A Collaborative and Decentralised Approach for Auditing of Distributed Workflows

Submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

By

ANTONIO NEHME

Faculty of Computing, Engineering and the Built Environment

January, 2020

DECLARATION

I hereby declare that the thesis entitled "A Collaborative and Decentralised Approach for Auditing of Distributed Workflows" submitted by me, for the award of the degree of *Doctor of Philosophy* to Birmingham City University is a record of bonafide work carried out by me under the supervision of Prof. Ali E. Abdallah, Dr. Khaled Mahbub, and Dr. Vitor Jesus.

I further declare that the work reported in this thesis has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place: Birmingham, United Kingdom

Date: **Signature of the Candidate**

CERTIFICATE

This is to certify that the thesis entitled "A Collaborative and Decentralised Approach for Auditing of Distributed Workflows" submitted by Mr. ANTONIO NEHME, Center for Cyber Security and Secure Networks, Birmingham City University, Birmingham for the award of the degree of *Doctor of Philosophy*, is a record of bonafide work carried out by him under my supervision, as per Birmingham City University's code of academic and research ethics.

The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university. The thesis fulfills the requirements and regulations of the University and in my opinion meets the necessary standards for submission.

Place: Birmingham, United Kingdom

Date: **Signature of the Guide**

Abstract

The world is on a continuous move towards collaborations between organisations. This practice is common in many domains including government, health, supply chain and engineering. Collaborations are enabled by inter-operable applications through which each organisation makes a contribution in a workflow. Depending on the application domain, a number of assurances are needed for the security and robustness of the workflow while non-repudiation is a common requirement. The contribution of this work revolves around assuring non-repudiation in distributed collaborations without relying on a single point of trust. In comparison with common practices, this thesis proposes an approach for auditing that does not trust a single entity to protect the integrity or availability of audit trails, or to generate or verify the correctness of audit records. To achieve this aim, security of applications within each organisation including their resilience and defence against intrusion needs to be covered as a pre-requisite of the security of the collaboration; availability and scalability of each application are also essential to fulfil a collaboration. Microservices architectural paradigm enables building scalable and maintainable applications and it is expected to become the default paradigm in the next five years. Microservices applications, however, are challenging to secure and the literature lacks a comprehensive reference that covers the specifications of this paradigm.

This research starts by targeting security of microservices-based applications and carries on to cover non-repudiation in distributed workflow collaborations. For the first part, a security reference architecture covering microservices specifications throughout the application development life cycle is presented, as well as an access control framework to limit vulnerabilities caused by following common old practices. As for the second part, a robust, confidentiality friendly and application-agnostic approach is offered to create verifiable audit trails that cover any degree of details in workflow collaborations and give auditing capabilities to any threshold of participants. This thesis presents an implementation of the proposed approach for auditing using an untrusted centralised server, and another using blockchain.

Keywords: *Microservices*, *Auditing*, *Workflows*, *Trust*, *Confidentiality*, *Blockchain*.

Acknowledgement

First and foremost, I wish to express my deep sense of gratitude to my supervisors, **Prof. Ali E. Abdallah** for his guidance, inspiring and insightful conversations and for paving my way into the research world, **Dr. Khaled Mahbub** for his academic and technical guidance, attention to details and continuous support over the course of this journey and **Dr. Vitor Jesus** for his deep knowledge, academic support, insightful arguments and immense help to focus the scope of my research. I wish to extend my gratitude to **Dr. Maya Chehab** for her guidance at the initial stage of my research.

I am also grateful to the centre of Cyber Security at Birmingham City University, namely to **Prof. Mark Josephs** and **Prof. Paul Kearney** who have always been helpful and supportive. I also acknowledge the role of the Cyber Security Research Group that enabled me to have an insight on cutting-edge research conducted by my colleagues, and of my fellow PhD students Thomas Wagner, Diana Haidar, Fatima Abdallah, Carolina Boye, Moojan Pordelkhaki, AlaaAllah ElSabaa and Akinola Siyanbola who were always ready for any technical, academic or moral support. I would also thank Xiaohu Zhou whose technical assistance was very helpful in this research.

A special acknowledgement goes also to the Doctoral Research College, namely to Ms. Sue Witton and Ms. Bernadette Allen for their administrative assistance and for being always helpful.

I would also like to thank the PhD committee who will invest the time and effort required to assess my research and evaluate my defence of this thesis.

My work in this thesis would not have been achieved without the help of Samia Nehme, Peter, Elena and Christopher Coveney, my parents Akram Nehme and Liliane Sammour and siblings Karen, Alex and Andrew who have given me the encouragement, love and support I needed to make it through this challenging journey. A special thanks goes to my friends, Camilio Harb, Mary Fadel and Iman Matar for every action they took throughout the years that helped me get to where I am today.

Place: Birmingham, United Kingdom

Date: 02/01/2020 **Antonio Nehme**

Table of contents

List of figures

List of terms and abbreviations

ACS Access Control Server 33–38, 42

API Application Programming Interface 1, 9, 10, 17, 20, 71, 89, 100

CGW Consumer Gateway 33, 36, 42, 92

CMS Consumer Microservice 30, 33, 36, 37

GW Micro-Gateway 30, 33–35

IDS Intrusion Detection System 20, 24

IETF Internet Engineering Task Force 19

JOSE Javascript Object Signing and Encryption 19

JSON JavaScript Object Notation 19

JWT JSON Web Token 19, 20

MSA Microservices-Based Application 1, 2, 18, 19, 25, 95, 97, 100, 101

OAuth Open Authorization 2, 7, 19, 20, 25, 27–29, 31–39, 41, 42, 95, 98

PAP Policy Administration Point 33

PDP Policy Decision Point 33, 34, 36

PEP Policy Enforcement Point 33, 34, 36, 37

PKI Public Key Infrastructure 18, 20, 58

PVSS Publicly Verifiable Secret Sharing 58, 59

REST Representational State Transfer 19

RGW Resource Gateway 33, 35–38, 40, 42, 92

RMS Resource Microservice 30, 33–37, 40, 41

SOA Service-Oriented Architecture 1, 3–5, 7, 10, 11

SOAP Simple Object Access Protocol 19

XACML eXtensible Access Control Markup Language 7, 19, 26, 27, 33, 34, 36, 37, 39, 43, 98

XML Extensible Markup Language 19

List of publications

1. Nehme, A., Jesus, V., Mahbub, K. and Abdallah, A., 2019. Securing Microservices. IT Professional, 21(1), pp. 42-49. https://doi.org/10.1109/MITP.2018.2876987

2. Nehme, A., Jesus, V., Mahbub, K. and Abdallah, A., 2018, November. Fine-Grained Access Control for Microservices. In International Symposium on Foundations and Practice of Security (pp. 285-300). Springer, Cham. https://doi.org/10.1007/978-3-030-18419-3_19

3. Nehme, A., Jesus, V., Mahbub, K. and Abdallah, A., 2019, August. Decentralised and Collaborative Auditing of Workflows. In International Conference on Trust and Privacy in Digital Business (pp. 129-144). Springer, Cham. https://doi.org/10.1007/978-3-030-27813-7_9

4. Zhou, X., Nehme, A., Jesus, V., Wang, Y., Josephs, M. and Mahbub, K., 2019, October. Towards Blockchain-Based Auditing of Data Exchanges. In International Conference on Smart Blockchain (pp. 43-52). Springer, Cham. https://doi.org/10.1007/978-3-030-34083-4_5

CHAPTER 1

Introduction

1.1 Introduction

Electronic systems dominate every aspect of daily lives and are the backbone of governments, health, industrial, financial as well as educational systems (Weber et al. 2016, Werner and Gehrke 2015, Odat 2012, Twining et al. 2013, Lightstep 2018). To keep up with consumer expectations, continuous improvement and delivery of existent systems are main requirements especially in large organisations, thus the adoption of agile as a rapid software development methodologies and continuous integration and delivery has evolved as default development practices (Solinski and Petersen 2016, NGINX 2015). Maintainability, defined as the ability to build on top of existent code, is another essential requirement to build systems in a cost and time-effective way (Dragoni et al. 2017). Other required characteristics for software to be considered as reliable are security, availability, robustness and scalability of the software.

