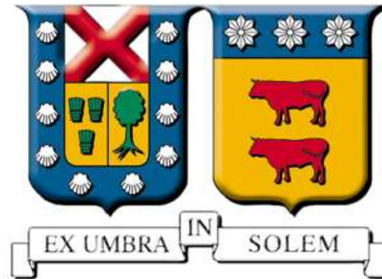
Universidad Técnica Federico Santa María
Departamento de Informática
Valparaíso - Chile



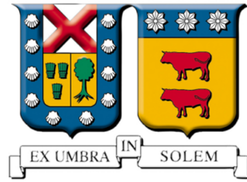
SUPPORTING TRIAGE OF MICROSERVICE SECURITY SMELLS, USING TRADE-OFF ANALYSIS

FRANCISCO LEONARDO PONCE MELLA
francisco.ponceme@usm.cl

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR IN INFORMATICS ENGINEERING

SUPERVISOR DR. HERNÁN ASTUDILLO
CO-SUPERVISOR DR. JACOPO SOLDANI AND DR. ANTONIO BROGI

2024



Universidad Técnica Federico Santa María
Departamento de Informática

TÍTULO DE LA TESIS:
SUPPORTING TRIAGE OF MICROSERVICE SECURITY SMELLS, USING TRADE-OFF ANALYSIS

AUTOR:
FRANCISCO LEONARDO PONCE MELLA

TRABAJO DE TESIS, presentado en cumplimiento parcial de los requisitos para la obtención del Grado de Doctor en Ingeniería Informática de la Universidad Técnica Federico Santa María.

COMISIÓN DE EXAMEN

Prof. Dr. Hernán Astudillo R.: Director de Tesis
(Depto. Informática – UTFSM)
Prof. Dr. Antonio Brogi.: Co- Director de Tesis
(Universidad de Pisa, Italia)
Prof. Dr. Jacopo Soldani.: Co- Director de Tesis
(Universidad de Pisa, Italia)
Prof. Dr. Mauricio Araya.: Correferente Interno
(Depto. Electrónica- UTFSM)
Prof. Dr. Santiago Matalonga.: Correferente Externo Internacional
(University of the West of Scotland)
Prof. PhD Riccardo Scandariato.: Correferente Externo Internacional
(Hamburg, University of Technology, Germany)
Prof. Dr. Mauricio Solar F.: Presidente Comisión
(Depto. Informática – UTFSM)

Valparaíso, Chile, Junio 2024.

Dedicated to my loved ones

“My philosophy is that worrying means you suffer twice.”

Newt Scamander

Acknowledgments

First, I want to express my gratitude to my supervisors, doctors Hernán Astudillo, Jacopo Soldani, and Antonio Brogi. During all these years I not only received their excellent advice and help professionally, but they also taught me the social aspect of this career and inspired me with their passion for research. Definitely without their guidance and help this thesis would not have been possible.

I also extend my thanks to all the colleagues I collaborated with during my PhD. All those incredible people who also influenced my professional career. I wholeheartedly wish you all the luck and success in your professional and personal development. A special mention to Pablo Cruz, Felipe Beroíza, Gastón Márquez, Philip Wizenty, and Florian Rademacher.

I also want to take this opportunity to thank all those people who helped me during my PhD. I especially want to thank Roberto Muñoz, because I still remember that conversation that ended with my enrollment in the doctoral program, and René Noël, Rodrigo Olivares, and Carlos Becerra who also had a positive influence on my professional development and I had the opportunity to work and collaborate with them.

I also want to thank my entire family for all their support and encouragement over the years. All my professional achievements have always been celebrated, and I have always received expressions of love and affection. A special mention to my parents Leonardo Ponce Donoso and María Mella Silva, and my sister and brother Valeska Ponce Mella and Leonardo Ponce Mella. Without their constant support, this thesis would not have been possible.

Last, but not least, I want to express my gratitude to a special person, Valentina Meléndez. All these years she has been there for me when I have needed it, in the good times celebrating me and encouraging me, but also in the bad times, listening to me and helping me get back on my feet. Without her unconditional love, this entire process would have been very different, so from the bottom of my heart, thank you very much, I love you!

Abstract

Microservice-based applications (MSAs) are pervading enterprise IT, as they enable building *cloud-native* applications that can fully exploit the capabilities of cloud computing, exhibiting distributed, dynamic, and fault-resilient behavior. MSAs introduce new security challenges, including the so-called *security smells*, i.e., symptoms of poor decisions that may impact MSA security. Security smell instances must be carefully checked, and possibly resolved via refactoring, to preempt authenticity, integrity, or confidentiality issues.

In this thesis ten smells for securing microservices are identified, and organized in a taxonomy, associating each security smell with the security properties it may violate and the refactorings enabling to mitigate its effects. Furthermore, this thesis also presents an end-to-end approach for resolving security smells in existing MSAs that automatizes smell detection and provides users with an interactive mechanism for smell resolution across the concerned MSA components.

On the other hand, choosing between tolerating a given microservice security smell instance and resolving it with a refactoring requires careful trade-off considerations, since both the smell and its refactoring may impact other quality attributes besides security, e.g., maintainability and performance. For example, the *centralized authorization* security smell harms the authenticity and time behavior of the MSA but favors its testability. Thus, resolving the security smell by applying the *use decentralized authorization* refactoring would favor the MSA authenticity and modularity, but would harm its testability and resource utilization. Making informed refactoring decisions requires assessing the trade-offs of impacts on multiple affected quality attributes.

This thesis also argues for trade-off analysis to help determine whether to keep a microservice security smell or to apply a refactoring, based on their positive/negative impacts on specific quality attributes and design soundness. The proposed method enacts and supports this trade-off analysis using *Softgoal Interdependency Graphs* (SIGs), a visual formalism that - in our case - enables a holistic view of the positive/negative impacts of microservice security smells and refactorings on software quality attributes and design soundness. Additionally, we systematically elicit possible impacts of smells and refactorings on applications' maintainability, performance efficiency, and adherence to microservices' key design principles, which were then validated through an online survey targeting experienced practitioners and researchers.

Since multiple security smell instances can affect multiple services in an MSA, architects must not only find trade-offs for each smell instance but also decide which smell instances to resolve first. Indeed, some smell instances may be more "urgent" than others because they affect services that implement core functionalities and/or quality attributes that are critical for a services effective functioning. Given the number of services forming an MSA, their quality requirements, and the multiple different impacts of security smells on quality attributes, it is inherently complex and costly to determine which security smell instances should be resolved first, being the most urgent.

Taking the above into consideration, this thesis also proposes a triage method to systematically associate security smell instances with "urgency codes", similar to what triage nurses do with patients who enter a hospital emergency room and describe their symptoms. The proposed method enables assigning each security smell instance (i.e., a security smell affecting a service in an MSA) an urgency code, which is computed by combining (i) the relevance of the service to the business, and (ii) the importance of the service quality attributes that are impacted by the smell instance. The method systematizes this process by assigning smell instances to urgency codes, which can be used by practitioners to decide which smell instances to resolve first (presumably, those with the highest urgency). The practical applicability of the proposed triage method is illustrated with a use case based on a third-party MSA, and its usefulness is evaluated with a controlled experiment involving 26 practitioners. Our results suggest that the proposed triage method eases the triage process and yields urgency codes on which practitioners are more confident.

Contents

1 Introduction **1**

1.1 Research objectives 3

1.2 Research contributions 3

1.3 Thesis structure 4

1.4 Related publications 6

2 Smells and Refactorings for Microservices Security **8**

2.1 Research Design 10

2.1.1 Search for Primary Studies 11

2.1.2 Selection of Primary Studies 12

2.1.3 Literature Analysis 15

2.1.4 Replication Package 16

2.2 Microservices Security Smells and Refactorings 16

2.2.1 Insufficient Access Control 19

2.2.2 Publicly Accessible Microservices 19

2.2.3 Unnecessary Privileges to Microservices 20

2.2.4 Own Crypto Code 21

2.2.5 Non-Encrypted Data Exposure 22

2.2.6 Hardcoded Secrets 22

2.2.7 Non-Secured Service-to-Service Communications 23

2.2.8 Unauthenticated Traffic 24
2.2.9 Multiple User Authentication 25
2.2.10 Centralized Authorization 26
2.2.11 Summary 27
2.3 Threats to Validity 28
2.3.1 Threats to External Validity 28
2.3.2 Threats to Construct and Internal Validity 29
2.3.3 Threats to Conclusions Validity 29
2.4 Related Work 30

3 Microservices Security: Bad vs. Good Practices 32
3.1 Research Design 33
3.1.1 Literature Selection 33
3.1.2 Literature Analysis 34
3.2 Bad vs. Good Practices for Microservices Security 35
3.2.1 Trust the Network vs. Zero-Trust Principle 36
3.2.2 No Layered Security vs. Defense-in-Depth 38
3.2.3 Non-Proactive vs. Proactive Security Measures 40
3.2.4 Non-scalable vs. Scalable Security Controls 41
3.3 Threats to Validity 43
3.4 Related Work 44

4 Model-Driven End-to-End Resolution of Security Smells in Microservice Architectures 45
4.1 Background 46
4.2 End-to-End Smell Resolution 48
4.2.1 Reconstruction 49
4.2.2 Smell Detection 53
4.2.3 Smell Resolution 55

4.3 Validation . . . 57
4.3.1 Research Questions . . . 57
4.3.2 Validation Implementation . . . 57
4.3.3 Answer to Research Questions . . . 60
4.4 Threats to Validity . . . 61
4.5 Related Work . . . 62

5 Should microservice security smells stay or be refactored? 64
5.1 Background: SIGs . . . 65
5.2 Towards a SIG-based trade-offs analysis . . . 66
5.3 Illustrative example . . . 67
5.4 Softgoal Interdependency Graphs . . . 69
5.4.1 Insufficient Access Control . . . 69
5.4.2 Publicly Accessible Microservices . . . 70
5.4.3 Unnecessary Privileges to Microservices . . . 71
5.4.4 Non-Secured Service-to-Service Communications . . . 73
5.4.5 Unauthenticated Traffic . . . 74
5.4.6 Multiple User Authentication . . . 75
5.5 Related work . . . 76

6 To Security and Beyond: On The Impacts of Microservice Security Smells and Refactorings 77
6.1 Background . . . 78
6.1.1 Smells and Refactorings for Microservice Security . . . 78
6.1.2 Softgoal Interdependency Graphs . . . 80
6.2 Survey Design . . . 81
6.2.1 Literature selection . . . 82
6.2.2 Thematic analysis . . . 82
6.2.3 Online survey . . . 83

6.3 Impacts of Security Smells . . . 83
6.3.1 Insufficient Access Control (IAC) . . . 84
6.3.2 Publicly Accessible Microservices (PAM) . . . 86
6.3.3 Unnecessary Privileges to Microservices (UPM) . . . 90
6.3.4 Non-Secured Service-to-Service Communications (NSC) . . . 93
6.3.5 Unauthenticated Traffic (UNT) . . . 95
6.3.6 Multiple User Authentication (MUA) . . . 97
6.3.7 Centralized Authorization (CNA) . . . 99
6.4 Threats to Validity . . . 103
6.5 Related Work . . . 104

7 Triaging Microservice Security Smells, with TriSS **105**
7.1 Motivating Scenario . . . 106
7.2 The TriSS Triage Method . . . 107
7.3 Use case . . . 109
7.4 Experimental evaluation . . . 112
7.4.1 Experimental Study Design . . . 112
7.4.2 Experimental Study Execution . . . 113
7.4.3 Triage Results . . . 114
7.4.4 Answers to Research Questions . . . 115
7.5 Discussion . . . 118
7.6 Related work . . . 119

8 Conclusions **120**
8.1 Summary of contributions . . . 121
8.2 Possible directions for future work . . . 122

List of Figures

1.1 Thesis workflow. 4

2.1 Taxonomy of microservices security (a) properties, (b) smells, and (c) refactorings. For the sake of readability, the association between security properties and smells is represented by aligning the corresponding boxes, whilst that between smells and refactorings is represented with arrows. 16

2.2 Coverage of the microservices security smells in the selected studies. 17

3.1 Literature review process. 34

3.2 (a) Bad and (b) good practices for microservice security. 36

4.1 Approach to resolve security smells in MSA. 49

4.2 MSA-based student management application. 50

4.3 Structure of the reconstruction framework. 50

4.4 Example of recovered domain concepts. 51

4.5 Example of recovered interface specifications. 52

4.6 Example of our approach to recover deployment specifications from docker artifacts (a) into a LEMMA operation model with microservice deployments (b) and infrastructure components (c). 53

4.7 LEMMA technology and operation models for security smell detection. 54

4.8 LEMMA Eclipse editor presenting the student microservice operation model with the *Publicly Accessible Microservices* smell detection. 55

4.9 LEMMA Eclipse editor presenting the student microservice operation model with the *Publicly Accessible Microservices* smell with the resolution strategy. 55

4.10 Artifacts created by the *Deployment_Base* and Zuul code generators. 56

4.11 Architecture of the Lakeside Mutual application. 58

4.12 Accessibility of running microservices in the original Lakeside Mutual application. 59

4.13 Accessibility of running microservices in the refactored version obtained with our approach. 59

5.1 An example of SIG. 65

5.2 SIG displaying the impact of keeping the *centralized authorization* smell and its corresponding refactoring on the considered softgoals. 68

5.3 SIG displaying the impact of keeping the *Insufficient Access Control* security smell and its corresponding refactoring. 70

5.4 SIG displaying the impact of keeping the *Publicly Accessible Microservices* security smell and its corresponding refactoring. 71

5.5 SIG displaying the impact of keeping the *Unnecessary Privileges to Microservices* security smell and its corresponding refactoring. 72

5.6 SIG displaying the impact of keeping the *Non-Secured Service-to-Service Communications* security smell and its corresponding refactoring. 73

5.7 SIG displaying the impact of keeping the *Unauthenticated Traffic* security smell and its corresponding refactoring. 74

5.8 SIG displaying the impact of keeping the *Multiple User Authentication* security smell and its corresponding refactoring. 75

6.1 Example of a Softgoal Interdependency Graphs 81

6.2 Research process. Grey boxes denote steps to elicit and assess possible impacts of smells and refactorings identified in our previous work [105] (whose steps are in white) 82

6.3 Agreement with the possible impacts of IAC 84

6.4 Distribution of agreement with IAC-related statements. 85

6.5 SIG that provides a holistic view of the validated impacts related to IAC 86

6.6 Agreement with the possible impacts of PAM 88

6.7 Distribution of agreement with PAM-related statements. 88
6.8 SIG that provides a holistic view of the validated impacts related to PAM 89
6.9 Agreement with the possible impacts of UPM 91
6.10 Distribution of agreement with UPM-related statements. 91
6.11 SIG that provides a holistic view of the validated impacts related to UPM 92
6.12 Agreement with the possible impacts of NSC 93
6.13 Distribution of agreement with NSC-related statements. 94
6.14 SIG that provides a holistic view of the validated impacts related to NSC 94
6.15 Agreement with the possible impacts of UNT 95
6.16 Distribution of agreement with UNT-related statements. 96
6.17 SIG that provides a holistic view of the validated impacts related to UNT 96
6.18 Agreement with the possible impacts of MUA 97
6.19 Distribution of agreement with MUA-related statements. 98
6.20 SIG that provides a holistic view of the validated impacts related to MUA 98
6.21 Agreement with the possible impacts of CNA 100
6.22 Distribution of agreement with CNA-related statements 101
6.23 SIG that provides a holistic view of the validated impacts related to CNA 102

7.1 Lakeside Mutual MSA. 106
7.2 SIG displaying the impact of keeping a *publicly accessible microservices* smell. 108
7.3 TriSS' color-coded triage matrix: *H* (high); *h* (medium to high); *M* (medium); *m* (low to medium); *L* (low); *l* (none to low); ∅ (none). 109
7.4 Distribution of urgency codes assigned by practitioners with 0-2 years of experience. Light and dark blue correspond to without and with the use of TriSS respectively. 114
7.5 Distribution of urgency codes assigned by practitioners with 3+ years of experience. Light and dark blue correspond to without and with the use of TriSS, respectively. 114
7.6 Subjects answers on (a) ease of triaging and (b) confidence on assigned urgency, both *with vs. without* TriSS. 116

7.7 Changes in (a) assigned urgency codes and (b) confidence on intuitively assigned urgency codes. 117

List of Tables

2.1 PICO terms of our research problem. 11

2.2 References, publication years, colours (viz., *white* or *grey* literature), and types (viz., *blog post*, *book*, *book chapter*, *conference* paper, *journal* article, or *video*) of the selected primary studies. 14

2.3 Classification of the selected studies according to the taxonomy in Figure 2.1. 18

2.4 Summary of the classification of selected primary studies. 27

3.1 Reference, year of publication, colour, and type of selected primary studies. 35

3.2 Coverage of the identified bad/good practices for microservices security in the selected primary studies. 37

4.1 Results from the reconstruction process. 58

6.1 Possible impacts of IAC 84

6.2 Possible impacts of PAM 87

6.3 Possible impacts of UPM 90

6.4 Possible impacts of NSC 93

6.5 Possible impacts of UNT 95

6.6 Possible impacts of MUA 97

6.7 Possible impacts of CNA 99

7.1 Security smell instances affecting services of the Lakeside Mutual MSA. 107

7.2 Service relevance and QA's importance for the motivating scenario. 110
7.3 Urgency codes assigned to security smell instances in Lakeside Mutual. 111
7.4 Triage tasks defined for the controlled experiment. 113

Chapter 1

Introduction

Microservice-based architectures (MSAs) enable building *cloud-native* applications [39] that can fully exploit the capabilities of cloud computing, by exhibiting distributed, dynamic, and fault-resilient behavior [149]. Microservice-based applications are essentially service-oriented applications adhering to an extended set of design principles [157], e.g., shaping services around business concepts, decentralization, and ensuring the independent deployability and horizontal scalability of microservices, among others. Such additional principles make microservice-based applications not only service-oriented but also highly distributed and dynamic. As a result, other than the classical security issues and best practices for service-oriented applications, MSAs introduce new security challenges [134], including the so-called *security smells*.

A security smell can be observed in an application, being it a possible symptom of a bad decision (though often unintentional) while designing or developing the application, which may impact the overall application's security [103]. The effects of security smells can be mitigated by refactoring the application or the services therein, while at the same not changing the functionalities offered to external clients.

This thesis reports on what is being said by practitioners and researchers about known security smells and refactorings enabling to mitigate their effects. It complements these results by also reporting on what is being said about bad/good practices for securing microservices.

Therefore, this thesis also presents an end-to-end approach for resolving security smells in existing MSAs that automatizes smell detection and provides users with an interactive mechanism for smell resolution across the concerned MSA components.

Choosing between tolerating a given microservice security smell and resolving it with a refactoring requires careful trade-offs consideration, since both the smell and its refactoring may impact other quality attributes besides security, e.g., maintainability and performance [107]. Suppose, for instance, that an

MSA is affected by the *centralized authorization* security smell, with only one of its components being used to centrally authorize the requests sent by external clients [105]. The requests exchanged among the microservices forming the MSA are instead trusted by design, with no further authorization controls. This makes the MSA itself prone to, e.g., confused deputy attacks, which may compromise its authenticity. The latter can be avoided by applying the *use decentralized authorization* refactoring, namely by enforcing fine-grained authorization controls for each of its microservices [105]. The *centralized authorization* security smell harms the authenticity and time behavior of the MSA but favors its testability. Thus, resolving the smell by applying the *use decentralized authorization* refactoring would favor the MSA authenticity and modularity, but would harm its testability and resource utilization. Making informed refactoring decisions requires assessing the trade-offs of impacts on several (possibly many) quality attributes [104].

This thesis introduces a first support for analyzing the possible trade-offs related to keeping a security smell in a microservice-based application or applying some refactoring. More precisely, it introduces a method to enact such trade-off analysis using *Softgoal Interdependency Graphs* (SIGs) [25], which provide a visual and holistic panorama of the positive/negative impacts of each security smell and refactorings on each software quality attribute and each microservices' key design principle.

Since multiple security smell instances can affect multiple services in an MSA, architects must not only find trade-offs for each smell instance but also decide which smell instances to resolve first [14]. Indeed, some smell instances may be more "urgent" than others because they affect services that implement core functionalities and/or quality attributes that are critical for a service to effectively serve its clients. Given the number of services forming an MSA, their several quality requirements, and the multiple different impacts of security smells on quality attributes, it gets inherently complex and costly to determine which security smell instances should be resolved first, being the most urgent.

Taking the above into consideration, this thesis also proposes TriSS (*Triage Security Smells*), a method that systematically associates security smell instances with "urgency codes", similar to what triage nurses do with patients that enter a hospital emergency room and describe their symptoms. TriSS enables assigning each security smell instance (i.e., a smell affecting a service in an MSA) an urgency code, which is computed by combining (i) the relevance of the service to the business, and (ii) the importance of the service quality attributes that are impacted by the smell instance. TriSS systematizes this process by assigning smell instances to urgency codes, which practitioners can use to decide which smell instances to resolve first (presumably, those with the highest urgency).

1.1 Research objectives

The primary aim of this research is to support systematic, repeatable reasoning on whether/how to refactor microservices' security smells in a microservices-based application. In particular, this thesis aims at advancing the state-of-the-art focusing on the following research questions:

- O_1: What are the effects of bad security decisions for microservice-based applications?
- O_2: How to detect/resolve security smells affecting a microservice-based application?
- O_3: What are the impacts of microservices' security smells on quality attributes besides security?
- O_4: How to determine which microservices' security smells to resolve first?

1.2 Research contributions

We hereby list the research contributions that are going to be presented in this thesis, which are presented by associating them with the research objective described above.

- O_1: We conducted a Multivocal Literature Review of the existing white and grey literature on securing microservice. As a result, we present a taxonomy of microservices' security smells and the refactorings that allow mitigating their effects. Additionally, we also introduce a taxonomy of microservices' security bad and good practices, linking each bad practice to the microservice security smell(s) that signal it.
- O_2: We propose an end-to-end model-driven approach for resolving security smells in existing MSAs that automatizes smell detection and provides users with an interactive mechanism for smell resolution across the concerned MSA components. Our approach recovers the software application architectural design using LEMMA models. These models address different viewpoints in the MSA development process and contain, among others, information about security aspects of Java-based MSAs and to automatically detect the two most recognized security smells for microservices (viz., *Publicly Accessible Microservices* and *Insufficient Access Control*).
- O_3: We systematically identified 42 potential impacts of microservices security smells. These were subsequently validated through an online survey involving practitioners and researchers working with MSAs. We introduce a visual and holistic panorama of the positive and negative impacts of microservice security smells and refactorings using Softgoal Interdependence Graphs. Finally, we have also

developed a software tool that contains the Softgoal Interdependence Graphs and that provides information about the impacts of microservices security smells and refactoring.

- O_4: We introduce the notion of *urgency* for microservice security smell instances, and propose a method, TriSS, to systematically *triage* them. TriSS allows assigning to each security smell instance an urgency code based on combining the service's business relevance and the smell's impact on security and other quality attributes, e.g., performance and maintainability.

1.3 Thesis structure

The thesis is organized as illustrated in Figure 1.1, which shows the logical workflow of the contributions presented in this thesis. Further information about the content of each chapter is given below:

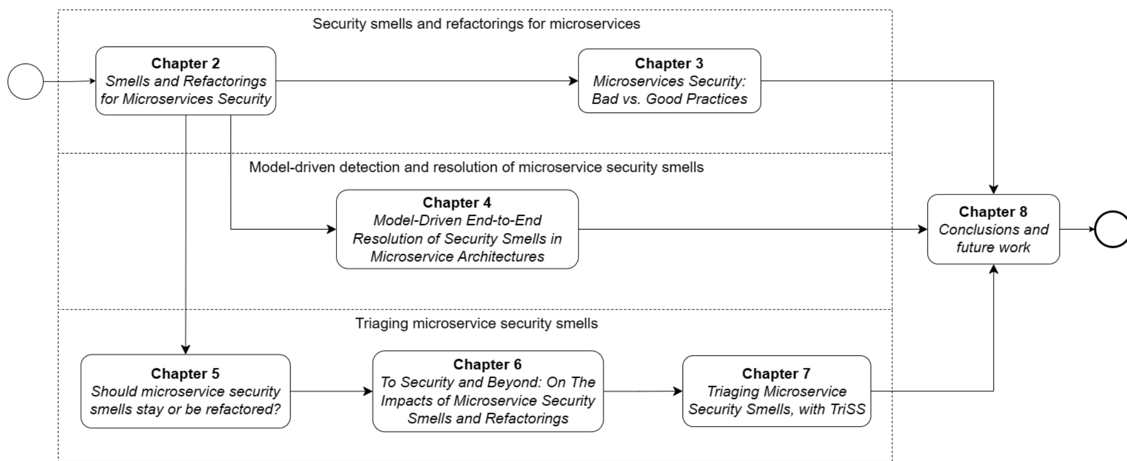


Figure 1.1: Thesis workflow.

Chapter 2 presents the results of a multivocal literature review of the existing white and grey literature on securing microservices. Ten bad smells for securing microservices are identified, which we organized in a taxonomy, associating each smell with the security properties it may violate and the refactorings enabling to mitigate its effects. The security smells and the corresponding refactorings have pragmatic value for practitioners, who can exploit them in their daily work on securing microservices.

The smells and refactorings presented in Chapter 2 were published in [105], which appeared in the "Journal of Systems and Software" and presented as journal-first paper at ICSA 2023.

Chapter 3 presents the results of a multivocal literature review that analyzes 44 primary studies discussing bad and good practices for microservice security. We were able to identify four bad and six good practices,

and to associate each bad practice with specific security smell(s) that signal it and with good practice(s) that avoid incurring in it.

The microservice security bad and good practices presented in Chapter 3 has been presented and published at ECSA Tracks and Workshops 2022 [106].

Chapter 4 proposes a end-to-end approach for resolving security smells in existing MSAs. The presented approach extends a modeling ecosystem for MSAs with (i) reconstruction capabilities that automatically map MSA source code to viewpoint-specific architecture models; (ii) validations that detect security smells from reconstructed models; and (iii) model refactorings that support the interactive resolution of security smells and solutions' reflection back to source code.

The content of Chapter 4 is the result of joint research with the IDiAL Institute (University of Applied Sciences and Arts Dortmund, Germany), which has produced three scientific publications. The first one has been presented and published at the 18th International Conference on Software Technologies (ICSOFT), as a conference paper under the title "Towards resolving security smells in microservices, model-driven" [151], and has been awarded the ICSOFT 2023 Best Student Paper award. The second one is an extension of the first paper that has been sent for publication under the title "Model-Driven Security Smell Resolution in Microservice Architecture Using LEMMA" for a book in the CCIS Series published by Springer. Finally, the third one has been accepted for publication at the 14th International Conference on Cloud Computing and Services Science (CLOSER 2024), as a conference paper under the title "Model-Driven End-to-End Resolution of Security Smells in Microservice Architectures".

Chapter 5 argues for a trade-off analysis method to help determine whether to keep a security smell or to apply a refactoring, based on their positive or negative impacts on specific quality attributes and design soundness. The method enacts and supports this trade-off analysis using *Softgoal Interdependency Graphs* (SIGs). We also illustrate our method with a detailed analysis of a well-known security smell and its possible refactoring, and apply it to other six security smells.

The method presented in Chapter 5 has been presented and published at ECSA 2022 [104].

Chapter 6 explores the impacts of microservice security smells –and of the refactorings known to mitigate their effects– beyond security. In particular, we systematically elicit possible impacts of smells and refactorings on applications' maintainability, performance efficiency, and adherence to microservices' key design principles. We then validate the elicited impacts through an online survey targeting experienced practitioners and researchers. We also provide a holistic view of these impacts, through Softgoal Interdependency Graphs (SIGs).

The impacts of microservice security smells presented in Chapter 6 have been presented and published

at CLEI 2023 [107]. We were also invited to submit an extended version of this paper to the CLEI'23 Special Issue in the CLEI Electronic Journal and we are currently waiting for its publication.

Chapter 7 introduces the notion of *urgency* for microservice security smell instances, and propose the TriSS method to *triage* them. TriSS enables assigning to each security smell instance with an urgency code based on combining the service's business relevance and the smell's impact on security and other quality attributes, e.g., performance and maintainability. The practical applicability of TriSS is illustrated with a use case based on a third-party MSA, and its usefulness is evaluated with a controlled experiment involving 26 practitioners. The experiment's results suggest that TriSS eases the triage process and yields urgency codes on which practitioners are more confident.

The TriSS method presented in Chapter 7 has been accepted for publication at the International Conference on Evaluation and Assessment in Software Engineering (EASE) 2024 [108].

Chapter 8 summarises our research contributions and provides perspectives for future work.

1.4 Related publications

1. Ponce, F., Márquez, G., and Astudillo, H. (2019). Migrating from monolithic architecture to microservices: A Rapid Review. In 2019 38th International Conference of the Chilean Computer Science Society (SCCC) (pp. 1-7). IEEE, https://doi.org/10.1109/SCCC49216.2019.8966423

2. Ponce, F. (2021). Towards resolving security smells in microservice-based applications. In Advances in Service-Oriented and Cloud Computing: International Workshops of ESOCC 2020, Heraklion, Crete, Greece, September 28–30, 2020, pp. 133-139. Springer International Publishing. https://doi.org/10.1007/978-3-030-71906-7_11

3. Ponce, F., Soldani, J., Astudillo, H., and Brogi, A. (2022). Smells and refactorings for microservices security: A multivocal literature review. Journal of Systems and Software, vol. 192, p. 111 393, 2022, issn: 0164-1212. https://doi.org/10.1016/j.jss.2022.111393

4. Ponce, F., Soldani, J., Astudillo, H., and Brogi, A. (2022). Microservices Security: Bad vs. Good Practices. In Tracks and Workshops of the European Conference on Software Architecture - ECSA (pp. 337-352). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-031-36889-9_23

5. Ponce, F., Soldani, J., Astudillo, H., and Brogi, A. (2022). Should microservice security smells stay or be refactored? towards a trade-off analysis. In European Conference on Software Architecture - ECSA (pp. 131-139). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-031-16697-6_9

6. Wizenty, P., Ponce, F., Rademacher, F., Soldani, J., Astudillo, H., Brogi, A., and Sachweh, S. (2023). Towards resolving security smells in microservices, model-driven. In 18th International Conference on Software Technologies (ICSOFT). https://doi.org/10.5220/0012049800003538

7. Ponce, F., Soldani, J., Taramasco, C., Astudillo, H., and Brogi, A. (2023). To Security and Beyond: On The Impacts of Microservice Security Smells and Refactorings. In 2023 XLIX Latin American Computer Conference (CLEI) (pp. 1-10). IEEE. https://doi.org/10.1109/CLEI60451.2023.10346146

8. Wizenty, P., Ponce, F., Rademacher, F., Soldani, J., Astudillo, H., Brogi, A., and Sachweh, S. (2024). Model-Driven Security Smell Resolution in Microservice Architecture Using LEMMA. ICSOFT 2022. Communications in Computer and Information Science, vol 2104. Springer, Cham. https://doi.org/10.1007/978-3-031-61753-9_3

9. Wizenty, P., Ponce, F., Rademacher, F., Soldani, J., Astudillo, H., Brogi, A., and Sachweh, S. (2024). Model-Driven End-to-End Resolution of Security Smells in Microservice Architectures. In Proceedings of the 14th International Conference on Cloud Computing and Services Science - CLOSER. SciTePress. https://doi.org/10.5220/0012671700003711

10. Ponce, F., Soldani, J., Taramasco, C., Astudillo, H., and Brogi, A. (2024). Beyond Security: Understanding the Multiple Impacts of Security Smells for Microservices. CLEI'23 Special Issue in the CLEI Electronic Journal (submitted for publication).

11. Ponce, F., Soldani, J., Taramasco, C., Astudillo, H., and Brogi, A. (2024). Triaging Microservice Security Smells, with TriSS. In 28th International Conference on Evaluation and Assessment in Software Engineering (EASE). https://doi.org/10.1145/3661167.3661282

Chapter 2

Smells and Refactorings for Microservices Security

Microservices are pervading for architecting enterprise applications nowadays, with big players in IT (e.g., Amazon, Netflix, Spotify, and X) already delivering their core businesses through microservices [139]. This is mainly because microservice-based applications are cloud-native, thus better exploiting the potentials of cloud hosting, and since they fully twin with DevOps and continuous delivery practices [8]. Microservices also bring various other advantages, such as ease of deployment, resilience, and scalability [88]. Together with their gains, however, microservices bring also some pains, and securing microservice-based applications is certainly one of those [134].

Microservice-based applications are essentially service-oriented applications adhering to an extended set of design principles [157], e.g., shaping services around business concepts, decentralisation, and ensuring the independent deployability and horizontal scalability of microservices, among others. Such additional principles make microservice-based applications not only service-oriented, but also highly distributed and dynamic. As a result, other than the classical security issues and best-practices for service-oriented applications, microservices bring new security challenges [134]. For instance, being much more distributed than traditional service-oriented applications, microservice-based application expose more endpoints, thus enlarging the surface prone to security attacks [67]. It is also crucial to establish trust among the microservices forming an application and to manage distributed secrets, whereas these concerns are of much less interest in traditional web services or monolithic applications [154]. Another example follows from the many communications occurring among the microservices forming an application, which —if not properly handled— can result in message data being intercepted and in malicious users inferring business operations

from such data [155].

Whereas there exist quite much literature on securing microservices, different pieces of literature deal with different aspects of security. As a result, the currently available information on securing microservices is scattered among a considerable amount of books, blog posts, research papers, videos, and whitepapers. This hampers consulting the body of knowledge on the topic, both for academic researchers wishing to delineate novel research directions and solutions for securing microservices, and for practitioners daily needing to secure microservice-based applications. To help both researchers and practitioners, this research tries to organise the scattered knowledge on securing microservices, by answering to the following two research questions:

(RQ1) What are the smells indicating possible security violations in microservice-based applications?

(RQ2) How to refactor microservice-based applications to mitigate the effects of security smells therein?

A security smell can be observed in an application, being it a possible symptom of a bad decision (though often unintentional) while designing or developing the application, which may impact on the overall application's security [103]. The effects of security smells can be mitigated by refactoring the application or the services therein, while at the same not changing the functionalities offered to external clients. Even if applying refactorings requires some efforts to application developers, it is known that they can help mitigating the effects of smells to improve the overall system quality [2, 11].

To answer to our research questions, in this chapter we report on what is being said by researchers and practitioners about known security smells and refactorings enabling to mitigate their effects. We provide a sort of "snapshot" of the state of the art and practice on such smells and refactorings, keeping the level of detail at which they are discussed in literature. We indeed systematically analyzed the available literature on securing microservices to elicit security smells well-known among researchers and practitioners, which may possibly result in security violations in microservice-based applications. We also elicited the refactorings proposed by researchers and practitioners that are known to mitigate the effects of such security smells. In particular, following the recommendations by Garousi et al. [43], we captured both the state of the art and the state of practice in the field by conducting a multivocal review of the existing literature. We analysed both white literature (viz., peer-reviewed papers) and grey literature (viz., blog posts, industrial whitepapers, books, and videos). We carefully selected 58 primary studies published since 2011 (when microservices were first discussed in an industrial workshop [70]) until the end of 2020, which we then systematically analysed by following the guidelines for conducting systematic reviews [44, 63]. As a result, we obtained a taxonomy that organises the identified security smells together with all refactorings that should be enacted to mitigate their effects. Finally, we exploited the taxonomy to classify the selected primary studies, in

order to distill the actual recognition of the identified smells and of their corresponding refactorings.

In this chapter, we illustrate the results of our multivocal review. We first present the taxonomy of security smells, including ten security smells and ten refactorings, organised around three security properties defined in the ISO/IEC 25010 standard for software quality [55], viz., integrity, authenticity, and confidentiality. Then we discuss each security smell by illustrating why it can possibly violate the security property it is associated with. For each smell, we also discuss the corresponding refactorings recommended by practitioners, by also explaining how such refactoring enables mitigating its effects.

We believe that the results of our study can provide benefits to both researchers and practitioners interested in microservices. A systematic presentation of the state of the art and practice on well-known security smells for microservices provides a body of knowledge to develop new techniques and solutions, to investigate and experiment research implications, and to set future research directions. At the same time, it can help practitioners to better understand the currently most recognised security smells for microservices, and to enact the corresponding refactorings to mitigate their effects. This is of pragmatic value for practitioners, who can use our study as a starting point for securing their microservice-based applications, as well as a reference in their day-by-day work with microservices.

2.1 Research Design

We identified microservices' security smells and refactorings by conducting a multivocal review [43], namely by searching and classifying both white and grey literature on the topic. Considering grey literature however comes with an intrinsic difficulty. Grey literature is indeed intended as materials and research produced by organizations outside the traditional commercial or academic publishing distribution channels, e.g., technical, research, or project reports, working/white papers, government documents, or videos and evaluations. The use of grey literature is hence risky, since there is often little or no scientific factual representation of data or analyses presented in grey literature itself [37], and because it lacks independent reviews assessing its quality [44]. On the other hand, a growing interest around using grey literature for helping software engineering practitioners, as well as combining it with white literature to determine the state of the art and practice around a topic is gaining a considerable interest in the field of software engineering [43, 134].

Given the above, and with the aim of maximizing the validity of our study, we followed a systematic approach based on that by Kitchenham and Charters [63] for conducting systematic literature reviews in software engineering. We first defined the PICO terms for defining the goal and scope of our study (Table 2.1), and we then enacted a systematic search and classification of primary studies, considering both white literature and grey literature. As for grey literature, following the guidelines by Garousi et al [43]

Table 2.1: PICO terms of our research problem.

Concern	Explanation
Population	(RQ1) microservices' security smells, (RQ2) refactorings for mitigating security smells in microservice-based applications
Intervention	characterization, internal/external validation, data extraction, synthesis
Comparison	comparison based on mapping primary studies to a taxonomy
Outcome	a taxonomy of security smells and refactorings for microservice-based applications

and the lessons learned in our former grey literature review [134], we varied the standard approach by Kitchenham and Charters [63] as follows:

- We exploited general web search engines for searching for grey literature.
- We adopted the effort bounded stopping criteria.
- We fixed the type of relevant grey literature to blog posts, whitepapers, industrial magazines, and videos.

The first variation was motivated by the fact that grey literature is publicly available on the web, but typically not indexed by indexing databases (as in the case of white literature). The latter two variations were instead aimed to limit the number of relevant search hits to consider to a manageable amount, and following the recommendations outlined by Garousi et al [44].

We hereafter detail the systematic approach that we followed, by starting from the structuring of the search string and by also describing the triangulation and inter-rater reliability assessment trials that we ran to enforce the validity of our findings.

2.1.1 Search for Primary Studies

The structuring of the search string was done by following the guidelines provided by Kitchenham and Charters [63]. We identified the search string guided by the *Population* terms of our research problem (Table 2.1), with search keywords taken from each aspect of our research problem. We anyhow decided —differently from what indicated by Kitchenham and Charters [63]— to not restrict our focus to specific research settings, as research settings are often not explicitly described in grey literature [44, 134]. As a result, our search string was formed by the following terms:

```
(microservice*) ∧ (security*) ∧
(smell* ∨ antipattern* ∨ bad practice* ∨ pitfall* ∨ refactor* ∨ reengineer* ∨ restructure*)
```

(where '*' matches lexically related terms). The search was restricted to primary studies published since the beginning of 2011 (when microservices were first discussed in an industrial workshop [70]) until the end of 2020.

The search of white literature was carried out by matching the search string against the title and abstract of the white literature indexed by the following databases: ACM Digital Library, DBLP, Google Scholar, IEEE Xplore, ISI Web of Science, Science Direct, Scopus, and SpringerLink. Given the recency of microservice-related studies and the well-known concerns with indexing, Google Scholar played a key role for the initial selection before the inclusion and exclusion stage.

The search for grey literature was instead carried out by exploiting the features natively supported by web search engines, which match search strings against the whole content of websites (including, e.g., Medium, Stack Overflow, and YouTube, often used by practitioners to share their experiences). The search engines we employed were the following: Google, Bing, and DuckDuckGo. Given the amount of results returned by the different combinations of the keywords in the search string (often, hundreds of thousands), and given that each search was repeated on each considered search engine, we adopted the effort bounded stopping criteria suggested by Garousi et al. [44]. In particular, for each search on each search engine, we considered the top 250 search hits (e.g., the first 25 pages of results returned by the Google search engine).

The above search resulted in around 6000 search hits, 5250 out of which were the grey literature matches identified with web search engines. These included multiple hits for the same piece of literature, hitted on different indexing platforms or web search engines, and in a high number of irrelevant studies. This held especially in the case of matching grey literature, since web search engines indeed look for search strings over the whole pages they index. We hence enacted a sample selection based on control factors, which we describe in the following section.

2.1.2 Selection of Primary Studies

We enacted a first refinement of the search hits based on the following inclusion criteria:

(i_1) A study is to be selected if it is written in English.

(i_2) A study is to be selected if it qualifies as white literature, or as a blog post, whitepaper, industrial magazine publication, or video authored by a practitioner.[1]

(i_3) A study is to be selected if it focuses on microservices.

[1] We combined i_2 with the following criteria to ensure that grey literature was authored by a practitioner. We indeed only considered the primary studies published on IT companies' websites, by the social media accounts of IT companies, or by people whose LinkedIn profiles indicate that they work in IT since 5+ years.

(i_4) A study is to be selected if it focuses on security.

The above criteria were designed to focus on studies written in English, either being white literature or grey literature of the form we decided to consider, whilst at the same time dealing with microservices' security. Such criteria enabled us to reduce the search hits to 136 candidate primary studies, which we further refined to align with our research questions, viz., eliciting well-known security smells in microservices and the refactorings enabling to resolve such smells. We indeed screened the 136 candidate primary studies by means of the following two additional inclusion criteria:

(i_5) A study is selected if it presents *at least one security smell* possibly resulting in a violation of a security property defined by the ISO/IEC 25010 software quality standard [55], such as confidentiality, integrity, and authenticity.

(i_6) A study is selected if it presents *at least one refactoring* for mitigating the effects of a security smell, even if the latter is not explicitly mentioned.

In particular, i_5 was checked by determining whether a primary study discusses a possible security smell, viz., a security issue deriving from bad decisions while designing or developing microservices. i_6 was instead checked by determining whether a primary study discusses a technical solution for resolving the occurrence of one such possible security smell. To reduce possible biases in applying i_5 and i_6, we enacted thematic coding [10]. The two criteria were applied to all 136 candidate primary studies by two researchers, who independently coded such studies as to be included/excluded. The inter-rater agreement on inclusion/exclusion of candidate primary studies was then measured by adopting the Krippendorff Kα coefficient, which measures the agreement between two lists of codes applied as part of content analysis [65]. The initial agreement on inclusion/exclusion of studies already reached 88.24%, already above the typical reference score of 80%. The 16 inclusion/exclusion mismatches were then resolved by triangulation, with other two researchers independently coding the corresponding 16 primary studies. A joint discussion session was then organised to agree on whether to finally include each of the 16 primary studies based on the four independent codings.

