M. J. Donahoo, and E. Song (2013). Aspect-driven, data-reflective and context-aware user interfaces design. *ACM SIGAPP Applied Computing Review 13*(4), 53–66.

Cerny, T., M. J. Donahoo, and M. Trnka (2018, January). Contextual understanding of microservice architecture: Current and future directions. *SIGAPP Appl. Comput. Rev. 17*(4), 29–45.

Cerny, T., J. Svacina, D. Das, V. Bushong, M. Bures, P. Tisnovsky, K. Frajtak, D. Shin, and J. Huang (2020). On code analysis opportunities and challenges for enterprise systems and microservices. *IEEE Access*, 1–22.

Chatley, G., S. Kaur, and B. Sohal (2016, 01). Software clone detection: A review. *International Journal of Control Theory and Applications 9*, 555–563.

CheckStyle (2019). Checkstyle: A development tool to help programmers write java code that adheres to a coding standard. https://checkstyle.sourceforge.io. Accessed March 27, 2020.

Chiba, S. (1998, October). Javassist – a reflection-based programming wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*.

Cicchetti, A., D. Di Ruscio, L. Iovino, and A. Pierantonio (2013, February). Managing the evolution of data-intensive Web applications by model-driven techniques. *Software & Systems Modeling 12*(1), 53–83.

Ciuciu, I., Y. Tang, and R. Meersman (2012). Towards evaluating an ontology-based data matching strategy for retrieval and recommendation of security annotations for business process models. In K. Aberer, E. Damiani, and T. Dillon (Eds.), *Data-Driven Process Discovery and Analysis*, Berlin, Heidelberg, pp. 103–119. Springer Berlin Heidelberg.

Counsell, S., H. Hamza, and R. M. Hierons (2010). The 'deception' of code smells: An empirical investigation. In *Proceedings of the ITI 2010, 32nd International Conference on Information Technology Interfaces*, pp. 683–688.

Cunningham, W. (1992). The wycash portfolio management system. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, New York, NY, USA, pp. 29–30. Association for Computing Machinery.

Das, D., A. Walker, V. Bushong, J. Svacina, T. Cerny, and V. Matyas (2021, February). On automated RBAC assessment by constructing a centralized perspective for microservice mesh. *PeerJ Computer Science 7*, e376.

DeMichiel, L. (2009). JSR 317: JavaTM persistence API, version 2.0. http://jcp.org/en/jsr/detail?id=317. Accessed 16 July 2020.

DeMichiel, L. and M. Keith (2006). JSR 220: Enterprise javabeans version 3.0. java persistence API. http://jcp.org/en/jsr/detail?id=220. Accessed 16 July 2020.

DeMichiel, L. and W. Shannon (2016). JSR 366: Java Platform, Enterprise Edition 8 Spec. https://jcp.org/en/jsr/detail?id=342. Accessed 27 March 2020.

Dietrich, J. (2012). Upload your program, share your model. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, New York, NY, USA, pp. 21–22. Association for Computing Machinery.

Ellson, J., E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull (2002). Graphviz - open source graph drawing tools. In P. Mutzel, M. Jünger, and S. Leipert (Eds.), *Graph Drawing*, Berlin, Heidelberg, pp. 483–484. Springer Berlin Heidelberg. http://www.graphviz.org. Accessed 16 July 2020.

Ferraiolo, D., J. Cugini, and R. Kuhn (1995, December). Role-Based Access Control (RBAC): Features and Motivations. In *Proceedings of the 11th Annual Computer Security Applications Conference*, pp. 241–248. IEEE.

Finnigan, K. (2018). *Enterprise Java Microservices*. Manning Publications.

Fontana, F. A. and M. Zanoni (2011). On investigating code smells correlations. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 474–475.

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*. USA: Addison-Wesley Longman Publishing Co., Inc.

Freudenthal, E., T. Pesin, L. Port, E. Keenan, and V. Karamcheti (2002). drbac: distributed role-based access control for dynamic coalition environments. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pp. 411–420.

Gabel, M., L. Jiang, and Z. Su (2008). Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, New York, NY, USA, pp. 321–330. ACM.

Granchelli, G., M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle (2017a). Towards recovering the software architecture of microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 46–53.

Granchelli, G., M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle (2017b). Towards recovering the software architecture of microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 46–53.