Microservices architectural paradigm is an enabler for manageability, technology diversity and rapid development of applications; it also enables scalability in a cost-effective way (Dragoni et al. 2017). A Microservices-Based Application (MSA) does not reflect a new architectural style when thinking of the Service-Oriented Architecture (SOA), a paradigm used to integrate interoperable electronic services belonging to the same or to different organisations (Zimmermann 2016). It is rather SOA implemented following current trends and technologies such as automation of infrastructure operations and the adoption of containers, and used as a replacement of the older style, the monoliths, to develop applications for a single organisation (Zimmermann 2016, Fowler 2014, Thönes 2015). This paradigm, however, is not straightforward to adopt and to secure (Dragoni et al. 2017); recent surveys reveal architectural flaws in existent implementations of microservices as well as exploited vulnerabilities leading to failure of systems in industries leading this paradigm (Neri et al. 2019, Sun et al. 2015). On the other hand, organisations need to collaborate with each other in a wide range of domains including health, supply chain and engineering, banking, e-government (de Vrieze and Xu 2018). These collaborations, enabled through Application Programming Interfaces (APIs) and the Service-Oriented Architecture, need to be auditable following a robust

approach to enable assigning accountability of wrong-doing with a high level of certainty in case of a dispute between organisations (Zawoad et al. 2013, 2016, Weber et al. 2016).

Similarly, in any business context, the potential damage caused by a compromised element is relative to the trust level that this element has. This work targets vulnerabilities caused by trust in microservices-based applications, and in the auditing of the collaborations connecting multiple organisations through these applications.

1.2 Thesis Motivation and Rationale

Microservices is a promising paradigm that, in addition to the scalability and maintainability benefits, enables the integration of smart devices (Jarwar et al. 2017). A survey conducted by NGINX in 2015 and combining 1800 IT professionals reveals that 44% of participating companies are already using microservices for development and in production, and 24% are investigating the adoption of microservices (NGINX 2015). Another survey in 2018 including 350 senior developers from a diversity of industries shows that 86% of the participants expect microservices to become the default development paradigm within the next five years (Lightstep 2018). In a nutshell, microservices architectural paradigm is an approach to develop a single application with a number of small and independently deployable services, developed with different technologies, and communicating through lightweight mechanisms (Fowler and Lewis 2014). The term 'microservices' dates back to 2011, but the first detailed introduction of the paradigm was presented in a blog by Lewis and Fowler in 2014 (Fowler and Lewis 2014, Zimmermann 2016, Yarygina and Bagge 2018). MSA is still maturing, and the literature is poor in architectural and security guidance (Zimmermann 2016, Dragoni et al. 2017, Neri et al. 2019, Di Francesco et al. 2017). In terms of sectors, Microservices are being introduced in all types of applications, including health, government, and critical applications (Fetzer 2016, Savchenko et al. 2015). The design principles of this paradigm increase the attack surface of an application, and although security challenges and practices are discussed in existent research, a holistic approach to ensure the security of microservices-based applications was not available to guide software practitioners at the initial stage of this research (Nehme et al. 2019*b*, Yarygina and Bagge 2018). In addition to that, open security standards for authentication and access control, designed originally for monolithic applications, do not take into account the specifications of microservices. As a result, the common integration of Open Authorization (OAuth), the currently most popular open standard for access delegation, with microservices-based applications leaves these applications vulnerable to token manipulation and privilege escalation attacks; these attacks can lead to data exfiltration and to incorrect behaviour of microservices-based applications (Nehme et al. 2018). A large

scale study on applications in real-world settings using OAuth 2 revealed that vulnerabilities in these applications enabled access tokens to be stolen in 91% of the systems that were evaluated (Sun and Beznosov 2012); although one can expect a decrease of these worrying numbers when best security practices are followed, microservices-based applications require special attention for the larger attack surface that they have in comparison to the monolithic applications considered in their study.

Moving to collaborations between organisations, enabled through SOA, transactions travel through a number of administrative and security domains (de Vrieze and Xu 2018). Common approaches for auditing of these collaborations rely on a trusted party to generate or store audit trails; collusion with this trusted party makes tampering with digital evidence and breaching the confidentiality of workflow transactions possible. Tackling collusion-related threats to tamper with evidence is work in progress, and there is room for improvement in the common audit mechanisms to construct reliable audit trails. Examples of application domains involving the collaboration of multiple organisations include banking, logistics, supply chain, e-government (Yao et al. 2010, Lim et al. 2012, Weber et al. 2016, Wouters et al. 2008, Kieseberg et al. 2016). A number of studies shed light on the importance of audit trails in systems involving multiple organisations. Werner et al. (Werner and Gehrke 2015) highlight the importance of reliable audit trails in the financial accounting sector. Kuntze et al. (Kuntze and Rudolph 2011) stress on the importance of data authenticity, integrity and privacy for evidence stored in forensic databases; they state that maintaining the confidentiality of parties involved in a chain of evidence is challenging. In the government domain, service processing and collaborative decision making is done through data exchanged between departments when an e-government system is in place (Hartmann and Steup 2015, Freudenthal and Willemson 2017, Thompson et al. 2015). Auditability of transactions to ensure accountability and non-repudiation is a core aspect of security (Hartmann and Steup 2015). A case study in Australia discusses the inadequacy of audit systems in Western Australia's Police and Health departments to assign accountability for operations on government records (Thompson et al. 2015). Existent approaches for audit either trust an entity to record evidence of cross-domain transactions, or rely on each domain to record what it sees of a transaction (Rudolph et al. 2009, Lim et al. 2012, Vahi et al. 2013, Wouters et al. 2008). Blockchain has been promoted as a trustless architecture that can be used for auditing; this architecture, however, has limitations including cost and scalability (Tian 2017). There is room for improvement in common practices for auditing where collusion with an entity trusted to record or store evidence can jeopardise the confidentiality, integrity and availability of records. The highlighted research gaps, in addition to the engagement with the digital transformation project for the Republic of Lebanon at its initial stages[1], are the motivations of this research.

[1] `https://digitaltransformation.gov.lb`

1.3 A Layered Approach for Security

This research follows a layered approach, inspired from Yarygina and Bagge (Yarygina and Bagge 2018), to target the security of distributed applications. This approach considers the security of small components confined within a single administrative domain to be the pre-requisite of secure interactions within and outside a domain. This thesis focuses on microservices-based architectural paradigm as an approach to develop applications. Looking at Figure 1.1, which shows the components of a distributed application in perspective, microservices interacting within a single domain form a microservices-based application, and the latter (i.e. microservices-based application) connects with other applications, potentially in different domains, to form a Service-Oriented Architecture (Yarygina and Bagge 2018). A number of studies including the first systematic review comparing microservices-based applications to SOA concluded that the difference between the two is not in the architectural style, but in the development and deployment paradigms enabled by recent technologies (Zimmermann 2016, Newman 2015, Neri et al. 2019). Hence, one can expect many of SOA's security approaches to be applicable to microservices-based applications, and to have specific security concerns for microservices-based applications caused by the new practices and technologies (Yarygina and Bagge 2018).

From an abstract point of view, the inner layer of distributed applications contains the application components (microservices) in each domain. Each microservice encapsulates a small functionality and is developed, deployed and maintained independently from other microservices. To perform a larger functionality, microservices communicate and trust instructions and data sent from each other within a single domain; this means that an erroneous input from a microservice, caused by a compromise of this component, propagates across a domain jeopardising the bigger function (Dragoni et al. 2017). Each microservice is a potential point of exploit; therefore, security of microservices needs to be carefully monitored throughout their life cycle starting from securely bootstrapping components to their safe decommissioning.