Finally, we decided to keep only one instance of each representative study, in case they were exactly replicated across other pieces of literature, as this often happen in the case of grey literature (e.g., we removed [41] and [116] from consideration, as they were replicating the blog posts [42] and [115], respectively).

As a result, 58 primary studies were selected to be analysed further. The selected primary studies are listed in Table 2.2 by providing a reference to their full bibliographic information available in the references of this thesis. The table also classifies each selected primary study by publication year, colour, and type.

Table 2.2: References, publication years, colours (viz., *white* or *grey* literature), and types (viz., *blog post*, *book*, *book chapter*, *conference* paper, *journal* article, or *video*) of the selected primary studies.

Ref.	Year	Colour	Type
[88]	2015	grey	book
[67]	2015	grey	blog post
[100]	2016	grey	blog post
[89]	2016	grey	video
[52]	2016	grey	book
[153]	2016	grey	book
[35]	2016	white	journal
[42]	2017	grey	blog post
[141]	2017	grey	blog post
[131]	2017	grey	blog post
[12]	2017	grey	blog post
[27]	2017	grey	blog post
[80]	2017	grey	blog post
[79]	2017	grey	blog post
[125]	2017	grey	blog post
[73]	2017	grey	video
[56]	2017	grey	book
[20]	2017	grey	book
[156]	2017	grey	book
[154]	2018	white	conference
[19]	2018	grey	blog post
[31]	2018	grey	blog post
[32]	2018	grey	blog post
[49]	2018	grey	blog post
[62]	2018	grey	blog post
[66]	2018	grey	blog post
[46]	2018	grey	video
[57]	2018	grey	video
[38]	2018	grey	book

Ref.	Year	Colour	Type
[82]	2018	grey	book
[54]	2018	white	book chapter
[120]	2018	white	conference
[85]	2018	white	conference
[123]	2019	grey	blog post
[1]	2019	grey	blog post
[15]	2019	grey	blog post
[68]	2019	grey	blog post
[132]	2019	grey	blog post
[136]	2019	grey	blog post
[148]	2019	grey	blog post
[33]	2019	grey	video
[128]	2019	grey	video
[96]	2019	grey	video
[126]	2019	grey	book
[24]	2019	grey	whitepaper
[18]	2019	white	conference
[90]	2019	white	conference
[86]	2019	white	journal
[115]	2020	grey	blog post
[58]	2020	grey	blog post
[59]	2020	grey	blog post
[83]	2020	grey	blog post
[91]	2020	grey	blog post
[114]	2020	grey	blog post
[129]	2020	grey	blog post
[130]	2020	grey	book
[78]	2020	white	conference
[117]	2020	white	book

Notably, the colour of 40 out of the 58 selected primary studies is grey, again witnessing the importance of grey literature in distilling the state of practice on microservices, as already noticed in previous reviews [134, 87].

It is also worth noting that, despite we searched for primary studies published from 2011 to 2020, only primary studies published from 2015 satisfied our selection criteria, therefore getting included in our analysis. A possible reason for this is that microservices started to spread mainly after Lewis and Fowler formalised the microservice-based architectural style in their blog post [70], dated 2014. Security smells and refactorigs were then getting discussed in grey and white literature only after early adopters of the microservice-based architectural style started experiencing security issues in their applications. The first primary studies actually sharing security smells and refactorings were indeed published only a year after Lewis and Fowler's blog post [70], namely in 2015.

2.1.3 Literature Analysis

To obtain the findings discussed in Section 2.2 we again adopted thematic coding [10] and Krippendorff $K\alpha$-based inter-rater reliability assessment [65]. The selected primary studies were subject to annotation and labelling with the goal of identifying the security smells and refactoring emerging from the analysed text. This process of analysis was executed in parallel over two 50% splits of the selected primary studies, to ensure avoidance of observer bias. The coders of the two splits were then inverted and an inter-rater evaluation was enacted between the two emerging lists of security smells and refactorings. Inter-rater reliability was then measured by applying the $K\alpha$ coefficient to measure the agreement among the emerging lists of security smells and of their corresponding refactorings by the two independent observers who individually coded 100% of the selected primary studies. The result of applying $K\alpha$ to measure the agreement between the two lists amounted to 91.90% agreement, above the typical reference score of 80%.

To further mitigate possible biases, we also enacted a triangulation step. Other two researchers were indeed involved in cross-checking the coding, without any prior knowledge on the coding itself. Feedback sessions were then organized to discuss the cross-checking enacted by this two researchers, by firstly discussing their feedback separately with the coders of the two splits, and by then organizing a plenary feedback sessions where all researchers were involved. As a result of the feedback sessions, we obtained the final coding discussed in Section 2.2.

2.1.4 Replication Package

To encourage the repeatability of our study and the verifiability of our findings, a replication package is publicly available on GitHub[2]. The replication package contains the intermediary artifacts and the final results of our study. The selected white and grey literature can instead be accessed online, by recovering the bibliographic information from the references listed in Table 2.2.

2.2 Microservices Security Smells and Refactorings

Figure 2.1 illustrates a taxonomy for the security smells pertaining to the considered security properties, and for the refactorings allowing to mitigate such smells. We obtained our taxonomy by following the guidelines for conducting systematic reviews in software engineering proposed by Kitchenham and Charters [63]:

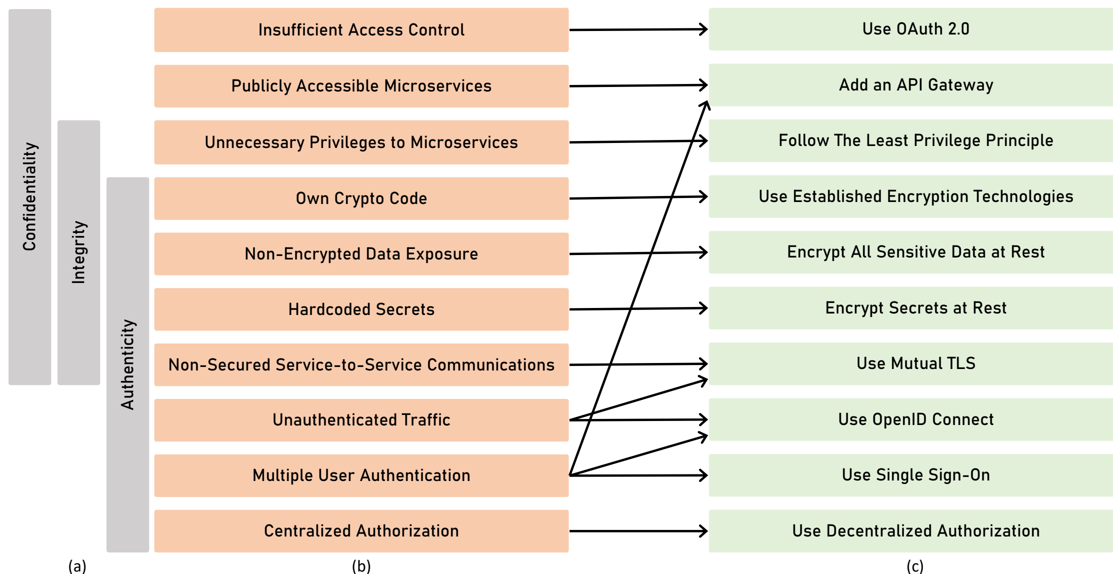


Figure 2.1: Taxonomy of microservices security (a) properties, (b) smells, and (c) refactorings. For the sake of readability, the association between security properties and smells is represented by aligning the corresponding boxes, whilst that between smells and refactorings is represented with arrows.

1. We identified the security smells by performing a first scan of the selected primary studies.

2. We excerpted the refactorings directly from the selected primary studies after additional scans.

The obtained security smells and refactorings were then manually organized to obtain a taxonomy. A first version of the taxonomy was obtained by grouping the security smells based on the security properties

[2] https://github.com/ms-security/smells-replication-package

they pertain to, by taking the security properties defined in the ISO/IEC 25010 standard [55] as a reference. Out of the five properties defined in the ISO/IEC 25010, only three of them resulted to be directly corresponding to some of the identified security smells. These are confidentiality (viz., the degree to which a product or system ensures that data are accessible only to those authorized to have access), integrity (viz., the degree to which a system, product, or component prevents unauthorized modification of computer programs or data), and authenticity (viz., the degree to which the identity of a subject or resource can be proved to be the one claimed). The taxonomy of security properties, smells, and refactorings underwent various iterations among the researchers. This resulted in some corrections and amendments to the first version of the taxonomy, which resulted in the final version of the taxonomy displayed in Figure 2.1. In the taxonomy, the refactorings associated to a smell should *all* be applied to mitigate its effects.

As we outlined previously, we aim at providing a "snapshot" of the state of the art and practice on smells that may affect the security of microservice-based applications and on refactorings enabling to mitigate such smells' effects. Therefore, despite the smells and refactorings in the taxonomy may be known to appear in distributed systems, our objective here is to analyse on how and how much they affect microservices, by reporting on what researchers and practitioners state in the selected primary studies.

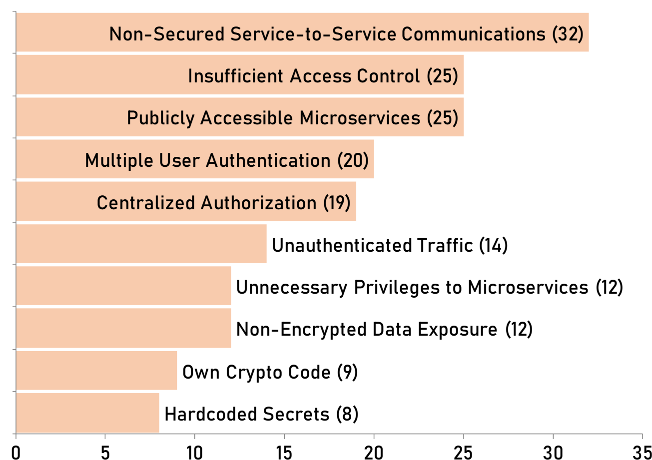


Figure 2.2: Coverage of the microservices security smells in the selected studies.

In this perspective, Table 2.3 shows the classification of all selected primary studies based on the taxonomy in Figure 2.1. The table provides a first overview of the coverage of the security smells over the selected primary studies, which is also displayed in Figure 2.2. We can observe that researchers and practitioners put more emphasis on securing, authenticating, and authorizing service interactions. Indeed, the more a security smell is related to service interactions, the higher is its coverage in the selected primary studies, as the authors consider it more impacting on the security of a microservice-based application. A reason for this is that microservices-based applications are heavily distributed, with many different services

Table 2.3: Classification of the selected studies according to the taxonomy in Figure 2.1.

	Insufficient access control	Publicly accessible microservices	Unnecessary privileges to microservices	Own crypto code	Non-encrypted data exposure	Hardcoded secrets	Non-secured service-to-service communications	Unauthenticated traffic	Multiple user authentication	Centralised authorization
[88]				✓	✓					✓
[67]	✓		✓				✓	✓		
[154]							✓			✓
[42]		✓		✓					✓	
[115]	✓					✓	✓			
[100]										✓
[89]	✓				✓		✓			
[52]	✓			✓	✓	✓		✓	✓	✓
[153]	✓						✓			
[35]							✓			
[141]	✓	✓		✓						
[131]		✓							✓	
[12]			✓						✓	
[27]			✓	✓	✓		✓		✓	
[80]	✓	✓	✓				✓			
[79]			✓				✓			
[125]						✓	✓			
[73]		✓				✓			✓	✓
[56]		✓	✓		✓		✓		✓	✓
[20]	✓	✓	✓				✓			
[156]	✓									✓
[19]								✓		
[31]	✓							✓		
[32]		✓					✓		✓	✓
[49]					✓		✓			
[62]	✓	✓		✓		✓				✓
[66]	✓	✓			✓					
[46]	✓	✓						✓	✓	
[57]			✓		✓	✓	✓			

	Insufficient access control	Publicly accessible microservices	Unnecessary privileges to microservices	Own crypto code	Non-encrypted data exposure	Hardcoded secrets	Non-secured service-to-service communications	Unauthenticated traffic	Multiple user authentication	Centralised authorization
[38]		✓					✓		✓	
[82]	✓	✓					✓	✓		✓
[54]	✓	✓					✓		✓	✓
[120]										✓
[85]	✓	✓						✓	✓	✓
[123]				✓		✓		✓		
[1]	✓	✓	✓					✓		
[15]			✓		✓		✓	✓		
[68]				✓			✓			
[132]	✓				✓		✓	✓	✓	
[136]	✓	✓								✓
[148]								✓		
[33]	✓						✓		✓	
[128]		✓					✓			
[96]	✓					✓				
[126]	✓						✓			
[24]							✓	✓	✓	
[18]		✓								
[90]							✓	✓	✓	✓
[86]	✓	✓							✓	✓
[58]		✓								
[59]		✓	✓				✓		✓	
[83]			✓				✓			
[91]	✓	✓		✓						
[114]							✓			
[129]										✓
[130]	✓	✓			✓		✓		✓	✓
[78]					✓		✓		✓	
[117]		✓					✓			✓

offering endpoints used by external clients and by the application services to interact with each other. The many endpoints and service interactions result in a broader attack surface, whose securing is fundamental [130]. Also, as we discuss hereafter, this is not only when an external service interacts with a frontend service of a microservice-based application, but also when the services forming a microservice-based application interact with each other. Securing, authenticating, and authorizing service interaction occurring between the boundaries of the network of services forming an application is indeed as important as securing, authenticating, and authorizing those coming from external services [38].

At the same time, Figure 2.2 also shows that all security smells in the taxonomy are significantly recognized by the authors of the selected primary studies, hence making them worthy to get discussed in detail. We hereafter provide a structured presentation of the identified smells. For each smell, we recall the *affected security properties* and provide a *description* of the smell, where we also illustrate how (as per what

emerges from the selected primary studies) the smell may possibly result in violating the corresponding security properties. We then illustrate the *suggested refactorings*, also classifying them based on their *type* (viz., use of established security protocols/libraries, design pattern implementation, or data encryption).

2.2.1 Insufficient Access Control

Affected Security Properties. CONFIDENTIALITY.

Description. The INSUFFICIENT ACCESS CONTROL smell occurs whenever a microservice-based application does not enact access control in one or more of its microservices, hence possibly violating the CONFIDENTIALITY of the data and business functions of the microservices where access control is lacking [141, 20]. If this smell is present, microservices can get exposed to, e.g., the "confused deputy problem", with attackers that can trick a service and get data that they should not be able to get [89]. At the same time, microservices are not suitable for traditional identity control models, since client details and permissions need to be verified as and when a request is sent [33], and they need a way to automatically decide whether to allow or reject calls between services [156]. In addition, microservices require development teams to establish and maintain the identity of users without introducing extra latency and contention with frequent calls to a centralized service [52].

Suggested Refactoring. From the 25 primary studies describing the INSUFFICIENT ACCESS CONTROL smell, it turns out that the effects of this smell can be mitigated if development teams USE OAUTH 2.0. Open Authorization (OAuth) 2.0 [53] is indeed the most used mechanism to manage access delegation. OAuth 2.0 is a token-based security framework for delegated access control that lets a resource owner grant a client access to a certain resource on their behalf. This access is for a limited time and with limited scope [82]. OAuth 2.0 is hence a natural candidate to enforce access control in microservice-based application at each level, therein included controlling the accesses to each microservice [68].

Refactoring Type. Use of established security protocols.

2.2.2 Publicly Accessible Microservices

Affected Security Properties. CONFIDENTIALITY.

Description. The PUBLICLY ACCESSIBLE MICROSERVICES smell occurs whenever the microservices forming an application are directly accessible by external clients [80, 91]. Each microservice can be accessed independently through its own API, and it needs a mechanism to ensure that each request is authenticated and authorized to access the set of functions requested [1]. However, if each microservice performs

authentication individually, the full set of a user's credentials is required each time, increasing the likelihood of CONFIDENTIALITY violations (e.g., with the exposure of long-term credentials) and reducing the overall maintainability and usability of the application. Also, each microservice is required to enforce the security policies that are applicable across all functions of the microservice-based application [82].

Suggested Refactoring. The PUBLICLY ACCESSIBLE MICROSERVICES smell is discussed in 25 out of the 58 selected primary studies. The authors of such 25 primary studies highlight that development teams can mitigate the effects of the PUBLICLY ACCESSIBLE MICROSERVICES smell if they ADD AN API GATEWAY. This enables to identify a set of microservices to be exposed through the newly introduced API Gateway, while the rest of the microservices are made unreachable from outside this domain. The API gateway centrally enforces security for all the requests entering the microservices application, including authentication, authorization, throttling, and message content validation for known security threats [130]. For instance, as we shall discuss in Section 2.2.9, the API gateway can be used to implement a single sign-on to the application. In addition, by using this approach development teams can also secure all microservices behind a firewall, allowing the API gateway to handle external requests and then communicate with the microservices behind the firewall [91]. Since the clients do not directly access the services, they cannot exploit the services on their own [59].

Refactoring Type. Design pattern implementation.

2.2.3 Unnecessary Privileges to Microservices

Affected Security Properties. CONFIDENTIALITY and INTEGRITY.

Description. The UNNECESSARY PRIVILEGES TO MICROSERVICES smell occurs when microservices are granted unnecessary access levels, permissions, or functionalities that are actually not needed by such microservices to deliver their business functions [20, 1]. The UNNECESSARY PRIVILEGES TO MICROSERVICES smell is described in 12 out of the 58 selected primary studies, all highlighting how such additional privileges given to microservices would potentially result in CONFIDENTIALITY and INTEGRITY issues. This happens, e.g., when a service can write or read data stored in databases or messages posted in messages queues, even if such databases or queues are not needed by the service to deliver its business function. As a result, resources are unnecessarily exposed, hence unnecessarily increasing the attack surface for CONFIDENTIALITY and INTEGRITY leaks: an intruder taking control of a service can indeed start reading or modifying all data and messages the service can access to [67].

Suggested Refactoring. The authors of the primary studies describing the UNNECESSARY PRIVILEGES TO MICROSERVICES smell also highlight that development teams can mitigate its effect if they FOLLOW

THE LEAST PRIVILEGE PRINCIPLE. The latter recommends that accounts and services should have the least amount of privileges they need to suitably perform their business function [56]. This principle should be used because even if a development team has ensured that the machine-to-machine communication is secured and there are appropriate safeguards with their firewall, there is always the risk that an attacker gets access to the microservice-based application [56, 15]. Development teams should indeed limit service privileges, by providing each service with access to only the resources they actually need to deliver their business functionalities.

Refactoring Type. Design pattern implementation.

2.2.4 Own Crypto Code

Affected Security Properties. CONFIDENTIALITY, INTEGRITY, and AUTHENTICITY.

Description. It is well-known that CONFIDENTIALITY, INTEGRITY, and AUTHENTICITY of data in software applications can get violated if development teams use their OWN CRYPTO CODE, viz., their own new encryption solutions and algorithms, unless they have been heavily tested [62, 91]. The authors of nine out of the 58 selected primary studies emphasize that microservice-based applications are not the exception: development teams that implement their own encryption solutions may end with improper solutions for securing microservices, which may result in possible CONFIDENTIALITY, INTEGRITY, and AUTHENTICITY issues. The use of OWN CRYPTO CODE may actually be even worse than not having any encryption solution at all, as it may produce a false sense of security [88].

Suggested Refactoring. In all the primary studies describing the OWN CRYPTO CODE smell, the authors point out that the way to mitigate this smell is through the USE OF ESTABLISHED ENCRYPTION TECHNOLOGIES. Development teams should indeed minimize the amount of encryption code they write on their own, but rather maximize the reuse of code coming from libraries that have already been heavily tested by the community [41, 91]. Development teams should also avoid the use of experimental encryption algorithms, as they may be subject to various kinds of vulnerabilities, which may be not yet known at the time of their use. Whatever are the programming languages used to implement the microservices forming an application, development teams always have access to reviewed and regularly patched implementations of established encryption algorithms [88].

Refactoring Type. Use of established security libraries.

2.2.5 Non-Encrypted Data Exposure

Affected Security Properties. CONFIDENTIALITY, INTEGRITY, and AUTHENTICITY.

Description. The NON-ENCRYPTED DATA EXPOSURE smell occurs when a microservice-based application accidentally expose sensitive data, e.g., because it was stored without any encryption in the data storage, or because the employed protection mechanisms are affected by security vulnerabilities or flaws [88, 15]. When sensitive data is exposed, its CONFIDENTIALITY and INTEGRITY can get violated because it could be acquired or modified by an intruder who gets direct access to the microservices forming an application. As a result, the intruder may access to or manipulate, e.g., some credentials for accessing other systems or business-critical data [52].

Suggested Refactoring. From the 12 selected primary studies that describe the NON-ENCRYPTED DATA EXPOSURE smell, it emerges that development teams can mitigate the effects of this smell if they ENCRYPT ALL SENSITIVE DATA AT REST, viz., when data is not actively moving from device to device or network to network, e.g., when data is stored on a hard drive. The microservice-based architectural style enables development teams to separate functions from data in each of the microservices forming an application [89]. As a recommendation, all sensitive data should always be encrypted, and it should be decrypted only when it needs to be used. Most database management systems provide features for automatic encryption, and disk-level encryption features are available at the operating-system level [130]. Application-level encryption is another option, in which the application itself encrypts the data before passing it over to the file system or a database. Finally, if a caching technology is used and the data is encrypted in the database, then development teams need to ensure that the same level of encryption is applied to such caching technology [56]. At the same time, development teams should also keep in mind that encryption is a resource-intensive operation that could have a considerable impact on the application performance [130]. As not all data needs the same level of security, and since in a microservice-based application it is common to have multiple data stores, development teams must perform a proper analysis to identify the critical ones and encrypt them.

Refactoring Type. Data encryption.

2.2.6 Hardcoded Secrets

Affected Security Properties. CONFIDENTIALITY, INTEGRITY, and AUTHENTICITY.

Description. The HARDCODED SECRETS smell occurs when a microservice of an application has hardcoded credentials in its source code, or when credentials are hardcoded in the deployment scripts for a microservice-based application, e.g., as environment variables passing secrets in a Dockerfile or a Docker Compose file [52, 73]. Microservices are indeed likely have secrets to be used for communicating with

authorization servers and other services. These secrets might be an API key, a client secret, or credentials for basic authentication [116]. The authors of eight out of the 58 selected primary studies clearly state that development teams should never store sensitive keys and other information in environment variables. In the latter case, the CONFIDENTIALITY and INTEGRITY of secrets may get violated, as they could be accidentally exposed, e.g., since exception handlers may grab and send the corresponding information to a logging platform. In addition, since child processes duplicate the parent's environment on startup, child processes may be another source of exposure of secrets in application logs, or they may be the reason why secrets could be unintentionally accessed by other services. In all such cases, we would end up with CONFIDENTIALITY and INTEGRITY leaks [62, 123].

Suggested Refactoring. The authors of the 8 primary studies describing the potential security leaks due to HARDCODED SECRETS all agree on saying that development teams can mitigate the effects of this smell if they ENCRYPT SECRETS AT REST. This would indeed help achieving that only authorized resources have access to secrets. In doing so, development teams should also adopt the following best practices: they should never store credentials alongside applications [52] or in the repositories used to host their source code [115], nor they should exploit environment variables to pass secrets to applications [62].

Refactoring Type. Data encryption.

2.2.7 Non-Secured Service-to-Service Communications

Affected Security Properties. CONFIDENTIALITY, INTEGRITY, and AUTHENTICITY.

Description. This smell occurs whenever two microservices in an application interact without enacting a secure communication channel, even if they are within the same network [130, 78]. As microservice-based applications are highly distributed, communication interfaces and channels proliferate, hence increasing the overall application attack surface. Each API exposed by each microservice indeed constitutes a potential attack vector that could be exploited by a malicious intruder, as it does each communication channel between any two microservices [59]. Microservices often need to communicate with each other to perform their business functions, and —if the communication channel is not secured— the data transferred can be exposed to man-in-the-middle, eavesdropping, and tampering attacks. This could not only result in CONFIDENTIALITY issues for service-to-service communications, as it could also break their INTEGRITY and AUTHENTICITY, e.g., intruders could intercept the communication between two microservices and change the data in transit to their advantage [27, 130].

Suggested Refactoring. The authors of the 32 primary studies describing the NON-SECURED SERVICE TO SERVICE COMMUNICATIONS all agree on saying that this smell can be mitigated with the USE OF MU-

TUAL TRANSPORT LAYER SECURITY. Mutual TLS [127] is indeed a widely-accepted solution to secure service-to-service communications, which enables encrypting the data in transit and ensuring its integrity and confidentiality. This is going to protect the microservice-based application context from man-in-the-middle, eavesdropping, and tampering attacks providing a bidirectional encryption channel. In addition, when Mutual TLS is used to secure communications between two microservices, each microservice can legitimately identify the microservice it is talking to, which means that microservices can authenticate each other [54, 130]. Hence, whereas there exist other solutions for encrypting data in transit and ensuring its integrity, Mutual TLS is suggested by experienced practitioners as it also helps mitigating the UNAUTHENTICATED TRAFFIC smell (as we will discuss next).

Refactoring Type. Use of established security protocols.

2.2.8 Unauthenticated Traffic

Affected Security Properties. AUTHENTICITY.

Description. The UNAUTHENTICATED TRAFFIC smell occurs in a microservice-based application not only when there are unauthenticated API requests coming from external systems, but also when there are unauthenticated requests between the microservices of the application themselves [1, 15]. To ensure AUTHENTICITY in microservice-based applications, it is critical to ensure that each request is authenticated and authorized. Therefore, it is necessary to have mechanisms that allow the authentication of traffic coming from outside the application, as well as that corresponding to messages between its microservices. It is indeed crucial each microservice can authenticate each other, especially when the interactions are due to some user transactions and a microservice is passing the logged-in user context to another microservice [130]. The problem here is that no information is typically shared among microservices, and the user context must be passed explicitly from one microservice to another. The challenge is hence to build trust between two interacting microservices, in such a way that the receiving microservice can accept the user context passed from the calling one [130]. Therefore, a way is needed to verify that the user context (and, more generally, the data) passed between microservices has not been modified. If the traffic is not authenticated, there is actually no guarantee that this is the case, and microservices are exposed to security attacks that may result in, e.g., tampering with data, denial of service, or elevation of privileges [90].

Suggested Refactoring. The UNAUTHENTICATED TRAFFIC smell is discussed in 14 of the selected primary studies, whose authors highlight the refactorings to be enacted to mitigate this smell are to USE MUTUAL TLS and to USE OPENID CONNECT. As we already discussed how Mutual TLS [127] enables authentication between any two interacting services (see 2.2.7), we hereafter focus on the use of OpenID Connect.

OpenID Connect is the most used mechanism to manage user authentication. It is based on the use of an ID token, typically a JSON Web Token that contains authenticated user information, including user claims and other relevant attributes [130]. The user ID token enables microservices to verify the user identity based on the authentication performed by an authorization server, as well as to obtain basic profile information about the end-user in an interoperable and REST-like manner. OpenID Connect hence provides a distributed identity mechanism for traffic authentication, which is also recognized to be easy to use, self-contained, and easy to replicate [31].

Refactoring Type. Use of established security protocols.

2.2.9 Multiple User Authentication

Affected Security Properties. AUTHENTICITY.

Description. The MULTIPLE USER AUTHENTICATION smell occurs when a microservice-based application provides multiple access points to handle user authentication [27]. Each access point constitutes a potential attack vector that can be exploited by an intruder to authenticate as an end-user, and having multiple access points hence result in increasing the attack surface to violate AUTHENTICITY in a microservice-based application [59]. The use of multiple access points for user authentication also results in maintainability and usability issues, since user login is to be developed, maintained, and used in multiple parts of the application [78].

Suggested Refactoring. The MULTIPLE USER AUTHENTICATION smell is discussed in 21 of the selected primary studies, whose authors point out that development teams can mitigate the effects of this smell if they USE A SINGLE SIGN-ON. The single sign-on approach suggests having a single entry point to handle user authentication and to enforce security for all the user requests entering the microservice-based application [59]. This approach facilitates log storage and auditing tasks [86], allowing the detection of abnormal situations that may occur. Single sign-on can actually be achieved if (i) we ADD AN API GATEWAY acting as a single entry point to the application, if not already there, and if (ii) we USE OPENID CONNECT to share the user context among the microservices. Both refactorings not only help implementing the single sign-on to mitigate the effects of the MULTIPLE USER AUTHENTICATION smell, but also contribute mitigating the effects of other smells. Indeed, we already discussed how the use of an API gateway helps mitigating the PUBLICLY ACCESSIBLE MICROSERVICES smell, as well as how OpenID Connect enables mitigating the effects of the UNAUTHENTICATED TRAFFIC smell (in Sections 2.2.2 and 2.2.8, respectively).

Refactoring Type. Design pattern implementation (USE SINGLE SIGN-ON, ADD AN API GATEWAY) and use of established security protocols (USE OPENID CONNECT).

2.2.10 Centralized Authorization

Affected Security Properties. AUTHENTICITY.

Description. In a microservice-based application, authorization can be enforced at the “edge” of the application (e.g., with the API gateway), by each microservice of the application, or both. The CENTRALIZED AUTHORIZATION smell occurs when the microservice-based application only handles authorization in one component, typically at the “edge” of the application, while it does not enact any fine-grained authorization control at the microservices-level. Such a kind of centralized authorization diminishes the advantages of having a distributed solution, such as that given by microservices, and it reduces performances and efficiency, since the central authorization point tends to become a bottleneck [52, 130]. A CENTRALIZED AUTHORIZATION may also result in violating AUTHENTICITY in microservice-based applications. For instance, when authorization is managed only at the edge, microservices are exposed to the so-called “confused deputy problem”: they trust the gateway based on its mere identity, exposing the microservices in an application to misuse if they are compromised [85]. It is worth noting that the “confused deputy problem” here occurs in a different way from when there is INSUFFICIENT ACCESS CONTROL, as here a centralized authorization results in an invoked microservice trusting its invoker without enacting any further authorization. In the case of INSUFFICIENT ACCESS CONTROL, instead, it is the invoker that has access to too many resources, hence potentially exposing such resources to attacks.

Suggested Refactoring. The CENTRALIZED AUTHORIZATION smell is discussed in 19 out of the 58 selected primary studies. From such 19 primary studies, it emerges that development teams can refactor microservice-based application to mitigate the CENTRALIZED AUTHORIZATION smell by enacting a decentralized authorization approach. They can indeed USE DECENTRALIZED AUTHORIZATION by simply developing a token-based authorization mechanism. This is achieved by transmitting an access token together with each request to a microservice, and access to such microservice is granted to the caller only if a known and correct token is passed [120]. In this way, authorization can be enforced also at the microservices-level, as it gives each microservice more control to enforce its own access-control policies. Among the 19 primary studies discussing the USE DECENTRALIZED AUTHORIZATION refactoring, JSON Web Token (JWT) is the most used mechanism to implement such a refactoring. A JWT is a standard for safely passing claims or data attributed to a user within an environment [56], which ensures that a man in the middle cannot change its content because the issuer of the JWT actually signs it [130].

Refactoring Type. Design pattern implementation.

Table 2.4: Summary of the classification of selected primary studies.

Smell	Affected Security Properties	Refactoring	Refactoring Type	Coverage
INSUFFICIENT ACCESS CONTROL	CONFIDENTIALITY	USE OAUTH 2.0	use established security protocols	25 / 58
PUBLICLY ACCESSIBLE MICROSERVICES	CONFIDENTIALITY	ADD AN API GATEWAY	design pattern implementation	25 / 58
UNNECESSARY PRIVILEGES TO MICROSERVICES	CONFIDENTIALITY, INTEGRITY	FOLLOW THE LEAST PRIVILEGE PRINCIPLE	design pattern implementation	12 / 58
OWN CRYPTO CODE	CONFIDENTIALITY, INTEGRITY, AUTHENTICITY	USE ESTABLISHED ENCRYPTION TECHNOLOGIES	use established security libraries	9 / 58
NON-ENCRYPTED DATA EXPOSURE	CONFIDENTIALITY, INTEGRITY, AUTHENTICITY	ENCRYPT ALL SENSITIVE DATA AT REST	data encryption	12 / 58
HARDCODED SECRETS	CONFIDENTIALITY, INTEGRITY, AUTHENTICITY	ENCRYPT SECRETS AT REST	data encryption	8 / 58
NON-SECURED SERVICE-TO-SERVICE COMMUNICATIONS	CONFIDENTIALITY, INTEGRITY, AUTHENTICITY	USE MUTUAL TLS	use established security protocols	32 / 58
UNAUTHENTICATED TRAFFIC	AUTHENTICITY	USE MUTUAL TLS, USE OPENID CONNECT	use established security protocols	14 / 58
MULTIPLE USER AUTHENTICATION	AUTHENTICITY	USE SINGLE SIGN-ON, ADD AN API GATEWAY, USE OPENID CONNECT	design pattern implementation, use established security protocols	20 / 58
CENTRALIZED AUTHORIZATION	AUTHENTICITY	USE DECENTRALIZED AUTHORIZATION	design pattern implementation	19 / 58

2.2.11 Summary

Table 2.4 shows a summary of the smells and refactorings discussed in this section. This also includes the security properties affected by them and how often they were discuss on the selected primary studies (viz., their *coverage*).

The table again confirm what we observed earlier, namely that researchers and practitioners put more emphasis on the security smells and refactorings related to service interactions. Interaction-related smells can indeed affect confidentiality and authenticity in microservice-based applications, mainly because such application are heavily distributed. Many different services indeed offer endpoints used by external clients and by the application services to interact with each other, hence increasing the surface for potential security attacks [130]. Table 2.4 also shows that, to mitigate the effects of smells possibly affecting confidentiality and authenticity, researchers and practitioners mainly recommend to leverage of existing solutions, namely to implement well-known design patterns and to use established security protocols [54, 1].

Leveraging of existing solutions is also widely recommended to deal with integrity-related smells, with researchers and practitioners recommending to use established security protocols and libraries to mitigate their effects [91]. In this case, a significantly recognized solution is also to encrypt sensitive data [57].

2.3 Threats to Validity

Wohlin et al. [152] define the potential threats to the validity of studies in empirical software engineering, four of which also apply to our study. These are the threats to the *external*, *internal*, *construct*, and *conclusions* validity, which we discuss hereafter.

2.3.1 Threats to External Validity

The external validity concerns the applicability of a set of results in a more general context [152]. Since the primary studies considered by our multivocal review were selected from a very large extent of online sources, the security smells and the corresponding refactorings may only be partly applicable to the broad area of disciplines and practices on microservices. This may hence result in threatening the external validity of our study.

To reinforce the external validity of our findings, we organized multiple feedback sessions with all researchers during our analysis of the selected studies (Section 2.1.3). The discussions held within and after the feedback sessions resulted in qualitative data, which we exploited to fine-tune the taxonomy of security smells and refactorings resulting from our study, obtaining that presented in Section 2.2. We also prepared a replication package (Section 2.1.4) storing the artifacts produced during our study, to make them publicly available to all who wish to deepen their understanding on the data we produced. We believe that this helps in making our results and observations more explicit and applicable in practice.

As for the external validity of our study, one may also consider our selection criteria as “too restrictive”. However, such criteria enable focusing only on representative studies, viz., studies discussing at least a security smell or a corresponding refactoring. There is however a potential risk of having missed some relevant literature, as a study might not explicitly mention the security smells in our taxonomy (Figure 2.1). To mitigate this potential threat, we carefully checked both our selection criteria against each candidate study. We indeed checked whether a study was discussing the security issues connected to some security smell, and whether it was discussing the concrete changes to apply to a microservice-based application to mitigate the effect of such smell. This enabled us to select also those studies that were not explicitly referring to a smell or refactoring in our taxonomy, but rather reporting on the corresponding security issues or to-be-applied changes.

Finally, another potential threat to the external validity of our study is having missed relevant grey literature. Practitioners may indeed share knowledge by exploiting a different terminology than ours, e.g., a practitioner’s blog post may discuss some security smells or refactorings, without explicitly mentioning the terms “smell” or “refactor”. To mitigate this threat to validity, we included relevant synonyms in the search

string, and we exploited the features natively supported by search engines, such as including related terms in string-based searches.

2.3.2 Threats to Construct and Internal Validity

Wholin et al. [152] define the internal validity of studies as concerning the method employed to study and analyse data, therein included the potential types of bias involved. They instead define the construct validity as the generalizability of the constructs under study.

To mitigate the potential threats to the construct and internal validity of our study, we exploited theme coding and inter-rater reliability assessment (Sections 2.1.3 and 3.1.1). These helped limiting potential biases, such as observer and interpretation biases, hence helping us to enhance the validity of the analysis we performed on the data we retrieved. In addition, the taxonomy organising the emerging lists of smells underwent various iterations among all the researchers to further avoid bias by triangulation. The same process was applied to the actual classification of the selected primary studies and to the results of the analysis.

2.3.3 Threats to Conclusions Validity

The conclusions validity is defined by Wohlin et al. [152] as concerning the degree to which the conclusions of a study are reasonably based on the available data.

To mitigate potential threats to the conclusions validity of our study, we exploited the above described inter-rater reliability assessment to limit potential biases in our observations and interpretations. Additionally, the observations and conclusions discussed in this chapter were drawn independently by the authors of this research. They were then discussed and double-checked against the selected primary studies in two joint discussion sessions.

2.4 Related Work

Various secondary studies analyse and classify the state of the art and practice on microservices. For instance, Pahl and Jamshidi [94] elicit potential research directions on microservices, after discussing agreed and emerging concerns on microservices and positioning microservices with respect to existing cloud and container technologies. Taibi et al [138] instead report on common architectural patterns for microservices, by discussing the advantages, disadvantages, and lessons learned of each pattern. However, neither Pahl and Jamshidi [94] nor Taibi et al [138] provide an overview on the smells possibly resulting in security issues in microservice-based applications, nor on the ways to mitigate their effects of such smells.

Other noteworthy examples are the systematic grey literature review by Soldani et al [134] and the industrial surveys by Di Francesco et al [30] and Ghofrani and Lübke [47], which all provide an overview on the state of practice on microservices. Soldani et al [134] identify the technical and operational advantages and disadvantages of microservices, therein included the difficulties in securing microservice-based applications. Di Francesco et al [30] and Ghofrani and Lübke [47] instead illustrate the results of surveys they conducted with practitioners daily working with microservices, also resulting in distilling the advantages and disadvantages of microservices. The studies by Soldani et al [134], Di Francesco et al [30], and Ghofrani and Lübke [47] differ from ours in their objective: they report on challenges and advantages of microservices, whereas we focus on distilling the smells that may possibly result in security issues in microservices, as well as on how to mitigate their effects.

Berardi et al [13] instead first report on microservices' security, by analysing white literature to identify the research communities where this is most discussed. Berardi et al [13] also distill the currently known security attacks to microservice-based applications, as well as the countermeasures that have been proposed to secure such applications from such attacks. However, Berardi et al's review [13] differs from ours in the objectives. They indeed focus on existing solutions to secure microservices from possible attacks. We instead focus on known security smells, namely on bad decisions that may hamper the security of microservice-based applications, as well as on the refactorings that are know to mitigate their effects. Also, Berardi et al [13] consider only white literature on securing microservices, while we also consider grey literature to analyse both the state of the art and the state of practice on microservices' security smells.

In this perspective, the industrial survey reported by Taibi and Lenarduzzi [137] is a step closer to ours, as they first explicitly defined 11 microservice-specific bad smells. The smells defined by Taibi and Lenarduzzi [137] span from the design of microservice-based application to their actual development, and each smell is equipped with the best practices enabling to avoid incurring in such smell. Our results complement those by Taibi and Lenarduzzi [137], as none of the smells they define is about securing microservices. We

instead precisely distill the refactorings that enable mitigating the effects of well-known security smells in microservice-based applications.

Similar considerations apply to the secondary studies presented by Bogner et al [16], Carrasco et al [21], and Neri et al [87], which all distill architectural smells for microservices. Bogner et al [16] present a systematic literature review identifying and documenting architectural smells in SOA-based architectural styles, including microservices. Although the main focus of their review is on the broader SOA, several smells apply also to microservices. Carrasco et al [21] and Neri et al [87] instead explicitly focus on microservices, with two multivocal reviews distilling architectural smells for microservices as well as architectural refactorings enabling to resolve such smells. It is however worth noting that the focus of Bogner et al [16], Carrasco et al [21], and Neri et al [87] is on ensuring that the architecture of microservice-based applications complies with the design principles defining microservices themselves. We hence complement the results of their studies, as we focus on another architectural aspect than complying with microservices' design principles, viz., securing microservice-based applications.

Finally, it is worth relating our study with the multivocal literature review by Pereira-Vale et al [99], and with the grey literature review by Mao et al [74], even if they focused on DevOps rather than on microservices. Pereira-Vale et al [99] report the state of art and practice of the security solutions that have been proposed for microservices, and they identified the most used ones. Mao et al [74] instead report on the state of practice of DevSecOps, by first overviewing the currently existing risks in classical DevOps practices, and by then illustrating the best practices in DevSecOps and how they enable addressing the security risks of DevOps. The reviews by Pereira-Vale et al [99] and by Mao et al [74] differ from ours because they focus on the solutions proposed to support securing microservices, whereas we focus on distilling smells that may result in security issues in microservices. At the same time, the results in their reviews and ours complement each other, since, e.g., the security solutions they review can be used to implement the refactorings we describe.

In summary, to the best of our knowledge, there is currently no study classifying the smells that can possibly result in security issues for microservice-based applications, nor eliciting the refactorings that enable mitigating their effects. The latter is precisely the scope of our study, which we have presented in this chapter.

Chapter 3

Microservices Security: Bad vs. Good Practices

Securing microservices has become crucial, due to their widespread use in enterprise IT nowadays [134]. However, the information on the state-of-the-art and state-of-practice on securing microservices is scattered among a vast amount of white and grey literature. This makes accessing the body of knowledge on the topic complex and time consuming, both for researchers aiming to propose novel research directions and/or solutions for securing microservice-based applications, and for practitioners daily working with microservices and needing to secure them.

With the perspective of helping researchers and practitioners, we first reviewed the white and grey literature on securing microservices to elicit known security smells, defined as possible symptoms of a bad, though unintentional, decisions while designing/developing microservices, which may impact on their security [105]. We also elicited the refactorings allowing to mitigate the effects of security smells [105]. To further help researchers and practitioners, this research aims at complementing our former review [105] by eliciting what are not just "symptoms", but actually bad practices for securing microservices together with the security issues they may cause, as well as the good practices that, if adopted, enable avoiding to incur in such security issues. We indeed aim to answer the following two research questions:

(RQ1) What are the known bad practices *when securing microservices?*

(RQ2) What are the good practices *known to avoid incurring in the security issues caused by the aforementioned bad practices?*

To answer to our research questions, we report on what is being said by researchers and practitioners about bad/good practices for securing microservices, to provide a sort of "snapshot" of the state-of-the-art and state-of-practice on the topic. In particular, we present a taxonomy including four bad practices for securing microservices, and mapping such bad practices to six good practices that, if adopted, avoid incurring in the corresponding security issues. We then separately discuss each bad practice, by illustrating the bad smell(s) signaling it and the security issues it may cause. For each bad practice, we also describe the good practices known to enable avoiding its corresponding security issues.