Gupta, V., P. Kapur, and D. Kumar (2016, 04). Modelling and measuring code smells in enterprise applications using tism and two-way assessment. *International Journal of System Assurance Engineering and Management 7.*

Habib, M. A., N. Mahmood, M. Shahid, M. U. Aftab, U. Ahmad, and C. M. Nadeem Faisal (2014). Permission based implementation of dynamic separation of duty (dsd) in role based access control (rbac). In *2014 8th International Conference on Signal Processing and Communication Systems (ICSPCS)*, pp. 1–10.

Headway Software Technologies Ltd. Structure101. https://structure101.com. Accessed 21 September 2020.

Hopkins, W. (2009). JSR 375: JavaTM EE security API. https://jcp.org/en/jsr/detail?id=375. Accessed 16 July 2020.

Hunsaker, C. (2015). REST vs SOAP: When is REST better for web service interfaces? https://stormpath.com/blog/rest-vs-soap. Accessed 14 August 2020.

Ibrahim, A., S. Bozhinoski, and A. Pretschner (2019). Attack graph generation for microservice architecture. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, New York, NY, USA, pp. 1235–1242. Association for Computing Machinery.

Jendrock, E., I. Evans, D. Gollapudi, K. Haase, C. Srivathsa, R. Cervera-Navarro, and W. Markito (2014, May). Working with realms, users, groups, and roles. In *The Java EE 7 tutorial: volume 2*. Addison-Wesley Professional.

Jia, Y. and M. Harman (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering 37*(5), 649–678. https://doi.org/10.1109/TSE.2010.62.

Keivanloo, I., C. K. Roy, and J. Rilling (2012). Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *Proceedings of the 6th International Workshop on Software Clones*, IWSC '12, Piscataway, NJ, USA, pp. 36–42. IEEE Press.

Keivanloo, I., C. K. Roy, and J. Rilling (2014). Sebyte: Scalable clone and similarity search for bytecode. *Science of Computer Programming 95*, 426–444. Special Issue on Software Clones (IWSC'12).

Khomh, F., M. Di Penta, and Y. Gueheneuc (2009). An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pp. 75–84.

Kratzke, N. and P.-C. Quint (2017). Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. *Journal of Systems and Software 126*, 1–16.

Kumar, K. S. and D. Malathi (2017, April). A novel method to find time complexity of an algorithm by using control flow graph. In *2017 International Conference on Technical Advancements in Computers and Communications (ICTACC)*, pp. 66–68.

Lau, D. (2018). An abstract syntax tree generator from java bytecode. https://github.com/davidlau325/BytecodeASTGenerator. Accessed 27 March 2020.

Le, D. M., P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic (2015). An empirical study of architectural change in open-source software systems. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 235–245.

Lee, S., J. Jo, and Y. Kim (2015). Method for secure restful web service. In *2015 IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS)*, pp. 77–81. https://doi.org/10.1109/ICIS.2015.7166573.

Li, W., Y. Lemieux, J. Gao, Z. Zhao, and Y. Han (2019). Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 122–1225.

Logarix. AI Reviewer. http://aireviewer.com. Accessed 21 September 2020.

Macia, I., J. Garcia, P. Daniel, A. Garcia, N. Medvidovic, and A. Staa (2012, 03). Are automatically-detected code anomalies relevant to architectural modularity? an exploratory analysis of evolving systems. *AOSD'12 - Proceedings of the 11th Annual International Conference on Aspect Oriented Software Development*.

Makai, M. (2019). Object-relational mappers (orms). https://www.fullstackpython.com/object-relational-mappers-orms.html. Accessed 27 March 2020.

Mantyla, M., J. Vanhanen, and C. Lassenius (2003). A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pp. 381–384.

Marinescu, R. (2005). Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 701–704.

Marinescu, R. and D. Ratiu (2004). Quantifying the quality of object-oriented design: the factor-strategy model. In *11th Working Conference on Reverse Engineering*, pp. 192–201.

Márquez, G. and H. Astudillo (2019). Identifying availability tactics to support security architectural design of microservice-based systems. In *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA '19, New York, NY, USA, pp. 123–129. Association for Computing Machinery.

Mathew, A. P. and F. A. Capela (2019). An analysis on code smell detection tools. *17th SC@ RUG 2019-2020*, 57.

Mayer, B. and R. Weinreich (2018). An approach to extract the architecture of microservice-based software systems. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pp. 21–30.

Mayr, H. C., C. Kop, and D. Esberger (2007). Business process modeling and requirements modeling. In *First International Conference on the Digital Society (ICDS'07)*, pp. 8–8.

McGraw, G. (2004). Software security. *IEEE Security Privacy 2*(2), 80–83. https://doi.org/10.1109/MSECP.2004.1281254.