The middle layer of Figure 1.1 represents microservices coordination to form a microservices-based application. Microservices trust each other, and often communicate over an insecure network within their domain (Otterstad and Yarygina 2017, Dragoni et al. 2017); even when the security of the microservices is assumed, the trust model in MSA and nature of interactions between microservices is an enabler for attacks on their applications. Due to relatively recent adoption and endorsement of microservices architectural paradigm, comprehensive security guidance was not available in the literature at the initial stage of this research (Yarygina and Bagge 2018, Nehme et al. 2019*b*). Open Authorization 2, an open security standard designed for access delegation of traditional architectural styles, has not yet been modified to consider microservices speci-

fications. Common adoption of some of these standards causes vulnerabilities leading to potential data exfiltration and faulty behaviour of applications (Nehme et al. 2019*a*). Similarly to the effect of a compromised microservice on an application, a compromise in a microservices-based application can spread to and affect applications in other domains of a Service-Oriented Architecture (Nehme et al. 2018).

Moving to the external layer of the figure, a Service-Oriented Architecture, considered as an aggregation of applications from different domains in the context of this thesis, relies on the two aforementioned layers. However, assuming the robustness of applications in different security domains does not necessarily imply the benevolence of the administrative domains maintaining these services. Auditing is a fundamental element in the security of SOA (Meier et al. 2009); recording evidence used for auditing must be done following a robust and trustworthy approach to make its data reliable. While trusting an audit system within a single domain might be a justifiable and understandable practice, the question remains who to trust with the generation and storage of evidence covering interactions spreading across multiple domains.

With this in mind, the security approach followed in this research starts by focusing on the security of microservices and microservices-based applications in Chapter 2; vulnerabilities related to microservices access control that affect their internal, as well as external domains that they interact with are then targeted in Chapter 3. As for the security of SOA, the contribution of this research is developing a robust auditing approach covering the interactions and data exchange between participants in different domains.

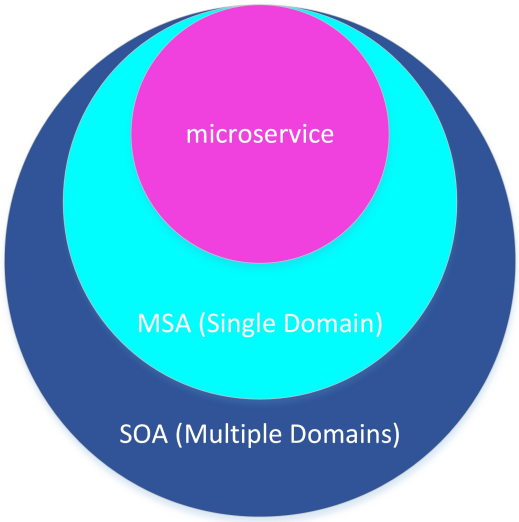

Fig. 1.1 Distributed Microservices-Based Applications in Perspective

1.4 Research Aims and Objectives

This thesis aims are to contribute to the security of MSA applications with a focus on access control, and to enable assigning accountability for actions and decisions made in workflows-based collaborationsin order to solve disputes between participants without relying on or trusting a single party.

Fulfilling the aim of this research requires meeting the objectives listed below:

- Objective 1: identifying existent security practices that are applicable to microservices-based applications.
- Objective 2: identifying weaknesses in practices for access control in MSA and designing an approach, with a proof of concept, to overcome these limitations.
- Objective 3: designing an approach to assign accountability for contributions in workflow-based collaborations between organisations without relying on a single point of trust to generate or store digital evidence.
- Objective 4: offering an implementation for the proposed approach for auditing with an an untrusted centralised server and another implementation with a blockchain.
- Objective 5: analysing and evaluating the two implementations with a focus on security and performance.

1.5 Thesis Structure

This work contributes to the security of microservices and their compositions within and across different administrative domains. In terms of structure, this thesis targets the security of the inner layers of Figure 1.1 in Chapters 2 and 3, and moves progressively to the outer layer in Chapters 4 and 5. Chapter 6 concludes this thesis by reflecting on the big picture, highlighting the limitations of the proposed approaches and shedding light on future research directions. A research paper covering each of the chapters in the body of this thesis is published; the list of publications, following the same order of the chapters is presented at the beginning of this thesis. This section presents an overview on each of the main chapters.

1.5.1 Chapter 2: A Security Reference Architecture for Microservices-Based Applications

This chapter gives an overview of microservices, their interactions to build microservices-based applications, as well as the technologies enabling the shift to this architectural paradigm. Chapter 2 gives a comprehensive but not exhaustive view on the security of microservices; it sheds light on security practices covering different dimensions throughout the development life cycle and presents a security reference architecture that covers fundamental security practices for microservices-based applications. While this research does not claim the completeness of the security guidelines for microservices, the presented framework helps to build security by design for microservices-based applications. The aim is to raise awareness among software practitioners and security architects, and point them towards simple practices, since the design phase and throughout the application life cycle, that help mitigate most common attacks. At the time of publishing this chapter, this was believed to be the first reference architecture giving a comprehensive view of microservices challenges and approaches to mitigate them.

1.5.2 Chapter 3: Fine-grained Access Control Approach for Microservices-Based Applications

Due to the flexibility, scalability, and agility of development of microservices, the adoption of a security solution requires it to be easily adaptable to the application context and requirements and reusable for different microservices. An approach that targets key security challenges of microservices-based applications access control is proposed. The access control model in Chapter 3 relies on a coordination of security components, and offers a fine-grained access control in order to minimise the risks of token theft, session manipulation, and a malicious insider; it also renders the system resilient against a privilege escalation attack known as the confused deputy problem. This approach is based on a combination of OAuth 2 and eXtensible Access Control Markup Language (XACML) open standards, and achieved through reusable security components integrated with microservices.

1.5.3 Chapter 4: Auditing of Distributed Workflow Collaborations (AuDiC)

Workflows, possible through SOA, involve actions and decision making at the level of each participant. Trusted generation, collection and storage of evidence is fundamental for these systems to assert accountability in case of disputes. Ensuring the security of audit systems requires reliable protection of evidence in order to cope with its confidentiality, its integrity at generation and storage phases, as well as its availability. Collusion with an audit authority is a threat that can affect all these security aspects, and there

is room for improvement in existent approaches that target this problem. This thesis presents AuDiC: a robust, collaborative, trustless and confidentiality friendly approach for workflow auditing, which targets security challenges of collusion-related threats and enables recording evidence to any degree of details. This approach is application agnostic, relies on participants verifying reported audit data of each other, and introduces a secure mechanism to share encrypted audit trails with participants while protecting their confidentiality. The approach presented in Chapter 4 offers auditing capabilities to any K out of N participants in a collaboration, and the adequacy of the approach is discussed to produce reliable evidence despite possible collusion to destroy, tamper with, or hide evidence.

1.5.4 Chapter 5: Adopting Blockchain With AuDiC

The initial implementation of AuDiC uses an untrusted centralised server only to display audit records to all participants. The availability of this server is crucial for the functionality of the scheme; this dependency makes the implementation prone to a single point of failure. Due to its distributed nature and its common use for audit, using blockchain is a natural extension to AuDiC. Chapter 5 considers the same approach with a blockchain-based implementation of the audit server. The same security goals are achieved with the first implementation were reached by making use of the advantages of blockchain and finding a way around its limitations.

CHAPTER 2

A Reference Architecture for Microservices-Based Applications

Microservices have drawn significant interest in recent years and are now successfully finding their way into different areas, from enterprise IT and the Internet of Things to critical applications. Although microservices enable building scalable, resource efficient and maintainable applications, the specifications of their architectural paradigm require special considerations for the security of microservices-based applications compared to older paradigms.