The results of our study can help both practitioners and researchers interested in securing microservices. By complementing the results of our former review [105], a systematic presentation of bad and good practices for microservices security can indeed help practitioners to better understand what to avoid when securing microservice-based applications, as well as which good practices should be adopted to avoid it. In addition, the results presented in this chapter, together with those in our former review [105], constitute a body of knowledge that can be used by researchers for developing new techniques and solutions, experimenting research implications, or delineating novel research directions.

3.1 Research Design

We hereafter illustrate how we selected the primary studies from which to extract bad/good practices for microservices security (Section 3.1.1), as well as the analysis process we enacted to elicit such bad/good practices from the selected primary studies (Section 3.1.2).[1]

3.1.1 Literature Selection

Our aim is to complement the analysis of microservices' security smells and refactoring in our previous work [105], by eliciting the known bad/good practices for microservices security. We already elicited the 136 white/grey primary studies providing the state-of-the-art/state-of-practice in microservices security in [105], by following the guidelines for conducting systematic literature reviews in [63], combined with those in [44] for systematically reviewing grey literature. We hence started from such 136 candidate studies, as shown in Figure 3.1. Still following the guidelines in [44, 63], we applied different selection criteria to align with our research questions RQ1 and RQ2, namely to elicit the bad and good practices in microservices security.

(i_1) A study is to be selected if it discusses at least one *bad practice* for microservices security.

[1] To encourage repeating our review process, a replication package (containing the sheets we used to run our review) has been released at `https://docs.google.com/spreadsheets/d/1fY4lq3cdFjWCf7ZqaT5itnR7rw5vse-e/edit`.

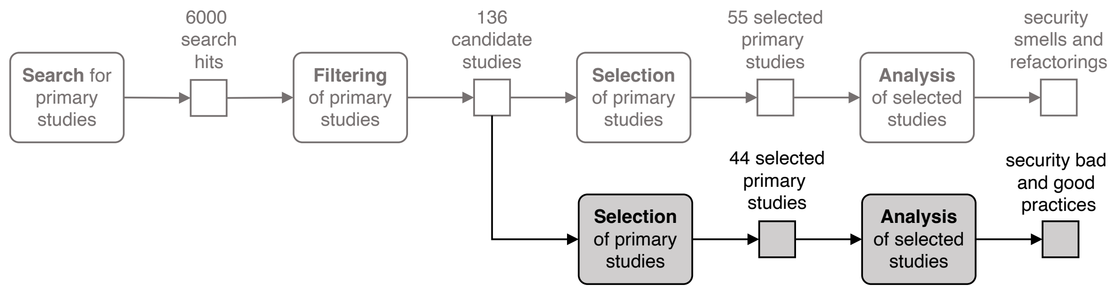


Figure 3.1: Literature review process.

(i_2) A study is to be selected if it discusses at least one *good practice* to avoid incurring in bad practices for microservices security.

Two authors of this research applied the criteria i_1 and i_2 to the 136 candidate primary studies, by independently coding such studies as to be included/excluded. The Krippendorf Kα coefficient [65] was then used to measure the inter-rater agreement on inclusion/exclusion of candidate primary studies. The measured, initial agreement was 87.5%, hence already significantly higher than the typical reference score of 80%. The missing 12.5% was caused by 17 mismatches, which were resolved by triangulation. Firstly, the other two authors of this research independently coded the corresponding 17 primary studies. Then, a joint discussion –involving all authors– decided whether to include each of these 17 primary studies, based on the four independent codings. This yielded 44 primary studies, shown in Table 3.1.

Table 3.1 also indicates the year of publication, "colour" (viz., *white* vs. *grey* literature), and type of the selected primary studies. Notably, 39 out of the 44 selected primary studies are grey literature, confirming an observation of previous reviews [105, 134], namely that grey literature constitutes the main source of information when analysing state-of-the-art/practice on microservices.

3.1.2 Literature Analysis

We analysed the selected primary studies to elicit bad practices for microservice security, as well as the good practices to avoid the corresponding security issues. The analysis was enacted by adopting thematic coding [10] and Krippendorf Kα-based inter-rater reliability assessment [65]. The first two authors annotated and labelled the selected primary studies to elicit bad/good practices for microservice security. The annotation and labelling were executed in parallel over two complementary partitions of the selected primary studies, with the aim of reducing potential observer biases. The coders were then switched to evaluate the inter-rater agreement on the two emerging lists of bad and good practices for microservice security. The inter-rater agreement was again measured by exploiting the Kα coefficient [65] to determine the agreement between

Table 3.1: Reference, year of publication, colour, and type of selected primary studies.

Ref.	Year	Colour	Type	Ref.	Year	Colour	Type
[1]	2019	grey	blog post	[83]	2020	grey	blog post
[12]	2017	grey	blog post	[86]	2019	white	journal article
[15]	2019	grey	blog post	[88]	2015	grey	book
[19]	2018	grey	blog post	[91]	2020	grey	blog post
[24]	2019	grey	whitepaper	[96]	2019	grey	video
[27]	2017	grey	blog post	[102]	2019	grey	video
[33]	2019	grey	video	[114]	2020	grey	blog post
[35]	2016	white	journal article	[115]	2020	grey	blog post
[49]	2018	grey	blog post	[123]	2019	grey	blog post
[52]	2016	grey	book	[125]	2017	grey	blog post
[54]	2018	grey	book chapter	[128]	2019	grey	video
[56]	2017	grey	book	[129]	2020	grey	blog post
[57]	2017	grey	video	[130]	2020	grey	book
[58]	2020	grey	blog post	[132]	2019	grey	blog post
[59]	2020	grey	blog post	[136]	2019	grey	blog post
[66]	2018	grey	blog post	[140]	2018	white	conference paper
[67]	2015	grey	blog post	[141]	2017	grey	blog post
[68]	2019	grey	blog post	[147]	2019	grey	blog post
[78]	2020	white	conference paper	[148]	2019	grey	blog post
[79]	2017	grey	blog post	[153]	2016	grey	book
[80]	2017	grey	blog post	[154]	2018	white	conference paper
[82]	2018	grey	book	[156]	2017	grey	book

the first two authors (who independently coded their partitions) on the emerging lists of bad and good practices for microservice security. The measured agreement amounted to 89.9%, again obtaining a value significantly higher than 80%, which is typically taken as reference score for inter-rater agreement [65].

A final triangulation step was then performed to further reduce potential biases. The last two authors cross-checked the coding performed by the first two authors, with no prior information on the coding itself. Their cross-checks were then discussed by organizing three feedback sessions: in the first two sessions, the feedback by the last two authors was separately discussed with the first two authors. Then, the independent codings and feedbacks were discussed in a plenary sessions involving all authors. This concluded our analysis process, which resulted in the taxonomy of bad and good practices shown in Figure 3.2.

3.2 Bad vs. Good Practices for Microservices Security

The identified bad/good practices for microservices security are displayed in Figure 3.2, and their appearance in the selected primary studies is reported in Table 3.2. To align with our research questions RQ1 and

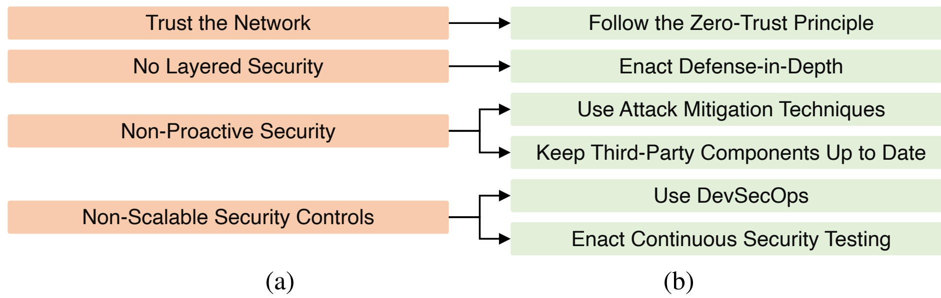


Figure 3.2: (a) Bad and (b) good practices for microservice security.

RQ2, we hereafter describe each identified bad practice, together with the good practices known to enable avoiding its corresponding security issues. We also map the identified bad practices to the security smells discussed in [105] that may signal them.

3.2.1 Trust the Network vs. Zero-Trust Principle

Bad practice. The authors of 13 selected primary studies identify TRUST THE NETWORK as a common bad practice when developing microservices. TRUST THE NETWORK is an approach that assumes that network communication between microservices is reliable and secure. This model employs a centralized security strategy, where all traffic is granted access by default, and security is upheld through network firewalls and other external security measures [35, 52]. In a microservices-based application, services are frequently added, removed, and updated, making it challenging to sustain a centralized security approach that can adapt to these fluctuations. This is essentially because the TRUST THE NETWORK approach does not take into account such a dynamic and distributed nature of microservices-based applications.

TRUST THE NETWORK indeed means designing microservices by blindly trusting other software components, with the latter even being microservices or integration components used within the microservice-based application under consideration, nor their possible addition, removal, or updates. For instance, a microservice-based application may be designed without enforcing security in service-to-service communications among its internal microservices, by just relying on network-level security to secure such communications. Another common mistake is to trust integration components by their mere identity, when such components are used to let microservices interoperate within the system (e.g., message brokers) or with external clients (e.g., API Gateway). When these or similar situations happens, the involved communication channels create potential attack vectors that can be exploited by an intruder to violate the communicating microservices [83, 130].

Related Security Smells. Blindly trusting software components and relying only on network-level security

Table 3.2: Coverage of the identified bad/good practices for microservices security in the selected primary studies.

	TRUST THE NETWORK	NO LAYERED SECURITY	NON-PROACTIVE SECURITY		NON-SCALABLE SECURITY CONTROLS	
	FOLLOW THE ZERO-TRUST PRINCIPLE	ENACT DEFENSE-IN-DEPTH	USE ATTACK MITIGATION TECHNIQUES	KEEP THIRD-PARTY COMPONENTS UP TO DATE	USE DEVSECOPS	ENACT CONTINUOUS SECURITY TESTING
[1]			✓			
[12]			✓			
[15]		✓	✓			
[19]		✓				
[24]			✓			
[27]			✓			
[33]		✓				
[35]	✓					
[49]			✓		✓	
[52]	✓	✓			✓	
[54]						✓
[56]			✓	✓		
[57]		✓	✓			
[58]		✓				
[59]		✓			✓	
[66]		✓			✓	✓
[67]	✓	✓				
[68]				✓		✓
[78]		✓				
[79]					✓	
[80]	✓		✓			
[82]	✓					
[83]	✓					
[86]			✓			✓
[88]	✓	✓				
[91]		✓				✓
[96]	✓					
[102]	✓					
[114]					✓	
[115]			✓	✓	✓	
[123]		✓				✓
[125]			✓			✓
[128]	✓					
[129]	✓					
[130]	✓	✓				
[132]			✓			
[136]		✓				
[140]		✓				
[141]		✓				
[147]		✓	✓		✓	✓
[148]		✓			✓	✓
[153]		✓				
[154]	✓	✓				
[156]			✓			

may result in introducing the following smells from [105]: NON-SECURED SERVICE-TO-SERVICE COMMUNICATIONS, UNAUTHENTICATED TRAFFIC and INSUFFICIENT ACCESS CONTROL. When any of such smells occurs in a microservice-based application, the latter's confidentiality, integrity, and authenticity may get compromised [105].

Good practice. All the authors of the primary studies discussing the TRUST THE NETWORK bad practice agree on saying that, to avoid incurring in the security issues it may cause, application architects should FOLLOW THE ZERO-TRUST PRINCIPLE to design the security of a microservice-based application. The zero-trust principle consists of assuming that the network is always hostile and untrusted, by never taking anything for granted. Each request must be authenticated and authorized at each node before being accepted for further processing [130]. Even if a request originates from a microservice at the same network-level, development teams should never assume that they can trust the source microservice and they should verify it based on what the credentials are [102]. In other words, since microservices communicate with each other over the network, an application's microservices should be treated in the very same way as external third-party applications [96]. To achieve this and realize the zero-trust principle, the authors of the primary studies discussing the FOLLOW THE ZERO-TRUST PRINCIPLE good practice share the following recommendations:

- Use Mutual TLS [127] to secure microservices communication, as this protects the application from man-in-the-middle, eavesdropping, and tampering attacks.
- Use OpenID Connect [92] to verify the user identity at each microservice, as well as to obtain basic profile information about the end-user.
- Use OAuth 2.0 [53] to enforce access control at each level, hence enabling each microservice to have its own fine-grained access controls.
- Use network segmentation to divide the network into smaller segments and isolate some components, as this helps reducing the attack surface for an application.

3.2.2 No Layered Security vs. Defense-in-Depth

Bad Practice. The authors of 21 of the selected primary studies discuss the NO LAYERED SECURITY bad practice. NO LAYERED SECURITY means designing a microservice-based application by relying only on perimeter defense, and/or by implementing a single security solution or practice [148]. Securing the perimeter or adopting a single security solution is not going to be enough, because microservices communicate with each other over the network and in disparate manners. Also, one such approach does not consider

the dynamic and distributed nature of microservices architectures, where services are continuously added, removed, and updated, e.g., possibly getting exposed after some update. Therefore, microservices-based applications call for a more robust and dynamic defense involving multiple layers of security controls, both at the perimeter and at the level of each microservice/software component forming an application [154].

Related Security Smells. Relying on perimeter defense and/or implementing a single security solution may result in introducing the following smells from [105]: INSUFFICIENT ACCESS CONTROL, NON-SECURED SERVICE-TO-SERVICE COMMUNICATIONS, UNAUTHENTICATED TRAFFIC, UNNECESSARY PRIVILEGES TO MICROSERVICES and NON-ENCRYPTED DATA EXPOSURE. When any of such smells occurs in a microservice-based application, the latter's confidentiality, integrity, and authenticity may get compromised [105].

Good Practice. The authors of the primary studies discussing NO LAYERED SECURITY agree on saying that application architects should ENACT DEFENSE-IN-DEPTH when designing the security of microservice-based application, as this would enable avoiding the security issues due to the NO LAYERED SECURITY bad practice. Defense-in-depth is a strategy in which several layers of security controls are introduced to protect an application from different types of threats. In a microservices-based application, this might involve implementing security controls at the network, host, and application levels. Critical microservices/components should be protected with multiple security layers, so that a potential attacker who has exploited one of the microservices of an application may not be able to do so to other microservices or layers [59]. This security strategy helps slowing down attacks when one security mechanism fails, or a security vulnerability is identified. One of the advantages of the microservice architecture is that it allows the diversification of security layers that can be designed and implemented [91]. A layered security strategy helps designing the application in such a way that each layer handles different types of attacks and repels an attacker at the outermost layer [130]. These layers depend on the criticality of the application and its microservices, but the general recommendations from the authors of the primary studies discussing the ENACT DEFENSE-IN-DEPTH good practice are the following:

- Secure all microservices behind a firewall to ensure that only the allowed traffic arrives to the application.
- Use an API Gateway for enforcing security on all requests incoming to an application, including authentication, authorization, throttling, and message content validation for known security threats [130].
- Use Mutual TLS [127], OAuth 2.0 [53], and OpenID Connect [92] (as also recommended in Section 3.2.1) to secure service interactions.

- Follow the *Least Privilege Principle*, by allowing each service to *only* access the resources it needs to deliver its businesses [15, 56].
- Encrypt all sensitive data and decrypt it only when needed [56, 130].
- Validate all input passed to the microservices to ensure it is in the correct format and does not contain malicious content.
- Use logging and monitoring tools to collect information about the activity of microservices, and to detect and respond to security incidents.

3.2.3 Non-Proactive vs. Proactive Security Measures

Bad Practice. Preventing exploits by intruders is only one part of securing a microservices-based application. Proactive detection and reaction are another essential part [116]. Though it is impossible to protect an application against all types of attacks, microservices must be provided with detection and prevention capabilities against, e.g., credential-stuffing and credential abuse attacks as well as the capability to detect malicious botnets [24]. This is highlighted by the authors of 16 of the selected primary studies, who identify NON-PROACTIVE SECURITY as a bad practice while designing microservice-based applications. Such a bad practice would indeed lead to not including security mechanisms that allow to proactively detect and react to potential or actual attacks. The lack of such proactive security measures can expose microservices-based applications to vulnerabilities, e.g., distributed denial-of-service attacks, with attacker attempting to (and possibly succeeding in) making an application unavailable to its users.

Related Security Smells. Without proactive detection and reaction to security issues, we may introduce the following smells from [105]: INSUFFICIENT ACCESS CONTROL, UNNECESSARY PRIVILEGES TO MICROSERVICES, OWN CRYPTO CODE, NON-ENCRYPTED DATA EXPOSURE, UNAUTHENTICATED TRAFFIC. When any of such smells occurs in a microservice-based application, the latter's confidentiality, integrity, and authenticity may get compromised [105].

Good Practices. The most discussed good practice to avoid the security issues due to NON-PROACTIVE SECURITY is USE ATTACK MITIGATION TECHNIQUES, which is discussed in 15 out of the 16 primary studies dealing with NON-PROACTIVE SECURITY. Attack mitigation techniques include all mechanisms and practices that are implemented to prevent security breaches and protect the application and its data from malicious attacks [56, 57]. Concrete examples of attack mitigation techniques for microservices-based applications are web application firewalls [147] and rate limiting [132].

- A web application firewall is an application firewall for HTTP applications applying protection rules to HTTP conversations [56]. A web application firewall can also monitor the volume of cache misses to proactively detect that, e.g., an API gateway is constantly performing middle-tier service calls due to cache misses, which would suggest that the cache is potentially suffering a malicious attack [12]. Web application firewalls can hence be used to provide instant protection against SQL injection, cross-site scripting, illegal resource access, remote code execution, remote file inclusion, and other OWASP Top-10 threats [150].

- Application architects should also introduce rate-limiting API calls to mitigate distributed denial-of-service (DDoS) attacks and protecting the backend application that process the API calls [132]. Rate limiting consists of counting how many requests a server is accepting in a period and rejecting new ones when a certain limit is reached [115].

Another good practice to avoid the security issues due to NON-PROACTIVE SECURITY is KEEP THIRD-PARTY COMPONENTS UP TO DATE, even if less discussed than the former. KEEP THIRD-PARTY COMPONENTS UP TO DATE means that application administrators should ensure to include all the latest security patches for the third-party components use in an application [56, 116]. A scanning software can be used on the source code repository to identify vulnerable dependencies (e.g., as done by GitHub bots), and this type of scan should be enacted at any phase of the deployment pipeline.

The actual implementation of proactive security measures for microservices-based applications (such as those described above) must be tailored to the specific technology stack and requirements of the application and its maintaining organization. It is crucial to recognize that maintaining the security of a microservice-based application is a continual process, and therefore, the security measures must be consistently evaluated and updated to remain efficient [115].

3.2.4 Non-scalable vs. Scalable Security Controls

Bad Practice. Microservices-based applications are typically designed and developed by adopting continuous DevOps practices. While dealing with microservices, there can indeed be quite frequent changes resulting in new releases [134]. Continuous integration/continuous deployment (CI/CD) pipelines are typically enacted to distribute microservice-based applications and to keep them updated in production. CI/CD pipelines are themselves a source of possible security issues, so they need fully automated, scalable security controls [59]. Suppose that, e.g., certain development team plan to upgrade a microservice-based application by changing or adding functionalities to some of its microservices. Without automation, they would be required to scan the added code and its integration with the existing code for vulnerabilities and weak-

nesses before they could deploy it [125]. This would go in contrast with the automation needs of CI/CD pipelines, hence resulting in what the authors of the selected primary studies consider the NON-SCALABLE SECURITY CONTROLS bad practice.

Related Security Smells. The lack of integration of security measures/testing in CI/CD pipeplines may result in introducing the following smells from [105], especially if considering that microservice-based applications are continuously updated: PUBLICLY ACCESSIBLE MICROSERVICES, UNNECESSARY PRIVILEGES TO MICROSERVICES, OWN CRYPTO CODE, NON-SECURED SERVICE-TO-SERVICE COMMUNICATIONS, UNAUTHENTICATED TRAFFIC. When any of such smells occurs in a microservice-based application, the latter's confidentiality, integrity, and authenticity may get compromised [105].

Good Practices. USE DEVSECOPS and ENACT CONTINUOUS SECURITY TESTING are the two good practices identified to avoid the security issues due to NON-SCALABLE SECURITY CONTROLS, both being highlighted in 9 out of the 15 primary studies discussing the NON-SCALABLE SECURITY CONTROLS bad practice. USE DEVSECOPS helps in integrating security processes, principles, good practices, and tools early in the development lifecycle, thereby encouraging collaboration among security experts and business analysts, architects, and development and operations teams, thus making everyone accountable and responsible for building secured systems [66]. On the other hand, automation is the key to integrating quality protection in a way that ensures quick feedback on the impact of new changes. Therefore, ENACT AUTOMATED SECURITY TESTING helps realizing the speed and flexibility needed in CI/CD pipelines, and it also ensures a faster recovery [148]. By incorporating automated security testing into the continuous integration and continuous delivery (CI/CD) pipeline, vulnerabilities can be identified and rectified at every stage of the development process. This approach enable application developers to proactively and early address any identified security issue, possibly before they can be really exploited by attackers in a production environment [115, 147]. Automated security testing can create and update baselines before every release with OWASP Top 10 [150] and advanced libraries that are informed by discovered abnormalities across clients. Through automation, DevOps teams can meet their responsibility to facilitate fast releases and quicker code fixes [148].

3.3 Threats to Validity

Following the taxonomy in [152], we examine four kinds of threats to the validity of our study (viz., *external*, *construct*, *internal*, and *conclusions*).

Threats to External Validity. The *external* validity of our study may be threatened since the selected primary studies were taken from a large extent of online sources, which may be only partly applicable to the broader, more generic area of practices on microservices. To strenghten the external validity of our study, we run three feedback sessions among the authors of this study during the analysis of the selected primary studies, also with the goal of fine-tuning the taxonomy organizing the emerging lists of bad/good practices for microservices security (Figure 3.2). We also carefully applied our selection criteria as follows: instead of checking whether a bad/good practice in our taxonomy was explicitly mentioned in a study, we rather checked whether a primary study was discussing security issues due to some bad practices, and whether it was discussing how to avoid such issues, even just by means of examples, which we later linked to and organized into the identified good practices. Finally, we prepared a replication package providing access to the sheets that we produced during our study, to encourage repeating it and ease the understanding of the data that we produced. All what above was aimed at making our results and observations more explicit, replicable, and applicable in practice.

Threats to Construct and Internal Validity. The *construct* and *internal* validity instead concern the method employed to study and analyse data, including the types of potentially involved biases [152]. To mitigate the potential threats to the construct and internal validity of our study, we selected and classified the primary studies by relying on validated techniques, namely theme coding, inter-rater reliability assessment, and triangulation (Section 3.1). These are known to limit potential biases [134], like observer and interpretation biases, hence allowing us to strenghten the validity of our analysis of the data that we collected.

Threats to Conclusions Validity. The aforementioned inter-rater reliability assessment also helped in mitigating the potential threats to the *conclusions* validity of our study, which concerns the degree to which our conclusions were reasonably based on the available data [152]. In addition to that, all authors of this research independently drawn the observations and conclusions discussed in this chapter, which were then discussed and double-checked against the selected primary studies in joint discussion session.

3.4 Related Work

Multiple secondary studies have been conducted on microservices, also aimed at classifying known issues and good practices. For instance, Carrasco *et al*. [21], Neri *et al*. [87], and Taibi *et al*. [137, 138] report on bad smells, refactorings, and architectural patterns for microservices, aimed at supporting developers in identifying them and improving their microservice-based applications. They however focus on aspects other than securing microservices, and our study is hence intended to complement their results (and those of similar studies) along that direction.

As for microservice security, only a few secondary studies have been conducted. Berardi *et al*. [13] analyses the available white literature on microservices security, to identify the research communities where this is most discussed, the known security attacks, and the countermeasures to enact to secure microservice-based applications from such attacks. Hence, [13] differs from our study in the objectives: we indeed aim to elicit bad and good practices for microservices security, by considering the state-of-the-art and state-of-practice on the topic, taken from white and grey literature, respectively.

In this perspective, Pereira-Vale *et al*. [99] and Mao *et al*. [74] are closer to our study, as they consider both white and grey literature. Pereira-Vale *et al*. [99] report on existing mechanisms to secure microservice-based applications, while also identifying those most used. Mao *et al*. [74] instead reviews currently existing DevSecOps mechanisms, including those adopted to secure microservices. Pereira-Vale *et al*. [99] and Mao *et al*. [74] hence differ from ours because they focus on which existing mechanisms can be used to secure microservices, whereas we focus on distilling bad/good practices that should not/should be adopted to secure them. It is anyhow worth noting that our results and those by Pereira-Vale *et al*. [99] and Mao *et al*. [74] complement each other. For instance, the security mechanisms reviewed by Pereira-Vale *et al*. [99] and Mao *et al*. [74] can be used to implement the good practices that we identified.

To summarise, to the best of our knowledge, there is currently no study classifying the bad practices potentially causing security issues in microservice-based applications, nor the good practices that, if adopted, avoid incurring in such security issues. Our study is aimed to precisely cover this gap, by providing a systematic review of the known bad/good practices for microservice security.

Chapter 4

Model-Driven End-to-End Resolution of Security Smells in Microservice Architectures

The content of Chapter 4 is the result of joint research with the IDiAL Institute (University of Applied Sciences and Arts Dortmund, Germany), and builds on top of our proposal published in [151].

The distributed nature of Microservice Architectures (MSAs) makes them inherently prone to *security smells*, which denote poor (often unintentional) design decisions that harm application security [104]. The resolution of *security smells* often requires adapting code in different places of the application with heterogeneous purposes, technologies, and software languages. For example, the resolution of the Publicly Accessible Microservices smell [105]—a bad practice in MSA engineering that exposes microservice interfaces to architecture-external callers instead of hiding them behind API gateways [8]—requires (i) introduction of gateway programming and configuration code; (ii) adaptation of microservice programming and configuration code to connect with the introduced gateway; and (iii) adaptation of deployment code to cover the gateway. This scattering of smell across heterogeneous architecture components significantly aggravates their detection and holistic resolution.

This chapter presents an end-to-end approach for resolving security smells in existing MSAs that automatizes smell detection and provides users with an interactive mechanism for smell resolution across the concerned MSA components. MSA security smells have been recently proposed by Ponce et al. [105], and how to automatically detect and resolve them is still an open issue [23]. Our approach relies on the previous

work on MSA security smells, as well as stakeholder-oriented Model-Driven Engineering (MDE) [26] of MSAs [110]. More precisely, it first maps the source code of existing MSAs to MSA-specific architecture models based on the Language Ecosystem for Modeling Microservice Architecture (LEMMA) [110]. These models are then validated to automatically detect security smells and make them visible to MSA stakeholders, who can then decide, per smell, for a model refactoring that solves the smell. In the final step, a LEMMA-based code generator reflects the refactorings to the original MSA implementation, thereby fixing all places in the application that pertain to a certain smell. As a result, our approach contributes support for the following actions in MSA engineering:

- Automated uncovering of security smells that are scattered across architecture components.
- Automated reporting of those smells via MSA-oriented architecture models that abstract from components' heterogeneity, thus facilitating stakeholder reasoning about smells.
- Deciding for the most suitable smell resolution and subsequent automatic reflection back to the original application code.

We assess the applicability and effectiveness of our approach by executing it on two microservice-based applications. First, we use the student management application to illustrate the steps for resolving security smells. Additionally, we validate our results and demonstrate their applicability on Lakeside Mutual, a standard case study in MSA research [135]. Our results show the effectiveness of recovering a software application design in architecture models, the capability to detect and resolve security smells in the recovered models, and the capability to resolve the smell in implementing the application.

4.1 Background

We hereafter provide the necessary background on microservice security smells and LEMMA for viewpoint-based microservices modeling.

Smells and Refactorings for Microservice Security. A microservice security smell is a symptom of a potentially bad decision (often unintentional), that can negatively impact the application's security [105]. The effects of security smells can be resolved by refactoring the application without altering the functionality provided to external clients. We hereafter recall two popular MSA security smells that are part of the taxonomy proposed in [105], and the refactorings that allow to resolve them. The other smells that are part of the taxonomy are *Unnecessary Privileges to Microservices, Non-Secured Service-to-Service Communications,*

Unauthenticated Traffic, Own Crypto Code, Non-Encrypted Data Exposure, Hardcoded Secrets, Multiple User Authentication, and *Centralized Authorization*.

Publicly Accessible Microservices. A microservice of an application is publicly accessible when external clients can directly access it. This increases the application's attack surface and reduces its overall maintainability and usability. Also, if each publicly accessible microservice performs authentication, the full set of a user's credentials is required each time, increasing the likelihood of confidentiality violations (e.g., with the exposure of long-term credentials).

The suggested refactoring is making such microservices accessible *only* through a newly added API Gateway, which would act as an entry point for the application. This would enable centralizing authentication, reducing the application's attack surface and simplifying the authentication itself.

Insufficient Access Control. This smell occurs on the microservices of an application that is not enforcing access control. This can possibly violate the confidentiality of the microservices where access control is lacking, as attackers can trick a service and get data that they should not have access to.

The possible effects of this smell can be resolved by exploiting OAuth 2.0, which would enable microservices to control accesses. OAuth 2.0 indeed provides a token-based access control system that lets a resource owner grant a client access to a particular resource on their behalf.

LEMMA. The Model-Driven Engineering ecosystem LEMMA provides a set of modeling languages to capture concerns in MSA engineering from stakeholder-oriented architecture viewpoints [109]. MSA models constructed with those languages can be integrated based on an import mechanism that enables referencing between elements of heterogeneous models to support reuse and increase the information content of captured viewpoints in a microservice architecture. The presented approach for security smell resolution in microservice architectures relies on the following LEMMA modeling languages.

Domain Data Modeling Language (DDML). The Domain Data Modeling Language (DDML) of LEMMA addresses the concerns of domain experts and microservice developers in the Domain Viewpoint. To this end, the language supports the construction of *domain models* that cluster the relevant concepts from the application domain. These concepts may be enriched with patterns from Domain-Driven Design (DDD) [36], which is a popular methodology for microservice design [45, 76, 81, 84, 88]. The DDML also implements LEMMA's type system so that domain concepts are usable, e.g., for the typing of parameters of modeled microservice operations. Among others, such typing relationships identify the portion of the application domain on which a microservice operates and for which it is thus responsible.

Technology Modeling Language (TML). LEMMA's TML targets the Technology Viewpoint on microservice architectures and allows for the construction of *technology models* that capture technology decisions

related to microservices and their implementation and deployment, e.g., communication protocols and deployment technologies. Additionally, the TML supports the definition of *technology aspects* that apply to specific elements in LEMMA models, e.g., modeled microservices and their interfaces, or infrastructure nodes. Given their flexibility, technology aspects can also be exploited to enable subsequent augmentation of LEMMA models with additional metadata.

Service Modeling Language (SML). LEMMA's SML reifies the Service Viewpoint in MSA engineering and provides modeling concepts to specify microservices, their interfaces, operations, endpoints, and dependencies to other microservices in *service models*. Among others, the SML integrates with the TML so that LEMMA service models can import LEMMA technology models to specify, e.g., protocol-dependent communication endpoints such as HTTP addresses, and the available methods to operate on them.

Operation Modeling Language (OML). LEMMA's OML focuses on MSA's Operation Viewpoint supporting the specification and configuration of microservice containers and infrastructure nodes, e.g., for service discovery, in *operation models*. Similarly to the SML, the OML integrates with the TML to cope with MSA's technology heterogeneity w.r.t. microservice operation and deployment [64]. More precisely, technologies for microservice deployment and infrastructure usage can flexibly be specified in technology models, making them referenceable from operation models.

Next to the model-based description of microservices and their operation, LEMMA also anticipates model processing, and in this context has already been used to foster MSA team integration by *model transformation* [135] and increase microservice development efficiency by *code generation* [112]. In the following, we rely on LEMMA's capabilities in model processing to identify microservices' security smells by *static analysis* of service and operation models and suggest resolution actions by *interactive model refactoring*.

4.2 End-to-End Smell Resolution

This section introduces our approach for end-to-end microservice security smell resolution. Figure 4.1 depicts the successive steps of resolving security smells using existing or specifically for this approach created LEMMA components.

The first step consists of the automated reconstruction of the MSA of an existing application (Section 4.2.1). For this purpose, we extended LEMMA's functionalities by integrating the Microservice Reconstruction Framework (MAR) to recover the architecture design of the application with a focus on domain concepts, API management, and deployment specifications as LEMMA models.

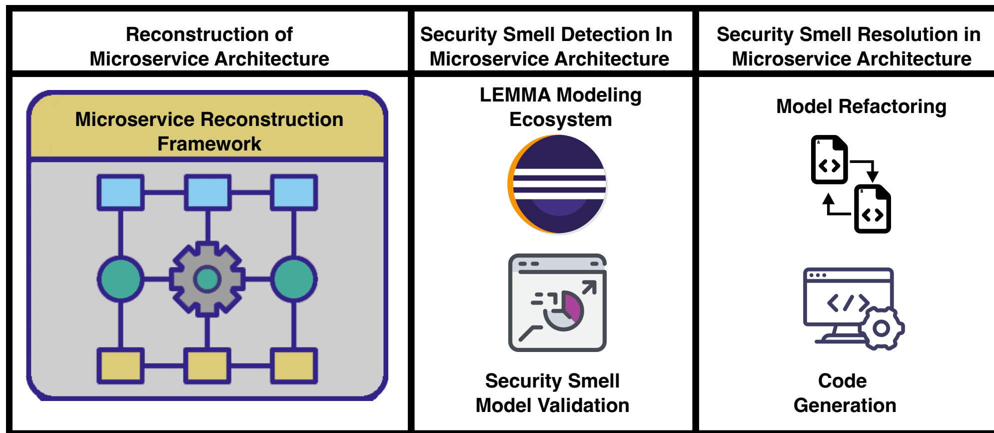


Figure 4.1: Approach to resolve security smells in MSA.

In the second step, we use LEMMA's existing modeling ecosystem to modify the reconstructed models from the previous step and extend LEMMA's model validation functionalities with the capability to detect microservice security smells (Section 4.2.2).

To resolve the security smell in the application architecture, the final step consists of model refactoring and code generation. The model refactoring is a new addition to LEMMA's functionalities for our security smell resolution approach to resolve the smell in the reconstructed models. Moreover, LEMMA's existing code generation functionalities also resolve the security smell in the existing source code.

The rest of this section consists of a detailed description of how our approach enables reconstructing the MSA of an existing application (Section 4.2.1), detecting the security smells therein (Section 4.2.2), and resolving them in the source code of the application (Section 4.2.3) on a concrete MSA-based application.

To illustrate our approach, we will use the MSA-based student management application, depicted in Figure 4.2 to highlight the separate steps to resolve the security smells and exemplify them with a descriptive example. The application consists of the `Student-` and `ExamService` as functional microservices. Additionally, the services rely on a `Database` and `Service Discovery` to provide their functionalities.

4.2.1 Reconstruction

Software Architecture Reconstruction (SAR) is a reverse engineering approach to recover the architectural design of an application that is outdated or to ensure conformance between implementation and design [11]. Figure 4.3 depicts the structure of our toolchain to reconstruct a technology heterogeneous software application architecture using an ecosystem of MDE tools.

The `Framework` orchestrates the process of deriving architecture information from static development

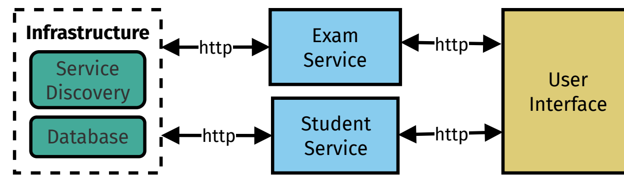


Figure 4.2: MSA-based student management application.

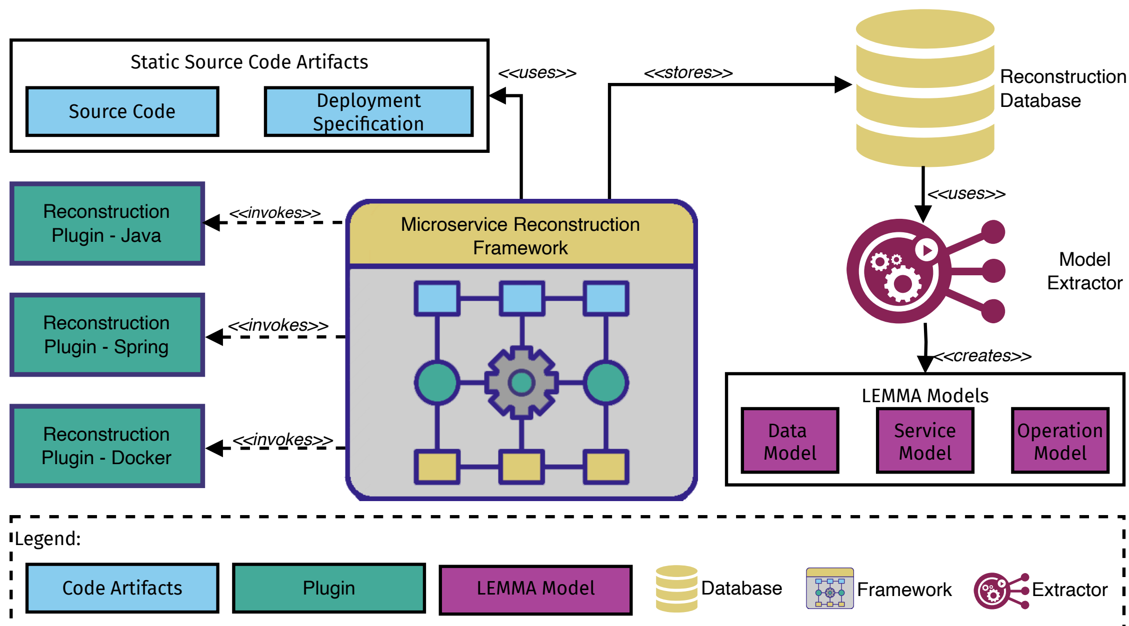
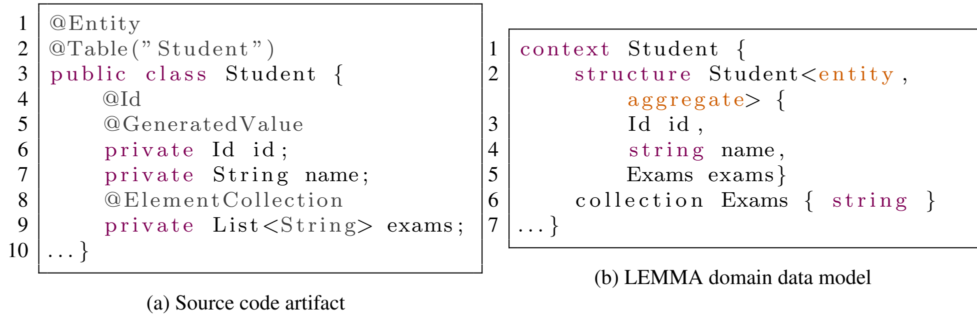


Figure 4.3: Structure of the reconstruction framework.

artifacts, e.g., `Source Code` and `Deployment Specification`. The framework provides a plugin functionality to support a heterogeneous technology stack to reconstruct the architectural design from heterogeneous source code artifacts, e.g., Spring annotations and Docker deployment specifications.

For this purpose, the framework manages the development artifacts and invokes the `Reconstruction Plugins`. The plugins implement functionalities for the technology-specific reconstruction of architectural information. When the framework invokes the plugins, they derive the architecture information and forward the reconstructed architecture information to the framework.

The next step of the process consists of aggregating the reconstructed information from the plugins into a coherent architectural design of the software application. The framework stores the design in a database to enable the possibility of enhancing the design with runtime information, e.g., traces or message broker logging information. The data format consists of the specific concepts for each viewpoint in MSA to store the reconstructed architecture design.


```
1  @Entity
2  @Table("Student")
3  public class Student {
4      @Id
5      @GeneratedValue
6      private Id id;
7      private String name;
8      @ElementCollection
9      private List<String> exams;
10 ...}
```

(a) Source code artifact

```
1  context Student {
2      structure Student<entity,
           aggregate> {
3          Id id,
4          string name,
5          Exams exams}
6      collection Exams { string }
7  ...}
```

(b) LEMMA domain data model

Figure 4.4: Example of recovered domain concepts.

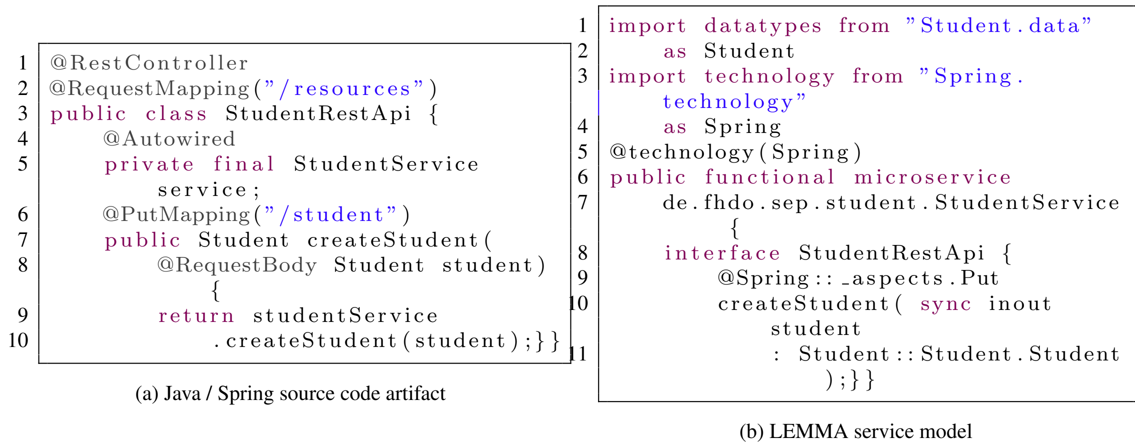
The final step to recovering the software application architecture is to derive viewpoint-specific models from the architectural design stored in the database. Therefore, the `LEMMA Model Extractor` [113] uses the information to create LEMMA models from it. The recovered models, with their corresponding viewpoints, address different stakeholders in the software engineering process for MSA, e.g., the domain data model captures domain concepts for service developers and domain experts. Figure 4.4 shows the source code artifact and the recovered domain data model using LEMMA DDML (Section 6.1).

Figure 4.4(a) shows the implementation of the `Student` Java class from the student microservice (Figure 4.2). The class consists of the two complex attributes `id` and `exams` and the primitive data type attribute `name`. Additionally, the annotations `Entity`, `Table`, and `Embedded/Id` enrich the class with technology-specific information, e.g., for database persistency and to enable the Inversion of Control (IoC) functionalities from the Spring Framework. Therefore, the Spring Reconstruction Plugin uses this information to derive domain concepts from the source code and maps them to the DDD pattern.