Mo, R., Y. Cai, R. Kazman, and L. Xiao (2015). Hotspot patterns: The formal definition and automatic detection of architecture smells. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pp. 51–60.

Moha, N. (2007). Detection and correction of design defects in object-oriented designs. In *OOPSLA '07*.

Moha, N., Y. Gueheneuc, L. Duchien, and A. Le Meur (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering 36*(1), 20–36.

Moha, N., Y. Gueheneuc, and P. Leduc (2006). Automatic generation of detection algorithms for design defects. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pp. 297–300.

Moha, N., Y.-G. Guéhéneuc, A.-F. Meur, L. Duchien, and A. Tiberghien (2010, 05). From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing 22*.

Moha, N., Y.-G. Guéhéneuc, A.-F. L. Meur, and L. Duchien (2008). A Domain Analysis to Specify Design Defects and Generate Detection Algorithms. In *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering*, Volume 4961 of *Lecture Notes in Computer Science*, pp. 276–291. Springer International Publishing.

Mohanty, H., J. Mohanty, and A. Balakrishnan (2016). *Trends in Software Testing*. Springer Singapore. https://doi.org/10.1007/978-981-10-1415-4.

Montesi, F. and J. Weber (2016, September). Circuit breakers, discovery, and api gateways in microservices. *arXiv:1609.05830 [cs]*. arXiv: 1609.05830.

Moonen, L. and A. Yamashita (2012). Do code smells reflect important maintainability aspects? In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, USA, pp. 306–315. IEEE Computer Society.

Mordani, R. (2016). JSR 250: Common Annotations for the JavaTM Platform. https://jcp.org/en/jsr/detail?id=250. Accessed 27 March 2020.

NGINX, Inc. (2015). The Future of Application Development and Delivery Is Now Containers and Microservices Are Hitting the Mainstream. https://www.nginx.com/resources/library/app-dev-survey. Accessed 27 March 2020.

Oberle, D., A. Eberhart, S. Staab, and R. Volz (2004). Developing and managing software components in an ontology-based application server. In H.-A. Jacobsen (Ed.), *Middleware 2004*, Berlin, Heidelberg, pp. 459–477. Springer Berlin Heidelberg.

Oliveira, R. (2016). When more heads are better than one? understanding and improving collaborative identification of code smells. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 879–882.

Omicini, A., A. Ricci, and M. Viroli (2005). Rbac for organisation and security in an agent coordination infrastructure. *Electronic Notes in Theoretical Computer Science 128*(5), 65–85. Proceedings of the 2nd International Workshop on Security Issues in Coordination Models, Languages, and Systems (SecCo 2004).

Oracle (2020). Securing RESTful web services using Java security annotations. https://docs.oracle.com/middleware/1212/wls/RESTF/secure-restful-service.htm#RESTF280. Accessed 14 August 2020.

Pallets Projects (2020). Flask documentation quickstart (1.1.x). https://flask.palletsprojects.com/en/1.1.x/quickstart. Accessed 14 August 2020.

Peters, R. and A. Zaidman (2012). Evaluating the lifespan of code smells using software repository mining. In *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 411–416.

Pigazzini, I., F. A. Fontana, V. Lenarduzzi, and D. Taibi (2020). Towards microservice smells detection. In *The 42nd International Conference on Software Engineering*, pp. 0.

PMD (2019). PMD: An extensible cross-language static code analyzer. https://pmd.github.io. Accessed March 27, 2020.

Pugh, B. (2015). Findbugs. http://findbugs.sourceforge.net. Accessed March 27, 2020.

Quay (2020). Clair: Vulnerability static analysis for containers. https://github.com/quay/clair. Accessed 11 December 2020.

Rademacher, F., S. Sachweh, and A. Zündorf (2020a). A modeling method for systematic architecture reconstruction of microservice-based software systems. In S. Nurcan, I. Reinhartz-Berger, P. Soffer, and J. Zdravkovic (Eds.), *Enterprise, Business-Process and Information Systems Modeling*, Cham, pp. 311–326. Springer International Publishing.

Rademacher, F., S. Sachweh, and A. Zündorf (2020b). A modeling method for systematic architecture reconstruction of microservice-based software systems. In S. Nurcan, I. Reinhartz-Berger, P. Soffer, and J. Zdravkovic (Eds.), *Enterprise, Business-Process and Information Systems Modeling*, Cham, pp. 311–326. Springer International Publishing.

Rademacher, F., J. Sorgalla, P. Wizenty, S. Sachweh, and A. Zündorf (2020). *Graphical and Textual Model-Driven Microservice Development*, pp. 147–179. Cham: Springer International Publishing.