This chapter introduces microservices and discusses how microservices-based applications can be secured at different levels and stages considering a common software development lifecycle. The findings are represented in a security reference architecture for microservices-based applications.

2.1 Introduction

Structuring web applications has substantially changed throughout the period following the invention of Web three decades ago (Lu et al. 2017). This is based on the demand of the market during periods of time (Salah et al. 2016). Client-server is one of the earliest architectures of web applications. It is based on two programs: a client program, which runs the user interface, requesting resources from a server which usually hosts a database (Salah et al. 2016). The server is normally placed behind a firewall in this architecture which serves as a barrier against potential unsafe packets and unauthorized access to the system.

After that, and due to the urge of information exchange and collaboration between organisations, web applications started following a decentralised architecture in which a client makes use of more than one resource. This is achieved with web services which are applications that offer their services in a machine-readable way through Application Programming Interfaces (APIs)(Maleshkova 2015). This evolution was facilitated by the enhancement of remote procedure calls and inter-process communication (Slater 2015, Salah et al. 2016). APIs lead to a radical change in software development practices enabling reuse of web services; in 2012, Twitter traffic analysis revealed that 75% of its traffic goes through its API (Maleshkova 2015). This evolution is known as the

Service-Oriented Architecture (SOA) which intends to develop applications through the collaboration of multiple connected units, the web services (Yu et al. 2016). Each unit is autonomous and loosely coupled from the other (Slater 2015, Salah et al. 2016, Yu et al. 2016). Through APIs, SOA enabled the collaboration of applications developed with different programming languages; these applications are orchestrated to achieve a business function. Considering an online store connected to a payment system as an example. The online store, combining a number of functionalities including authentication, searching for items, adding to basket and rating items, is connected to a point of sale for payments to be made; the latter system also includes different functionalities including credit check, card validation, credit update and email notifications. The online store and point of sale are two connected systems exposing an API each.

However, individual applications in a SOA chain are developed with programming languages that, although enabled breaking the application into modules, produce single executable artefacts that rely on sharing resources on the same machine; these are referred to as monoliths (Dragoni et al. 2017, Rahman and Gao 2015). In the example (i.e. the online store connected to a payment system), the functionalities of each application can be developed as modules, and each application is a single executable artefact. Although split into modules, monolithic applications covering different func-

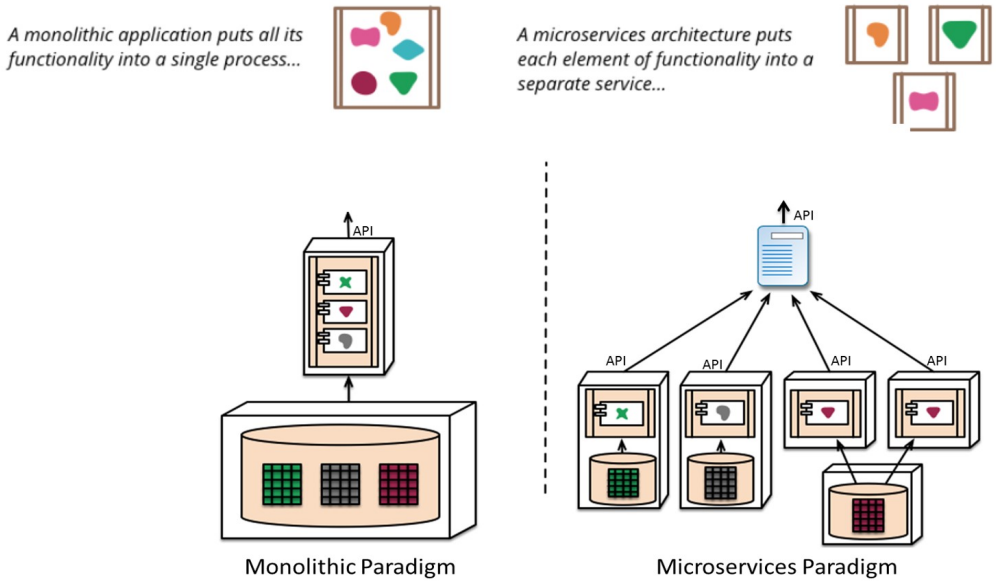

Fig. 2.1 Monolithic vs. Microservices Paradigm. Modified from (Fowler and Lewis 2014)

tionalities suffer from general issues, mostly related to maintainability and scalability. Another paradigm had to be followed to overcome the limitations of monoliths; thus, microservices have emerged.

Being a direct evolution of SOA, but also an application development practice, security for Microservices needs to be progressive where it shares commonalities with

general software development security. In contrast, new approaches are needed to cope with the fundamentally different approach to application development. In this sense, this chapter serves as guidance from architectural design (specific to Microservices) to implementation and maintenance (general to software development). It includes discussing the fundamentals of this paradigm and how it evolved from SOA as well as security practices considering microservices specifications and design principles throughout the application life cycle. This is complemented with a set of references concerning industry initiatives, known security challenges and lessons learned so far, relevant projects and standardization efforts.

2.2 Overview of Microservices

Consider the following scenario from a point of sale (PoS) system as an example: when somebody pays a bill, a transaction is sent along with a balance check; assuming the availability of funds, the person's credit gets updated, and email notifications are sent as a receipt of payment. The described scenario is triggered through an API call to the PoS system, and this system can either be implemented as a monolith or a microservices-based application.

2.2.1 From Monoliths to Microservices-Based Applications

A monolithic application is a reference to a source code that deployed as a single executable artefact (JAVA WAR files) (Dragoni et al. 2017, Sun et al. 2015), or as a set of files mapped to the same directory (Ruby on Rails and Node.js applications) (Slater 2015, Richardson and Smith 2016), and developed using one programming language. These applications can be structured into different modules, being classes, packages, or any internal structure depending on the nature of the application, the programming language or the programming paradigm being used (functional, Object Oriented, etc.) (Zimmermann 2016) and the communication between these components is done through internal calls.

While monolithic implementations of SOA enabled rich applications, their limitations soon became apparent:

- Large monolithic applications are complex and hard to maintain as tracking bugs turned into a difficult task requiring perusals through a long code (Dragoni et al. 2017).
- Maintaining the codebase of a large application introduces time-consuming tasks such as long building and deploying phases since a small change affects the entire application (Dragoni et al. 2017).

– Regular and fast delivery cycles are impractical, and all services in the monolithic application to be disrupted during deployment (Villamizar et al. 2015).

– The entire life of the application is limited to the initial choices of technologies (Dragoni et al. 2017).

– Allocation of resources is inefficient given that scaling one popular service requires resources to be allocated to the entire system (Dragoni et al. 2017, Villamizar et al. 2015).

– Monolithic applications are prone to single points of failure: a load on a single functionality or a bug in one module of the application (a memory leak for example), can bring the entire application down (Villamizar et al. 2015).

A clear solution is to decompose monolithic applications into small and independently deployable services with each as simple as possible, performing one small business function, and running independently from others (Thönes 2015). The resulting components, in larger numbers but individually much simpler, communicate with each other, in order to achieve the same level of functionality as in a large monolithic architecture.