In this case, Figure 4.4(b) features the technology-agnostic recovered LEMMA domain data model. The model contains the `Student` context in accordance with the *Bounded Context* [36] in DDD. The excerpt of the recovered context includes the complex data structure `Student`. The concept is derived from the Java class with the eponymous name. The `Entity` annotation from the Spring Framework, in combination with the name of the Java class, maps to the complex data structure in the domain data model, including the `entity` and `aggregate` pattern from DDD.

For the specification of microservices APIs and service dependencies in the recovered architectural design, our approach uses LEMMA service models to display this information for stakeholders such as service developers in the development process of MSA. Figure 4.5 shows the Java source code for interface specification from the student microservice, including technology-specific information and the corresponding LEMMA service model.

Figure 4.5(a) contains the implementation of a REST controller by using the `RestController` and



(a) Java / Spring source code artifact

(b) LEMMA service model

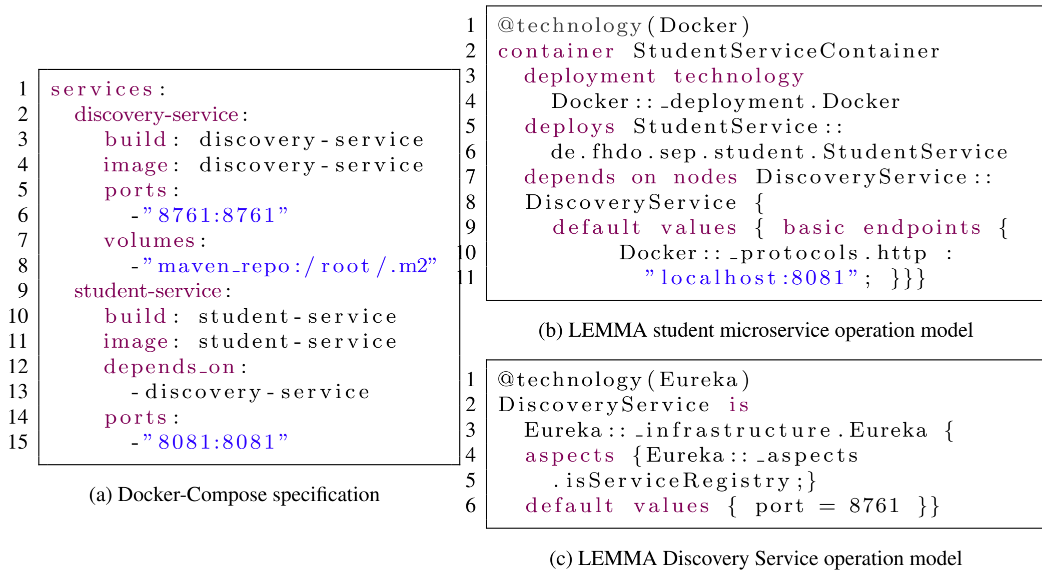
Figure 4.5: Example of recovered interface specifications.

`RequestMapping` annotation. The figure shows the `createStudent` endpoint of the microservice, including URI and method specification. The listing specifies the incoming `student` data type as a `RequestBody` and outgoing `student` parameter.

The service model in Figure 4.5(b) features the interface specification of the Student microservice derived from the REST controller specification. The figure starts with import statements for domain data and technology models. The `datatype` statement imports the student data model from Figure 4.4(b), used as data types in the interface specification. Additionally, to enhance the service model with technology-specific information, the subsequent import statement enables using the `Spring` technology model in the microservices interface modeling. Figure 4.5(b) is then completed by the modeling of the `Student` functional microservice, including the fully qualified name and technology, including the specification of the `StudentRestAPI` interface derive from the Java source code. The specification includes the REST method and URI specification. Furthermore, the imported data types from the student domain model define the incoming and outgoing parameters.

LEMMA operation models contain the deployment specification for microservices and infrastructure components of the application and, therefore, address the concerns of the service developers and operators. Our approach currently supports the recovery of Docker-specific deployment specifications. It can, therefore, be used to recover information from the *docker-compose* file of the student management application, an excerpt of which is in Figure 4.6(a). Figure 4.6(a) displays the deployment specification for the infrastructure component `discovery-service` and microservice `student-service`. The Discovery-Service and StudentService deployment specifies, among others, the `image`, `port`, and `depends_on` dependencies in the software application.

The reconstruction process uses these specifications for reconstructing operation models, capturing the



(a) Docker-Compose specification

(b) LEMMA student microservice operation model

(c) LEMMA Discovery Service operation model

Figure 4.6: Example of our approach to recover deployment specifications from docker artifacts (a) into a LEMMA operation model with microservice deployments (b) and infrastructure components (c).

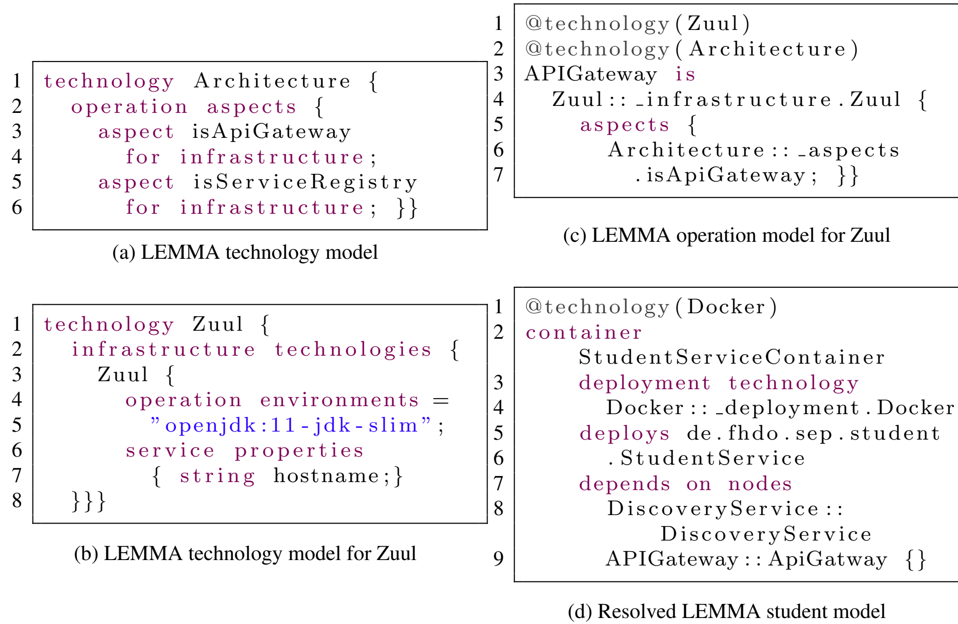
architectural design from the operation viewpoint. The operation model in Figure 4.6(b) describes the deployment of the Student microservice referencing the recovered service model from Figure 4.5(b). The specification includes `Docker` as the deployment technology for the Student microservice and the runtime dependency to the infrastructure component of a `DiscoveryService` (Figure 4.6(c)). Since Eureka is part of the Netflix OSS stack and implements the architecture pattern of a *Service Registry* [11], the `discovery-service` is reconstructed as an infrastructure node named `DiscoveryService` with the aspect `isServiceRegistry` indicating the reference to the eponymous pattern for microservices.

The result of the end-to-end resolution process of the reference application is the recovered MSA of the student management application. The recovered MSA is in the form of LEMMA models addressing different software engineering viewpoints used to detect smells, as described in the following section.

4.2.2 Smell Detection

The detection step uses the LEMMA models recovered in Section 4.2.1 to identify microservice security smells. Therefore, our approach leverages the expressiveness of LEMMA's aspect functionality [111] to enhance models with metadata, enabling the possibility to include architecture and security-specific information into the models that can be used to identify security smells. Figure 4.7 represents the LEMMA models associated with the smell detection process.

Figure 4.7(a) presents a LEMMA technology model that contains the specification of metadata to enrich



(a) LEMMA technology model

(b) LEMMA technology model for Zuul

(c) LEMMA operation model for Zuul

(d) Resolved LEMMA student model

Figure 4.7: LEMMA technology and operation models for security smell detection.

operation models with information related to architectural patterns, e.g., *Service Registries* or *API Gateways* [119], to enable the identification of infrastructural components in the software systems architecture. To identify those patterns in the recovered architecture, the model specifies metadata as `operation aspects`. Specifically, the eponymous infrastructure components `isApiGateway` and `isServiceRegistry`.

Figure 4.7(b) displays the `Zuul` technology model containing the technological specification for a concrete implementation for an API Gateway. The Zuul technology model is created to be used in an operation model to specify the deployment and operation of an API Gateway.

Figure 4.7(c) then imports both the architecture and Zuul technology models to define the infrastructure node with the name `APIGateway`. The node uses the infrastructure technology `Zuul` for the implementation specified in Figure 4.7(b). The operation aspect `isApiGateway` from Figure 4.7(a) is applied to the node, making it identifiable as an API Gateway.

For security smell identification in MSA, the model validation functionalities of LEMMA allow one to analyze the models regarding the occurrence of security smells, e.g., the absence of infrastructure components that implement specific microservice patterns or the lack of authorization specifications. The model incorporates these patterns by using LEMMA technology aspects.

In the case of the deployment specification of the student management application (Figure 4.8), the model validation gives a warning because the student microservice misses the dependency on an API Gateway, which is normally modeled leveraging LEMMAs abstract concept (c.f. Figure 4.7a). Therefore, the

missing gateway may lead to the *Publicly Accessible Microservices* smell. Figure 4.9 shows the interactive user interface that presents the smell resolution strategies.

```
 5 @technology(Docker)
 6 container StudentContainerContainer
 7     deploy
 8     deploy
 9     depend
10     defaul
11         basic endpoints {
12             Docker::_protocols.http : "localh
13         }
14     }
15 }
```

Publicly Accessible Microservice Security Smell detected
1 quick fix available:
Select resolution strategy for Security Smell: Publicly Accessible Microservices.
Press 'F2' for focus

Figure 4.8: LEMMA Eclipse editor presenting the student microservice operation model with the *Publicly Accessible Microservices* smell detection.

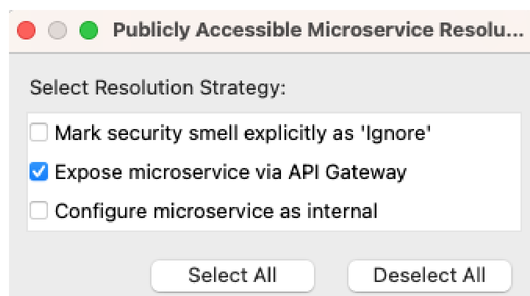


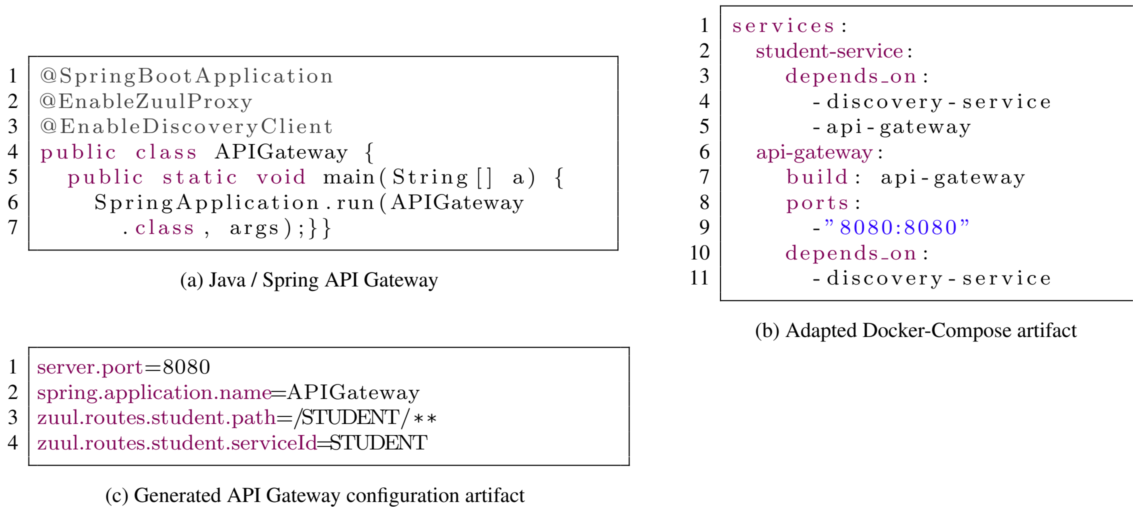
Figure 4.9: LEMMA Eclipse editor presenting the student microservice operation model with the *Publicly Accessible Microservices* smell with the resolution strategy.

LEMMA provides the functionality to refactor the models automatically to resolve the identified microservice smells. Figure 4.7(d) presents the automatically refactored operation model with the dependency on an API Gateway. In addition to the refactored operation model, the refactoring process also creates technology (b) and operation (c) models if they are not already present in the software architecture, including architectural and infrastructure components.

4.2.3 Smell Resolution

The security smell resolution uses the refactored LEMMA models to implement the refactoring in the application's source code. Since we have detected the *Publicly Accessible Microservices* security smell in the student management application, our approach provides three strategies as shown in Figure 4.9.

The first strategy explicitly ignores the security smell, e.g., because the software architect intentionally exposed the microservice, which implements some gateway functionality. The second and third strategies resolve the security smell by exposing the microservices interfaces via an API Gateway or configuring the microservice not to expose it externally (e.g., by removing its external network access configuration).



(a) Java / Spring API Gateway

(b) Adapted Docker-Compose artifact

(c) Generated API Gateway configuration artifact

Figure 4.10: Artifacts created by the *Deployment_Base* and Zuul code generators.

Hereafter, we assume that the user selected the second strategy to integrate an API Gateway into the student management application automatically. The implementation of this refactoring requires the following tasks:

1. Source code generation for the API Gateway.
2. Enable the request routing by the API Gateway.
3. Adapt the deployment specification of the microservices.

To execute these tasks, LEMMA model processing functionalities provide the *Deployment_Base* and *Spring Cloud Zuul* code generators to adapt or create source code artifacts. The Zuul code generator uses the technology model from Figure 4.7(b) and the operation model from Figure 4.7(c) to generate the API Gateway implementation, including source code and configuration files based on the Spring Cloud technology stack and using Java as a programming language. Moreover, the *Deployment_Base* generator adapts or creates deployment specifications for the refactored architecture for Docker technology, e.g., *docker-compose* files for service composition and a *Dockerfile* for containerization.

Figure 4.10(a) shows the source code of the API Gateways implementation using the `@EnableZuulProxy` annotation to enable the routing functionalities of the node. In addition to that, the code generator for the API gateway also generates a configuration file containing the routing information based on the `depends_on` dependencies between the functional service and the API gateway in LEMMA operation models (Figure 4.7(d)).

For service composition, the *Deployment_Base* generator adapts the *docker-compose* file from Figure 4.6(a) to address the resolved security smells in the source code. Therefore, the code generator removes

the exposure of the ports for the `student` service for accessing the endpoints. The code generator adds the `api-gateway` component to enable further access to the microservices interfaces. The code generator creates executable source code that is runnable without further configuration, so the end-to-end security smell resolution provides all artifacts needed for the refactoring implementation.

4.3 Validation

In this section, we validate our approach to model-driven end-to-end resolution of security smells in MSA-based applications. For validation purposes, we use the Lakeside Mutual[1] microservice reference application (Figure 7.1) to assess our proposed approach since the *Publicly Accessible Microservices* and the *Insufficient Access Control* security smells occur in its implementation. Moreover, we raise research questions (RQ) to investigate the results of the different stages of the presented approach.

4.3.1 Research Questions

In the validation process for our presented approach, we aim to answer the following research questions:

RQ1: *How accurate is the reconstructed architecture design of the application?* This research question addresses the accuracy of the reconstructed architecture design related to concepts, e.g., recovered microservices, interfaces, endpoints, data structures, deployment specifications, and infrastructure components.

RQ2: *Could the security smells be detected in the reconstructed models?* The detection of the *Publicly Accessible Microservices* and *Insufficient Access Control* security smell in the reconstructed models is addressed by this research question.

RQ3: *Could the security smell be resolved in the reconstructed models?* We verify if the detected security smell can be resolved in the recovered models capturing the application's architectural design.

RQ4: *Is resolving the security smell in the application's source code possible?* This research question addresses the end-to-end smell resolution, and we check if the smell is also resolved in the application's source code.

4.3.2 Validation Implementation

As described in Section 4.2.1, the end-to-end security smell resolution process starts by recovering an application's MSA, whose modeling in LEMMA is then used to enact smell resolution. Therefore, it is

[1] https://github.com/Microservice-API-Patterns/LakesideMutual

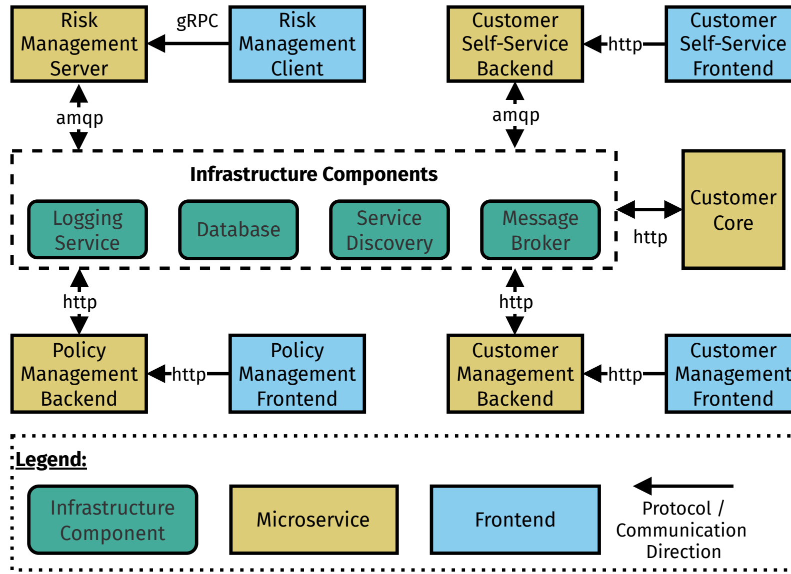


Figure 4.11: Architecture of the Lakeside Mutual application.

Element	Expected	TP	FP	FN	Recall	Precision	$F_{measure}$
Microservices	5	4	0	1	80%	100%	88%
Interfaces	16	14	0	2	87%	100%	93%
Endpoints	61	50	3	8	86%	94%	90%
Data Structures	161	117	29	14	89%	80%	84%
Deployment	5	4	0	1	80%	100%	88%
Infrastructure	2	2	0	0	100%	100%	100%
Sum	250	191	32	26	88%	86%	87%

Table 4.1: Results from the reconstruction process.

important to assess the effectiveness of the reconstruction step, and here, we measure it for the reference application we considered in our case study.

Table 4.1 presents the results of the reconstruction process by relying on *Recall*(eq. (4.1)), *Precision*(eq. (4.2)), and $F_{measure}$ to measure its overall effectiveness.

$$Recall = \frac{TP}{TP+FN}, \; Precision = \frac{TP}{TP+FP} \tag{4.1}$$

$$F_{\text{measure}} = 2 * \frac{Recall * Precision}{Recall + Precision} \tag{4.2}$$

The table shows that the reconstruction framework and plugins effectively supported the reconstruction of the microservices composing the Lakeside Mutual application but for the case of the *Risk-Management*

server, which is developed in NodeJS, which is not yet supported by our approach, but could be seamlessly integrated with a plugin. A similar consideration applies to the reconstructed data structures, viz., the operation's in- and outgoing data types. The discrepancy in the data structures results from the fact that the plugins do not support the reconstruction of external dependencies, where the source code is not present for the reconstruction, e.g., Spring dependencies. The framework and plugins also recover complex data types as data structures with LEMMA's domain models. Moreover, we recover the deployment specifications for all Spring-based microservices, including infrastructure components, e.g., the Eureka Server or the Spring Boot Admin application.

```
NAMES                                  PORTS
lm-customer-self-service-frontend-1    0.0.0.0:3000->80/tcp
lm-risk-management-server-1            0.0.0.0:50051->50051/tcp
lm-policy-management-frontend-1        0.0.0.0:3010->80/tcp
lm-customer-self-service-backend-1     0.0.0.0:8080->8080/tcp
lm-customer-management-frontend-1      0.0.0.0:3020->80/tcp
lm-customer-management-backend-1       0.0.0.0:8100->8100/tcp
lm-policy-management-backend-1         0.0.0.0:8090->8090/tcp,
lm-customer-core-1                     0.0.0.0:8110->8110/tcp
lm-eureka-server-1                     0.0.0.0:8761->8761/tcp
```

Figure 4.12: Accessibility of running microservices in the original Lakeside Mutual application.

```
NAMES                                  PORTS
lm-customer-self-service-frontend-1    80/tcp
lm-customer-self-service-backend-1     8080/tcp
lm-customer-management-frontend-1      80/tcp
lm-policy-management-frontend-1        80/tcp
lm-risk-management-server-1            50051/tcp
lm-customer-management-backend-1       8100/tcp
lm-policy-management-backend-1         8090/tcp, 61613/tcp, 61616/tcp
lm-customer-core-1                     8110/tcp
lm-apigateway-1                        0.0.0.0:80->80/tcp
lm-eureka-server-1                     8761/tcp
```

Figure 4.13: Accessibility of running microservices in the refactored version obtained with our approach.

We used the reconstructed LEMMA models to automatically detect the microservices' security smells present in the application (Section 4.2.2), by focusing on the two smells currently supported by our approach, viz., *Publicly Accessible Microservices* and *Insufficient Access Control*, which we observed on the application by inspecting its source code and a running instance of the system security smells [104] (while also proposing resolution strategies for both of them). However, due to the expressiveness of LEMMA's aspect functionality to include metadata in the models, we believe it is possible to extend our current approach to detect and resolve other security smells. For instance, while running the Lakeside Mutual application, we observed that all its microservices were configured to be exposed on different ports of the hosts, therefore all being affected by instances of the *Publicly Accessible Microservices* smell, and possibly subject to direct attacks by external, malicious clients. We also observed that no access control was configured in Lakeside

Mutual, meaning that its microservices were all affected by instances of the *Insufficient Access Control* smell.

With the above knowledge in mind, we checked whether the smell detection and refactoring featured by our approach was capable of detecting and resolving the *Publicly Accessible Microservices* and *Insufficient Access Control* smells affecting Lakeside Mutual's microservices by processing the reconstructed LEMMA models of its MSA. Notably, all smell instances were detected by our LEMMA-based framework, and we selected the suggested refactorings to resolve them. More precisely, we first selected the resolution of *Insufficient Access Control* smells, which adapted the LEMMA models by specifying that all microservices should use OAuth 2.0, as recommended in the literature [105]. We then selected the introduction of an API gateway to resolve the *Publicly Accessible Microservices* smells, and this adapted the LEMMA models by introducing the infrastructural components implementing the gateway itself, and configuring the Lakeside Mutual's microservices not to be exposed externally.

The resolution of *Publicly Accessible Microservices* smells was also automatically implemented by our LEMMA-based framework by adapting the Docker-based deployment to reflect what is specified in LEMMA. This can be observed in Figure 4.13, which shows that (a) Lakeside Mutual's microservices were all publicly accessible in the original version of the application, whereas (b) they were reachable only internally after the refactoring by relying on an API gateway to expose them externally. As a result, Lakeside Mutual's microservices were no more exposed to direct attacks by malicious external clients, with the gateway allowing the enforcement of further security measures, e.g., firewalling or rate limiting.

4.3.3 Answer to Research Questions

This subsection elaborates on the results of the validation processes related to RQ1 to RQ4.

Answer to RQ1: *The overall accuracy of the recovered architecture design of the Lakeside Mutual application is 87 percent.* Table 4.1 shows the results of the reconstructed microservice-specific concepts for the architecture design of the software application. The unrecovered concepts, e.g., the missing microservice, occurred due to a yet unsupported technology of the MAR framework. However, due to the extensibility of the plugin functionality, the technology can be integrated seamlessly. The reconstructed LEMMA models, adapted source code, and recovery results for the student management application are provided in the auxiliary materials[2].

Answer to RQ2: *LEMMA's validation functionalities can detect both security smells in the reconstructed models.* Our approach extended the validation functions (c.f. Figure 4.8) of LEMMA to detect the mi-

[2] https://drive.google.com/drive/folders/1W1WE0P_YSc_xx-q_DWHSAaTcRpLngjGG?usp=drive_link

croservice security smell of Insufficient Access Control and Publicly Accessible Microservices in the reconstructed models of capturing the architecture design of the Lakeside Mutual Application.

Answer to RQ3: *LEMMA's model refactoring functionality enables the resolution of both security smells in the reconstructed models.* For the end-to-end security smell resolution approach, we extended LEMMA with the functionality to resolve the detected security smells (c.f. Figure 4.7) by using a model-to-model transformation [26] and, therefore, adapt the model automatically to resolve the smell by a given smell-specific refactoring strategy.

Answer to RQ4: *The security smells are also resolved in the source code using code generation based on the refactored models (c.f. Figure 4.13)* The final step of our approach uses code generation to resolve the security smell in the implementation of the application. The security smell of Publicly Accessible Microservices is resolved automatically without manual adaption of the source code. However, to resolve the security smell of Insufficient Access Control, LEMMA functionality provides a guide to enforce sufficient access control by manually adapting the source code.

4.4 Threats to Validity

Three potential threats to validity classified by Wohlin et al. [152] may apply to our approach. These are the threats to the external, internal, and construct validity, which we discuss hereafter.

The *external* validity concerns the applicability of a set of results in a more general context. Currently, our approach is tailored specifically for Java-based MSAs, posing a potential threat to external validity. Notably, this is a challenge that we acknowledge, and it is important to highlight that our current focus on Java-based MSAs serves as a demonstrator. We intend to extend support for other programming languages in our ongoing and future work.

The *construct and internal* validity of a study concern the generalizability of the constructs under study and the method employed to study and analyze data, respectively. To mitigate potential threats to the construct and internal validity of our proposed approach, we have transparently reported the specifics of the reconstruction, detection, and refactoring processes. We have also provided the overall accuracy of the automatic reconstruction architecture design, which allows the models to be updated automatically as the MSA evolves. Finally, we have also provided auxiliary materials associated with the reconstructed LEMMA models.

4.5 Related Work

Microservice security smells have been recently proposed in [105], and how to automatically detect and refactor them is still an open issue [23]. Indeed, to the best of our knowledge, the only available work in this direction are [29] and [104]. [29] introduces KubeHound, a tool for detecting security smells in MSAs deployed with Kubernetes. [105] instead proposes a trade-off analysis to support deciding whether to refactor smells, assuming them to have already been detected. Our approach is, therefore, the first enabling to automatically detect security smells and to support deciding how to refactor them, while also enabling the automatic implementation of chosen refactorings.

Methods and tools for securing MSAs exist, however. For instance, [142] proposes Pomegranate, a fully automated test tool suite that can help developers detect security issues in MSAs. Pomegranate essentially encapsulates open-source vulnerability scanning tools into one suite, exploiting them to detect security vulnerabilities in MSAs. We differ from Pomegranate in our objectives, as we focus on detecting security smells in MSAs and on supporting developers in choosing the refactoring for resolving detected smells while also enabling the implementation of chosen refactorings automatically.

Other solutions for securing MSAs are given by the production-ready tools for security analysis, e.g., Kubesec.io, Checkov, and SonarQube. These analysis tools provide validated solutions for vulnerability assessment and security weaknesses detection, which can also be used for microservices applications. Our proposal complements the analyses enacted by the above-listed tools, enabling the detection and refactoring of the microservice security smells proposed in [105], in addition to the vulnerabilities and security weaknesses they identify. Additionally, production-ready tools such as Fortify and Coverity analyze the source code for the occurrence of code smells, whereby our proposed solution addresses security and architecture smells, resulting from a bad design.

Additional existing approaches provide the possibility to identify and resolve other types of smells for microservices. [101], [146], and [133] propose two different solutions for detecting architectural smells in MSAs. They both share our baseline idea of starting from smells identified with industry-driven reviews, with [101] and [146] picking those from [137], while [133] picking those from [87]. [133] also shares our baseline idea of using MDE to detect and refactor smells. The main difference between [101], [133], and our proposal however relies on the considered types of smells, with [101] and [133] focusing on architectural smells. We rather complement their results by enabling the automatic detection and refactoring of microservice smells from [105].

Similar considerations apply to [9] and [51], which both organize information retrieved from practitioners or industry-scale projects into guidelines for designing microservice applications while avoiding the

inclusion of well-known architectural smells. We complement these works by enabling the detection of microservices' security smells and refactoring them to resolve their possible effects.

Finally, it is also worth relating our microservice-oriented proposal with existing solutions for detecting smells in classical services. For instance, [4], [40], and [124] present three different MDE approaches to detect architectural smells in a service, with [4] and [40] relying on UML to model services, while [124] relying on Archery. [6] and [144] instead allow to analysis of the source code of a service to detect the smells therein, also supporting refactoring to resolve the occurrence of identified smells. Similarly to the above-discussed approaches, the difference between our proposal and those in [4], [6], [40], [124], and [144] resides in the considered smells, with our proposal complementing their results by enabling to detect and refactor microservices' security smells.

Chapter 5

Should microservice security smells stay or be refactored?

Microservices pose new security challenges, including so-called *security smells* [105]. These are possible symptoms of a bad decisions (though often unintentional) while designing or developing an application, which may impact the overall application's security. The effects of security smells can be mitigated by refactoring the application or the services therein while not changing the functionalities offered to clients. Even if applying refactorings requires effort from development teams, they can help improving the overall application quality [11].

Consider, for instance, the *centralized authorization* security smell, which occurs when a single component centrally authorizes all external requests sent to a microservice-based application [105]. When this occurs, the requests exchanged among an application's microservices are trusted without further authorization controls, making them prone to, e.g., confused-deputy attacks, which may compromise the application's authenticity [55]. To mitigate the effects of *centralized authorization*, application architects may *use decentralized authorization*, which consists of refactoring applications to enforce fine-grained authorization controls at microservices-level [105]. This would improve the application's authenticity, while also impacting on other quality attributes and on the adherence of the application to microservices' key design principles.

Whether to keep a security smell, or to apply a refactoring to mitigate its effects, is a design decision that deserves careful analysis. This can indeed impact on several aspects of an application, including other quality attributes besides security, like performances and maintainability [28], and on its adherence to microservices' key design principles [87]. Architects must therefore analyze the possible different trade-offs

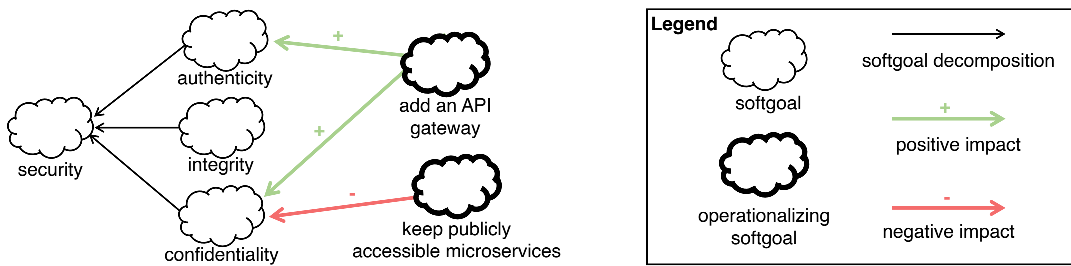


Figure 5.1: An example of SIG.

among such aspects, to balance them and take informed decisions [61]. This is inherently complex, as it requires to holistically consider the impact of keeping a smell/applying a refactoring on security and other quality attributes, and on the adherence to microservices' key design principles.

In this chapter, we introduce a first support for analyzing the possible trade-offs related to keeping a security smell in a microservice-based application or applying some refactoring. More precisely, we introduce a method to enact such trade-off analysis using *Softgoal Interdependency Graphs* (SIGs) [25], which provide a visual and holistic panorama of the positive/negative impacts of each security smells and refactorings on each software quality attribute and each microservices' key design principle.

We also illustrate our method by applying it to the *centralized authorization* security smell and its possible refactoring. In particular, we show the SIG displaying their possible impacts on microservices' key design principles and on ISO 25010 security, performance efficiency, and maintainability properties [55], and we discuss how it helps reasoning on whether it is worthy to apply the refactoring or keep the smell in the application.

5.1 Background: SIGs

SIGs allow the systematic modeling of the impact of design decisions (called *operationalizing softgoals*) on quality attributes (called *softgoals*). More precisely, they allow to model the positive or negative impact of different design decisions on the considered quality attributes, hence providing a visual support for an application architect to start a trade-off analysis process [25].

Figure 6.1 illustrates an example of SIG, displaying the impact of two design decisions (viz., the operationalizing softgoals *keep publicly accessible microservices* or *add an API gateway*) on *security*, with the latter decomposed in three different properties (viz., the softgoals *confidentiality*, *integrity*, and *authenticity*). The positive/negative impact of each decision is depicted as a green/red arrow. Impacts are labeled with +/− to denote that a design decision helps achieving/hurts a quality attribute. Other possible labels

are ++/-- to denote that a design decision ensures that a quality attribute is realized/broken, and S+/S- to indicate an expected positive/negative impact, yet to be empirically confirmed.

5.2 Towards a SIG-based trade-offs analysis

We propose the use of SIGs to support the analysis of trade-offs related to keeping a security smell in a microservice-based application or applying the refactorings known to mitigate its effects. Each SIG is intended to provide a holistic view of the positive/negative impacts (and their rationale) on software quality attributes and microservices' key design principles. Following the recommendations of Oliveira et al. [28], we propose each SIG to display the impact of security smells and refactorings on at most four softgoals. Adding more softgoals will complicate the visualization of the impacts, hence making the trade-off analysis to get too complex to manage.

Given that our ultimate goal is to analyze the impact of microservices' security smells and refactorings, we introduce softgoals for *security* and *microservices' key design principles*. Specifically, we consider *security* as per its definition by the ISO 25010 software quality standard [55], and we take the microservices' key *design principles* defined by Neri et al. [87].

These two softgoals are the main ones that we work with, and should be part of every generated SIG. The other two softgoals depend on the desiderata of an application architect. In this research, to illustrate our method, we also consider *performance efficiency* and *maintainability* from the ISO 25010 software quality standard [55] as two concrete examples.

To enact a more accurate analysis, we decompose the considered softgoals, hence considering all the different aspects of each softgoal separately. In this case, this means to decompose the above-mentioned softgoals as follows:

- *Security* is decomposed into *confidentiality*, *integrity*, and *authenticity*. These are the sub-properties of *security* in the ISO 25010 software quality standard [55], which are also known to be affected by security smells [105].
- Microservices' key *design principles* are decomposed into *independent deployability*, *horizontal scalability*, *failure isolation*, and *decentralization*, following the taxonomy proposed by Neri et al. [87].
- *Performance efficiency* is decomposed into its known sub-properties, as per its definition in the ISO 25010 software quality standard [55], i.e., *resource utilization*, *time behaviour*, and *capacity*.
- *Maintainability* is decomposed into its known sub-properties, as per its definition in the ISO 25010

software quality standard [55], i.e., *modularity*, *reusability*, *analysability*, *modifiability*, and *testability*.

We exploit SIGs to analyze two different types of design decisions (or *operationalizing softgoals*). These are (i) keeping a security smell or (ii) spending efforts in applying the refactorings allowing to mitigate its effects. This will enable the analysis of the impacts of the security smells and refactorings that we systematically elicited in our previous work [105].

The actual impact of security smells and refactorings is modeled by arcs connecting design decisions to softgoals. Each arc is labelled with "++" or "+", if the design decision makes or helps realizing a softgoal, respectively. An arc is instead labelled with "−−" or "−", if it breaks or complicates realizing a softgoals, respectively. We use such labels for the impacts known thanks to our former work done in [87, 105]. There are however other impacts of the smells that we expect to occur, even if they are yet to be empirically validated. For these, we use "S−" and "S+", depending on whether the expected impact is negative or positive, respectively. The validation of such expected impacts, as well as of those of other known security smells and refactorings, is in the scope of our future work.

5.3 Illustrative example

Figure 5.2 illustrates our method applied to the *centralized authorization* security smell, which occurs when a single component is used to centrally authorize external requests sent to a microservice-based application [105]. The *centralized authorization* smell hurts the *authenticity* of the application since the requests exchanged among microservices are trusted without further authorization controls, making them prone to, e.g., *confused-deputy attacks*. This smell also hurts the *decentralization* design principle, since it happens when microservices depend on a central component to manage authorization [87].

We also expect the *centralized authorization* smell to hurt the *time behavior* of the application, since the central authorization component can become a bottleneck if the application considerably grows in the number of microservices. We also expect this smell to hurt the *analysability* of the application, since having a central authorization component makes complex to assess the impact of a change in the authorization mechanism on all microservices. Finally, we expect a positive impact on the *testability* property of the application, since the authorization control should be tested only in one component.

To mitigate the effects of *centralized authorization*, application architects may *use decentralized authorization*, which consists of refactoring the application to enforce fine-grained authorization controls at the microservices-level [105]. The *use decentralized authorization* refactoring helps achieve the *authenticity*

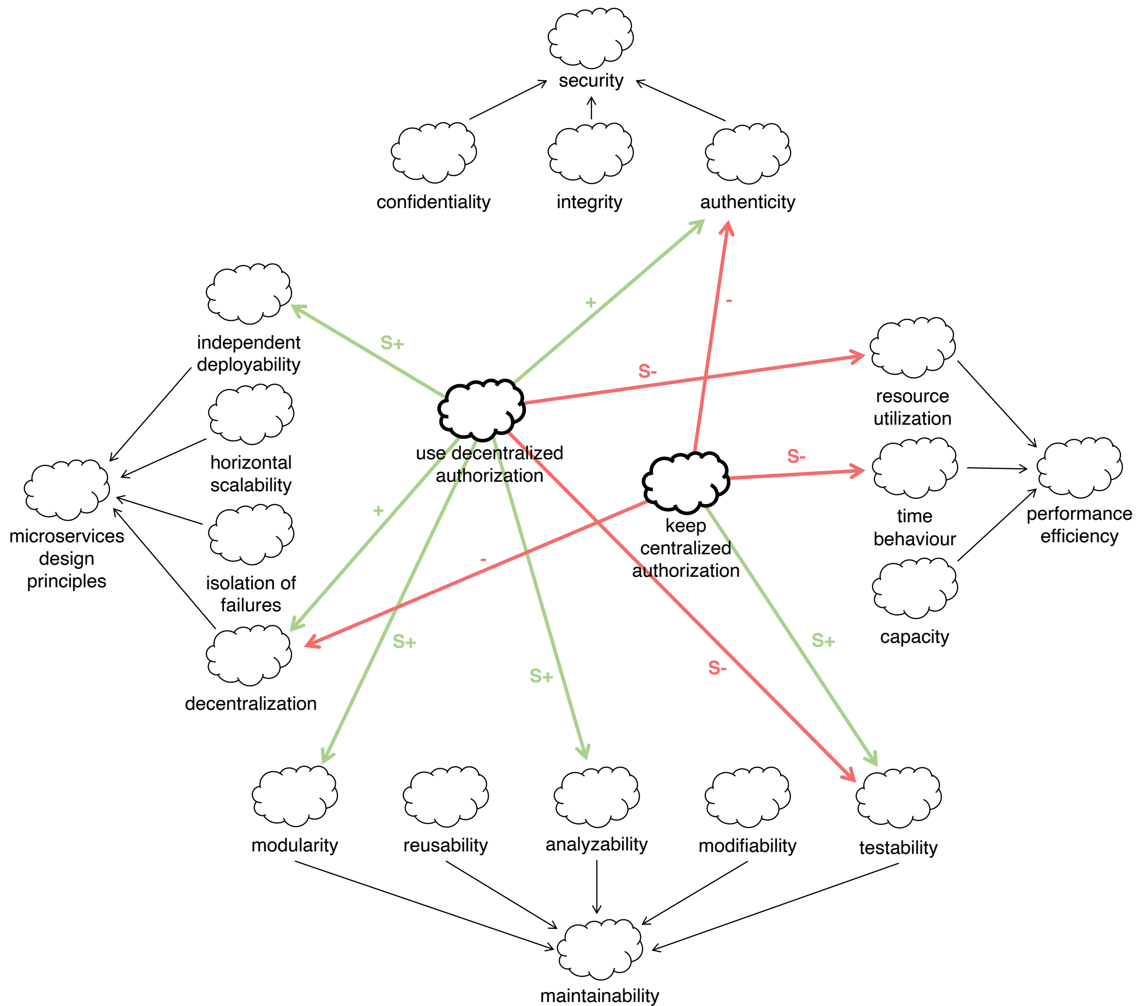


Figure 5.2: SIG displaying the impact of keeping the *centralized authorization* smell and its corresponding refactoring on the considered softgoals.

security property. This refactoring also helps achieve the *decentralization* design principle, since it suggests decoupling the central authorization component and that each microservice has its own fine-grained controls.

By applying the refactoring, microservices do not depend on a central authorization component. We expect that this helps achieving the *independent deployability* design principle. We also expect that this refactoring has a positive impact on *modularity* and *analysability* since each fine-grained authorization control can be modified independently, minimizing the impact that this has on the rest of the application. On the other hand, we expect the considered refactoring to have a negative impact on the *testability* of the application, since the fine-grained authorization controls need to be tested independently, hence scaling in complexity with the number of microservices. We also expect a negative impact on the *resource utilization*

property, since this approach requires more information to be transmitted for each request (e.g., always including authorization tokens), therefore causing a higher consumption of network resources.

Figure 5.2 hence presents a holistic view of the positive/negative impacts of the *centralized authorization* security smell, and its refactoring *use decentralized authorization*. With the above described impacts, we can start then a trade-off analysis. We can indeed reason on whether it is worthy to apply the refactoring, or whether it is more convenient to keep the smell in our microservice-based application. We can start by defining which of these softgoals is more important, which depends on the context and desiderata of the application architecture. This SIG allows us to prioritize the softgoals and select the design decision that best suits the application needs. For example, if *resource utilization* is our priority (e.g., our application runs on a platform with limited resources, or increasing resource utilization is too costly), we may decide to keep the smell. Instead, if we have no restrictions on *resource utilization*, and if *testability* has a lower priority than the other softgoals visualized in Figure 5.2, then applying the *used decentralized authorization* refactoring seems to be the best option.

5.4 Softgoal Interdependency Graphs

Using the method described in Section 5.2 and as illustrated in Section 5.3, we can generate the SIGs of other security smells. For this purpose, we will consider the information collected in previous works [87, 105], and we will add the rationale behind the expected positive/negative as claims in the SIGs.

5.4.1 Insufficient Access Control

This smell occurs on the microservices of an application that is not enforcing access control. This can possibly violate the confidentiality of the data/business functions of the microservices where access control is lacking, as attackers can trick a service and get data that they should not be able to get. The possible effects of this smell can be mitigated by exploiting *OAuth 2.0*, which would enable microservices to control accesses. Figure 5.3 shows the impacts of *Insufficient Access Control* and its refactoring.