Rao, A. and K. Reddy (2008, 03). Detecting bad smells in object oriented design using design change propagation probability matrix. *Lecture Notes in Engineering and Computer Science 2168*.

Red Hat Inc (2020a). Keycloak. https://www.keycloak.org. Accessed 14 August 2020.

Red Hat Inc (2020b). Keycloak authorization services guide. https://www.keycloak.org/docs/latest/authorization_services. Accessed 14 August 2020.

Reeshti, R. Sehgal, R. Nagpal, and D. Mehrotra (2019). Measuring code smells and anti-patterns. In *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*, pp. 311–314.

Ribeiro, J. C. B., F. F. de Vega, and M. Zenha-Rela (2007). Using dynamic analysis of java bytecode for evolutionary object-oriented unit testing. In *25th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)*.

Richards, M. (2015, February). Layered architecture. In *Software Architecture Patterns*. O'Reilly Media, Inc.

Roy, C. K., J. R. Cordy, and R. Koschke (2009, May). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program. 74*(7), 470–495.

Sae-Lim, N., S. Hayashi, and M. Saeki (2017). How do developers select and prioritize code smells? a preliminary study. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 484–488.

Salah, T., M. Jamal Zemerly, Chan Yeob Yeun, M. Al-Qutayri, and Y. Al-Hammadi (2016). The evolution of distributed systems towards microservices architecture. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, pp. 318–325.

Sandhu, R. S. (1990). Separation of duties in computerized information systems. In *DBSec*, pp. 179–190. Citeseer.

Sandhu, R. S., E. J. Coyne, H. L. Feinstein, and C. E. Youman (1996, February). Role-based access control models. *Computer 29*(2), 38–47. https://doi.org/10.1109/2.485845.

Sandhu, R. S. and P. Samarati (1994). Access control: principle and practice. *IEEE Communications Magazine 32*(9), 40–48. https://doi.org/10.1109/35.312842.

Scarioni, C. and M. Nardone (2019). Spring Security Architecture and Design. In C. Scarioni and M. Nardone (Eds.), *Pro Spring Security: Securing Spring Framework 5 and Boot 2-based Java Applications*, pp. 69–116. Berkeley, CA: Apress.

Scheepers, M. J. (2014). Virtualization and containerization of application infrastructure : A comparison.

Selim, G. M. K., K. C. Foo, and Y. Zou (2010, October). Enhancing source-based clone detection using intermediate representation. In *2010 17th Working Conference on Reverse Engineering*, pp. 227–236.

Sharma, T. (2016). Designite - A Software Design Quality Assessment Tool.

Singh, S. and N. Singh (2016). Containers docker: Emerging roles future of cloud technology. In *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, pp. 804–807.

Smid, A., R. Wang, and T. Cerny (2019). Case study on data communication in microservice architecture. In *Proceedings of the Conference on Research in Adaptive and Convergent Systems*, RACS '19, New York, NY, USA, pp. 261–267. Association for Computing Machinery.

Son, S., K. S. Mckinley, and V. Shmatikov (2013). Fix me up: Repairing access-control bugs in web applications. In *In Network and Distributed System Security Symposium*.

SpotBugs (2019). Spotbugs: Find bugs in java programs. https://spotbugs.github.io. Accessed March 27, 2020.

Srivastava, V., M. D. Bond, K. S. McKinley, and V. Shmatikov (2011). A security policy oracle: Detecting security holes using multiple api implementations. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, New York, NY, USA, pp. 343–354. Association for Computing Machinery.

Steinegger, R., P. Giessler, B. Hippchen, and S. Abeck (2017, 04). Overview of a domain-driven design approach to build microservice-based applications. SOFTENG: The Third International Conference on Advances and Trends in Software Engineering.

Su, F.-H., J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara (2016). Code relatives: Detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, New York, NY, USA, pp. 702–714. ACM.

Sudhakar, A. (2011). Techniques for securing rest. *CA Technology Exchange*, 32.

Suryanarayana, G., G. Samarthyam, and T. Sharma (2014). *Refactoring for Software Design Smells: Managing Technical Debt* (1st ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Svacina, J., J. Simmons, and T. Cerny (2020). Semantic code clone detection for enterprise applications. In *Proceedings of the The 35th ACM/SIGAPP Symposium On Applied Computing*, ACM SAC '20, pp. 1–3. ACM.