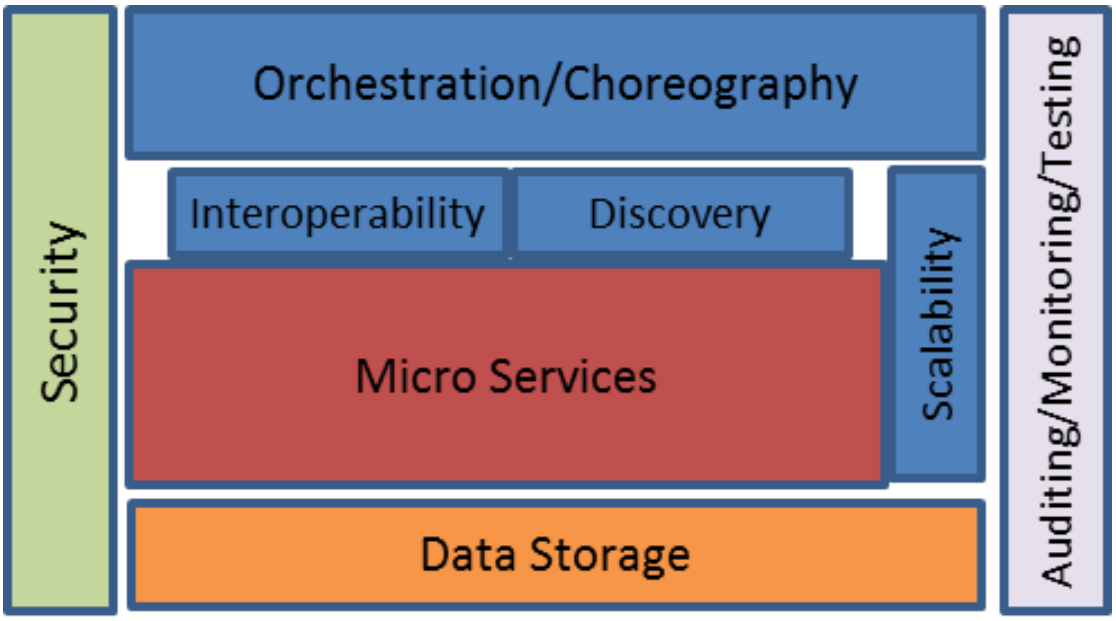

Fig. 2.2 Microservices and Overarching Challenges

Due to their granular nature, microservices entail components reflecting different challenges summarized in Figure 2.2. Microservices need to scale while remaining discoverable and interoperable. Each microservice, reflecting a functionality is independent, and therefore can be scaled individually. The discovery service keeps track of the active instances of the microservice, and is used to direct the traffic within the appli-

cation; Eureka[1] and Ribbon[2] are examples of components used service discovery and interoperability respectively. Similar to any service-oriented approach, microservices support orchestration and choreography; communication in microservices-based applications is tied to interoperability, discovery and scalability of microservices. Microservices are 'volatile', meaning that instances of a single microservice can be deployed or retired on demand (Heinrich et al. 2017); therefore, the data storage is normally separate from a microservice and each service has its database and adopts the database technology which best fits its functionality (Neri et al. 2019). Finally, the entire application needs to be secured, tested and monitored.

2.2.2 Microservices Principles and Enablers

Microservices architectural style is considered an evolution of the traditional monolithic implementation of a single application. It emphasises on dividing systems into small services (the microservices) that perform cohesive business functions (Alshuqayran et al. 2016). Cohesive in this context means implementing functionalities only related to the concern of the business function that a microservice implements. Microservices have to be loosely coupled, meaning that each service should have the ability to be deployed on its own (Dragoni et al. 2017). They should also have a bounded context so that a service should function without knowing anything about other services (Dragoni et al. 2017, Fowler and Lewis 2014). Each should also be autonomous and independently deployable (Heinrich et al. 2017, Fowler and Lewis 2014).

With this approach, complex systems are developed by joining independent microservices which communicate with each other via lightweight mechanisms over a network (Alshuqayran et al. 2016); This implies, for example, favouring REST interfaces over the complexity and heavy processing weight of SOAP.

2.2.3 Selective Scaling and Fast Delivery

Each of these services typically has its own management, programming language and database, which is often referred to as polyglot persistence. This ensures loose coupling between microservices and allows each service to use the database technology that best suits its needs (Namiot and Sneps-Sneppe 2014). Also, this allows each service to have a fit for purpose hosting and programming environment: for example, an image processing component written in C++ can be allocated a to high processing power while lower specs are allocated to another service performing logical or mathematical operations and written in Python. A further advantage is that this allows factorising the workload among different services which scale independently on demand (Sill 2016, Neri

[1] `https://github.com/Netflix/eureka`
[2] `https://github.com/Netflix/ribbon`

et al. 2019); this enables efficient scaling, for example, in a cloud environment (Thönes 2015, Linthicum 2016). Microservices architectural style is also considered an enabler for DevOps and Continuous Deployment which are software development practices aiming for rapid and frequent deployments; this is achieved by having independent development and deployment pipelines for each microservice (Heinrich et al. 2017). This methodology is believed to enhance the system agility and was adopted during the last few years by big software companies like Netflix, Amazon and Linkedin (Villamizar et al. 2015). Applications for Internet-of-Things can exploit this feature to the fullest when compared to monolithic applications (Namiot and Sneps-Sneppe 2014).

2.2.4 Containerisation

Each microservice should be kept as simple as possible which has the further advantage of requiring fewer resources. Given the independent deployability requirement, this currently poses a problem as the lowest-spec server (e.g., on a public cloud provider) is usually too expensive for the resources a microservice needs. Also, setting up virtual machines becomes a complex matter with the diversity of dependencies (Ebert et al. 2016).

Containers are used to mitigate this problem. The concept of a container is not new; Linux containers, LXC, have been supported since 2008 (Souppaya et al. 2017). In a typical setting, many containers, each running a service, run on the same kernel and hardware while being (logically) isolated from each other (Karmel et al. 2016, Neri et al. 2019). While a virtual machine has its own OS stack and depends on hardware-level isolation provided by a hypervisor, containers share the kernel of the host operating systems that they run on; containers are therefore lighter but less isolated than virtual machines (Ciuffoletti 2015). A popular implementation of containers is Docker, and it is supported by many tools, referred to as orchestrators, like Kubernetes, Mesos, and Docker Swarm (Heinrich et al. 2017, Souppaya et al. 2017). Orchestrators monitor containers health and resource consumption, restart and scale containers following their configuration (Souppaya et al. 2017). These Docker containers are relatively easy to clone with the availability of its registry services, DockerHub (Neri et al. 2019).

2.2.5 End-to-End Coordination of Microservices

In order to perform a complete business function, applications require a consistent and correct cooperation and communication between the microservices. Similar concepts in SOA still apply. There are two main mechanisms: orchestration, requiring a central service (the conductor) to send requests and organise the workflow, and choreography, where each service reacts according to events or triggers (Newman 2015).

Orchestration is normally executed at the gateway level, which is a single entry

point to the system which makes it ideal for storing logs and auditing tasks. The orchestrator, referred to as the conductor, is a central service that includes the logic of the application (Butzin et al. 2016, Dragoni et al. 2017); this, potentially, makes microservices tightly coupled to it (Villamizar et al. 2015, Namiot and Sneps-Sneppe 2014). As with choreography, the application logic is collectively known by microservices which reacts to some events or triggers to establish collaboration (Dragoni et al. 2017, Butzin et al. 2016, Newman 2015). An event store can be used with the ability to store and publish events, and events should be divided in categories to which microservices can subscribe. For microservices-based applications, choreography is preferred over orchestration, since orchestration implies a dependency on a central process which contradicts the decoupling principle of microservices (Dragoni et al. 2017, Newman 2015).