We expect that the *Insufficient Access Control* smell hurts *Isolation of failures*, since if this smell is present and a microservice has been compromised, an attacker could abuse the lack of access control, provoking a cascade of security failures. We also expect this smell to hurt *analyzability*. Because in the event of an attack, it becomes more complex to diagnose the extent of the damage caused, as microservices, e.g., can be exposed to the Confused Deputy Problem.

With regards to the impacts of its refactoring, we expect that the use of *OAuth 2.0* hurts *resource uti-*

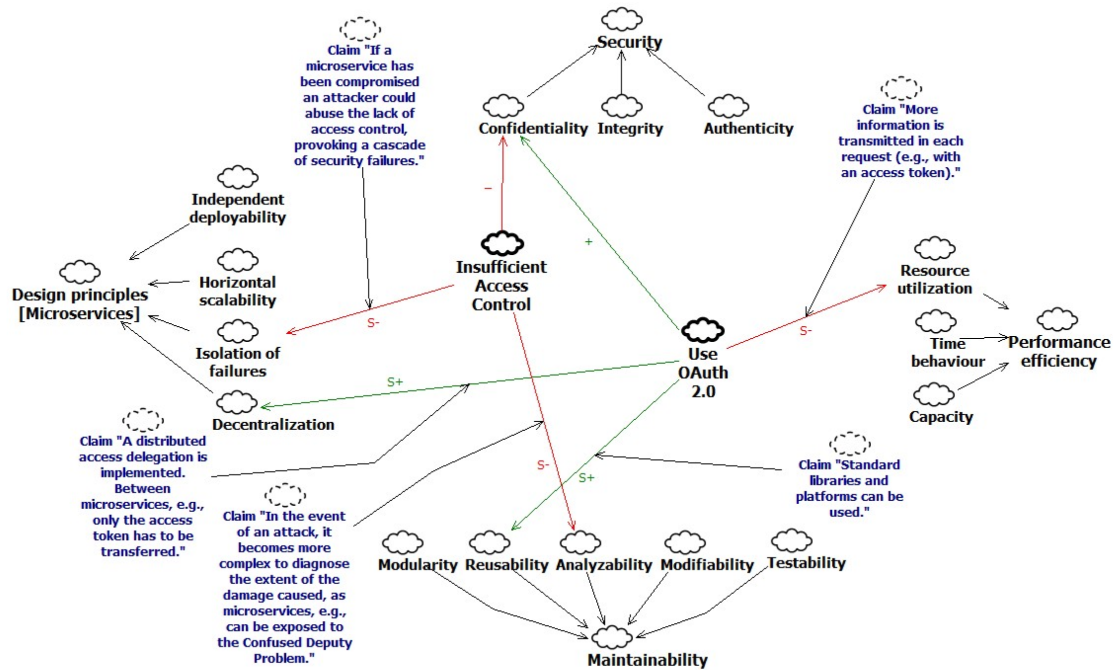


Figure 5.3: SIG displaying the impact of keeping the *Insufficient Access Control* security smell and its corresponding refactoring.

lization since more information is transmitted in each request (e.g., with an access token). We also expect a positive impact on *reusability*, because standard libraries and platforms can be used. Finally, we expect a positive impact on *decentralization*, since a distributed access delegation is implemented and between microservices, e.g., only the access token has to be transferred.

5.4.2 Publicly Accessible Microservices

A microservice of an application is publicly accessible when it can be directly accessed by external clients. This increases the application's attack surface and reduces its overall maintainability and usability. The suggested refactoring is making microservices accessible *only* through a newly added *API Gateway*, which would act as an entry point for the application. Figure 5.4 shows the impacts of *Publicly Accessible Microservices* and its refactoring.

We expect that the *Publicly Accessible Microservices* smell hurts the application *testability* since there are multiple access points to the microservices-based application, each one of them should be heavily tested independently. We also expect that it hurts *modifiability*, because if a security policy changes this may imply a change to all entry points of the application. Finally, we expect a negative impact on *isolation of failures*.

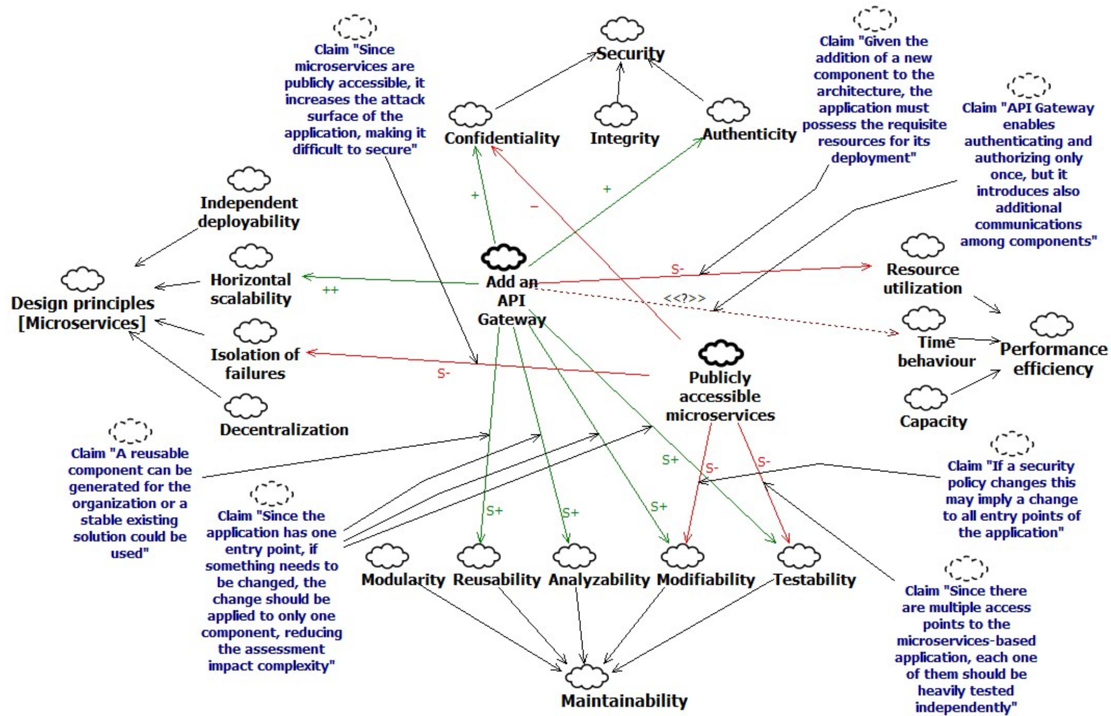


Figure 5.4: SIG displaying the impact of keeping the *Publicly Accessible Microservices* security smell and its corresponding refactoring.

With regards to the impacts of its refactoring, we expect that adding an *API Gateway* hurts *resource utilization* since a new component is added to the application, it is necessary to have the necessary resources available. We also expect a positive impact on *reusability*, because a reusable component can be generated for the organization, or a stable existing solution could be used. Finally, we expect a positive impact on *analizability, modifiability, and testability* since the application now has one entry point, if something needs to be changed, the change should be applied to only one component, reducing the assessment impact complexity.

5.4.3 Unnecessary Privileges to Microservices

This smell occurs on microservices that are granted privileges, e.g., access levels or permissions, which they do not need to deliver their business functions. The suggested refactoring is *following the least privilege principle*, namely ensuring that microservices have access *only* to the least set of functionalities and data needed. Figure 5.5 shows the impacts of *Unnecessary Privileges to Microservices* and its refactoring.

We expect that the *Unnecessary Privileges to Microservices* smell hurts the *resource utilization*. If a

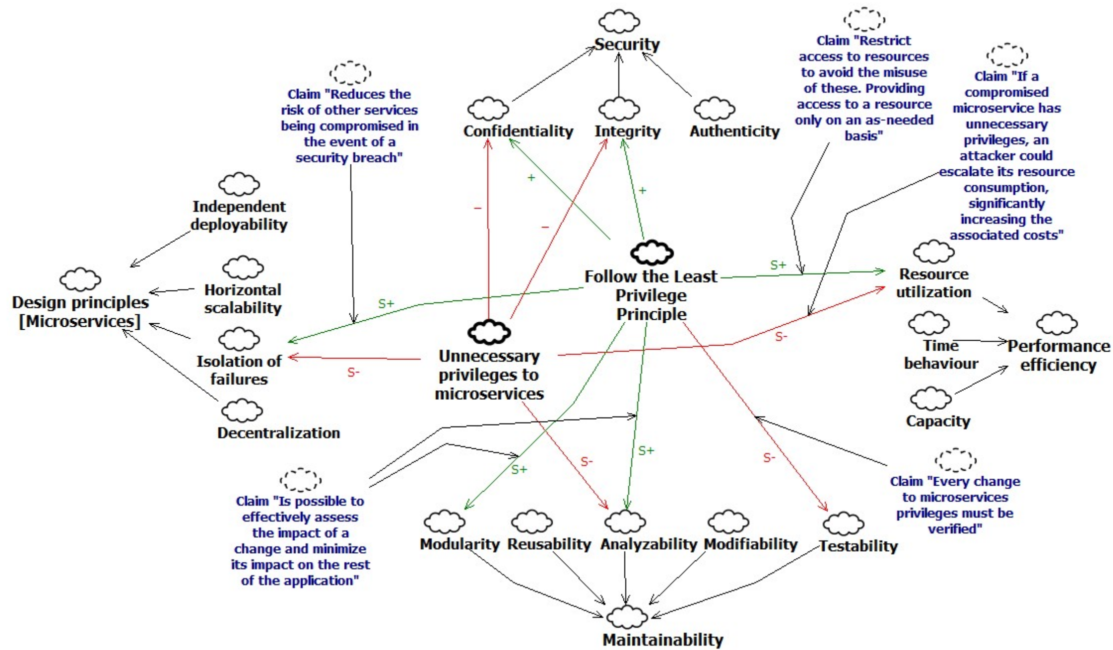


Figure 5.5: SIG displaying the impact of keeping the *Unnecessary Privileges to Microservices* security smell and its corresponding refactoring.

compromised microservice has unnecessary privileges, an attacker could escalate its resource consumption, significantly increasing the associated costs. We also expect a negative impact on *isolation of failures*, because this smell increases the risk of other services being compromised in the event of a security breach. Finally, we expect a negative impact on *analizability*, since this security smell increases the complexity of assessing the impact of a change.

With regards to the impacts of its refactoring, we expect that *Following the Least Privilege Principle* has a positive impact on *resource utilization*. This principle restricts the access to resources to avoid the misuse of these, providing access to a resource only on an as-needed basis. We expect a positive impact on *isolation of failures* because it reduces the risk of other services being compromised in the event of a security breach. We also expect a positive impact on *modularity* and *analyzability*, because reduces the complexity of effectively assessing the impact of a change, while minimizing its impact on the rest of the application. Finally, we expect a negative impact on *testability*, since every change to microservices privileges must be carefully verified.

5.4.4 Non-Secured Service-to-Service Communications

This smell occurs whenever microservices in an application interact without establishing a secure communication channel, even if they are running in the same internal network. Microservices should rather follow a "zero-trust" approach, by relying on *Mutual TLS* to secure service-to-service communications. Figure 5.6 shows the impacts of *Non-Secured Service-to-Service Communications* and its refactoring.

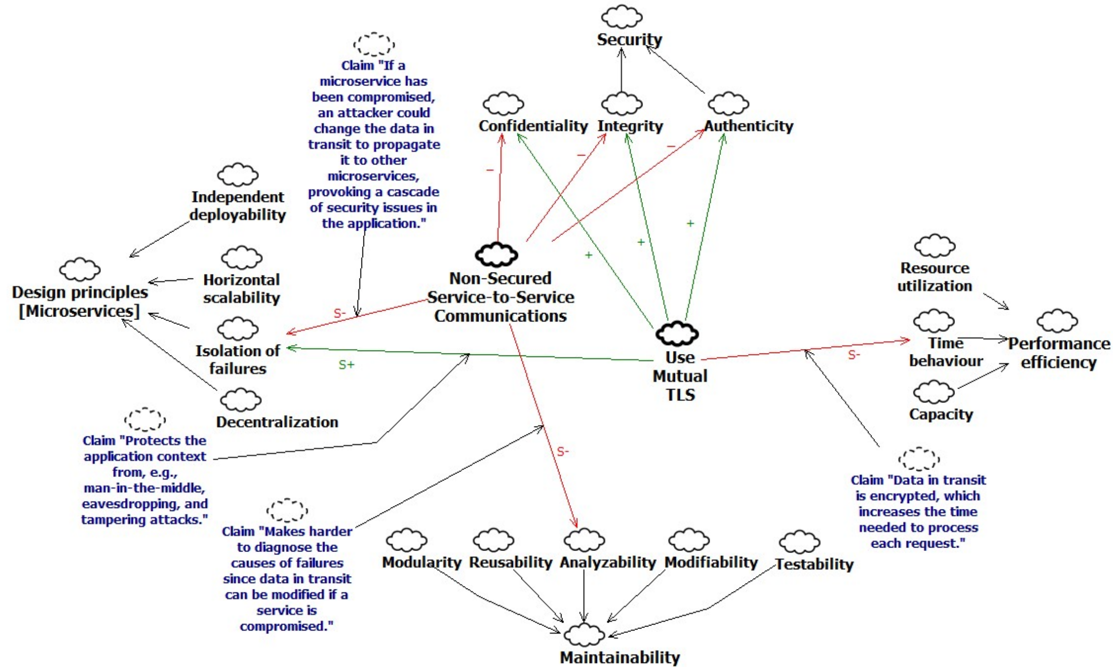


Figure 5.6: SIG displaying the impact of keeping the *Non-Secured Service-to-Service Communications* security smell and its corresponding refactoring.

We expect that the *Non-Secured Service-to-Service Communications* smell hurts *isolation of failures*. If a microservice has been compromised, an attacker could change the data in transit to propagate it to other microservices, provoking a cascade of security issues in the application. We also expect a negative impact on *analyzability*, because this smell makes it harder to diagnose the causes of failures since data in transit can be modified if a service is compromised.

With regards to the impacts of its refactoring, we expect that *Mutual TLS* hurts the application *time behavior*, since data in transit is encrypted, which increases the time needed to process each request. We also expect a positive impact on *isolation of failures*, since this protects the application context from, e.g., man-in-the-middle, eavesdropping, and tampering attacks.

5.4.5 Unauthenticated Traffic

This smell occurs when a microservice receives unauthenticated requests from external clients or from other microservices of its same application, which may result in violating the application's authenticity. The suggested refactorings are using *Mutual TLS*, as well as *OpenID Connect*. The latter exploits ID tokens containing cryptographically signed user claims, which can be checked for integrity, and which can be used to perform access control at the microservice level. Figure 5.7 shows the impacts of *Unauthenticated Traffic* and its refactoring.

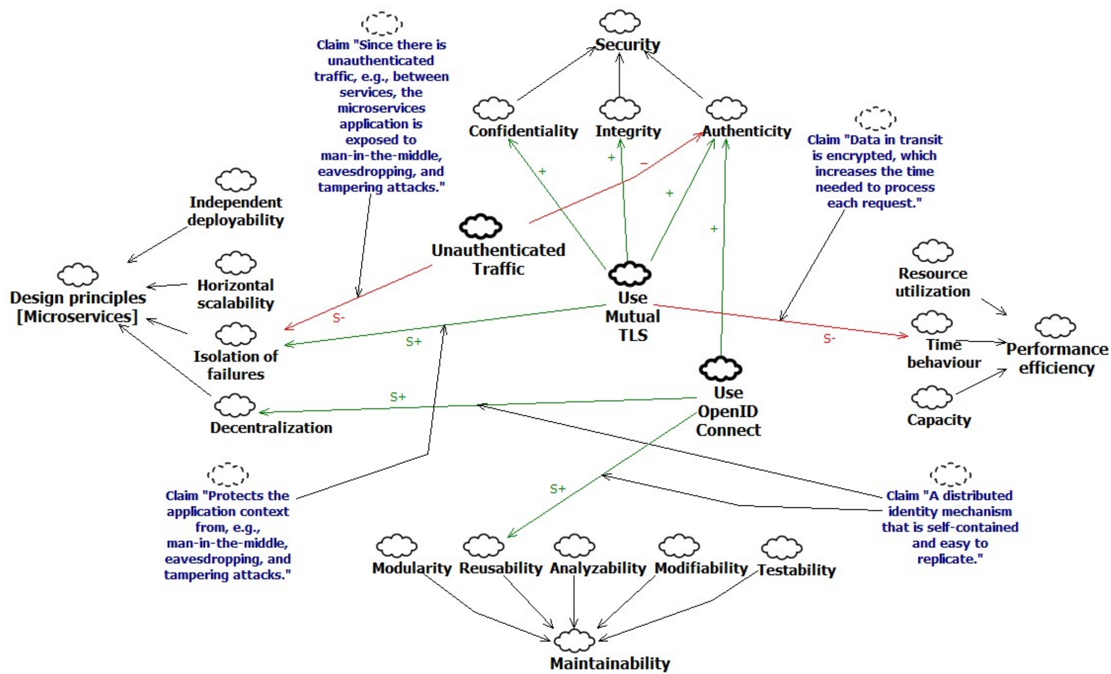


Figure 5.7: SIG displaying the impact of keeping the *Unauthenticated Traffic* security smell and its corresponding refactoring.

We expect that the *Unauthenticated Traffic* smell hurts *isolation of failures* since there is unauthenticated traffic, e.g., between services, the microservice application is exposed to man-in-the-middle, eavesdropping, and tampering attacks.

With regards to the impacts of its refactoring, besides the ones already defined for *Mutual TLS*, we expect that *OpenID Connect* helps to improve the *decentralization* and *reusability* of the application, since it is a distributed identity mechanism that is self-contained and easy to replicate.

5.4.6 Multiple User Authentication

This smell occurs when a microservice application provides multiple endpoints for user authentication, which can be exploited by an intruder to authenticate as an end-user. This increases the application's attack surface, and could also result in maintainability and usability issues. The suggested refactoring is using a *Single Sign-On*, namely using a single entry point to handle user authentication and to enforce security for all the user requests entering a microservice application. This approach facilitates log storage and auditing tasks, by providing a centralized entry point that performs user authentication. The single sign-on can be realized by adding an *API gateway* and using *OpenID connect*, which are refactoring known to resolve other security smells, as recapped earlier in this section. Figure 5.8 shows the impacts of the *Multiple User Authentication* security smell and its refactoring.

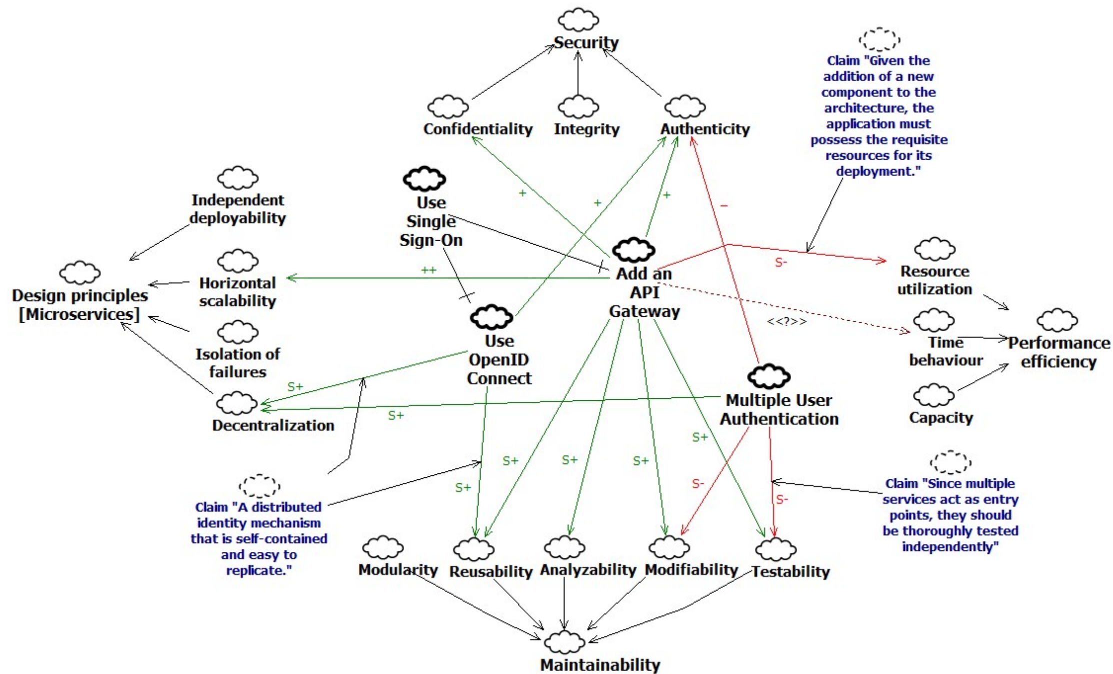


Figure 5.8: SIG displaying the impact of keeping the *Multiple User Authentication* security smell and its corresponding refactoring.

We expect that the *Multiple User Authentication* smell hurts the *testability* of the application. Since multiple services act as entry points, they should be thoroughly tested independently. With regards to the impacts of its refactoring, we consider the impacts of *Adding an API Gateway* and *using OpenID Connect*.

5.5 Related work

Some previous research has been done to enable trade-off analysis for decision-making process by software architects. Thus, Orellana et al. [93] provided a systematic approach to classify architectural patterns and analyze trade-offs among them using SIGs; This approach differs from ours in the objectives, since they focus on the impact of architectural patterns on systems' quality, but we focus instead on the impact of microservices' security smells and refactorings; i.e., not only on the quality of microservice-based applications, but also on their adherence to microservices' key design principles [87].

Pasquale et al. [97] proposed a requirements-driven approach to automate security trade-off analysis, using a security goal model that allows software engineers to identify and analyze trade-offs among them. In a similar vein, the proposal of Elahi and Yu [34] extends the $i*$ notation by providing a goal-oriented approach to model and analyze security trade-offs of multi-actor systems. Our study differs from these two in their objectives: they focus on security requirements and goals, respectively, but we focus instead on known security smells while also considering microservices' key design principles.

Another related approach was proposed by Márquez et al. [75], the first one that used SIGs with microservices. They proposed to use SIGs to illustrate the impact of microservices design patterns on high availability properties. The resulting SIG can be used to evaluate whether a given design pattern should be included in a microservice-based architecture. However, they addressed only high availability properties, and we focus instead on security smells and their refactorings, examining their impact on multiple quality attributes and microservices' key design principles.

Martini et al. [77] also support reasoning on the trade-off between keeping an identified smell and applying a refactoring, but rather based on technical debt. They indeed measure the negative impact of a smell as the interest paid by keeping the smell, comparing this to the principal cost of applying a refactoring to resolve such smell. They however consider impacts only under the technical debt perspective, in terms of its principal and interest, whereas we propose to trade-off among multiple aspects, like those we discussed in this chapter.

Finally, it is worth mentioning the work of Oliveira et al. [28], which presented a systematic method for architectural trade-off analysis based on patterns. This research aimed to help microservice-based application architects to identify and understand the patterns that best suit their specific needs in a project. Our research uses SIGs to provide a holistic view of the impact of microservices' security smells and refactorings on quality attributes and microservices' key design principles.

Chapter 6

To Security and Beyond: On The Impacts of Microservice Security Smells and Refactorings

Keeping a security smell or applying a refactoring to mitigate its effects are design decisions that deserve careful analysis. Such design decisions indeed impact multiple different aspects of an application, including the adherence to microservices' key design principles [87] and other quality attributes besides security, such as maintainability and performance efficiency, which are crucial in microservice applications [134].

Consider the *centralized authorization* security smell, which occurs when a single component centrally authorizes all external requests sent to a microservice-based application [105]. When this security smell is present, the requests exchanged among the application's microservices are trusted without additional authorization controls, leaving them vulnerable to various attacks such as confused-deputy attacks, which can compromise the application's authenticity [55]. To mitigate the effects of *centralized authorization*, application architects may *use decentralized authorization*, which involves refactoring the application to implement fine-grained authorization controls at the microservices level [105]. This would not only improve the authenticity of the application but also have a ripple effect on other quality attributes and strengthen the adherence to key microservices design principles.

As knowing such impacts helps to make informed decisions about keeping a security smell or applying a refactoring [104], the research question we try to answer in this chapter is the following:

Are the security smells and refactorings in [105] impacting on maintainability, performance efficiency, and adherence to microservices' key design principles?

In this perspective, we start from the selected literature from [105] and we apply thematic analysis [10, 65] to systematically elicit 42 possible impacts of security smells and refactorings on applications' maintainability, performance efficiency, and adherence to microservices' key design principles. We then validate the elicited impacts, by analyzing the results of an online survey targeting practitioners and researchers working with microservice applications. Our analysis also shows that the interviewed practitioners and researchers are mostly aligned in their agreement with elicited impacts. A few misalignments emerged on possible impacts on testability and performance efficiency, with practitioners being more cautious in agreeing with them compared to researchers.

Consequently, we address our main research question by unveiling 35 validated impacts of microservices' security smells and refactorings on maintainability, performance efficiency, and key design principles. For each impact, we provide a statement describing the impact type (positive or negative), the property it affects, and the rationale behind it, together with the agreement of the interviewed practitioners and researchers.

Furthermore, to answer the need of visualizing impacts outlined in our previous work [104], we provide a comprehensive visual representation, using softgoal interdependency graphs [25], to illustrate both the positive and negative impacts of each microservice security smell and refactoring on maintainability, performance efficiency, and microservices' key design principle. Additionally, we also provide a tool that displays the validated impacts facilitating access to this information.

6.1 Background

This section provides a quick recap of the microservice security smells and refactorings described in Chapter 2 and softgoal interdependency graphs.

6.1.1 Smells and Refactorings for Microservice Security

We hereafter recall the smells and refactorings for microservice security proposed in our previous work [105], focusing on those whose impact is analyzed in this chapter.

Insufficient Access Control. This smell occurs on the microservices of an application that are not enforcing access control. This can possibly violate the confidentiality of the data and business functions of the microservices where access control is lacking, as attackers can trick a service and get data that they should not

be able to get.

The possible effects of this smell can be mitigated by exploiting OAuth 2.0, which would enable microservices to control accesses. OAuth 2.0 indeed provides a token-based access control system that lets a resource owner grant a client access to a particular resource on their behalf. OAuth 2.0 is hence a natural candidate to enforce access control in a microservice-based application at each level, thereby including controlling the accesses to each microservice.

Publicly Accessible Microservices. A microservice of an application is publicly accessible when it can be directly accessed by external clients. This increases the application's attack surface and reduces its overall maintainability and usability. Also, if each publicly accessible microservice performs authentication, the full set of a users credentials is required each time, increasing the likelihood of confidentiality violations (e.g., with the exposure of long-term credentials).

The suggested refactoring is making microservices accessible *only* through a newly added API Gateway, which would act as an entry point for the application. This would enable centralizing authentication, overall reducing the attack surface of the application and simplifying the authentication itself. In addition, by using this approach development teams can also secure all microservices behind a firewall, allowing the API gateway to handle external requests and then communicate with the microservices behind the firewall.

Unnecessary Privileges to Microservices. This smell occurs when microservices are granted unnecessary access levels, permissions, or functionalities that they do not need to deliver their business functions. As a result, resources are unnecessarily exposed, increasing the attack surface, and the risk for confidentiality and integrity violations.

The suggested refactoring is to follow the least privilege principle, namely ensuring that microservices have access *only* to the least set of functionalities and data needed to suitably perform their business function. This would help to contain attacks' effects, e.g., if an attacker gets control of a microservice by exploiting its software vulnerabilities.

Non-Secured Service-to-Service Communications. This smell occurs whenever microservices in an application interact without establishing a secure communication channel, even if they are running in the same internal network. This could result in confidentiality, integrity, and authenticity issues, e.g., intruders could intercept the communication between two microservices and change the data in transit to their advantage.

Microservices should rather follow a zero-trust approach, by relying on Mutual TLS to secure service-to-service communications Mutual TLS indeed creates a secure communication channel that features data encryption and mutual authentication, hence preventing, e.g., man-in-the-middle attacks.

Unauthenticated Traffic. This smell occurs when a microservice receives unauthenticated requests from

external clients or from other microservices of its same application, which may result in violating the application's authenticity. The microservices in an application should rather always authenticate and authorize incoming requests, to ensure that data arriving from external clients and exchanged among the application's microservices is trusted and has not been modified. If the traffic is not authenticated microservices are exposed to security attacks that may result in, e.g., tampering with data, denial of service, or elevation of privileges.

The suggested refactorings are using Mutual TLS, as well as OpenID Connect. The latter exploits ID tokens containing cryptographically signed user claims, which can be checked for integrity, and which can be used to perform access control at the microservice level.

Multiple User Authentication. This smell occurs when a microservice application provides multiple endpoints for user authentication, which can be exploited by an intruder to authenticate as an end-user. This increases the application's attack surface, and could also result in maintainability and usability issues.

The suggested refactoring is using a single sign-on, namely using a single entry point to handle user authentication and to enforce security for all the user requests entering a microservice application. This approach facilitates log storage and auditing tasks, by providing a centralized entry point that performs user authentication. The single sign-on can be realized by adding an API gateway and using OpenID connect, which are refactoring known to resolve other security smells, as recapped earlier in this section.

Centralized authorization. This smell occurs when a microservice application only handles authorization in one component, typically at the gateway of the application, while it does not enact any fine-grained authorization control at the microservices level. Such centralized authorization may result in authenticity violations, since microservices would trust the gateway based on its mere identity.

The suggested refactoring is using decentralized authorization, by transmitting an access token with each request, e.g., a JSON Web Token. The token provides a mechanism to safely pass user claims or data, together with a digital signature that guarantees its authenticity. Authorization can then be enforced also at the microservices level, as it gives each microservice more control to enforce its own access-control policies.

6.1.2 Softgoal Interdependency Graphs

Softgoal Interdependency Graphs facilitate the systematic modeling of the impact of design decisions (called *operationalizing softgoals*) on quality attributes (called *softgoals*). Specifically, they enable the representation of both positive and negative impacts of different design decisions on quality attributes, hence providing visual support for application architects to start a trade-off analysis process [25].

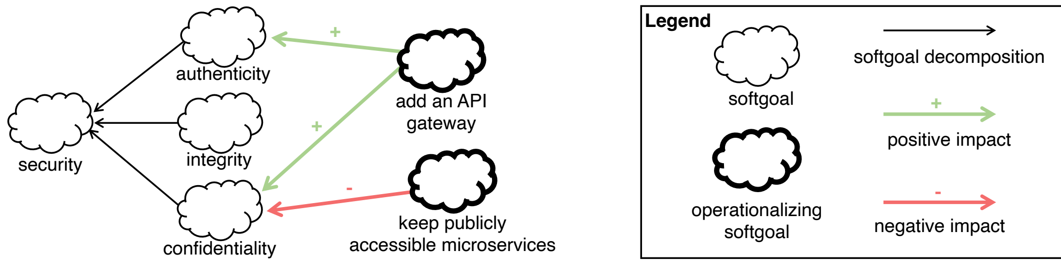


Figure 6.1: Example of a Softgoal Interdependency Graphs

Figure 6.1 presents a Softgoal Interdependency Graph exemplifying the impacts on *security* of two design decisions, namely, the operationalizing softgoals *keep publicly accessible microservices* or *add an API gateway*. Security is decomposed into three different security properties (viz., the softgoals *confidentiality*, *integrity*, and *authenticity*). The positive or negative impacts of each design decision are represented by green or red arrows, respectively. Impacts are also labeled with + or − to denote that a design decision helps to achieve or hurts a quality attribute. Other possible labels are ++ or −− to denote that a design decision unequivocally ensures or compromises a quality attribute.

In previous work [104], we introduced a method for conducting trade-off analyses utilizing Softgoal Interdependency Graphs (SIGs) [25]. SIGs offer a comprehensive visual representation of the positive or negative impacts of each security smell and refactoring on individual software quality attributes and microservices' key design principles. In this chapter, we will use the method introduced in [104] to illustrate the validated impacts of each security smell and refactoring.

6.2 Survey Design

We aim to complement the analysis of microservices' security smells and refactoring in our previous work [105], by assessing the possible impacts of both smells and refactorings on other properties than security. More precisely, we focus on two quality properties defined by the ISO 25010 standard [55], viz., *performance efficiency* and *maintainability*, which are crucial to microservices [134], as well as on the adherence of an application to microservices' key design principles [87]. We hereafter illustrate how we selected the primary studies from which to extract such possible impacts, as well as the process we enacted to analyze and assess such impacts.

6.2.1 Literature selection

The 55 white/grey primary studies providing the state-of-the-art/state-of-practice on smells and refactorings for microservices security were already identified and classified in [105], by following the guidelines for conducting systematic literature reviews in [63], combined with those in [44] for systematically reviewing grey literature. We hence started from such 55 primary studies and the smells and refactorings discussed therein, as shown in Figure 6.2

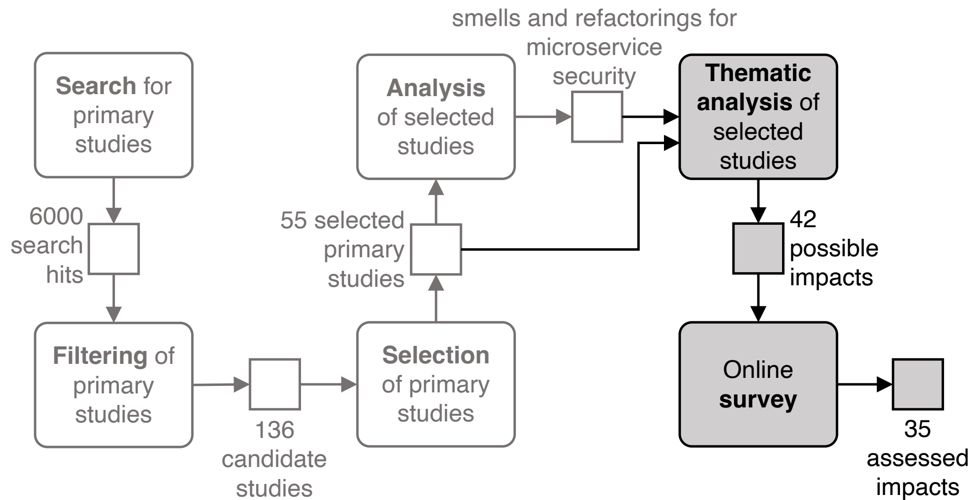


Figure 6.2: Research process. Grey boxes denote steps to elicit and assess possible impacts of smells and refactorings identified in our previous work [105] (whose steps are in white)

6.2.2 Thematic analysis

We analyzed the selected primary studies to elicit the possible impacts of the smells and refactorings from [105] on performance efficiency, maintainability, and adherence to microservices' key design principles.

The analysis was enacted by adopting thematic coding [10] and Krippendorf Kα-based inter-rater reliability assessment [65]. The first two authors annotated and labeled the selected primary studies to elicit possible impacts of smells and refactorings on the considered properties. The annotation and labeling were executed in parallel over two complementary partitions of the selected primary studies, to reduce potential observer biases. The coders were then switched to evaluate the inter-rater agreement on the two emerging lists of impacts. The inter-rater agreement was again measured by exploiting the Kα coefficient [65] to determine the agreement between the first two authors (who independently coded their partitions) on the emerging lists of bad and good practices for microservice security. The measured agreement was already higher than 80%, which is typically taken as a reference score for inter-rater agreement [65].

A final triangulation step was then performed to reduce potential biases further. The last three authors cross-checked the coding performed by the first two authors, with no prior information on the coding itself. This concluded our analysis process, which resulted in a total of 42 possible impacts, which are listed and discussed in Section 6.3.

6.2.3 Online survey

To further assess the identified impacts, as well as to understand whether they are differently perceived by industrial practitioners and academic researchers, we distributed an online survey to ask whether they agree with them. The survey was structured with initial profiling questions, used to distinguish industrial practitioners and academic researchers, and to ensure that respondents have experience in working with microservices. If this was the case, respondents were exposed to 42 statements, each presenting a different identified impact and its rationale, and they were asked to explicitly select their agreement with the statement, viz., *strongly disagree*, *disagree*, *neutral*, *agree*, or *strongly agree*.

The survey was published online from November 2022 to March 2023, collecting answers from a total of 22 respondents who explicitly declared that they were working with microservices. Out of such 22 respondents, 13 were industrial practitioners with different roles (1 software developer, 4 software engineers, and 8 software architects), 8 of whom declared 3+ years of experience in working with microservices. The other 9 respondents were academic researchers, 5 of whom declared 3+ years of experience working with microservices.[1]

6.3 Impacts of Security Smells

We hereafter present the candidate impacts of the security smells and refactorings for microservices, by considering one smell from our previous work [105] at a time. We also show whether/how interviewed experts agreed with such statements. The agreement will be displayed with bar plots, in which green is used to denote agreement, gray to denote neutrality, and yellow to denote disagreement. Darker shades of green/yellow denote stronger agreement/disagreement.

Also, we consider a candidate impact as confirmed if the absolute majority of respondents explicitly agree with it. At least 12 respondents must therefore agree or strongly agree with a candidate impact for the latter to be confirmed.

[1] All collected responses are publicly available on Zenodo at `https://doi.org/10.5281/zenodo.7828029`.

6.3.1 Insufficient Access Control (IAC)

IAC occurs when microservices lack proper access control, and its effects can be mitigated by exploiting OAuth 2.0. Indeed, IAC is known to negatively impact the confidentiality of a microservice application, whereas the use of OAuth 2.0 is positively impacting on it [105].

Table 6.1: Possible impacts of IAC

ID	Statement
IAC1	Insufficient access control may deteriorate the isolation of failures since an attacker can exploit a compromised microservice to provoke additional failures
IAC2	Insufficient access control may deteriorate the analysability of the application, making it more complex to diagnose the extension of damage (e.g., due to the confused deputy problem)
IAC3	OAuth 2.0 may increase the adherence to the decentralization principle of microservices, being it a distributed access delegation protocol
IAC4	OAuth 2.0 may increase the resource utilization of the application since, with this protocol, more information is transmitted on each request
IAC5	OAuth 2.0 may increase the reusability of the application since the organization can rely on standard libraries and platforms compatible with this industry-standard protocol

Five other possible impacts of IAC and the use of OAuth 2.0 emerged from the analyzed literature. These are listed in Table 6.1, with IAC1-2 predicating over the IAC smell itself, while IAC3-5 predicate over its possible refactoring. Each statement IAC1-5 highlights the possible impact, the affected property, and the rationale behind the impact.

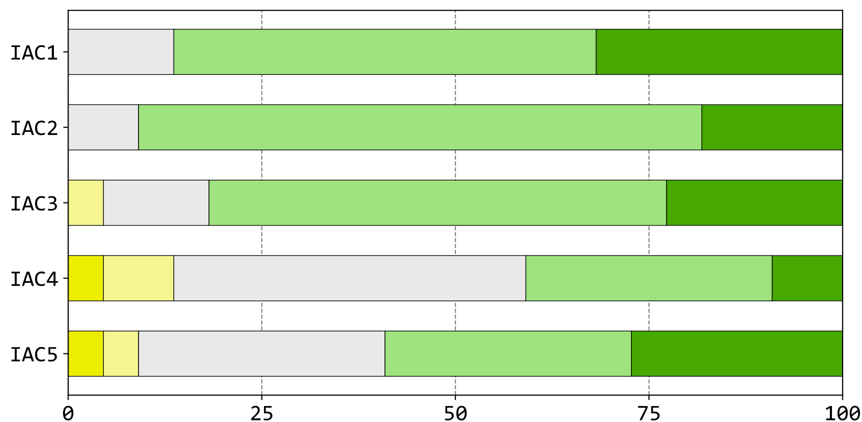
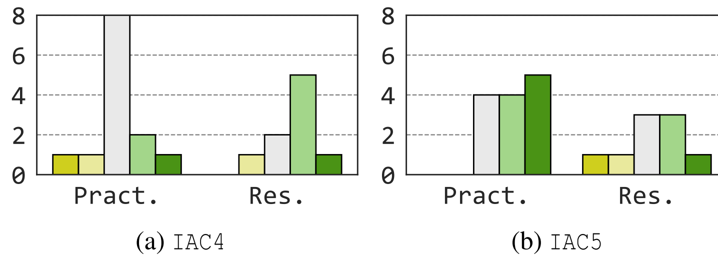


Figure 6.3: Agreement with the possible impacts of IAC

The agreement of respondents with the statements IAC1-5 is plotted in Figure 6.3. From the figure, we can observe that the agreement with IAC1, IAC2, and IAC3 is above 75%. Thus, most of the interviewed practitioners and researchers confirmed that IAC may deteriorate the isolation of failures and analysability of microservice applications, and that the use of OAuth 2.0 helps to achieve the microservices' decentral-

ization principle.

The majority of interviewed practitioners and researchers also agree with IAC5, confirming that the use of OAuth 2.0 can increase the reusability of the application. The answers for this statement are shown in Figure 6.4(b), from which we can observe practitioners mostly agree with IAC5, with a tendency towards strong agreement. Researchers also overall agree, but the tendency is more towards light agreement/neutrality.



(a) IAC4 (b) IAC5

Figure 6.4: Distribution of agreement with IAC-related statements.

IAC4 is instead not confirmed, as the overall agreement is lower than 50% (see Figure 6.3), meaning that respondents do not agree with OAuth 2.0 increasing the resource utilization of microservice applications. Figure 6.4(a) shows the actual answers given by practitioners and researchers. From the figure, we can observe that practitioners are mostly neutral about the statement, while researchers mostly agree with it. Finally, Figure 6.5 provides a comprehensive visual representation of the positive and negative impacts of the Insufficient Access Control (IAC) security smell and its corresponding refactoring (Use OAuth 2.0).