Swinhoe, D. (2020). The 15 biggest data breaches of the 21st century. https://www.csoonline.com/article/2130877/the-biggest-data-breaches_of-the-21st-century.html. Accessed 14 August 2020.

Syaikhuddin, M. M., C. Anam, A. R. Rinaldi, and M. E. B. Conoras (2018). Conventional software testing using white box method. *Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control 3*(1), 65–72.

Tahir, A., J. Dietrich, S. Counsell, S. Licorish, and A. Yamashita (2020). A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. *Information and Software Technology 125*, 106333.

Taibi, D. and V. Lenarduzzi (2018, May). On the definition of microservice bad smells. *IEEE Software 35*(3), 56–62.

Tarjan, R. (1971). Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pp. 114–121.

The Kubernetes Authors (2021). Minikube: Run kubernetes locally. https://github.com/kubernetes/minikube. Accessed 14 April 2021.

Thio, L. (2020). Role-based Authorization — Flask-User v1.0 documentation. https://flask-user.readthedocs.io/en/latest/authorization.html. Accessed 14 August 2020.

Tihomirovs, J. and J. Grabis (2016). Comparison of soap and rest based web services using software evaluation metrics. *Information Technology and Management Science 19*(1), 92–97.

Torres, A., R. Galante, and M. S. Pimenta (2009). Towards a uml profile for model-driven object-relational mapping. In *2009 XXIII Brazilian Symposium on Software Engineering*, pp. 94–103. IEEE.

Trnka, M., J. Svacina, T. Cerny, E. Song, J. Hong, and M. Bures (2020, June). Securing internet of things devices using the network context. *IEEE Transactions on Industrial Informatics 16*(6), 4017–4027.

Tufano, M., F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk (2015). When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Volume 1, pp. 403–414.

Tufano, M., F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. De Lucia, and D. Poshyvanyk (2017). When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering 43*(11), 1063–1088.

Van Emden, E. and L. Moonen (2002). Java quality assurance by detecting code smells. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, WCRE '02, USA, pp. 97. IEEE Computer Society.

VMware Inc (2020). Building a RESTful web service. https://spring.io/guides/gs/rest-service. Accessed 14 August 2020.

Von Zitzewitz, A. (2019). Mitigating technical and architectural debt with sonargraph. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pp. 66–67.

Vural, H., M. Koyuncu, and S. Guney (2017). A systematic literature review on microservices. In O. Gervasi, B. Murgante, S. Misra, G. Borruso, C. M. Torre, A. M. A. Rocha, D. Taniar, B. O. Apduhan, E. Stankova, and A. Cuzzocrea (Eds.), *Computational Science and Its Applications – ICCSA 2017*, Cham, pp. 203–217. Springer International Publishing.

Wagh, D. K. and R. Thool (2012, 07). A comparative study of soap vs rest web services provisioning techniques for mobile host. *Journal of Information Engineering and Applications 2*, 12–16.

Walker, A. and T. Cerny (2020). On cloud computing infrastructure for existing code-clone detection algorithms. *SIGAPP Appl. Comput. Rev. 20*(1), 5–14.

Walker, A., D. Das, and T. Cerny (2020, January). Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study. *Applied Sciences 10*(21), 7800.

Walker, A., J. Svacina, J. Simmons, and T. Cerny (2020a). On automated role-based access control assessment in enterprise systems. In K. J. Kim and H.-Y. Kim (Eds.), *Information Science and Applications*, Singapore, pp. 375–385. Springer Singapore.

Walker, A., J. Svacina, J. Simmons, and T. Cerny (2020b). On automated role-based access control assessment in enterprise systems. In K. J. Kim and H.-Y. Kim (Eds.), *Information Science and Applications*, Singapore, pp. 375–385. Springer Singapore.

Walls, C. (2016). *Spring Boot in Action* (1st ed.). USA: Manning Publications Co.

Wolff, E. (2016). *Microservices: Flexible Software Architectures*. CreateSpace Independent Publishing Platform.

Xu, D., L. Thomas, M. Kent, T. Mouelhi, and Y. Le Traon (2012). A model-based approach to automated testing of access control policies. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, New York, NY, USA, pp. 209–218. Association for Computing Machinery.

Yamashita, A. and S. Counsell (2013). Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software 86*(10), 2639–2653.

Yamashita, A. and L. Moonen (2013a). Do developers care about code smells? an exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 242–251.

Yamashita, A. and L. Moonen (2013b). Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 682–691.

Zhou, X., X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao (2018). Benchmarking microservice systems for software engineering research. In M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman (Eds.), *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pp. 323–324. ACM.