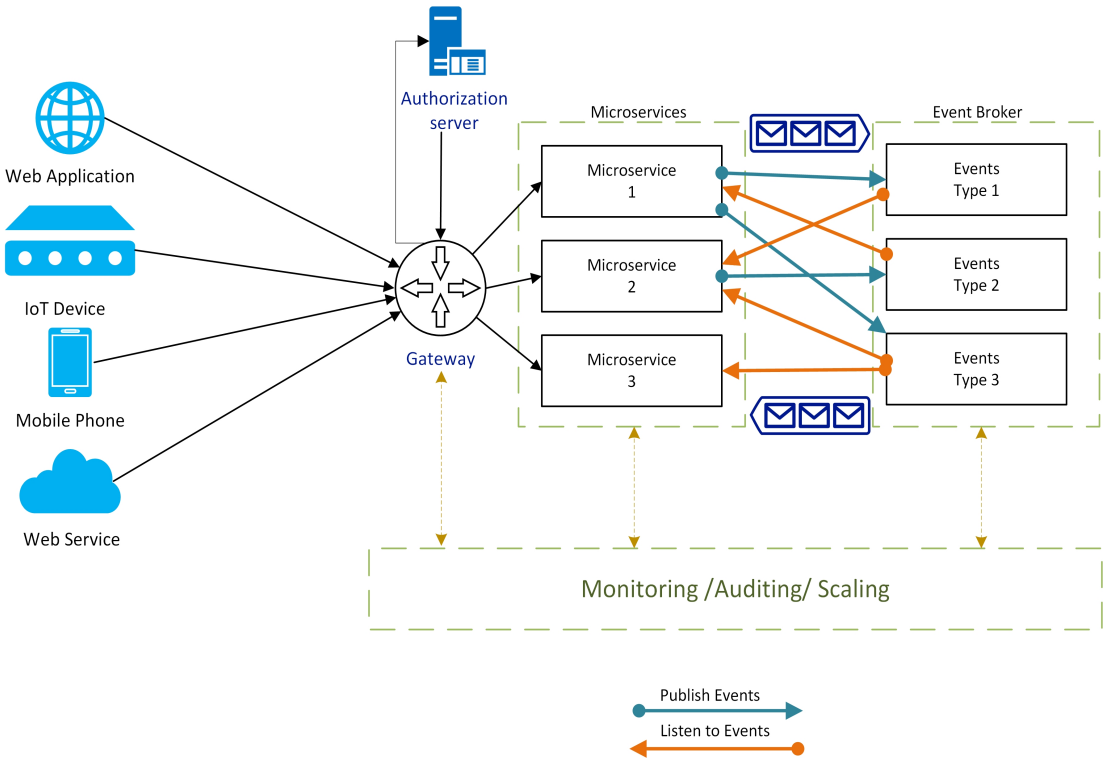

Fig. 2.3 Representative Model of a Microservices-Based Application

2.2.6 A Representative Model of Microservices-Based Applications

In terms of infrastructure, a typical end-to-end model is depicted in Figure 2.3. As shown in the figure, the gateway handles requests from a diversity of external clients. Also, by having a central position, it can return tailored responses according to the client type (Newman 2015). It also handles access management by communicating with an authorization server. The gateway forwards requests to be processed by microservices. To achieve a particular business functionality, microservices communicate

with each other by producing and consuming events using an Event Broker, whose role is to publish and store state-changing events. As shown in the figure, each microservice can publish and subscribe to events of different categories. An event broker is an autonomous application and, due to its role, is essential in auditing and monitoring.

To illustrate with the PoS use-case previously introduced at the beginning of the section, *Transactions*, *Balance*, and *Notifications* could be examples of microservices. Users, machines, and other services authenticate by giving their credentials. A request to the authentication server verifies the authentication material. If valid, a transaction gets sent with an amount to a *Transactions* microservice. This triggers an event for the *Balance* microservice which, given available funding, publishes another event allowing the transaction to complete. The *Transaction* service listens to this event, allows the transaction to complete, and publishes an event of a successful transaction on a channel to which *Balance* and *Notifications* services are subscribed. Balance gets updated and a notification gets sent to the user.

2.3 Microservices Security in the Literature

This section presents an overview of the literature discussing microservices security, as well as attempts to tackle a number of security challenges for microservices. It gives a good idea on the contribution of the literature to the security of microservices-based applications, and clarifies the novelty of the work in this chapter.

Multiple studies in the literature shed light on microservices security challenges (Dragoni et al. 2017, Esposito et al. 2016, Savchenko et al. 2015, Zimmermann 2016), and highlight the need for architectural guidance for microservices-based applications. Souppaya et. al. (Souppaya et al. 2017) discuss security concerns associated with the use of containers, and present guidelines aimed to reduce the attack surface of this virtualisation technology; security of microservices-based architecture and of the infrastructure on which the containers are hosted is out of the scope of their report. Combe et al. (Combe et al. 2016) discuss the specifications of docker ecosystem including the kernel and file system of docker containers, registry and images and suggest practices to harden the security of containers; microservices are only mentioned as a development paradigm suitable for the use of containers, but the security of microservices-based applications was not focused on in their work. Pantanjali et al. (Patanjali et al. 2015) only focus on authentication and authorization in the security analysis of their application; although security is not the main focus of their work, the authors detail the handling of access tokens in their microservice based application providing a dashboard for cost management and analysis with cloud service providers. Jander et al. (Jander et al. 2018) discuss security at the level of the network of microservices-based applications; they focus on protecting the confidentiality of internal calls between microservices and suggest

encrypting services internal communications within the applications and authenticating the requests at the level of the microservices. Sun et al. (Sun et al. 2015) highlight the complexity of monitoring internal traffic for microservices applications hosted in the cloud and propose a virtual network monitoring approach to help detect internal intrusion in a microservices-based application. Their approach, however, relies on Software Defined Network capabilities in a cloud infrastructure.

Lu et al. (Lu et al. 2017) discuss the applicability of microservices architectural patterns and common security practices to IoT systems, and the benefit of adopting this on the security of IoT applications. While some security practices for microservices are mentioned in their paper including access control and the use of secure containers, the robustness of these practices was assumed and microservices security was not the context of their work. Fetzer (Fetzer 2016) investigates the adequacy of microservices isolation through secure containers to build critical systems when the execution is supported by CPU security extensions; they use Intel Software Guard Extensions (SGX) enclave to protect the integrity and confidentiality of microservices executions running inside secure containers. Fetzer focuses on ensuring the correct execution of the code of individual microservices, but does not discuss secure development of microservices or security in the context of microservices-based applications. Sill (Sill 2016) highlights the importance of the design phase and the use of standards for building microservices-based applications. The author discusses standards and considerations for data formats and information exchange, API endpoints and documentation, secure messaging and communication patterns. Although this work points to practices that can contribute to the security of microservices-based applications, the focus of the author is the maintainability and manageability of applications. Otterstad and Yarygina (Otterstad and Yarygina 2017) also discuss the security benefits of the isolation of microservices deployment, and of the diversity of their software and execution environment; as a security-enhancing property, they suggest maximising the diversity of technologies used to for microservices including the programming languages, compilers, containers operating systems and images. They also suggest minimising unnecessary interactions between microservices to decrease their attack surface.

In a chapter of his book covering the concept of microservices, Newman (Newman 2015) gives a brief overview of some security aspects of microservices applications, and presents options, including available standard, approaches and products for authentication and authorization, data security and defence in depth; the author highlights the risk of intrusion on the application network and of implicit trust between microservices. While Newman's book gives a comprehensive view on microservices principles and software development practices, the security discussion in his work is not meant to be comprehensive. This chapter reflects on the security practices in Newman's book, and extends the discussion to cover the security of containers, secure bootstrapping

and handling cryptography material. Yarygina and Bagge (Yarygina and Bagge 2018) present an overview on microservices security threats with mitigation techniques for common threats that these applications are prone to. They suggest the use of a self hosted Public Key Infrastructure (PKI) scheme and the verification of access tokens by microservices; their work appeared after the submission of the contribution of this chapter for publication, and the authors' suggestions have been later considered in this chapter. While (Newman 2015, Yarygina and Bagge 2018) discuss security practices for microservices-based applications, this chapter aims to give a more comprehensive view on the security of these applications; it is an attempt to shed light on security practices covering different security layers throughout the development lifecycle of MSAs. The aim is to raise awareness among software practitioners and security architects, and to point them towards simple practices that help mitigate most common attacks.