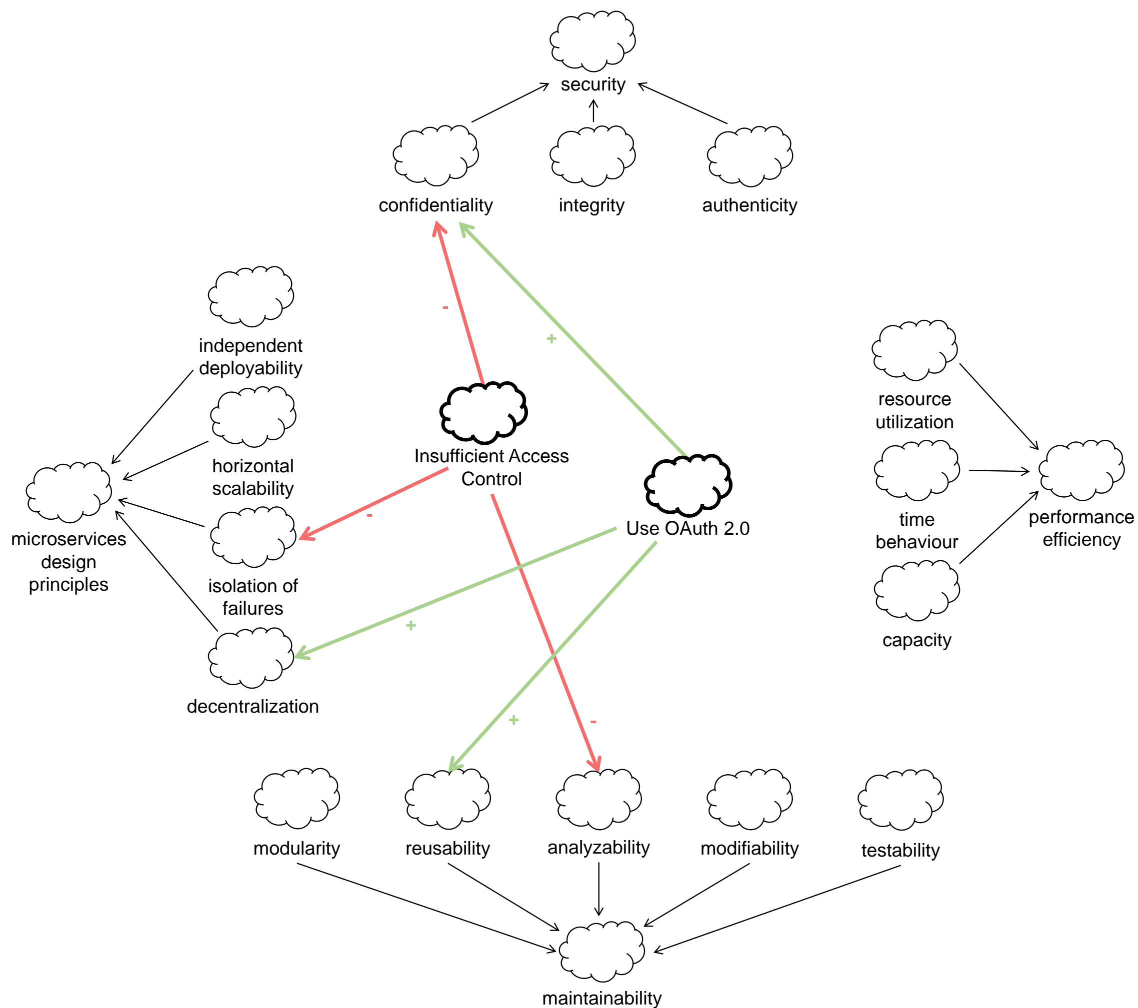


Figure 6.5: SIG that provides a holistic view of the validated impacts related to IAC

6.3.2 Publicly Accessible Microservices (PAM)

PAM occurs when microservices are directly accessible by external clients, and this can negatively impact on the confidentiality of a microservice application [105]. The effects of PAM can be mitigated by adding an API Gateway, to be used as an entry point for the application.

Nine other possible impacts of PAM and its refactoring emerged from the analyzed literature. These are listed in Table 6.2, which states the possible impacts, the property they affect, and their rationale. In the figure, `PAM1`, `PAM3`, and `PAM8` predicate over the PAM smell itself, while the others predicate over its refactoring.

The agreement of respondents with the statements `PAM1-9` is plotted in Figure 6.6. From the figure,

Table 6.2: Possible impacts of PAM

ID	Statement
PAM1	Having multiple publicly accessible microservices may deteriorate the testability of an application since there are multiple access points, each to be tested independently
PAM2	Adding an API Gateway may improve the testability since changes related to access to the application need to be tested only in the Gateway rather than on multiple entry points (being these services or other components)
PAM3	Having multiple publicly accessible microservices may deteriorate the isolation of failures of an application since they increase the attack surface which can be exploited to cause cascading failures
PAM4	Adding an API Gateway may increase the resource utilization of the application since additional resources are needed to actually run the API Gateway
PAM5	Adding an API Gateway may increase the time behavior of the application since it introduces additional communications/interactions between the Gateway itself and the other components
PAM6	Adding an API Gateway may improve the modifiability of an application since changes in how to access the application can be applied only to the Gateway, hence minimizing the impact on the other components
PAM7	Adding an API Gateway may improve the reusability of an application since the organization can rely on existing solutions for implementing a Gateway or reuse its own one for other applications
PAM8	Having multiple publicly accessible microservices may deteriorate the modifiability of an application, since a change in how to access the application may require applying such change to all publicly accessible microservices
PAM9	Adding an API Gateway may improve the analysability of an application since it simplifies evaluating the impact of changes in how to access the application (compared to the case of having multiple publicly accessible microservices)

we can observe that all respondents agree with PAM7, confirming that adding an API Gateway improves the reusability of microservice applications. Adding an API Gateway also improves the modifiability and analysability of microservice applications, as witnessed by the large agreement with PAM6 and PAM9. The interviewed practitioners and researchers also largely agree with PAM3 and PAM8, hence confirming that the presence of PAMs may deteriorate the isolation of failures and the modifiability of a microservice application.

The majority of interviewed practitioners and researchers also agree with PAM1 and PAM2, confirming that having publicly accessible microservices may deteriorate the testability of an application, while adding an API Gateway may improve it. However, being the agreement with PAM1 and PAM2 lower than 75%, they deserve a closer look to check for possible differences in practitioners' and researchers' answers. These are shown in Figure 6.7(a-b), from which we can observe that most practitioners and researchers are aligned in mostly agreeing with both statements.

PAM4 is instead not confirmed, as the overall agreement is exactly 50% and we only consider impacts as confirmed only if the absolute majority of respondents explicitly agree. Figure 6.7(c) shows the distribution

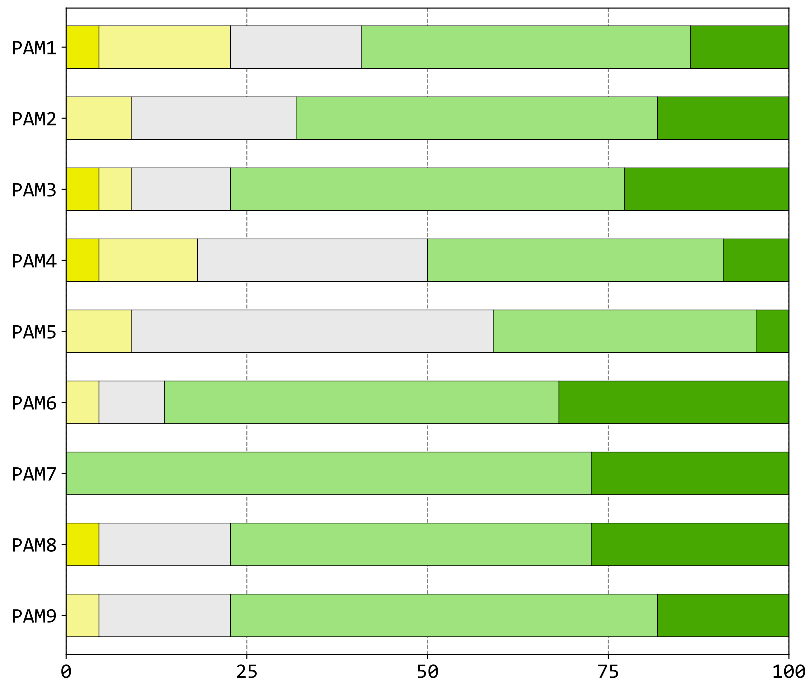
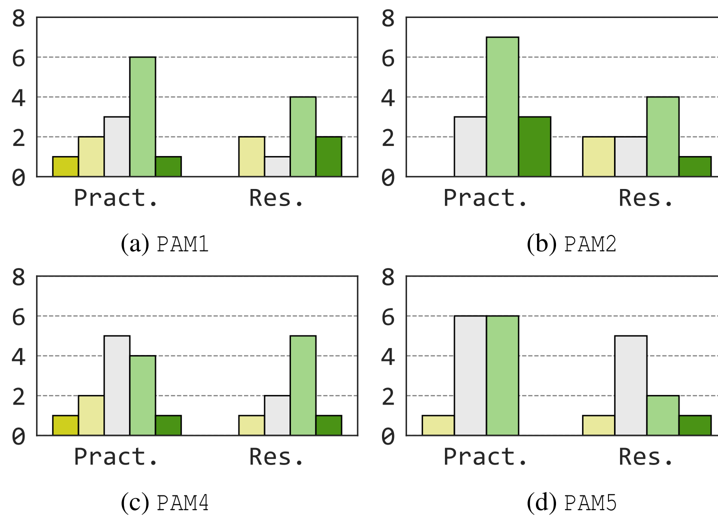


Figure 6.6: Agreement with the possible impacts of PAM



(a) PAM1 (b) PAM2

(c) PAM4 (d) PAM5

Figure 6.7: Distribution of agreement with PAM-related statements.

of answers given by interviewees. From the figure, we can observe that practitioners are more neutral than researchers, who mostly agree. So, there is not enough support from practitioners to confirm that adding an API Gateway would negatively impact on the resource utilization of an application.

PAM5 is not confirmed either, as the overall agreement is lower than 50%, mainly because of neutral

opinions on the statement (Figure 6.6). Figure 6.7(d) shows the actual answers given by interviewees, by distinguishing between practitioners and researchers. From the figure, we can observe that practitioners are mostly divided between neutrality and agreement, while researchers are mostly neutral with `PAM5`. This means that, overall, respondents do not agree that adding an API Gateway may increase the time behavior of an application. Finally, Figure 6.8 provides a comprehensive visual representation of the positive and negative impacts of the Publicly Accessible Microservices (PAM) microservice security smell and its corresponding refactoring.

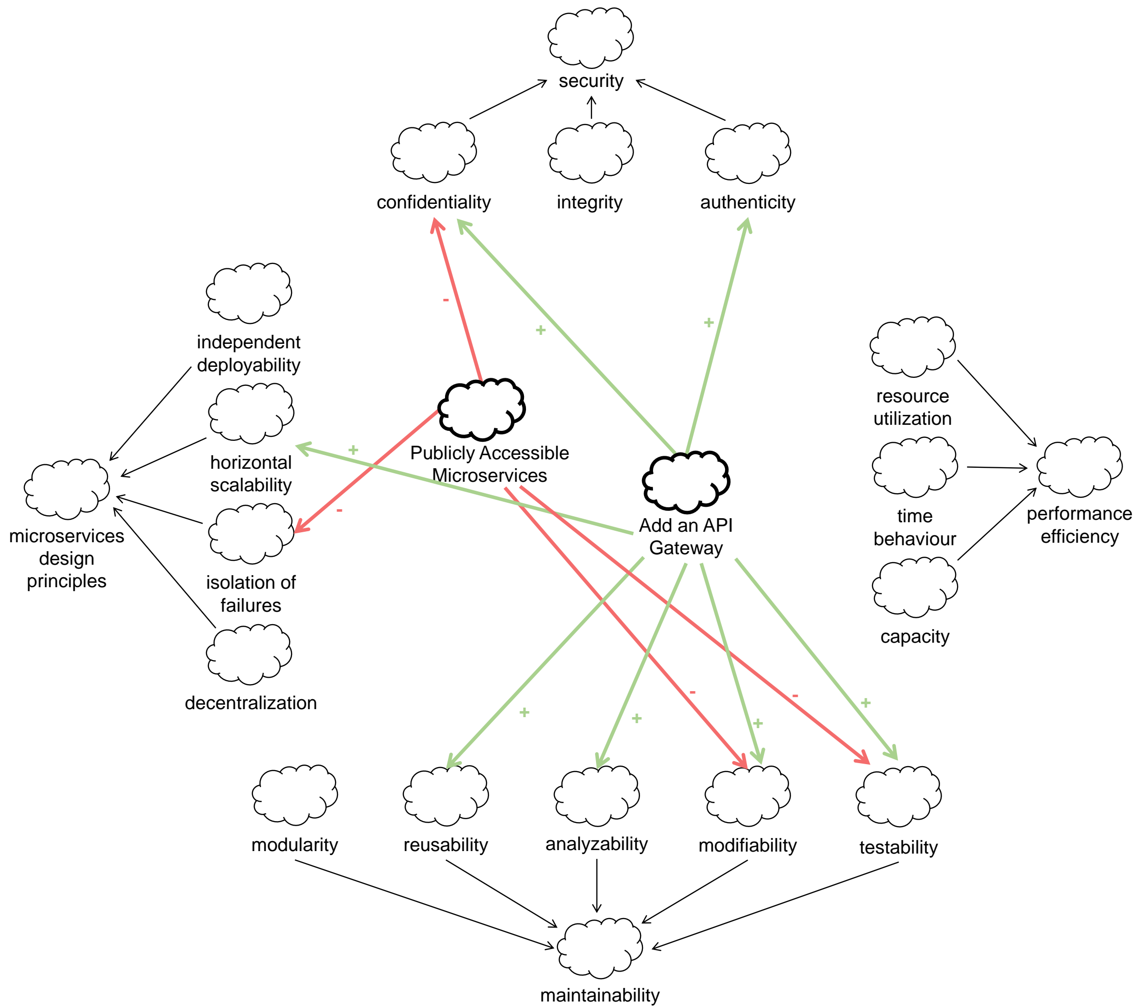


Figure 6.8: SIG that provides a holistic view of the validated impacts related to PAM

6.3.3 Unnecessary Privileges to Microservices (UPM)

UPM occurs when microservices are granted unnecessary privileges (e.g., access levels, permissions, or functionalities) that are not needed to deliver their business functions, and this may negatively impact on the confidentiality and the integrity of microservice applications. The impact can be reversed, if developers rather follow the least privilege principle [105].

Eight other possible impacts of UPM and its refactoring emerged from the analyzed literature. These are listed in Table 6.3, with `UPM1`, `UPM3`, and `UPM5` predicating over the UPM smell itself, while the others predicate over following the least privilege principle.

Table 6.3: Possible impacts of UPM

ID	Statement
`UPM1`	Providing unnecessary privileges to microservices may deteriorate the resource utilization of an application since an attacker can exploit a compromised microservice to increase the resources consumed by the application.
`UPM2`	Following the Least Privilege Principle may improve the modularity of an application since it reduces the resources that a microservice can access, hence meaning that a change in a microservice could impact only on such resources.
`UPM3`	Providing unnecessary privileges to a microservice may deteriorate the analysability of an application since they complicate the impact of a change in such microservices on the rest of the application.
`UPM4`	Following the Least Privilege Principle may improve the isolation of failures in an application since it reduces the resources that a microservice can access, hence also reducing the services that can fail in cascade to a compromised microservice.
`UPM5`	Providing unnecessary privileges to microservices may deteriorate the isolation of failures in an application since an attacker can exploit the unnecessary privileges to cause failures in other microservices.
`UPM6`	Following the Least Privilege Principle may improve the analysability of an application since it reduces the resources that a microservice can access, hence limiting the analysis of the effects of a change in a microservice only to such resources.
`UPM7`	Following the Least Privilege Principle may improve the resource utilization of an application since access to a resource is provided to microservices only on an as-needed basis (hence reducing the waste of resources due to misuse).
`UPM8`	Following the Least Privilege Principle may deteriorate the testability of an application since it is more complex to test whether a microservice can access only the resources it actually needs.

The agreement of interviewed practitioners and researchers with the statements `UPM1-8` is plotted in Figure 6.9. From the figure, we can observe that the agreement with `UPM1` and `UPM5` is above 75%, meaning that the UPM smell may deteriorate the resource utilization and isolation of failures of microservice applications. Respondents also overall agree with `UPM4` and `UPM7`, which indicates that following the least privilege principle may improve the isolation of failures and resource utilization.

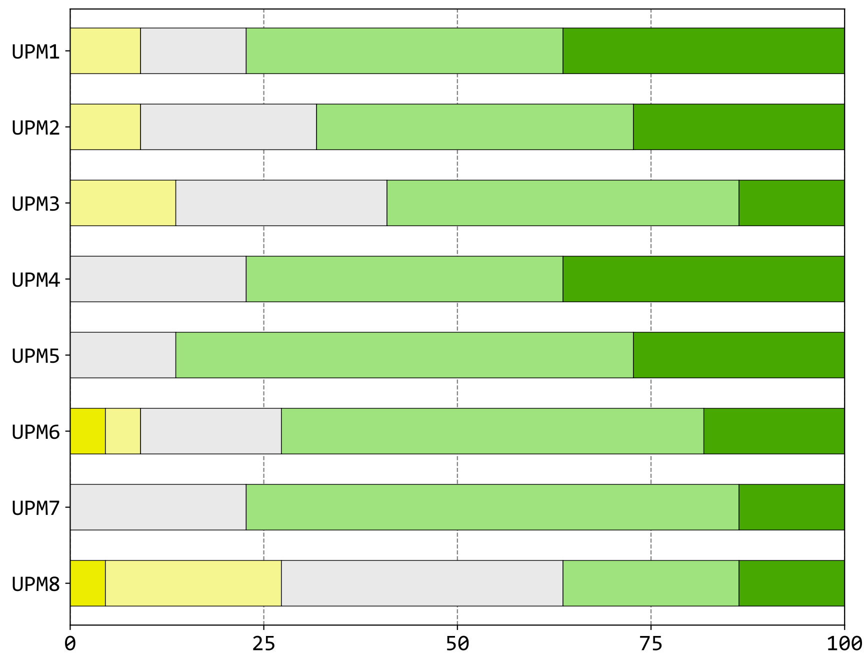
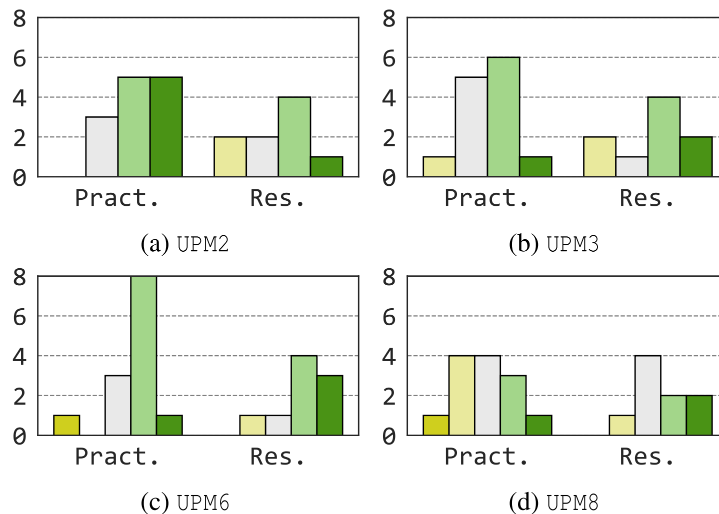


Figure 6.9: Agreement with the possible impacts of UPM

The majority of practitioners and researchers also agree with `UPM2`, `UPM3`, and `UPM6`, confirming that refactoring an application by following the least privilege principle improves the application's modularity and analysability, whereas the UPM smell deteriorates its analysability. Figure 6.10(a-c) show that practitioners and researchers are aligned in mostly agreeing with these three statements.



(a) `UPM2` (b) `UPM3`

(c) `UPM6` (d) `UPM8`

Figure 6.10: Distribution of agreement with UPM-related statements.

`UPM8` instead is not confirmed, as the overall agreement is lower than 50%, mainly because of the neutral

opinions from both practitioners and researchers on the statement. From Figure 6.10(d) we can observe that practitioners are mostly divided between neutrality, disagreement, and agreement, while researchers are mostly neutral. This means that respondents do not overall agree that following the least privilege principle may deteriorate the testability of microservice applications. Figure 6.11 then provides a comprehensive visual representation of the positive and negative impacts of the Unnecessary Privileges to Microservices (UPM) security smell and its corresponding refactoring.

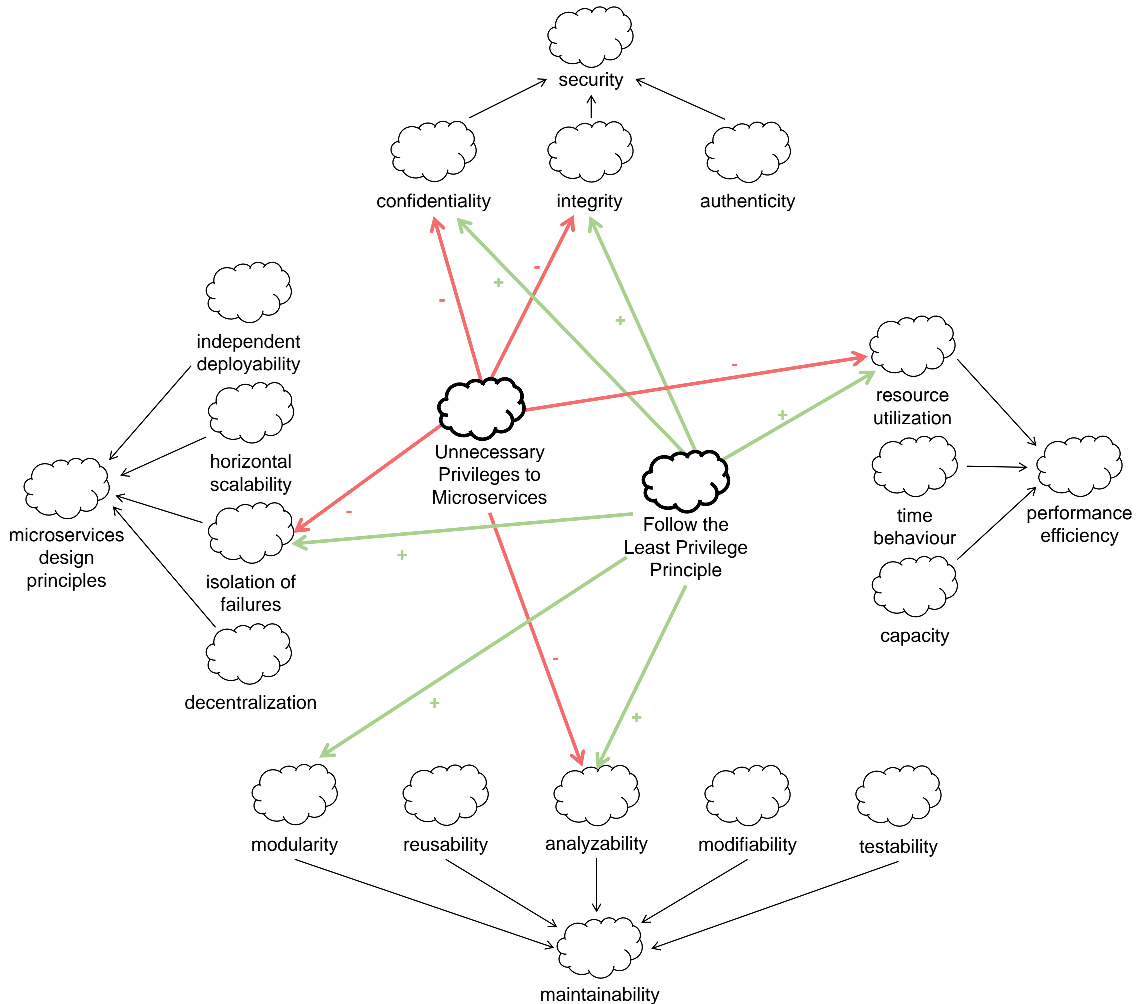


Figure 6.11: SIG that provides a holistic view of the validated impacts related to UPM

6.3.4 Non-Secured Service-to-Service Communications (NSC)

NSC occurs when some microservices in an application interact without establishing a secure communication channel, and its effects can be mitigated by using Mutual TLS. NSC is a security smell that negatively impacts on the confidentiality, integrity, and authenticity of microservice applications, whereas its refactoring has a positive impact on them [105].

Table 6.4: Possible impacts of NSC

ID	Statement
NSC1	Non-secured service-to-service communications may deteriorate the analysability of an application, making it harder to diagnose the causes of failures (e.g., since data in transit can be modified if a service is compromised).
NSC2	Mutual TLS may improve the isolation of failures of microservices since this protects service-to-service communication from, e.g., man-in-the-middle, eavesdropping, and tampering attacks.
NSC3	Non-secured service-to-service communications may deteriorate the isolation of failures in an application since an attacker can exploit them to propagate failures across microservices.
NSC4	Mutual TLS may deteriorate the time behavior of the application since it encrypts service-to-service communications.

Other four possible impacts of NSC and its refactoring are listed in Table 6.4. In the figure, NSC1 and NSC3 predicate over NSC itself, while NSC2 and NSC4 pertain to its Mutual TLS-based refactoring. Figure 6.12 then shows the level of agreement among respondents concerning statements NSC1-4, with NSC3 confirmed by the vast majority of them, hence NSC may contribute to deteriorate isolation of failures in a microservice application.

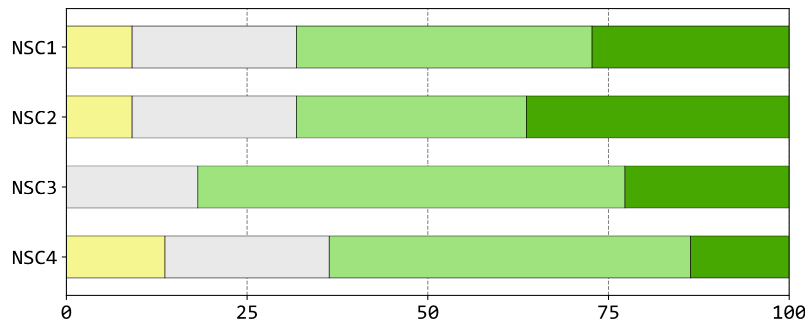


Figure 6.12: Agreement with the possible impacts of NSC

The majority of practitioners and researchers also agree with NSC1, which confirms the negative impact of the NSC smell on the analysability of microservice applications. The agreement also is reached for NSC2 and NSC4, confirming that the use of Mutual TLS improves an application's isolation of failures, at the price of negatively impacting on its time behavior. As observed in Figure 6.13, practitioners and researchers are mostly aligned in their agreement with these three statements, with practitioners mostly

strongly agreeing with NSC2. Finally, Figure 6.14 provides a comprehensive visual representation of the positive and negative impacts of the Non-Secured Service-to-Service Communications (NSC) security smell and its corresponding refactoring.

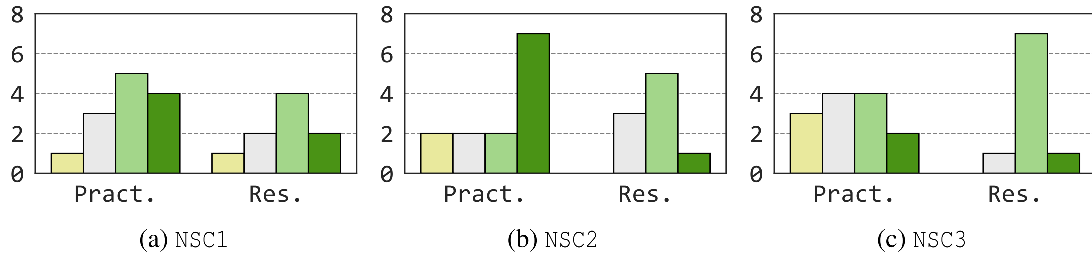


Figure 6.13: Distribution of agreement with NSC-related statements.

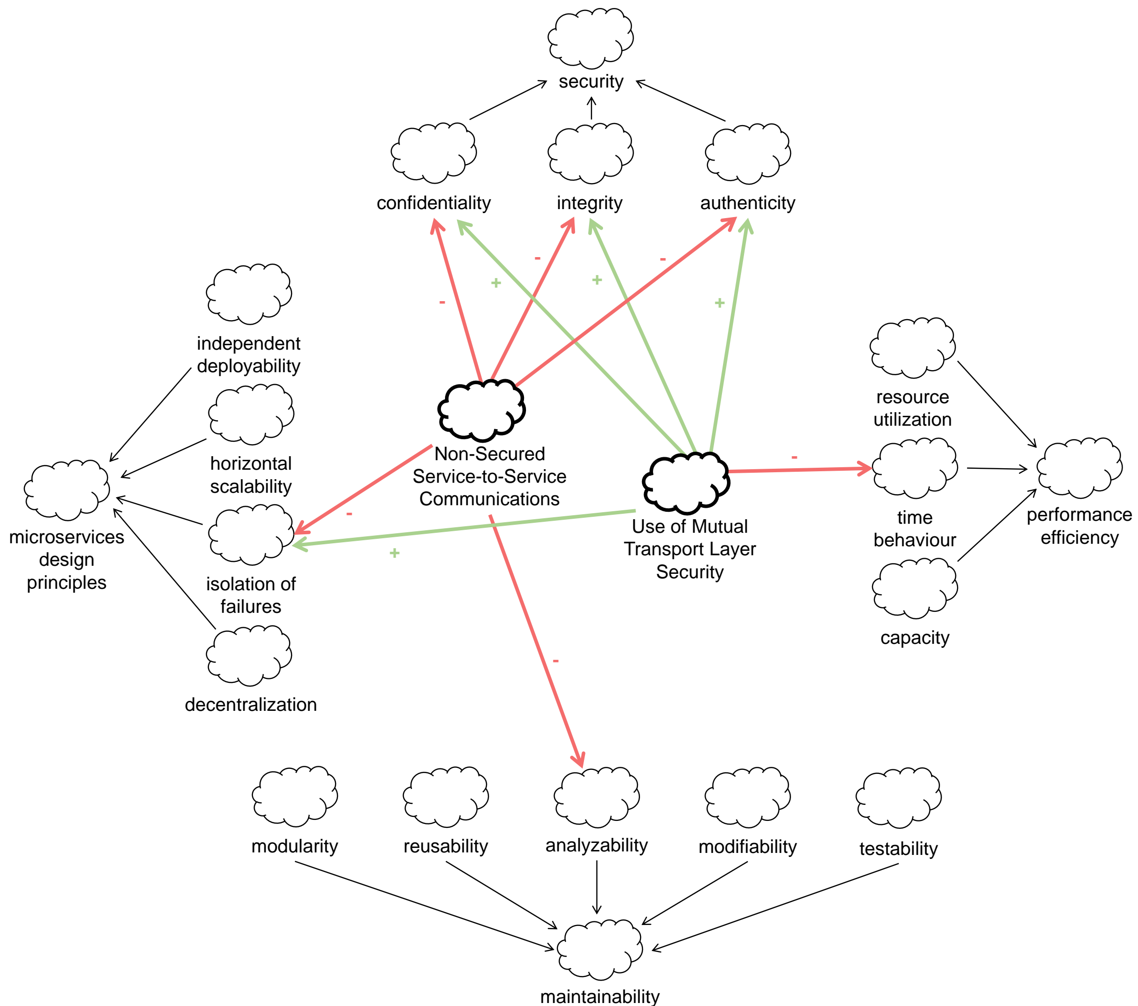


Figure 6.14: SIG that provides a holistic view of the validated impacts related to NSC

6.3.5 Unauthenticated Traffic (UNT)

UNT occurs when a microservice receives unauthenticated requests from external clients or other microservices of the application, which may negatively impact on the authenticity of microservice applications. The effects of UNT can be mitigated by using Mutual TLS and OpenID Connect [105].

In addition to the already discussed impacts of using Mutual TLS (Section 6.3.4), three additional impacts emerged from the analyzed literature. These impacts are listed in Table 6.5, with `UNT1` predicating over UNT itself, while `UNT2` and `UNT3` pertain to the use of OpenID Connect.

Table 6.5: Possible impacts of UNT

ID	Statement
`UNT1`	Having unauthenticated traffic may deteriorate the isolation of failures since an attacker can generate (unauthenticated) traffic to propagate failures across microservices.
`UNT2`	Using OpenID Connect may increase the adherence to the decentralization principle of microservices, being it, a distributed identity mechanism allowing to decentralize authentication.
`UNT3`	Using OpenID Connect may improve the reusability of the application since it can be realized by reusing existing software whose configuration is easy to reuse/replicate.

The agreement with the UNT statements is then shown in Figure 6.15. From the figure, we can observe that the agreement with `UNT1` is above 75%, which indicates that having unauthenticated traffic in a microservice application can lead to the deterioration of isolation of failures.

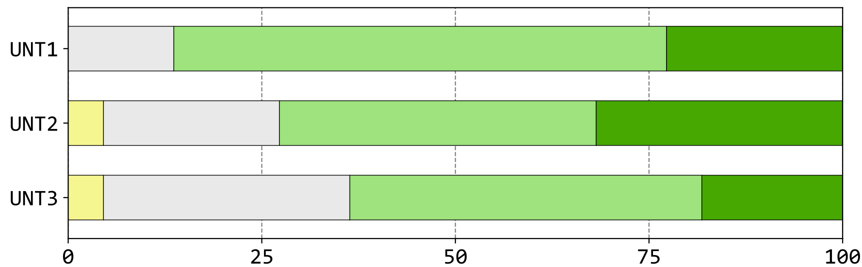


Figure 6.15: Agreement with the possible impacts of UNT

`UNT2` and `UNT3` are also confirmed by the majority of interviewees, meaning that OpenID Connect positively impacts on reusability and decentralization in microservice applications. By looking at Figure 6.16, we can observe that both practitioners and researchers are mostly aligned in their agreement with `UNT2`, whereas `UNT3` is mostly agreed by practitioners, with researchers having a more neutral reaction to such statement. Finally, Figure 6.17 provides a comprehensive visual representation of the positive and negative impacts of the Unauthenticated Traffic (UNT) security smell and its corresponding refactoring.

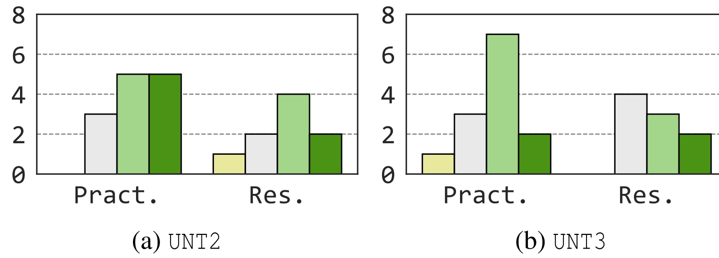


Figure 6.16: Distribution of agreement with UNT-related statements.

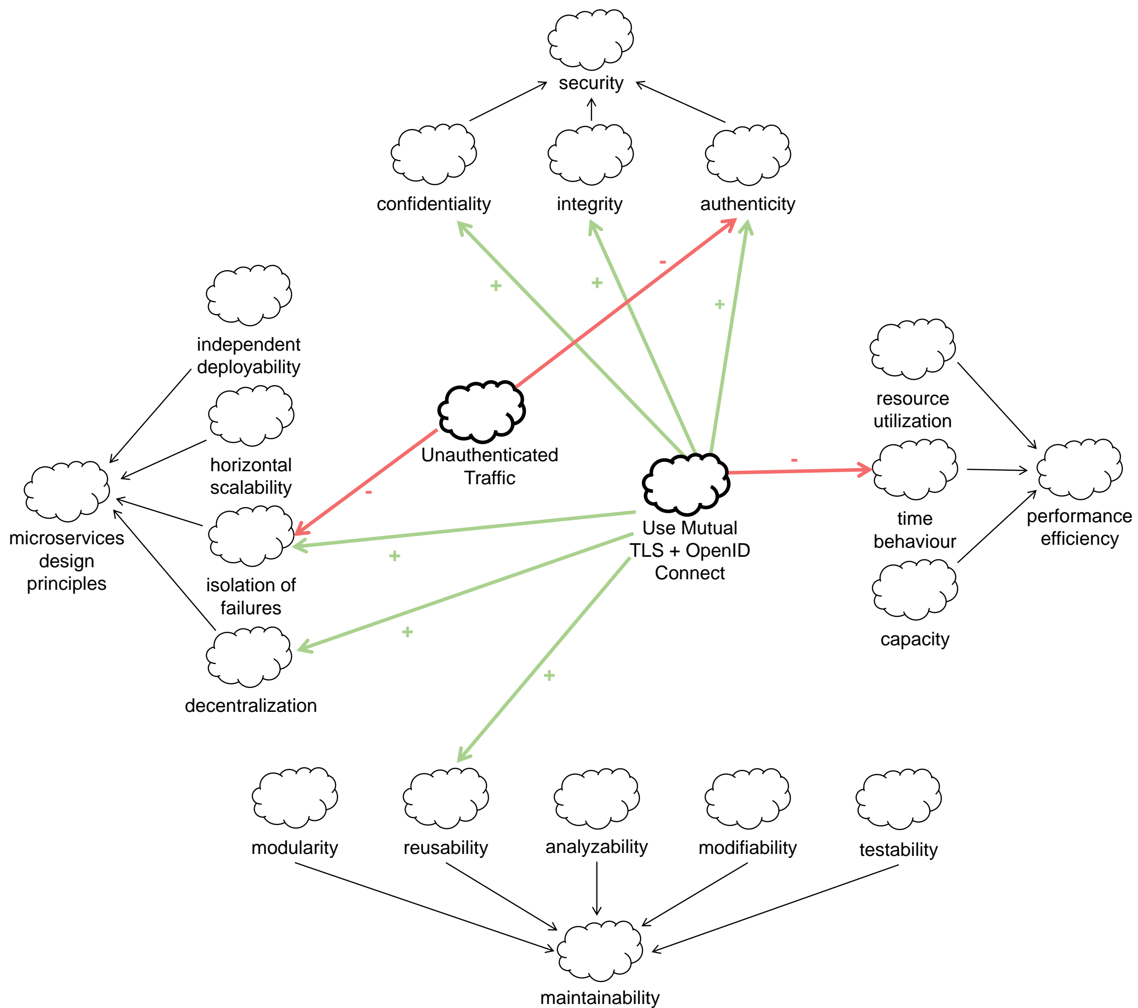


Figure 6.17: SIG that provides a holistic view of the validated impacts related to UNT

6.3.6 Multiple User Authentication (MUA)

MUA occurs when a microservice application provides multiple endpoints for user authentication, and its effects can be mitigated by using a single sign-on approach. MUA is a security smell that may negatively impact on the authenticity of microservice applications, whereas its refactoring has a positive impact on confidentiality and authenticity [105].

Table 6.6: Possible impacts of MUA

ID	Statement
MUA1	Authenticating users in multiple different services may deteriorate the testability of an application since the multiple user authentication points should be tested independently.
MUA2	Authenticating users in multiple different services may deteriorate the modifiability of an application since a change related to user authentication should be applied to all user authentication points.
MUA3	Having multiple user authentication points may increase the adherence to the decentralization principle of microservices since access to the application is distributed across all entry points.

Three additional possible impacts of MUA have emerged from the analyzed literature, all predicating over the MUA smell itself (Table 6.6). The agreement among interviewees concerning MUA statements is shown in Figure 6.18, which confirms MUA1 and MUA2, namely that MUA may deteriorate the testability and modifiability of microservice applications. Such agreement mainly comes from researchers in the case of MUA1 (Figure 6.19(a)), whereas practitioners and researchers are aligned in agreeing with MUA2 (Figure 6.19(b)).

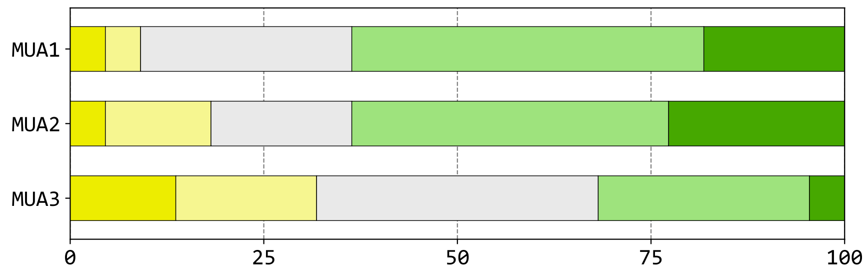
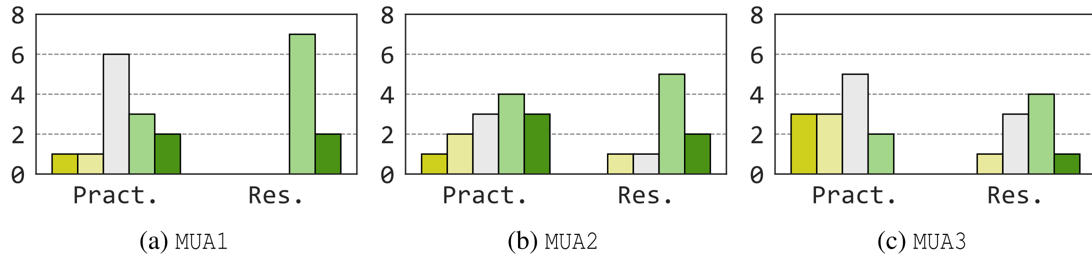


Figure 6.18: Agreement with the possible impacts of MUA

MUA3 instead is not confirmed, as the overall agreement falls below 50% (Figure 6.18). This is mainly because the vast majority of practitioners disagree or are neutral to MUA3, as opposed to researchers who mostly agree with such statement.

Finally, Figure 6.20 provides a comprehensive visual representation of the positive and negative impacts of the Multiple User Authentication (MUA) security smell and its corresponding refactoring.



(a) MUA1 (b) MUA2 (c) MUA3

Figure 6.19: Distribution of agreement with MUA-related statements.

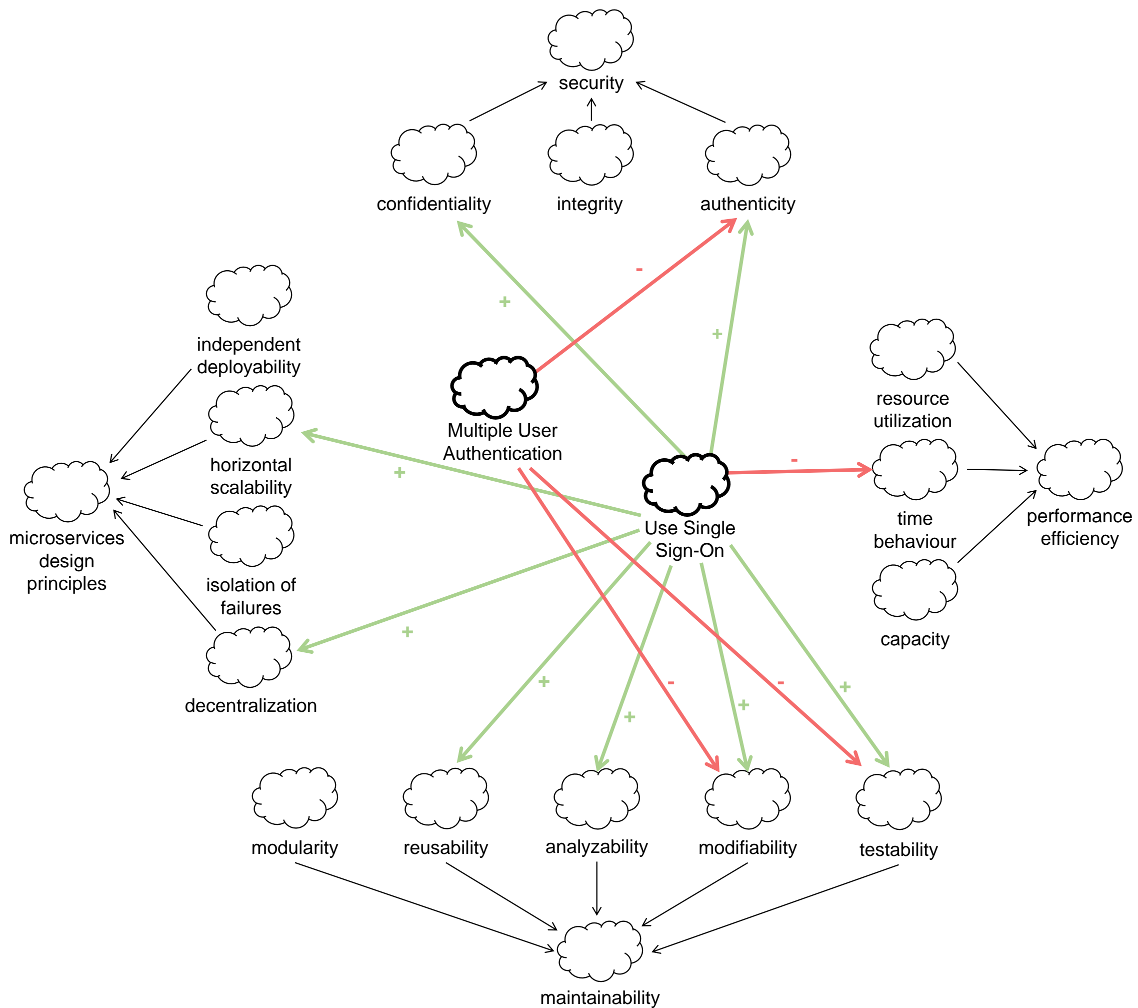


Figure 6.20: SIG that provides a holistic view of the validated impacts related to MUA

6.3.7 Centralized Authorization (CNA)

CNA occurs when a microservice application only handles authorization in one component, and its effects can be mitigated by using a decentralized authorization approach (e.g., transmitting a token with each request). CNA negatively impacts the authenticity of microservice applications, whereas its refactoring has a positive impact on it [105].

Table 6.7: Possible impacts of CNA

ID	Statement
CNA1	A centralized authorization may deteriorate the time behavior of an application, with the authorization server becoming a bottleneck when many microservices access it.
CNA2	Using decentralized authorization may increase the adherence to the decentralization principle of microservices, since access control is distributed across the application, with each microservice having its fine-grained access controls.
CNA3	A centralized authorization may deteriorate the analysability of an application, since it requires extensively considering all possible impacts (on all services in the application) of a change in the access control ruled by the authorization server.
CNA4	Using decentralized authorization may improve the modularity of an application, since each microservice is associated with its own fine-grained access controls, hence limiting the impact of changes only to such microservice.
CNA5	Using decentralized authorization may deteriorate the testability of an application, since more efforts are needed to independently test the access control on each microservice.
CNA6	Using decentralized authorization may improve the analysability of an application, since each microservice is associated with its own fine-grained access controls, hence requiring assessing the impact of changes only on such microservice.
CNA7	A centralized authorization may improve the testability of an application, since changes related to access control should be tested only in the central authorization server.
CNA8	Using decentralized authorization may increase the adherence to the independent deployability principle of microservices, since it avoids coupling microservices to an authorization server.
CNA9	A centralized authorization may reduce the adherence to the decentralization principle of microservices since authorization is centralized and fully managed by only one component.
CNA10	Using decentralized authorization may increase resource utilization, since more information is to be transmitted on each request (e.g., token containing roles and privileges).

Ten additional possible impacts of CNA and its refactoring have emerged from the analyzed literature. These impacts are listed in Table 6.7, with `CNA1`, `CNA3`, `CNA7` and `CNA9` focusing on the CNA smell, while the others on using a decentralized authorization approach.

The agreement with CNA-related impacts is illustrated in Figure 6.21. From the figure, we can observe that the vast majority of interviewed practitioners and researchers agree with `CNA2`, hence confirming that – as expected – decentralizing authorization contributes to realizing the decentralization design principle of microservices.

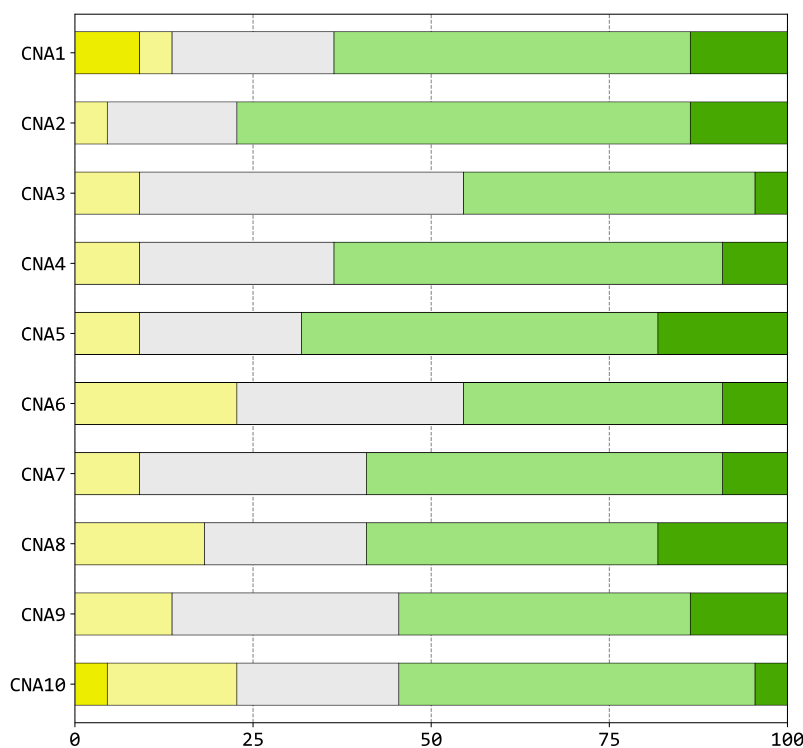
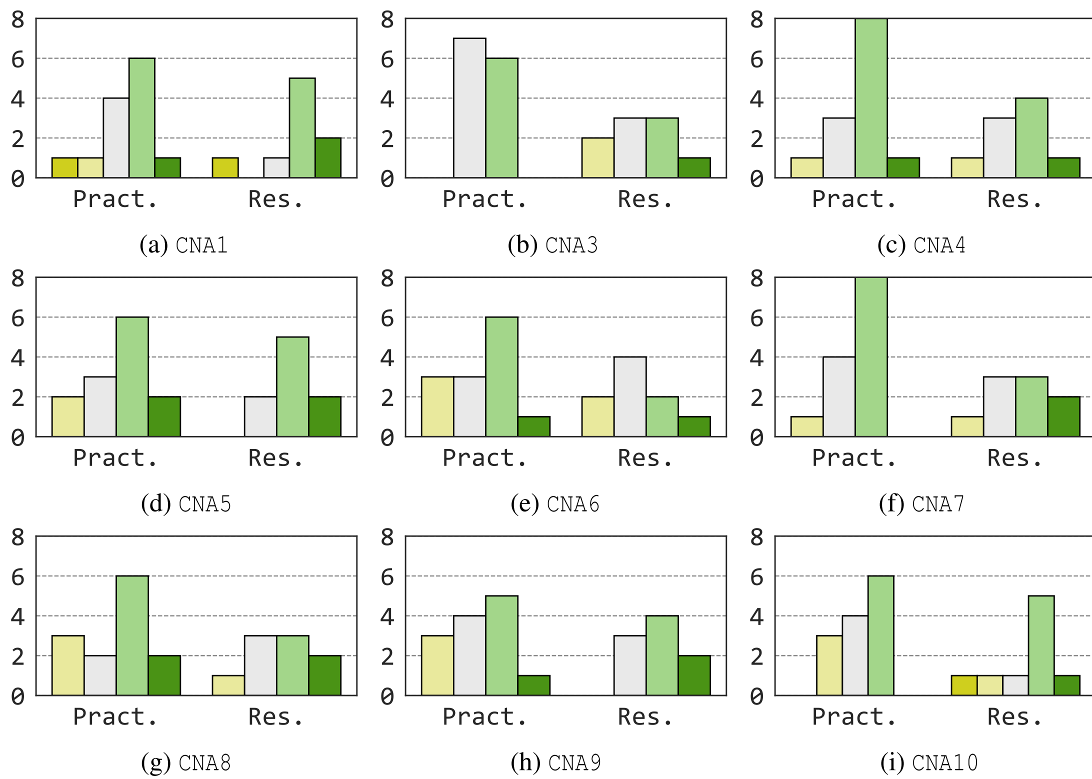


Figure 6.21: Agreement with the possible impacts of CNA

Respondents also mostly agree with CNA1 and CNA9, confirming the negative impact of CNA on an application's time behavior and decentralization, whereas the agreement with CNA7 confirms that CNA can improve the testability of an application. From Figure 6.22, we can observe that the agreement with CNA1, CNA7, and CNA9 comes both from practitioners and from researchers, who are aligned in mostly agreeing with such statements.

As shown in Figure 6.21, the majority of respondents also agreed with CNA4, CNA5, CNA8, and CNA10, which are about the refactoring of CNA. This confirms that decentralizing authorization helps improving the modularity of a microservice application and its adherence to the independent deployability principle of microservices, at the price of deteriorating the application's testability and increasing its resource utilization. Again, the agreement with CNA4, CNA5, CNA8, and CNA10 comes both from practitioners and from researchers, who are aligned in mostly agreeing with such statements.

The candidate impacts CNA3 and CNA6 are instead not confirmed, meaning that CNA and its refactoring are not considered to impact on the analysability of a microservice application. For CNA3, this is mainly because of the neutrality of practitioners, with researchers' answers being more distributed among disagreement, neutrality, and agreement (Figure 6.22(b)).



(a) CNA1 (b) CNA3 (c) CNA4 (d) CNA5 (e) CNA6 (f) CNA7 (g) CNA8 (h) CNA9 (i) CNA10

Figure 6.22: Distribution of agreement with CNA-related statements

In the case of CNA6, both practitioners' and researchers' answers are divided among disagreement, neutrality, and agreement, with a tendency towards agreement from practitioners and a tendency towards neutrality/disagreement for researchers (Figure 6.22(e)). Finally, Figure 6.23 provides a comprehensive visual representation of the positive and negative impacts of the Centralized Authorization (CNA) security smell and its corresponding refactoring.

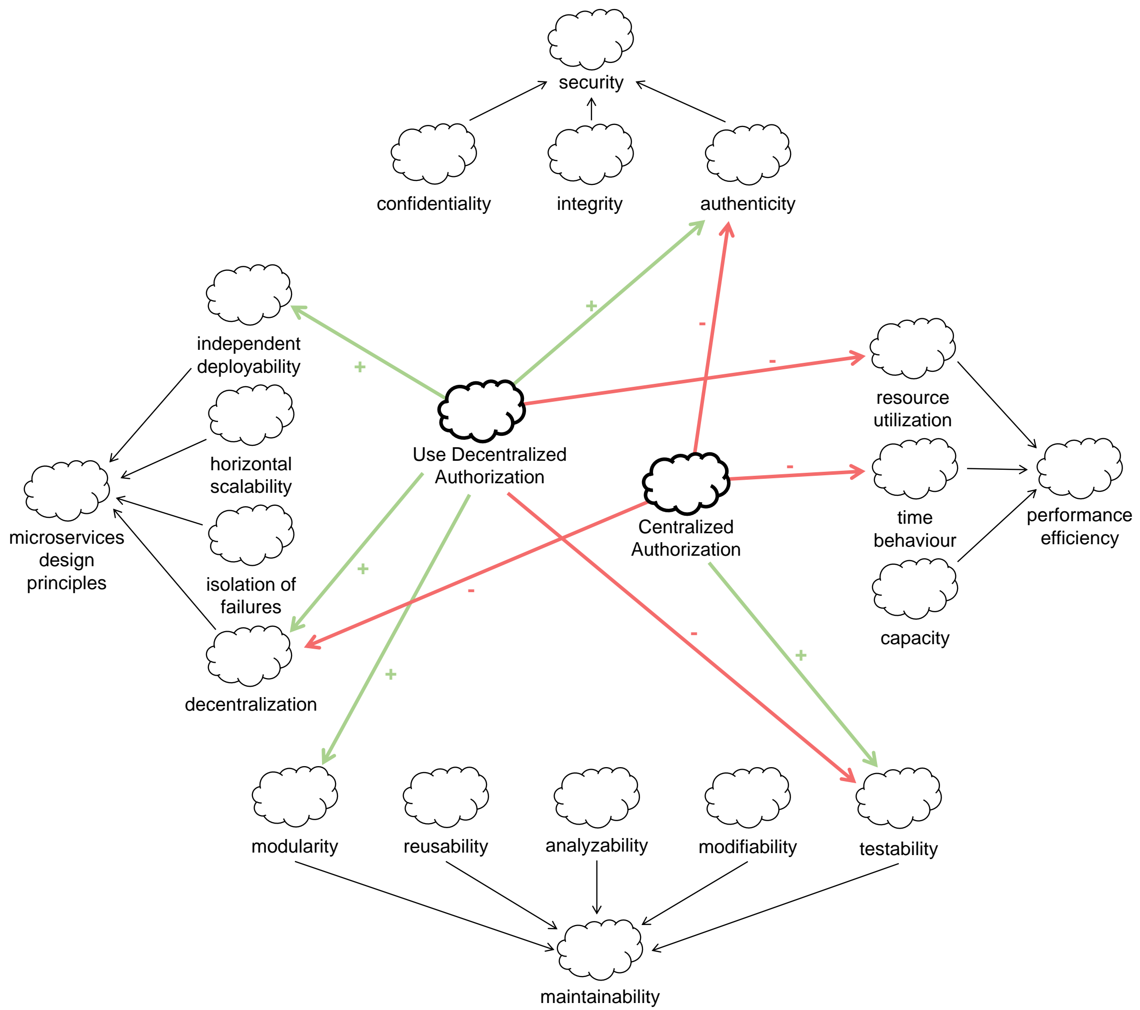


Figure 6.23: SIG that provides a holistic view of the validated impacts related to CNA

6.4 Threats to Validity

Among the potential threats to validity classified by Wohlin et al. [152], four may apply to our survey. These are the threats to the *external*, *internal*, *construct*, and *conclusions* validity, which we discuss hereafter.

External Validity. The external validity concerns the applicability of a set of results in a more general context [152]. Since the primary studies considered by our thematic analysis were selected from a very large extent of online sources, the elicited impacts of security smells and refactorings may only be partly applicable to the state-of-the-art and state-of-practice on microservices. This may hence result in threatening the external validity of our study.

Actions to mitigate this potential threat were already taken in the literature selection (e.g., to avoid missing relevant literature), as we thoroughly described in our previous work [105]. To further reinforce the external validity of our findings, we organized multiple feedback sessions with all authors during our analysis of the selected studies, and we considered as validated *only* the impacts with which the majority of interviewed researchers/practitioners explicitly agreed. We indeed considered neutral opinions as non-agreement (rather than disagreement), as each neutral opinion was not providing enough evidence on the fact that an elicited impact was considered a true impact by a respondent. Also, because the analyzed literature was not covering some other impacts, we left blank space for respondents to propose impacts that were not emerging in our systematic analysis, getting very few suggestions — which were not enough to consider additional impacts to be added, even if we plan to further investigate on suggestion as part of our future work.

Construct and Internal Validity. The construct and internal validity of a study concern the generalizability of the constructs under study and the method employed to study and analyze data, respectively [152]. The construct and internal validity may hence be threatened by the potential types of bias involved when running a study, which we tried to mitigate by exploiting systematic analysis methods, such as thematic analysis and inter-rater reliability assessment (Section 6.2). Such systematic methods helped limiting potential biases, such as observer and interpretation biases, hence contributing to enhancing the validity of the analysis we performed on the data we retrieved.

Conclusions Validity. The conclusions validity is defined as the degree to which the conclusions of a study are reasonably based on the available data [152]. To mitigate potential threats to the conclusions' validity of our study, we exploited the above-described inter-rater reliability assessment to limit potential biases in our observations and interpretations. Additionally, we exploited the online survey to validate the elicited impacts, considering as valid impacts only those with which the majority of interviewed researchers/practitioners explicitly agreed.

6.5 Related Work

Various existing research works focus on microservice smells, e.g., [87], [137], [16], and [21]. For instance, the industrial survey reported in [137] explicitly defined 11 microservice-specific bad smells, which span from the design of the microservice applications to their actual development, also eliciting the best practices enabling to avoid incurring such smells. [16, 21, 87] instead mainly focus on architectural smells, geared towards checking whether the architecture of microservice applications adheres to the design principles that define microservices themselves. We here focus on security smells and their possible refactorings, which (to the best of our knowledge) have first been elicited in our previous work [105], which we, therefore, take as a starting point for our work.

Beyond the type of considered smells, it is worth relating our work with other research works exploiting surveys to shed light on microservices. In this perspective, the closest work to ours is [118], which also surveyed practitioners with questions on microservice security. The results of the survey in [118] highlight that microservices present unique security challenges. [118] also shows that access to security discussions (including design decisions, challenges, or solutions relating to security) is beneficial for making security decisions. Our work differs from [118] in its objectives, as we rather focus on the possible impacts of security smells and refactoring beyond security, namely on maintainability, performance efficiency, and adherence to microservices' key design principles.

Similar considerations apply to other studies exploiting surveys to shed light on microservices, such as, e.g., [75], [17], [149], and [22]. [75] explores the impacts of frameworks on the high availability of microservice applications, while [17] targets microservice-oriented maintainability assurance techniques. [149] and [22] instead focus on the commercial adoption of microservices and on migration from monoliths to microservices, respectively. Our work hence differs from [75], [17], [149], and [22] in its objectives, as we focus on the impacts of security smells and refactorings on other quality attributes (namely, maintainability and performance efficiency) and on the adherence to microservices' key design principles.

Chapter 7

Triaging Microservice Security Smells, with TriSS

Choosing between tolerating a given microservice security smell instance and resolving it with a refactoring requires careful trade-offs considerations, since both the smell and its refactoring may impact other quality attributes besides security, e.g., maintainability and performance [107]. For example, the *centralized authorization* security smell harms the authenticity and time behavior of the MSA, but favors its testability. Thus, resolving the security smell by applying the *use decentralized authorization* refactoring would favor the microservice application (MSA) authenticity and modularity, but would harm its testability and resource utilization. Making informed refactoring decisions requires assessing the trade-offs of impacts on several (possibly many) quality attributes [104].

Since multiple security smell instances can affect multiple services in an MSA, architects must not only find trade-offs for each smell instance but also decide which smell instances to resolve first [14]. Indeed, some smell instances may be more "urgent" than others because they affect services that implement core functionalities and/or quality attributes that are critical for a service effective functioning. Given the number of services forming an MSA, their quality requirements, and the multiple different impacts of security smells on quality attributes, it is inherently complex and costly to determine which security smell instances should be resolved first, being the most urgent [151].

To support practitioners in this decision process, we propose TriSS (*Triage Security Smells*), a method that systematically associates security smell instances with "urgency codes", similar to what triage nurses do with patients that enter a hospital emergency room and describe their symptoms. TriSS enables assigning each security smell instance (i.e., a smell affecting a service in an MSA) an urgency code, which is computed

by combining (i) the relevance of the service to the business, and (ii) the importance of the service quality attributes that are impacted by the smell instance. TriSS systematizes this process by assigning smell instances to urgency codes, which practitioners can use to decide which smell instances to resolve first (presumably, those with the highest urgency). We illustrate the applicability of TriSS with a concrete use case based on an existing MSA, i.e., Lakeside Mutual[1].

Finally, we validate the usefulness of the TriSS method with a controlled experiment where 26 practitioners were asked to triage microservice security smells *with* and *without* TriSS. The practitioners' feedback suggests that TriSS simplifies the triage process, while increasing the confidence in the urgency codes assigned to the security smell instances affecting the services of an MSA.

7.1 Motivating Scenario

Lakeside Mutual is an MSA simulating an insurance company (Figure 7.1). It is a demonstrator MSA, com-

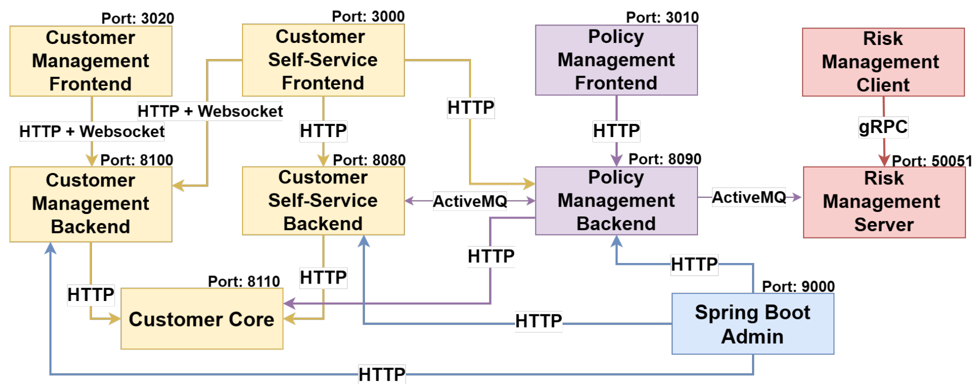


Figure 7.1: Lakeside Mutual MSA.

monly used to assess the results of MSA-related research, e.g., [60, 95, 135, 151]. By manually inspecting its (online) source code, we found that its services show 32 different instances of five known security smells (Table 7.1). Thus, it provides a simple yet effective example of how complex can be to assess the priority of MSA security smells.

Indeed, we might pose several questions on this MSA: How does each security smell instance impact the MSA overall quality? Is the *hardcoded secrets* instance that affects *Customer Self-Service Backend* more/less severe than the one affecting *Risk Management Server*? How to compare the instances of the *hardcoded secrets* smell with those of other smells? More generally, which security smell instances are more "urgent" to address?

[1] https://github.com/Microservice-API-Patterns/LakesideMutual

Table 7.1: Security smell instances affecting services of the Lakeside Mutual MSA.

Security smell	**Affected services**
Publicly accessible microservices	*Customer Core*, *Customer Management Backend*, *Customer Management Frontend*, *Customer Self-Service Backend*, *Customer Self-Service Frontend*, *Policy Management Backend*, *Policy Management Frontend*, *Risk Management Server*, *Spring Boot Admin*
Insufficient access control	*Customer Management Backend*, *Customer Self-Service Backend*, *Policy Management Backend*, *Spring Boot Admin*
Unauthenticated traffic	*Customer Management Backend*, *Customer Management Frontend*, *Customer Self-Service Backend*, *Customer Self-Service Frontend*, *Policy Management Backend*, *Policy Management Frontend*, *Spring Boot Admin*
Hardcoded secrets	*Customer Management Backend*, *Customer Self-Service Backend*, *Policy Management Backend*, *Risk Management Server*
Non-secured service-to-service communications	*Customer Self-Service Backend*, *Customer Self-Service Frontend*, *Customer Management Backend*, *Customer Management Frontend*, *Policy Management Backend*, *Policy Management Frontend*, *Customer Core*, *Spring Boot Admin*

Answering these questions requires considering several aspects, like the relevance of each affected service to the business and the importance of ensuring the affected services' quality. The TriSS method comes precisely to this purpose, as we illustrate next.

7.2 The TriSS Triage Method

When multiple instances of different security smells appear in an MSA, practitioners need to identify the security smell instances that demand urgent attention. This requires considering that in any MSA, some services are more "relevant" to the business than others, and that different services may have different requirements on quality attributes. For example, services handling sensitive data may prioritize confidentiality and integrity over availability, whereas frontend services may do the opposite.

To account for this, we propose TriSS, a triage method for security smell instances affecting an MSA. The method combines (1) the business relevance of the service affected by the security smell instance, and (2) the impact of the smell instance over the quality attributes required by the affected service. The combination yields an "urgency code", which is assigned to the security smell instance, enabling comparisons of resolution urgency among security smell instances that affect the same MSA. The urgency code assignment allows allocating security smell instances to "urgency classes," which can be used to determine which to address first, e.g., starting with those associated with the highest urgency code.

The TriSS method relies on stakeholders to specify the relevance (*None*, *Low*, *Medium*, or *High*) of the services forming an MSA, since they own the contextual knowledge needed to specify it. For example, the

Policy Management Backend service in our motivating scenario (Section 7.1) might be assigned with *High* relevance because it handles the insurance policies sold by Lakeside Mutual, whilst *Spring Boot Admin* might have relevance *None* since it monitors and administrates other services in the MSA.

TriSS also relies on stakeholders to classify the importance of quality attributes (QA) for each service, using the same scale *None–High*. For example, in our motivating scenario, if leaking customer information were considered important but less so than altering operating data, then *Policy Management Backend*'s requirements on QAs might have *High* relevance to integrity and authenticity but *Medium* relevance to confidentiality.

Instead, the impact of security smells on quality attributes can be directly extracted from our previous work [107], where we systematically elicited the impacts of microservice security smells and refactorings beyond security. For example, Figure 7.2 depicts the impacts of the *publicly accessible microservices* security smell [107] by using a softgoal interdependency graph (SIG) [25], as we have proposed in [104]. It shows that keeping (i.e., not resolving) an instance of such security smell on a given service negatively impacts its confidentiality, modifiability, and testability.

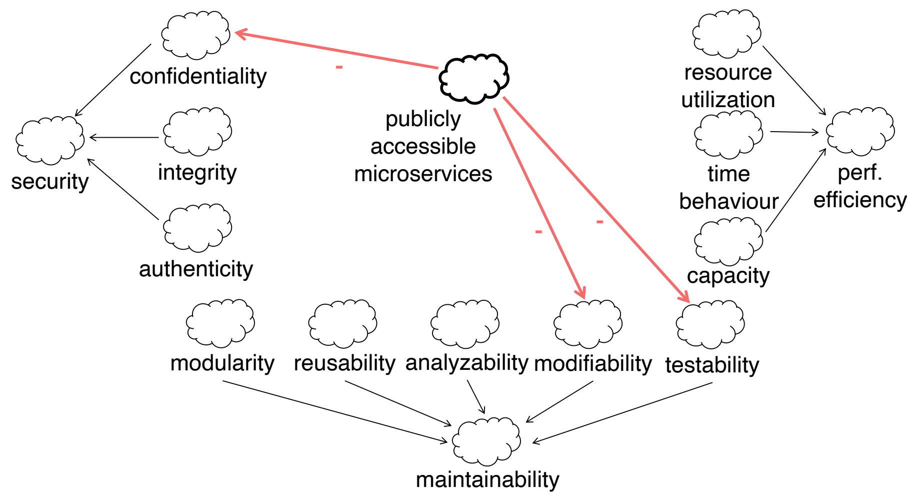


Figure 7.2: SIG displaying the impact of keeping a *publicly accessible microservices* smell.

The SIGs showing the impacts of all other microservice security smells are accessible online[2].

With this information, TriSS computes the urgency code for a security smell instance using the affected service's relevance and its QA requirements. This is done with the triage matrix in Figure 7.3, which enables assigning each security smell instance with an urgency code varying from *H* (highly urgent) to ∅ (non-urgent). While stakeholders directly give the service relevance (x-axis), the value of the impact on quality attributes (y-axis) is determined by considering the highest priority of a negatively impacted QA.

[2] https://ms-security.github.io

Impact on QAs				
High	m	M	h	H
Medium	L	m	M	h
Low	l	L	m	M
None	$\varnothing$	l	L	m
	None	Low	Medium	High
				Service relevance

Figure 7.3: TriSS' color-coded triage matrix: H (high); h (medium to high); M (medium); m (low to medium); L (low); l (none to low); $\varnothing$ (none).

For example, consider a security smell instance that negatively impacts *confidentiality* and *analyzability*. If the relevance of these quality attributes to an affected service has been assessed as *Low* and *Medium*, respectively, then the overall impact on quality attributes of this specific security smell instance is computed as *Medium* (i.e., the highest between *Low* and *Medium*). Finally, suppose the affected service's relevance is *Low*. In that case, TriSS will assign to this specific security smell instance with the urgency code m (Figure 7.3), i.e., low-to-medium urgency.

7.3 Use case

Consider again the 32 security smell instances affecting the Lakeside Mutual application (Section 7.1). We can apply the TriSS method to assign each security smell instance with an urgency code, if we know for each service its relevance in the Lakeside Mutual MSA and the importance of its quality attributes. Suppose that we are given the information that is in Table 7.2, which contains the service relevance and quality attributes importance for the motivating scenario (Section 7.1). For instance, *Customer Core* is assigned with a medium relevance and prioritizing security properties, since it handles customers' personal information.

With the information in Table 7.2, we can exploit TriSS to assign the security smell instances in our motivating scenario with urgency codes. Consider, for instance, the *publicly accessible microservice* smell instance affecting the *Customer Self-Service Backend* service. Keeping such *publicly accessible microservice* smell instance negatively impacts the confidentiality, modifiability, and testability of *Customer Self-Service Backend*, as displayed by the SIG in Figure 7.2. At the same time, *Customer Self-Service Backend* sets the importance of confidentiality, modifiability, and testability to *High*, *Low*, and *None*, respectively (Table 7.2), which means that the impact on QAs of the considered security smell instance is *High* (since the confidentiality and modifiability have *High* importance and they are both negatively impacted by such smell instance). This, combined with the *High* relevance of *Customer Self-Service Backend*, results in the

Table 7.2: Service relevance and QA's importance for the motivating scenario.

Service Name	**Relevance**	**Importance of QAs**
Customer Core	*Medium*	*High*: confidentiality, integrity. *Medium*: time behaviour. *Low*: modifiability.
Customer Management Backend	*Medium*	*High*: integrity. *Medium*: authenticity, time behaviour. *Low*: modifiability.
Customer Management Frontend	*Low*	*High*: integrity. *Medium*: authenticity, time behavior. *Low*: modifiability.
Customer Self-Service Backend	*High*	*High*: confidentiality, time behaviour. *Medium*: integrity, authenticity. *Low*: modifiability, resource utilization.
Customer Self-Service Frontend	*Medium*	*High*: confidentiality, time behaviour. *Medium*: integrity, authenticity. *Low*: modifiability, resource utilization.
Policy Management Backend	*High*	*High*: integrity, authenticity. *Medium*: confidentiality. *Low*: analyzability.
Policy Management Frontend	*Medium*	*High*: integrity, authenticity. *Medium*: confidentiality. *Low*: analyzability.
Risk Management Client	*Low*	*High*: time behavior, resource utilization. *Medium*: integrity. *Low*: reusability.
Risk Management Server	*Low*	*High*: time behavior, resource utilization. *Medium*: integrity. *Low*: reusability.
Spring Boot Admin	*None*	*High*: resource utilization. *Medium*: authenticity, time behaviour. *Low*: reusability.

publicly accessible microservice smell instance affecting the *Customer Self-Service Backend* service getting assigned with the urgency code H, i.e., it is highly urgent.

By systematically applying the TriSS method to the 32 security smell instances, as we already illustrated above, we obtain the urgency codes displayed in Table 7.3 for our motivating scenario (Section 7.1). The table also partitions the security smell instances into urgency classes, which provide practitioners and stakeholders with information about which security smell instances are more urgent than others, thus probably needing to be resolved first.

Table 7.3: Urgency codes assigned to security smell instances in Lakeside Mutual.

Urgency	Smell instances
H	⟨*Publicly accessible microservices*, *Customer Self-Service Backend*⟩, ⟨*Unauthenticated traffic*, *Customer Self-Service Backend*⟩, ⟨*Unauthenticated traffic*, *Policy Management Backend*⟩, ⟨*Hardcoded secrets*, *Customer Self-Service Backend*⟩, ⟨*Hardcoded secrets*, *Policy Management Backend*⟩, ⟨*Non-secured service-to-service communications*, *Customer Self-Service Backend*⟩, ⟨*Non-secured service-to-service communications*, *Policy Management Backend*⟩, ⟨*Insufficient access control*, *Customer Self-Service Backend*⟩
h	⟨*Publicly accessible microservices*, *Customer Core*⟩, ⟨*Publicly accessible microservices*, *Customer Self-Service Frontend*⟩, ⟨*Publicly accessible microservices*, *Policy Management Backend*⟩, ⟨*Unauthenticated traffic*, *Customer Self-Service Frontend*⟩, ⟨*Unauthenticated traffic*, *Policy Management Frontend*⟩, ⟨*Hardcoded secrets*, *Customer Management Backend*⟩, ⟨*Non-secured service-to-service communications*, *Customer Management Backend*⟩, ⟨*Non-secured service-to-service communications*, *Customer Self-Service Frontend*⟩, ⟨*Non-secured service-to-service communications*, *Policy Management Frontend*⟩, ⟨*Non-secured service-to-service communications*, *Customer Core*⟩, ⟨*Insufficient access control*, *Policy Management Backend*⟩
M	⟨*Publicly accessible microservices*, *Policy Management Frontend*⟩, ⟨*Unauthenticated traffic*, *Customer Management Backend*⟩, ⟨*Non-secured service-to-service communications*, *Customer Management Frontend*⟩,
m	⟨*Publicly accessible microservices*, *Customer Management Backend*⟩, ⟨*Unauthenticated traffic*, *Customer Management Frontend*⟩, ⟨*Hardcoded secrets*, *Risk Management Server*⟩
L	⟨*Publicly accessible microservices*, *Customer Management Frontend*⟩, ⟨*Unauthenticated traffic*, *Spring Boot Admin*⟩, ⟨*Non-secured service-to-service communications*, *Spring Boot Admin*⟩, ⟨*Insufficient access control*, *Customer Management Backend*⟩
l	⟨*Publicly accessible microservices*, *Risk Management Server*⟩,
$\varnothing$	⟨*Publicly accessible microservices*, *Spring Boot Admin*⟩, ⟨*Insufficient access control*, *Spring Boot Admin*⟩

7.4 Experimental evaluation

To evaluate the usefulness of the TriSS method, we run a controlled experiment.[3] The experiment consisted of asking practitioners to triage a subset of the microservice security smell instances affecting the Lakeside Mutual application (Section 7.1), while considering the services' relevance and importance of quality attributes displayed in Table 7.2. The ultimate goal was to extract answers for the following two research questions:

RQ_1) *Is TriSS easing the triage of microservice security smells?*

RQ_2) *Is TriSS increasing the confidence of practitioners in the results of the triage process?*

7.4.1 Experimental Study Design

Subjects were asked to complete three triage tasks, each consisting of assigning urgency codes to a subset of security smell instances that affect the Lakeside Mutual MSA. The authors of this research chose the subset to expose subjects to the several dimensions that make triaging complex, i.e., affected services and instantiated smells. The first task focuses on affected services, asking subjects to assign urgency codes to several instances of the *same* security smell, but each affecting a *different* service. The second task focuses instead on microservice security smells, with subjects asked to triage instances of *different* smells but affecting the *same* service. Finally, the third task exposes subjects to the full complexity of triaging, asking them to triage instances of *different* security smells that affect *different* services.

The experimental activities were (a) doing the triage intuitively, i.e., *without* the TriSS method, and (b) doing the triage *with* TriSS. Thus, the triaging tasks were organized into five steps:

1. Receiving an introduction to microservice security smells and their impacts on security and quality attributes, and to the Lakeside Mutual MSA.

2. Executing the three triage tasks intuitively, i.e., *without* TriSS, using only the MSA documentation and information on the services' relevance and QA's importance (Table 7.2). A questionnaire collected data on subjects' confidence (on a 1–5 Likert scale) in their urgency assessments.

3. Receiving an introduction to the TriSS method.

4. Executing the three triage tasks *with* TriSS. Again, a questionnaire collected data on subjects' confidence in their urgency assessments.

[3] A replication package containing all the sources of the controlled experiment can be found here `https://zenodo.org/doi/10.5281/zenodo.10987857`.

5. Responding to a final questionnaire on (a) work experience and (b) easiness perception of performing Steps 2 and 4.

The questionnaires allowed us to get answers for:

- RQ_1 (i.e., assessing whether subjects found it easier to triage using *with vs. without* TriSS) from the answers of Step 5.
- RQ_2 (i.e., assessing subjects' confidence on the assigned urgency codes *with vs. without* TriSS) from the answers of Steps 2 and 4.

The experimental study design was carried out by the author of this thesis, and it was then validated by submitting a first version of the controlled experiment to the other authors of this research. They were then independently running the experimental activities and providing their feedback on each activity. The feedback was analyzed in a joint discussion session, which resulted in the controlled experiment described above, which was then submitted to practitioners. As a result, the experiment was set to include the triage tasks defined in Table 7.4, to which the subjects were exposed during the experiment.

Table 7.4: Triage tasks defined for the controlled experiment.

Task	Subtask	Description
1	1-1	Insufficient Access Control affects Spring Boot Admin service
	1-2	Insufficient Access Control affects Customer Management Backend service
	1-3	Insufficient Access Control affects Customer Self-Service Backend service
2	2-1	Non-Secured Service-to-Service Communications affects Customer Management backend service
	2-2	Unauthenticated Traffic affects Customer Management backend service
	2-3	Publicly Accessible Microservices affects Customer Management backend service
3	3-1	Insufficient Access Control affects Policy Management Backend service
	3-2	Publicly Accessible Microservices affects Customer Management Frontend service
	3-3	Unauthenticated Traffic affects Customer Self-Service Frontend service
	3-4	Hardcoded Secrets affects Risk Management Server service

7.4.2 Experimental Study Execution

The experimental study was conducted as a seminar in January 2024. It involved 26 practitioners, all owning a professional degree in Computer Science or Computer Engineering, but with varying work experience.

Practitioners were asked to complete the triage tasks in Table 7.4. Each task consisted of assigning an urgency code to a specific subset of the security smell instances affecting the Lakeside Mutual MSA, using the information provided in Table 7.2.

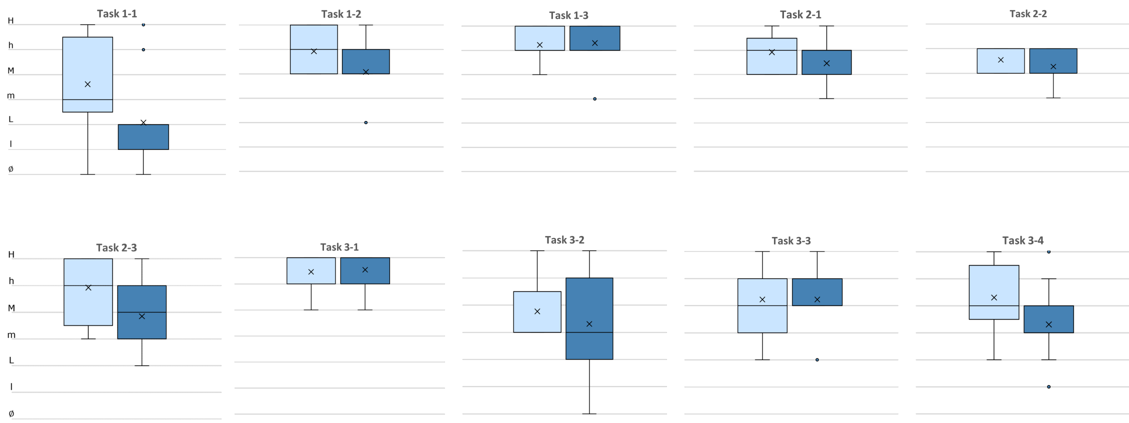


Figure 7.4: Distribution of urgency codes assigned by practitioners with 0-2 years of experience. Light and dark blue correspond to without and with the use of TriSS respectively.

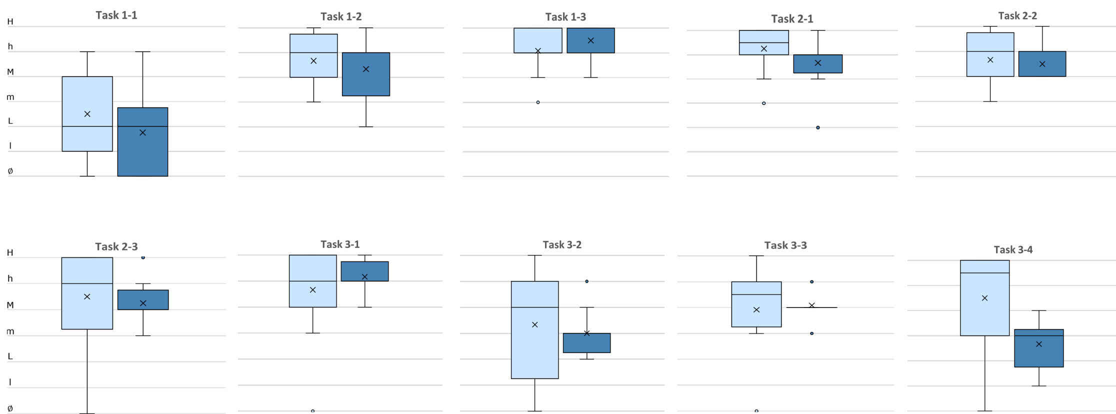


Figure 7.5: Distribution of urgency codes assigned by practitioners with 3+ years of experience. Light and dark blue correspond to without and with the use of TriSS, respectively.

7.4.3 Triage Results

The urgency codes assigned by practitioners while running the given triage tasks can be observed in the box plots in Figure 7.4 and Figure 7.5. The figures show the triage results by distinguishing subjects' experience in software development, viz., a group of 14 practitioners with 0-2 years experience in software development (Figure 7.4), and a group of 12 practitioners with 3+ years experience (Figure 7.5). In both figures, the light blue boxes represent the urgency codes assigned without using TriSS, while the dark blue boxes represent urgency codes assigned afterwards when using TriSS.

Figures 7.4 and 7.5 show that, overall, the agreement among practitioners in assigned urgency codes increased after knowing and applying the TriSS method (as it can be observed by the lower distribution plotted by dark blue boxes, if compared to light blue boxes). Additionally, the urgency codes assigned by

practitioners lowered when using TriSS. This suggests that, initially, practitioners overestimated the urgency of the security smell instances they were asked to triage and realized the overestimate thanks to TriSS.

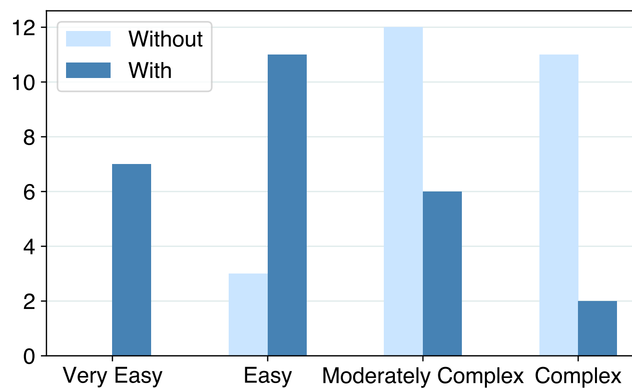
During the first task, practitioners were asked to assign urgency codes to instances of the *insufficient access control* security smell affecting the *Spring Boot Admin*, *Customer Management Backend*, and *Customer Self-Service Backend* services. Subtasks 1-1 and 1-2 (Figures 7.4 and 7.5, top left) showed a general reduction in assigned urgency codes, with practitioners reconsidering their initial triage. A noteworthy case emerged in subtask 1-3, where the distribution of assigned urgency codes remained consistent regardless of whether TriSS was used. This was observed independently of the experience of practitioners, and it was mainly motivated by the fact that subtask 1-3 was straightforward. The task was indeed asking to classify a security smell instance impacting important QAs for a business central service, which one can readily map to highly urgent.