2.4 Considerations for Microservices Security

As a new fast-growing application development paradigm, yet still maturing, new challenges are introduced, and security comes at the forefront. This chapter discusses Microservices from a security perspective. Rather than addressing this topic from a specific angle, this chapter tries to lay out a comprehensive approach by discussing all phases of a typical project lifecycle and related Security context: design, development and testing, "business-as-usual" (maintenance, verification, monitoring, etc.), infrastructure and interfaces with external parties.

Security is multidimensional in the sense that it needs to be present at multiple layers of an application and at all stages of its development. For microservices, our security model has four broad dimensions:

- the *internal* and *external interfaces* for service-to-service and inter-domain communications respectively.
- *the application architecture* and the potential need of specific security components or elements, such as instrumentation and detection.
- *the underlying infrastructure*, such as the containers, Operating Systems and the network.
- the microservice *components* themselves, from design to implementation and throughout the application lifecycle.

Before diving into the details of the security model, security standards for web services interactions are introduced while highlighting the applicable and commonly used ones with microservices-based applications. This is followed with an elaboration on each of the security dimensions and an aggregation of the findings in a reference architecture.

2.4.1 Security Standards

Simple Object Access Protocol (SOAP) was the first protocol used by Web Services for Interprocess Communication over HTTP enabling them to exchange Extensible Markup Language (XML)-based messages. Representational State Transfer (REST) is a simpler and lighter approach for point-to-point communications and is suitable for devices with low processing power (Vandikas and Tsiatsis 2016). While SOAP uses XML extensively, REST is independent of the data type including XML and JavaScript Object Notation (JSON) (Gorski et al. 2014*a*).

Extensive research has been done resulting in a number of standards for the security of SOAP Web Services including WS-Security covering encryption and integrity checks of messages, WS-Policy for policy enforcement on end-points, and WS-Trust for trust establishments in different security environments (Tang et al. 2015, Gorski et al. 2014*a*). Security was considered at the early stage of SOAP standardization effort, which resulted in the maturity of the security framework (Gorski et al. 2014*b*). In contrast with SOAP-based web services, RESTful web services were not developed with security in mind, and therefore they are not supported with a fully developed security framework (Gorski et al. 2014*a*, Yarygina 2017); moreover, standards designed for SOAP are not compatible with REST services (Masood and Java 2015). As a result, REST security is heavily reliant on best practices or implementation-oriented recommendations by framework or published by software engineering companies (Gorski et al. 2014*a*). An initiative to create security standards suitable for REST services started with Javascript Object Signing and Encryption (JOSE). This is a working group founded in 2011 within the Internet Engineering Task Force (IETF) with the purpose of creating standards for signing and encrypting JSON data structure (Gorski et al. 2014*a*, Siriwardena 2014). JOSE includes four specifications as proposed standards: JSON web Encryption(JWE), JSON Web Signature (JWS), JSON Web Key (JWK), and JSON Web Algorithm (JWA) (Gorski et al. 2014*a*). JWE (Hildebrand and Jones 2015) describes how a message is encrypted in a JSON, JWS (Bradley et al. 2015) defines how this message is signed, JWK (Jones 2015*b*) refers to the public key used for the signature, and JWA (Jones 2015*a*) describes a set of cryptographic algorithms that can be used for JWE, JWS and JWK (Gorski et al. 2014*a*). JSON Web Token (JWT) (Jones et al. 2015) is a standard to transfer a set of claims between parties in a JSON format. JWT standard uses JOSE specifications and is used to transfer access tokens to RESTful web services (Jones et al. 2015, Siriwardena 2014). These specifications are applicable to MSA given the reliance on REST for microservices communications (Vandikas and Tsiatsis 2016).

Also, a number of standards can be used for REST web services including OAuth 2 for access delegation, and XACML for access control (Gorski et al. 2014*a*, Siriwardena 2014). Applying these standards to secure RESTful applications is sometimes done by

software developers, and some of these standards are complicated to adopt in a secure manner and create vulnerabilities when they are not (Argyriou et al. 2017, Sun and Beznosov 2012). Common vulnerabilities in microservices applications result from misusing OAuth 2; the confused deputy problem is briefly introduced in the following section and discussed with other vulnerabilities in details in Chapter 3.

2.4.2 Secure and Trusted Services Interactions

Authentication and authorization are essential steps towards securing services. Microservices should only be invoked after requesting authentication and, ideally, authorisation if levels of privileges are available. OAuth (currently in version 2.0) and OpenID Connect are frameworks adopted in typical implementations of microservices that use RESTful APIs (Patanjali et al. 2015). In essence, an access token is issued by an authorization server to a trusted client application. Note that trust is directly relatable to the coordination model. Verifying the access token at the gateway level makes it vulnerable to the Confused Deputy Problem (Newman 2015). This vulnerability is caused by microservices trusting the gateway based on its mere identity (sometimes even an IP address), which makes it open to misuse if compromised. Having access control enabled and fine-grained scopes for the access token checked by microservices prior to responding to a request is a possible mitigation. A self-hosted PKI enabling Mutual Transport Layer Security is another good practice to protect the confidentiality of internal interactions between services from a malicious intruder on the network (Yarygina and Bagge 2018). Note that having a dedicated service acting as an authorization server provides three main benefits: decoupling and isolation in case the system is compromised, helping in the separation of concerns, and acting as a possible auditing point (Patanjali et al. 2015). OpenID Connect is built on top of OAuth 2, and uses JWT for identity tokens (Saito et al. 2016, Patanjali et al. 2015).

2.4.3 Secure Architecture

A common model for microservices uses API Gateways. Being a dedicated element that does not directly participate in the application itself, it can also act as an Intrusion Detection System (IDS). However, IDS for microservices can be challenging as the signatures for the services need to be, typically, customised to the application, depending on the level of traffic inspection. Availability is also a key component of a Security model; it can be achieved by having elements to detect (by querying, for example) services that are down. If a microservice fails, the gateway and the coordination logic should be updated in order to action failover mechanisms.

Moreover, architectural decisions should be carefully thought of to avoid incidents similar to Netflix compromise in 2015, which was due to allowing access to all users

cookies from any subdomain. This allowed an adversary to use Netflix.com services from one compromised subdomain (Sun et al. 2015).

A final mention should be made to key management. Given the large number of services, managing cryptographic material is likely to require using Key Vaults and hardware modules.

2.4.4 Secure Infrastructure

By infrastructure, one means network, servers, devices, specialised elements (such as gateways) and the containers themselves along with Operating Systems.

More than any other paradigm, Microservices depend on fast network messaging given the granularity of each component. Further, inter-service traffic should follow policies derived from the application logic. Finally, note that several applications, each with a large number of services, can coexist together in the same infrastructure and network. Starting at the network level becomes essential, given that microservices brings the potential of increasing the attack surface when compared to a monolithic architecture (Dragoni et al. 2017). Moreover, due to the containerisation trend, special attention is required for the risk of inadvertently facing 0-day vulnerabilities in the components that come from public repositories like Dockerhub (Combe et al. 2016, Yarygina and Bagge 2018). A survey by BayanOps in 2015 revealed that three out of four official Docker images created during that year have relatively easy to exploit vulnerabilities, which can potentially have high impact (Gummaraju et al. 2015). A good approach is to use Docker Security Scanning add-on prior to using images, and verify the image origin. Another survey by Cloud Passage referred to in their webinar (CloudPassage 2017) reveals that more than 90% of Docker images run a root user; this increases the change of escalating privilege to the host domain, and of controlling other microservices consequently. A good practice is to plan security roles within containers by applying the principle of least privilege rather than running root users, and to provide a strong isolation of containers by minimising shared databases and libraries (Yarygina and Bagge 2018).