The second task asked practitioners to triage instances of the *non-secured service-to-service communications*, *unauthenticated traffic* and *publicly accessible microservices* security smells, affecting the *Customer Management Backend* service. In the cases of subtasks 2-1 and 2-3, we again observed a reduction of the actual urgency codes assigned by practitioners after using TriSS. The case of subtask 2-2 (Figures 7.4 and 7.5, top right) was instead similar to subtask 1-3. Again, when submitted to a straightforward triage task, with a security smell impacting on quality attributes crucial to a medium relevant service, practitioners already classified such smell instance as medium-to-highly urgent — even before knowing about TriSS.

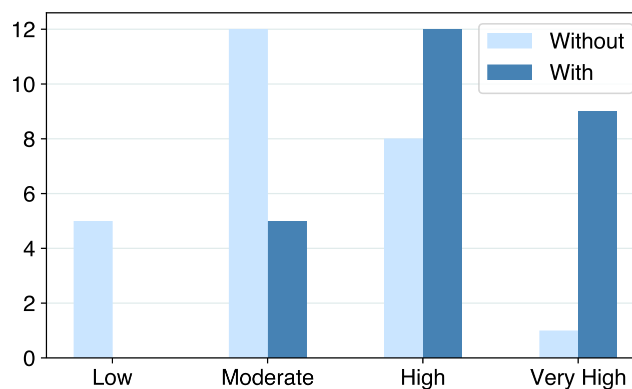
In the third task, practitioners were asked to triage an instance of the *insufficient access control* security smell affecting the *Policy Management backend* service, an instance of *publicly accessible microservices* affecting the *Customer Management Frontend* service, an instance of *unauthenticated traffic* affecting *Customer Self-Service Frontend*, and an instance of *hardcoded secrets* affecting the *Risk Management Server*. Also in this case (Figures 7.4 and 7.5, bottom right), overall, practitioners reduced the assigned urgency codes after using TriSS, and the decrease was quite significant in the case of experienced practitioners. This suggests that — in this case — TriSS helped experienced practitioners to realize that they were significantly overestimating their intuitively assigned urgency codes.

7.4.4 Answers to Research Questions

Figure 7.6 summarizes the data collected on (a) ease of triaging and (b) confidence in assigned urgency codes. The histogram in Figure 7.6a provide an answer to RQ_1: most subjects found the given triaging tasks moderately complex or complex when working *without* the TriSS method, whilst they found the same tasks easy or very easy when done *with* TriSS.



(a)



(b)

Figure 7.6: Subjects answers on (a) ease of triaging and (b) confidence on assigned urgency, both *with vs. without* TriSS.

Figure 7.6b instead provides a first answer to RQ_2: the practitioners were generally more confident in their results when triaging *with* the TriSS method than *without* it. Indeed, the majority of practitioners indicated high or very high confidence in the urgency codes assigned in the given triage tasks *with* TriSS, but moderate confidence when doing such tasks *without* TriSS.

Also for RQ_2, we compared results according to the subjects' experience in software development: one group of 14 practitioners with 0-2 years, and one group of 12 with 3+ years.

- Most (70.2%) of the urgency codes assigned intuitively were revised upon using the TriSS method, with 47.6% of the urgency codes actually revised *downwards* (Figure 7.7a).
- Upon using the TriSS method, *most* experienced practitioners *reduced their self-assessed confidence* on their intuitive triage assignments (Figure 7.7b), although this effect was not present for novice practitioners.

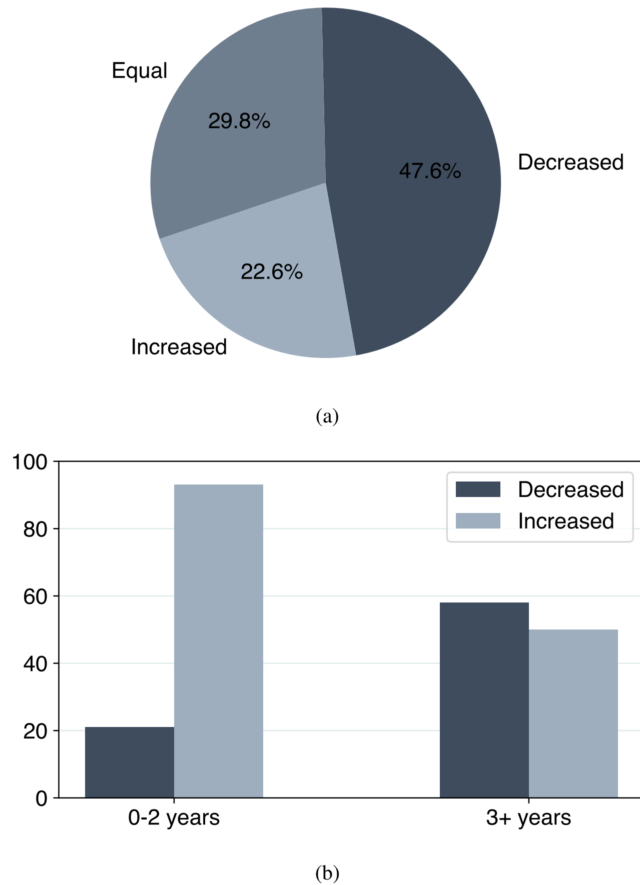


Figure 7.7: Changes in (a) assigned urgency codes and (b) confidence on intuitively assigned urgency codes.

- Independently of their experience, most practitioners had higher confidence in urgency codes assigned *with* the TriSS method as compared to those assigned *without* it (Figure 7.7b).

The above suggests that (i) intuitive triage assignments tend to overestimate the urgency of security smell instances, and that (ii) using the TriSS method yields assessments on which practitioners are more confident. Point (ii) holds especially for experienced practitioners, who *realize* upon formalizing the triage process that they have been "overconfident", whereas novices had no previous intuition on which over-trust.

7.5 Discussion

By looking at the urgency codes assigned by practitioners (Figures 7.4 and 7.5), we can observe two distinct behaviors, which provide insights into the usefulness of our proposed triage method:

- In two cases (specifically, tasks 1-3 and 2-2), the agreement on urgency codes assigned by practitioners was similar, independently of whether they were using TriSS and of their actual experience. This was because, in these particular tasks, practitioners had to triage security smell instances that affected a relevant service for the business, and some of the quality attributes affected by these instances were assigned with high importance for the specific service.

- In all other cases, the agreement on urgency codes was higher when using TriSS than when urgency codes were assigned intuitively. In these cases, practitioners had to triage security smell instances affecting services classified as having low and medium relevance for the business, making it, therefore, much more complex to estimate the urgency for such security smell instances.

The above suggests that, while there are straightforward cases where triage decisions are evident (such as when security smell instances impact critical quality attributes for highly relevant services), our findings suggest that our proposed triage method provides value in the more complex scenarios.

Additionally, the greater the number of security smell instances affecting an MSA, the greater the importance and usefulness of the triage process itself. Although the MSA considered in our motivating scenario is relatively small, it already includes enough security smell instances to showcase how the value of TriSS can be exploited to tackle the — already complex — triage process. This suggests that the value given by TriSS might be even more useful in real-world industrial MSAs, which might include hundreds of interacting services.

On the other hand, one may argue that relying on stakeholders to provide the input information needed by TriSS may introduce subjectivity into the proposed triage method (because, e.g., different stakeholders may assign different business relevance values to a service or different importance to quality attributes). However, the stakeholders of TriSS are those who actually own contextual knowledge on a considered MSA, namely the business relevance of the services therein and the importance of quality attributes for such services, making their involvement crucial in the triage process. Thus, while subjectivity cannot be eliminated, consistency can be enhanced by involving a diverse set of stakeholders, e.g., business analysts, developers, and architects. Also, subjectivity is not a limitation of TriSS but rather an intrinsic characteristic of triaging, as it actually happens when patients are triaged after entering to emergency rooms.

7.6 Related work

Resolving issues affecting modern applications is crucial [14, 69]. However, despite there exist solutions for detecting microservice security smell instances and reasoning about their resolution, e.g., [29, 151], the problem of triaging the "most urgent" security smell instances is still open [23]. Indeed, several triage approaches have been developed for triaging issue resolution, but, to the best of our knowledge, TriSS is the first to allow triage of security smell instances in MSAs.

The existing approaches differ from TriSS as they typically focus on resolving so-called *code smells* [143], instead of the *MSA security smells* identified in [105]. Some proposals focus on the triage of known *code smells* in object-oriented applications. For example, [50] assesses the severity of code smells by analyzing execution traces obtained with runtime tests, whereas [48] and [72] exploit several source code static analysis tools to estimate the severity of code smells in an object-oriented application. These approaches do not allow triage of security smells in an MSA, which is composed of several services possibly developed in several (perhaps non-object-oriented) programming languages.

Some approaches estimate the severity of code smells affecting the source code of a target application [5, 3], or use machine learning models to rank the code smells affecting an application according to their predicted severity [7, 71, 98]. Others prioritize the resolution of code smell instances considering the context where they occur. Thus, priorities may differ (even for instances of the same smell) depending on the affected application components. In particular, [121, 122] compute a context-relevant index by processing the information available in the issue tracking system of the affected component, whereas [145] ranks code smells using a combination of three criteria, namely past modifications of the affected components, their actual modifiability, and the known relevance of the smells themselves. Differently from TriSS, the methods mentioned above focus on code smells for component-based applications rather than security smells in MSAs.

Chapter 8

Conclusions

Securing microservice-based applications (MSAs) is crucial. *Security smells* denote symptoms of bad – though often unintentional– design decisions, which may result in violating security properties, and which can be resolved via refactoring. This thesis has reported on what is being said by practitioners and researchers about known security smells and refactorings enabling to mitigate their effects. It further complemented these findings by reporting on what is being said about bad/good practices for securing MSA.

An end-to-end model-driven approach for resolving security smells in existing MSAs was also proposed in this thesis. This approach automatizes smell detection and provides users with an interactive mechanism for smell resolution across the concerned MSA components. Additionally, this thesis has also introduced a method to enact the trade-off analysis of microservice security smells using SIGs [25], which provides a visual and holistic panorama of the positive/negative impacts of each security smell and refactorings on software quality attributes and microservices' key design principles.

Finally, with regards to the microservice security smells instances affecting an MSA, stakeholders take into account the service's business value, problem criticality, and available resources to decide which smells to resolve or leave alone, but making such decisions is inherently complex for MSAs with many services, possibly affected by multiple security smells instances. Borrowing from hospital emergency room triage practices, which assign an urgency code to incoming patients, this thesis has introduced the notion of *urgency* for microservice security smell instances, and proposed the TriSS method to *triage* them. TriSS enables assigning to each security smell instance with an urgency code based on combining the service's business relevance and the smell's impact on security and other quality attributes, e.g., performance and maintainability. In this chapter we summarise the research contributions contained in this thesis (see Section 8.1) and we also give some perspectives for future work (see Section 8.2).

8.1 Summary of contributions

This section recaps the contributions presented in this thesis, by linking them to each of the research objectives stated in Section 1.1

O_1: What are the effects of bad security decisions for microservice-based applications?

In Chapter 2 we presented the results of a multivocal literature review focused on identifying the smells denoting possible security violations in MSAs, and on the refactorings (proposed by practitioners) enabling to mitigate the effects of such smells. The taxonomy associates each smell with the ISO/IEC 25010 [55] security properties it can violate, and with the refactorings that should all be applied to mitigate its effects. We also provided an overview of the actual recognition of each smell in the selected literature, we discussed the effects of such smells in detail, and we showed how each corresponding refactoring enables mitigating their effects. Our research hence complements other existing studies on smells and refactorings for microservices, e.g., Carrasco et al. [21] and Neri et al. [87], by covering the security aspects of MSAs.

In Chapter 3 we provided a "snapshot" of the bad and good practices for securing MSAs, by means of a multivocal literature review reporting on what is being said by researchers and practitioners on the topic. This results complements our former review of security smells and refactorings [105], as well as existing studies on known issues and solutions for microservices [21, 87, 137], by covering the aspect of bad/good practices for microservices security. Also, and together with our review presented in Chapter 2, it provides a first body of knowledge that can be exploited by researchers as a starting point to study new techniques/solutions for securing MSAs, or to delineate novel directions for future research on the topic.

O_2: How to detect/resolve security smells affecting a microservice-based application?

In Chapter 4 we have introduced an end-to-end model-driven approach for resolving microservices security smells in MSAs. Our approach recovers the software application architectural design using LEMMA models. The models address different viewpoints in the MSA development process and contain, among others, information about security aspects of Java-based MSAs and to automatically detect the two most recognized security smells for microservices (viz., *Publicly Accessible Microservices* and *Insufficient Access Control*). We demonstrated that our approach enables selecting the refactorings to apply to resolve detected security smells, as well as how it automatically updates LEMMA models and adapts the microservices' source code by implementing the selected refactoring when it is possible or providing detailed information about manual refactoring possibilities.

O_3: What are the impacts of microservices' security smells on quality attributes besides security?

In Chapter 5 we have introduced a method based on Softgoal Interdependency Graphs (SIGs) to support

trade-off analysis related to keeping security smells in MSAs or applying some refactorings. Each SIG aims to provide a holistic view of the positive and negative impacts of a microservice security smell and its possible refactorings on software quality attributes and microservices' key design principles.

In Chapter 6 we presented an analysis of the impacts of microservice security smells and refactorings on two other quality attributes that are crucial to microservices, viz., maintainability and performance efficiency, as well as on applications' adherence to microservices' key design principles. More precisely, we systematically elicited 42 possible impacts, which we validated through an online survey that confirmed 35 of such impacts. In analyzing the survey's results, we also commented on the answers given by interviewed practitioners and researchers, who were mostly aligned in agreeing with elicited impacts. A few misalignments emerged on possible impacts on testability and performance efficiency, with practitioners being more cautious in agreeing with them compared to researchers.

We believe that the elicited impacts can help in deciding whether to keep a security smell or apply a refactoring to mitigate its effects, based on their impacts beyond security itself. For this reason, and to feature the visualization needed to make informed decisions [104], we developed an open-source, visual tool that is publicly accessible online.[1]

O_4: How to determine which microservices' security smells to resolve first?

In Chapter 7 we considered the problem of triaging the security smell instances in MSAs. We indeed introduced TriSS, a triage method to systematically assign urgency codes to the microservice security smell instances appearing in an MSA. TriSS computes the urgency of a smell instance using the business value of the affected service and the importance of the affected quality attributes for such service.

The applicability of TriSS was illustrated with a use case based on an existing, third-party MSA. The usefulness of TriSS has also been validated with a controlled experiment where 26 practitioners were asked to triage security smell instances *with vs. without* the TriSS method. The practitioners' feedback suggests that TriSS generally *simplifies* the triage process and *increases the confidence* in its results.

8.2 Possible directions for future work

We plan to extend the current implementation of the end-to-end model-driven approach for resolving microservices security smells presented in Chapter 4, into a full-fledged prototype, featuring model-driven detection and refactoring of *all* the microservice security smells presented in Chapter 2. We also plan to exploit the full-fledged prototype to validate our method on real-world applications and demonstrate how

[1] https://ms-security.github.io.

our approach facilitates the development process of MSAs by providing means for security smell resolution.

Additionally, we plan to further assist practitioners by supporting them in deciding whether to refactor a detected microservice security smell, e.g., by integrating with trade-off analyses like that proposed in Chapter 5, and by extending our approach to work with other microservice-related smells, e.g., the architectural smells from [87] or [137]. Moreover, we aim to allow for the generic extensibility of our approach in that developers can add new resolutions of security smells based on the abstracted specification of (i) model traversals and element filtering; and (ii) operations for model-based refactorings on traversed elements.

On the other hand, we also plan to develop automated support for practitioners to resolve security smell instances affecting concrete MSA, in line with the desiderata identified by [77]. In particular, a graphical tool will display the security smell instances affecting an MSA (e.g., detected by KubeHound [29]). The graphical tool will also enable visualizing the urgency of the smell instances in an MSA, with such urgency automatically computed by implementing TriSS' systematic combination of the relevance of affected services and the importance of their quality attributes. This will also support reasoning on whether/how to apply refactorings using a trade-off analysis, as we have proposed in Chapter 5. Finally, another interesting direction for future work is to relate this research to technical debt, given that, e.g., the results of the triage process might consider keeping certain security smells instances in an MSA.

Bibliography

[1] F. Abasi. Securing modern api- and microservices-based apps by design. IBM Developer, https://developer.ibm.com/technologies/api/articles/securing-modern-api-and-microservices-apps-1/, 2019.

[2] A. Almogahed and M. Omar. Refactoring techniques for improving software quality: Practitioners' perspectives. *Journal of Information and Communication Technology*, 20(4):511–539, 2021.

[3] T. Alshammari and M. Alshayeb. Toward a software bad smell prioritization model for software maintainability. *Arabian Journal for Science and Engineering*, 46, 2021.

[4] D. Arcelli, V. Cortellessa, and D. D. Pompeo. Automating performance antipattern detection and software refactoring in UML models. In X. Wang, D. Lo, and E. Shihab, editors, *2019 International Conference on Software Analysis, Evolution and Reengineering,* , pages 639–643. SANER 2019, IEEE Computer Society, 2019.

[5] F. Arcelli Fontana, V. Ferme, M. Zanoni, and R. Roveda. Towards a prioritization of code debt: A code smell intensity index. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 16–24, 2015.

[6] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto. Arcan: A tool for architectural smells detection. In I. Malavolta and R. Capilla, editors, *2017 IEEE International Conference on Software Architecture Workshops,* , pages 282–285. ICSA 2017 Workshops, IEEE Computer Society, 2017.

[7] T. W. W. Aung, Y. Wan, H. Huo, and Y. Sui. Multi-triage: A multi-task learning framework for bug triage. *Journal of Systems and Software*, 184:111133, 2022.

[8] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[9] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn. Microservices migration patterns. *Software: Practice and Experience*, 48(11):2019–2042, 2018.

[10] T. Basit. Manual or electronic? the role of coding in qualitative data analysis. *Educational Research*, 45(2):143–154, 2003.

[11] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[12] S. Behrens and B. Payne. Starting the avalanche: Application ddos in microservice architectures. The Netflix Tech Blog, https://netflixtechblog.com/starting-the-avalanche-640e69b14a06, 2017.

[13] D. Berardi, S. Giallorenzo, J. Mauro, A. Melis, F. Montesi, and M. Prandini. Microservice security: a systematic literature review. *PeerJ Comput. Sci.*, 8:e779, 2022.

[14] T. Besker, A. Martini, and J. Bosch. Technical debt triage in backlog management. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 13–22, 2019.

[15] E. Boersma. Top 10 security traps to avoid when migrating from a monolith to microservices. Sqreen, https://blog.sqreen.com/top-10-security-traps-to-avoid-when-migrating-from-a-monolith-to-microservices/, 2019.

[16] J. Bogner, T. Boceck, M. Popp, D. Tschechlov, S. Wagner, and A. Zimmermann. Towards a collaborative repository for the documentation of service-based antipatterns and bad smells. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 95–101, United States, 2019. IEEE Computer Society.

[17] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. Limiting technical debt with maintainability assurance: an industry survey on used techniques and differences with service- and microservice-based systems. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, New York, NY, USA, 2018. Association for Computing Machinery.

[18] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. Microservices in industry: Insights into technologies, characteristics, and software quality. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 187–195, United States, 2019. IEEE Computer Society.

[19] R. Budko. Five things you need to know about api security. The New Stack, https://thenewstack.io/5-things-you-need-to-know-about-api-security/, 2018.

[20] J. Carnell. *Spring Microservices in Action*. Manning Publications, Shelter Island, NY, United States, 1st edition, 2017.

[21] A. Carrasco, B. v. Bladel, and S. Demeyer. Migrating towards microservices: Migration and architecture smells. In *Proceedings of the 2nd International Workshop on Refactoring*, IWoR 2018, page 1–6, New York, NY, USA, 2018. Association for Computing Machinery.

[22] L. Carvalho, A. Garcia, W. K. Assunção, R. de Mello, and M. J. de Lima. Analysis of the criteria adopted in industry to extract microservices. In *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, pages 22–29. IEEE, 2019.

[23] T. Cerny et al. Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study. *Journal of Systems and Software*, 206:111829, 2023.

[24] R. Chandramouli. Security strategies for microservices-based application systems. NIST Special Publication 800-204, https://doi.org/10.6028/NIST.SP.800-204, 2019.

[25] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. The NFR framework in action. In *Non-Functional Requirements in software engineering*, pages 15–45. Springer, 2000.

[26] B. Combemale, R. B. France, J.-M. Jézéquel, B. Rumpe, J. Steel, and D. Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. CRC Press, first edition, 2017.

[27] R. da Silva. Best practices to protect your microservices architecture. Medium, https://medium.com/@rcandidosilva/best-practices-to-protect-your-microservices-architecture-541e7cf7637f, 2017.

[28] T. de Oliveira Rosa, J. a. F. L. Daniel, E. M. Guerra, and A. Goldman. A method for architectural trade-off analysis based on patterns: Evaluating microservices structural attributes. In *EuroPLoP*, 2020.

[29] G. Dell'Immagine, J. Soldani, and A. Brogi. Kubehound: Detecting microservices' security smells in kubernetes deployments. *Future Internet*, 15(7), 2023.

[30] P. Di Francesco, P. Lago, and I. Malavolta. Migrating towards microservice architectures: An industrial survey. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 29–38, United States, 2018. IEEE Computer Society.

[31] B. Doerfeld. How to control user identity within microservices. NordicAPIs, https://nordicapis.com/how-to-control-user-identity-within-microservices/, 2015.

[32] M. Douglas. Microservices authentication & authorization best practice. CodeBurst, https://codeburst.io/i-believe-it-really-depends-on-your-environment-and-how-well-protected-the-different-pieces-are-7919bfa6bc86, 2018.

[33] Edureka. Microservices security: Best practices to secure microservicess. https://youtu.be/wpA0N7kHaDo, 2019.

[34] G. Elahi and E. Yu. Modeling and analysis of security trade-offs – a goal oriented approach. *Data & Knowledge Engineering*, 68(7):579–598, 2009.

[35] C. Esposito, A. Castiglione, and K. Choo. Challenges in delivering software in the cloud as microservices. *IEEE Cloud Computing*, 3(5):10–14, 2016.

[36] E. Evans. *Domain-Driven Design*. Addison-Wesley, 2004.

[37] D. Farace and J. Schöpfel. *Grey Literature in library and information studies*. K.G. Saur, 2010.

[38] V. Feitosa Pacheco. *Microservice Patterns and Best Practices*. Packt Publishing, Birmingham, United Kingdom, 1st edition, 2018.

[39] D. Gannon, R. Barga, and N. Sundaresan. Cloud-native applications. *IEEE Cloud Computing*, 4(5):16–21, 2017.

[40] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering,* , pages 255–258, USA, 2009. CSMR 2009, IEEE Computer Society.

[41] Z. Gardner. Security in the microservices paradigm. Keyhole Software, https://keyholesoftware.com/2017/03/13/security-in-the-microservices-paradigm/, 2017.

[42] Z. Gardner. Security in the microservices paradigm. DZone, https://dzone.com/articles/security-in-the-microservices-paradigm, 2017.

[43] V. Garousi, M. Felderer, and M. V. Mäntylä. The need for multivocal literature reviews in software engineering: Complementing systematic literature reviews with grey literature. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, pages 1–6, New York, NY, USA, 2016. Association for Computing Machinery.

[44] V. Garousi, M. Felderer, and M. V. Mantyla. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106:101–121, 2019.

[45] M. Garriga. Towards a taxonomy of microservices architectures. In *Software Engineering and Formal Methods*, pages 203–218. Springer, 2018.

[46] G. Gebel and D. Brossard. Securing apis and microservices with oauth, openid connect, and abac. Axiomatics, Webinar, https://www.youtube.com/watch?v=TnCPJUV9RnA, 2018.

[47] J. Ghofrani and D. Lübke. Challenges of microservices architecture: A survey on the state of the practice. In N. Herzberg, C. Hochreiner, O. Kopp, and J. Lenhard, editors, *Proceedings of the 10th Workshop on Services and their Composition (ZEUS 2018)*, pages 1–8, Aachen, Germany, 2018. CEUR-WS.org.

[48] A. Gupta and N. K. Chauhan. A severity-based classification assessment of code smells in kotlin and java application. *Arabian Journal for Science and Engineering*, 47, 2022.

[49] N. Gupta. Security strategies for devops, apis, containers and microservices. Imperva, https://www.imperva.com/blog/security-strategies-for-devops-apis-containers-and-microservices/, 2018.

[50] T. Haendler, S. Sobernig, and M. Strembeck. Towards triaging code-smell candidates via runtime scenarios and method-call dependencies. In *Proceedings of the XP2017 Scientific Workshops*, XP '17. ACM, 2017.

[51] S. Haselböck, R. Weinreich, and G. Buchgeher. Decision models for microservices: Design areas, stakeholders, use cases, and requirements. In A. Lopes and R. de Lemos, editors, *Software Architecture,* , pages 155–170, Cham, 2017. Springer International Publishing.

[52] M. Hofmann, E. Schnabel, and K. Stanley. *Microservices Best Practices for Java*. IBM Redbooks, United States, 1st edition, 2016.

[53] IETF OAuth Working Group. Open authorization (oauth), version 2.0. https://oauth.net/2/, 2012.

[54] K. Indrasiri and P. Siriwardena. Microservices security fundamentals. In *Microservices for the Enterprise: Designing, Developing, and Deploying*, pages 313–345, Berkeley, CA, 2018. Apress.

[55] ISO. Systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. ISO/IEC 25010:2023, 2023.

[56] N. Jackson. *Building Microservices with Go*. Packt Publishing, Birmingham, United Kingdom, 1st edition, 2017.

[57] C. Jain. Top 10 security best practices to secure your microservices - appsecusa 2017. OWASP, https://youtu.be/VtUQINsYXDM, 2018.

[58] M. Kamaruzzaman. Microservice architecture and its 10 most important design patterns. Towards Data Science, https://towardsdatascience.com/microservice-architecture-and-its-10-most-important-design-patterns-824952d7fa41, 2020.

[59] J. Kanjilal. 4 fundamental microservices security best practices. SearchAppArchitecture, https://searchapparchitecture.techtarget.com/tip/4-fundamental-microservices-security-best-practices, 2020.

[60] S. Kapferer and O. Zimmermann. Domain-driven service design: Context modeling, model refactoring and contract generation. In *Service-Oriented Computing*, pages 189–208. Springer, 2020.

[61] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193)*, pages 68–78, 1998.

[62] A. Khan. How to secure your microservices: Shopify case study. Dzone, https://dzone.com/articles/bountytutorial-microservices-security-how-to-secur, 2018.

[63] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, 2007.

[64] H. Knoche and W. Hasselbring. Drivers and barriers for microservice adoption – a survey among professionals in Germany. *Enterprise Modelling and Information Systems Architectures*, 14(1):1–35, 2019. German Informatics Society.

[65] K. Krippendorff. *Content Analysis: An Introduction to its Methodology*. Sage Publications, Thousand Oaks, CA, United States, 2nd edition, 2004.

[66] T. Krishnamurthy. Transition to microservice architecture - challenges. BeingTechie, https://www.beingtechie.io/blog/transition-to-microservices-challenges, 2018.

[67] G. Lea. Microservices security: All the questions you should be asking. https://www.grahamlea.com/2015/07/microservices-security-questions/, 2015.

[68] R. Lemos. App security in the microservices age: 4 best practices. TechBeacon, https://techbeacon.com/app-dev-testing/app-security-microservices-age-4-best-practices, 2019.

[69] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. Arcelli Fontana. A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software*, 171:110827, 2021.

[70] J. Lewis and M. Fowler. Microservices. A definition of this new architectural term. https://martinfowler.com/articles/microservices.html, 2014.

[71] S. Lim, S. Zaidi, H. Woo, and C.-G. Lee. Toward an effective bug triage system using transformers to add new developers. *Journal of Sensons*, 2022, 2022.

[72] R. Malhotra and P. Singh. Exploiting bad-smells and object-oriented characteristics to prioritize classes for refactoring. *International Journal of System Assurance Engineering and Management*, 11, 2020.

[73] J. Mannino. Security in the land of microservices. AppSec EU 2017, https://www.youtube.com/watch?v=JRmWlLY8MGE, 2017.

[74] R. Mao, H. Zhang, Q. Dai, H. Huang, G. Rong, H. Shen, L. Chen, and K. Lu. Preliminary findings about devsecops from grey literature. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 450–457, United States, 2020. IEEE Computer Society.

[75] G. Márquez, J. Soldani, F. Ponce, and H. Astudillo. Frameworks and high-availability in microservices: An industrial survey. In *CIbSE*, pages 57–70, 2020.

[76] G. Márquez, M. M. Villegas, and H. Astudillo. A pattern language for scalable microservices-based systems. In *Proc. of the 12th Europ. Conf. on Software Architecture: Companion Proceedings*, ECSA '18, pages 24:1–24:7. ACM, 2018.

[77] A. Martini, F. A. Fontana, A. Biaggi, and R. Roveda. Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company. In *European conference on software architecture*, pages 320–335, 2018.

[78] N. Mateus-Coelho, M. Cruz-Cunha, and L. G. Ferreira. Security in microservices architectures. In *CENTERIS - International Conference on ENTERprise Information Systems / ProjMAN - International Conference on Project MANagement / HCist - International Conference on Health and Social Care Information Systems and Technologies*, Procedia Computer Science, pages 1–12, Amsterdam, Netherlands, 2020. Elsevier.

[79] S. Matteson. 10 tips for securing microservice architecture. TechRepublic, https://www.techrepublic.com/article/10-tips-for-securing-microservice-architecture/, 2017.

[80] S. Matteson. How to establish strong microservices security using ssl, tls, and api gateways. TechRepublic, https://www.techrepublic.com/article/how-to-establish-strong-microservice-security-using-ssl-tls-and-api-gateways/, 2017.

[81] G. Mazlami, J. Cito, and P. Leitner. Extraction of microservices from monolithic software architectures. In *2017 Int. Conf. on Web Services (ICWS)*, pages 524–531. IEEE, 2017.

[82] M. McLarty, R. Wilson, and S. Morrison. *Securing Microservices APIs*. O'Reilly, Newton, MA, United States, 1st edition, 2018.

[83] V. Mody. From zero to zero trust. Teleport, https://goteleport.com/blog/zero-to-zero-trust/, 2020.

[84] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly, 2016.

[85] A. Nehme, V. Jesus, K. Mahbub, and A. Abdallah. Fine-grained access control for microservices. In N. Zincir-Heywood, G. Bonfante, M. Debbabi, and J. Garcia-Alfaro, editors, *Foundations and Practice of Security*, pages 285–300, Cham, 2019. Springer International Publishing.

[86] A. Nehme, V. Jesus, K. Mahbub, and A. Abdallah. Securing microservices. *IT Professional*, 21(1):42–49, 2019.

[87] D. Neri, J. Soldani, O. Zimmermann, and A. Brogi. Design principles, architectural smells and refactorings for microservices: a multivocal review. *SICS Software-Intensive Cyber-Physical Systems*, 35(1):3–15, 2020.

[88] S. Newman. *Building Microservices*. O'Reilly, Newton, MA, United States, 1st edition, 2015.

[89] S. Newman. Security and microservices. Devoxx, https://youtu.be/ZXGaC3GR3zU, 2016.

[90] P. Nkomo and M. Coetzee. Software development activities for secure microservices. In S. Misra, O. Gervasi, B. Murgante, E. Stankova, V. Korkhov, C. Torre, A. M. A. Rocha, D. Taniar, B. O. Apduhan, and E. Tarantino, editors, *Computational Science and Its Applications – ICCSA 2019*, pages 573–585, Cham, 2019. Springer International Publishing.

[91] L. O'Neill. Microservice security - what you need to know. CrashTest Security, https://crashtest-security.com/microservice-security-what-you-need-to-know/, 2020.

[92] OpenID. Openid connect. https://openid.net/connect/, 2014.

[93] C. Orellana, M. M. Villegas, and H. Astudillo. Assessing architectural patterns trade-offs using moment-based pattern taxonomies. In *XLV Latin American Computing Conference (CLEI)*, 2019.

[94] C. Pahl and P. Jamshidi. Microservices: A systematic mapping study. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*, CLOSER 2016, pages 137–146, Setúbal, Portugal, 2016. SciTePress.

[95] S. Panichella, M. I. Rahman, and D. Taibi. Structural coupling for microservices. *arXiv preprint arXiv:2103.04674*, 2021.

[96] A. Parecki. Oauth: When things go wrong. Okta Developer, https://www.youtube.com/watch?v=H6MxsFMAoP8, 2019.

[97] L. Pasquale, P. Spoletini, M. Salehie, L. Cavallaro, and B. Nuseibeh. Automating trade-off analysis of security requirements. *Requirements Engineering*, 21(4), 2016.

[98] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia. Developer-driven code smell prioritization. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 220–231. ACM, 2020.

[99] A. Pereira-Vale, E. B. Fernandez, R. Monge, H. Astudillo, and G. Márquez. Security in microservice-based systems: A multivocal literature review. *Computers & Security*, page 102200, 2021.

[100] S. Perera. Walking the wire: Mastering the four decisions in microservices architecture. Medium, https://medium.com/systems-architectures/walking-the-microservices-path-towards-loose-coupling-few-pitfalls-4067bf5e497a, 2016.

[101] I. Pigazzini, F. A. Fontana, V. Lenarduzzi, and D. Taibi. Towards microservice smells detection. In *Proceedings of the 3rd International Conference on Technical Debt*, TechDebt '20, page 92–97, New York, NY, USA, 2020. Association for Computing Machinery.

[102] N. Poddar. Simplifying microservices security with a service mesh. Cloud Native Computing Foundation, Webinar, https://youtu.be/Ai8HlkI7Mm4, 2019.

[103] F. Ponce. Towards resolving security smells in microservice-based applications. In C. Zirpins et al., editors, *Advances in Service-Oriented and Cloud Computing*, pages 133–139, Cham, 2021. Springer International Publishing.

[104] F. Ponce, J. Soldani, H. Astudillo, and A. Brogi. Should microservice security smells stay or be refactored? towards a trade-off analysis. In *Software Architecture: 16th European Conference,*

ECSA 2022, Prague, Czech Republic, September 19–23, 2022, Proceedings, pages 131–139, Cham, 2022. Springer, Springer International Publishing.

[105] F. Ponce, J. Soldani, H. Astudillo, and A. Brogi. Smells and refactorings for microservices security: A multivocal literature review. *Journal of Systems and Software*, 192:111393, 2022.

[106] F. Ponce, J. Soldani, H. Astudillo, and A. Brogi. Microservices security: Bad vs. good practices. In *Software Architecture. ECSA 2022 Tracks and Workshops*, pages 337–352, Cham, 2023. Springer International Publishing.

[107] F. Ponce, J. Soldani, C. Taramasco, H. Astudillo, and A. Brogi. To security and beyond: On the impacts of microservice security smells and refactorings. In *2023 XLIX Latin American Computer Conference (CLEI)*, pages 1–10, 2023.

[108] F. Ponce, J. Soldani, C. Taramasco, H. Astudillo, and A. Brogi. Triaging microservice security smells, with triss - accepted for publication. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, New York, NY, USA, 2024. ACM.

[109] F. Rademacher. *A Language Ecosystem for Modeling Microservice Architecture*. PhD thesis, University of Kassel, 2022.

[110] F. Rademacher et al. Graphical and textual model-driven microservice development. In A. Bucchiarone et al., editors, *Microservices: Science and Engineering*, pages 147–179. Springer, 2020.

[111] F. Rademacher, S. Sachweh, and A. Zündorf. Aspect-oriented modeling of technology heterogeneity in microservice architecture. In *2019 IEEE Int. Conf. on Software Architecture (ICSA)*, pages 21–30. IEEE, 2019.

[112] F. Rademacher, S. Sachweh, and A. Zündorf. Deriving microservice code from underspecified domain models using DevOps-enabled modeling languages and model transformations. In *2020 46th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA)*, pages 229–236. IEEE, 2020.

[113] F. Rademacher, S. Sachweh, and A. Zündorf. A modeling method for systematic architecture reconstruction of microservice-based software systems. In *Enterprise, Business-Process and Information Systems Modeling*, pages 311–326. Springer, 2020.

[114] Radware. Microservice architectures challenge traditional security practices. https://blog.radware.com/security/2020/01/microservice-architectures-challenge-traditional-security-practices/, 2020.

[115] M. Raible. 11 patterns to secure microservice architectures. DZone, https://dzone.com/articles/11-patterns-to-secure-microservice-architectures, 2020.

[116] M. Raible. Security patterns for microservice architectures. Okta Developer, https://developer.okta.com/blog/2020/03/23/microservice-security-patterns, 2020.

[117] C. Rajasekharaiah. *Cloud-Based Microservices: Techniques, Challenges, and Solutions*. Apress, New York, NY, United States, 1st edition, 2020.

[118] A. Rezaei Nasab, M. Shahin, P. Liang, M. E. Basiri, S. A. Hoseyni Raviz, H. Khalajzadeh, M. Waseem, and A. Naseri. Automated identification of security discussions in microservices systems: Industrial surveys and experiments. *Journal of Systems and Software*, 181:111046, 2021.

[119] C. Richardson. *Microservices Patterns*. Manning Publications Co., Shelter Island, NY, United States, 2018.

[120] D. Richter, T. Neumann, and A. Polze. Security considerations for microservice architectures. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, pages 608–615, Setúbal, Portugal, 2018. SciTePress.

[121] N. Sae-Lim, S. Hayashi, and M. Saeki. Context-based code smells prioritization for prefactoring. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.

[122] N. Sae-Lim, S. Hayashi, and M. Saeki. Revisiting context-based code smells prioritization: on supporting referred context. In *Proceedings of the XP2017 Scientific Workshops*, XP '17, New York, NY, USA, 2017. Association for Computing Machinery.

[123] V. Sahni. Best practices for building a microservice architecture. Vinay Sahni, https://www.vinaysahni.com/best-practices-for-building-a-microservice-architecture, 2019.

[124] A. Sanchez, L. S. Barbosa, and A. Madeira. Modelling and verifying smell-free architectures with the archery language. In *Software Engineering and Formal Methods*, pages 147–163. SEFM 2015, Springer, 2015.

[125] R. Sass. Security in the world of microservices. ITProPortal, https://www.itproportal.com/features/security-in-the-world-of-microservices/, 2017.

[126] S. Sharma. *Mastering Microservices with Java*. Packt Publishing, Birmingham, United Kingdom, 3rd edition, 2019.

[127] P. Siriwardena. Mutual authentication with tls. In *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*, pages 47–58, Berkeley, CA, 2014. Apress.

[128] P. Siriwardena. Microservices security landscape. WSO2 Integration Summit 2019, https://youtu.be/6jGePTpbgtI, 2019.

[129] P. Siriwardena. Challenges of securing microservices. Medium, https://medium.facilelogin.com/challenges-of-securing-microservices-68b55877d154, 2020.

[130] P. Siriwardena and N. Dias. *Microservices Security in Action*. Manning Publications, Shelter Island, NY, United States, 1st edition, 2020.

[131] T. Smith. How do you secure microservices? DZone, https://dzone.com/articles/how-do-you-secure-microservices, 2017.

[132] T. Smith. How to secure apis. DZone, https://dzone.com/articles/how-to-secure-apis, 2019.

[133] J. Soldani, G. Muntoni, D. Neri, and A. Brogi. The μtosca toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software Pract. Exper.*, 51(7):1591–1621, 2021.

[134] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215 – 232, 2018.

[135] J. Sorgalla, P. Wizenty, F. Rademacher, S. Sachweh, and A. Zündorf. Applying model-driven engineering to stimulate the adoption of devops processes in small and medium-sized development organizations: the case for microservice architecture. *SN Computer Science*, 2(6):459, 2021.

[136] SumoLogic. Improving security in your microservices architecture. https://www.sumologic.com/insight/microservices-architecture-security/, 2019.

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

[138] D. Taibi, V. Lenarduzzi, and C. Pahl. Architectural patterns for microservices: A systematic mapping study. In *Proc. of the 8th Int. Conf. on Cloud Computing and Services Science - Volume 1: CLOSER,*, pages 221–232, Setúbal, Portugal, 2018. SciTePress.

[139] J. Thönes. Microservices. *IEEE Software*, 32(1):116–116, 2015.

[140] K. A. Torkura, M. I. Sukmana, A. V. Kayem, F. Cheng, and C. Meinel. A cyber risk based moving target defense mechanism for microservice architectures. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud*

Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), pages 932–939, United States, 2018. IEEE Computer Society.

[141] M. Troisi. 8 best practices for microservices app sec. TechBeacon, https://techbeacon.com/app-dev-testing/8-best-practices-microservices-app-sec, 2017.

[142] B. Ünver and R. Britto. Automatic detection of security deficiencies and refactoring advises for microservices. In *2023 IEEE/ACM Int. Conf. on Software and System Processes (ICSSP)*, pages 25–34, 2023.

[143] R. Verma, K. Kumar, and H. K. Verma. Code smell prioritization in object-oriented software systems: A systematic literature review. *Journal of Software: Evolution and Process*, 35(12):e2536, 2023.

[144] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi. Jspirit: a flexible tool for the analysis of code smells. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6. IEEE, 2015.

[145] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace. An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23:501–532, 2016.

[146] A. Walker, D. Das, and T. Cerny. Automated code-smell detection in microservices through static analysis: A case study. *Applied Sciences*, 10(21), 2020.

[147] Wallarm. A CISO's guide to cloud application security. https://www.wallarm.com/resources/a-cisos-guide-to-cloud-application-security, 2019.

[148] Wallarm. Shift to microservices: Evolve your security practices & container security. https://lab.wallarm.com/shift-to-microservices-evolve-your-security-practices-container-security/, 2019.

[149] Y. Wang, H. Kadiyala, and J. Rubin. Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering*, 26(4):63, 2021.

[150] D. Wichers and J. Williams. Owasp top-10 2017. OWASP Foundation, 2017.

[151] P. Wizenty., F. Ponce., F. Rademacher., J. Soldani., H. Astudillo., A. Brogi., and S. Sachweh. Towards resolving security smells in microservices, model-driven. In *Proceedings of the 18th International Conference on Software Technologies - ICSOFT*, pages 15–26. INSTICC, SciTePress, 2023.

[152] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Amsterdam, Netherlands, 2000.

[153] E. Wolff. *Microservices: Flexible Software Architecture*. O'Reilly, Newton, MA, United States, 1st edition, 2016.

[154] T. Yaryginam and A. Bagge. Overcoming security challenges in microservice architectures. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 11–20, United States, 2018. IEEE Computer Society.

[155] D. Yu, Y. Jin, Y. Zhang, and X. Zheng. A survey on security issues in services communication of microservices-enabled fog applications. *Concurrency and Computation: Practice and Experience*, 31(22):e4436, 2019.

[156] T. Ziade. *Python Microservices Development*. Packt Publishing, Birmingham, United Kingdom, 1st edition, 2017.

[157] O. Zimmermann. Microservices tenets. *Computer Science - Research and Development*, 32(3):301–310, Jul 2017.