A further challenge is filtering and monitoring traffic for microservices at a level close to the application, as deep-inspection rules need to be made custom to the application. Common web attacks such as SQL injection, are easily detected by commercial Web-application firewalls; however, these are not particularly suitable for microservices. Containers firewalls attempts do exist, however, with Project Calico[3] being an example. This project can be integrated with Kubernetes, and allows creating policies and firewall rules at the pods level. Pods, holding containers, will scale with the firewall rules.

[3] `https://www.projectcalico.org/micro-service-firewall/`

Overall, securing the network and server infrastructure can use a mix of current technologies to protect up to the container level. Past the container or hypervisor, application security becomes challenging as discussed in the next section.

2.4.5 Securing the Development Lifecycle and Governance

At this stage, a microservices architecture should draw on well-known secure software development best practices as, at the end of the day, this is software development as usual. Automated testing and verification becomes crucial as typically these applications are developed using Agile methodologies and rely on fast iteration cycles. In general, a comprehensive Secure Software Development Lifecycle (S-SDLC) comprises of

- Early risk assessment before design starts (e.g., handling trust, cryptographic material, etc), relevant at the architecture layout phase but also when selecting and assessing tools and frameworks for their own S-SDLC
- Adding accessory functions to the core functionality in order to support security monitoring, auditing, testing and interfaces with external security elements (Fowler 2014)
- Development with security in mind, following each language and framework recommendations and, ideally, third-party code reviews
- Deployment of the application along with security tests and verification tools – that should be continuous and periodic and include vulnerability management
- Secure and safe retirement of components and modules

NIST SP 800-190 (Souppaya et al. 2017) is in draft stage and offers guidance regarding containers. In a nutshell, the takeaway advice consists of

- Always use container-specific OSes, which are hardened to reduce attack surface.
- Adopt vulnerability management tools for containers, in addition to traditional ones for the host environment.
- Execute highly critical microservices in especially hardened containers and monitor them in depth.
- Handle trust by specialised hardware, a root point, which holds container images, cryptographic material, registries, configurations and any critical information.
- Enforce separation of duties (which involves access control) and segregation of traffic and roles between services and applications.

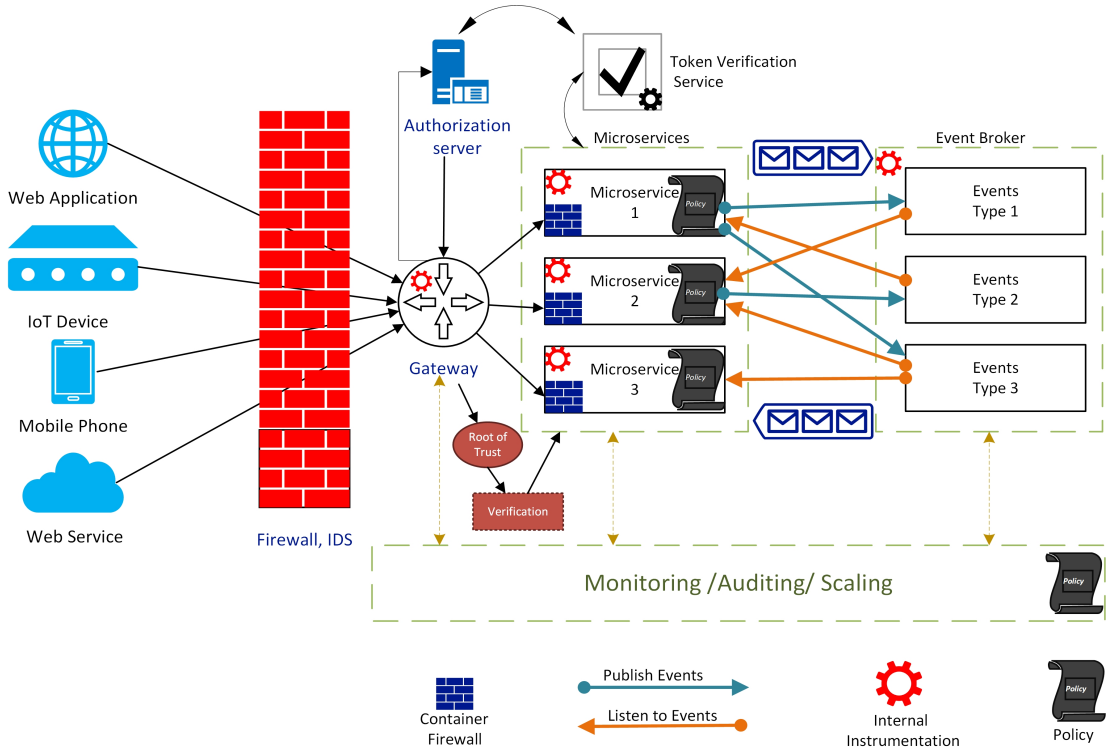

Fig. 2.4 A Security Reference Architecture for Microservices-Based Applications

2.4.6 A Secure Reference Model for Microservices-Based Applications

Given all considerations so far, Figure 2.4 is a modification of the reference model represented in Figure 2.3 in order to embed security by design. The changes from the purely functional architecture of Figure 2.3 reside on three aspects:

- Network elements are inserted in order to apply policies at the network level, from simple traffic rules to deep-packet inspection looking for malicious traffic or drawing intelligence. Policies should also be defined, enforced and verified to segregate inter-service communication and access. The token verification checks the validity of the access token rather than fully trusting the gateway, and policies can define the access rights of the token. This is one mitigation measure against potential vulnerabilities arising from the gateway being a confused deputy if compromised.

- A subsystem of monitoring, testing and verification is added. These components should interface directly the instrumentation components at the microservice level (represented by gears). These agents are, ideally, an integral part of the skeleton of any service and should be enrichened with service-specific metrics. The containers themselves are to be monitored, and one also expects support from the underlying OS and orchestration tools.

– A root of trust supports bootstrapping processes by holding containers images, cryptographic material, and configurations. This is used to retrieve configurations when containers are to be scaled, and to ensure authenticity of software components.

Any request from the outside world must pass through a Firewall and IDS, and container firewalls should inspect requests from the gateway or any potential internal traffic to verify the authenticity of the source. Access tokens should also be verified for authenticity at the microservices level and processed for access control by microservices policy rules. Further, every critical part of the system should be systematically monitored and verified, internal and external traffic should be audited, and container images and configurations must be validated against the trusted hardware.

2.5 Conclusion

Microservices lead to a promising paradigm to develop scalable and maintainable applications that, nevertheless, presents security challenges on its own. Whereas some of the current technologies and practices are directly applicable to microservices-based applications, others need to be developed and adopted in order to reach the needed level of security maturity. This chapter provided an overview on microservices architectural paradigm as well as a discussion of Security for Microservices by looking at different angles and, wherever possible, reusing current practices. The security recommendations are presented in a security reference architecture with the aim of raising awareness of common practices that can help to avoid design flaws and vulnerabilities weakening the security of an application. While many sources in the literature as well as industrial surveys suggest the shift towards microservices and predict it would become the next default paradigm in the near future, more work needs to be done to build systems that are secure by design and to develop specialised elements that support security for microservices (such as IDSes). The large attack surface caused by numerous components and interfaces co-existent on the same machine, the implicit trust very often assumed within the application network, the absence of standards tailored for microservices and the poor academic and industry guidance lead to making this initiative. This study lays the foundations of application security in this thesis; however, compromising a microservice with a zero-day vulnerability and a privileged insider on the network are still a threat, and common approaches for microservices access control exacerbate the impact of a successful compromise. The next chapter proposes an approach for access control that minimises trust between microservices to help contain a compromise.

CHAPTER 3

Fine-grained Access Control for Microservices-Based Applications

Chapter 2 discussed common security practices and standards that are applicable to microservices and proposed a security reference architecture for microservices-based applications. This chapter targets two vulnerabilities, *powerful tokens* theft and the *confused deputy* attack, caused by the common use of OAuth 2: an open standard that was not designed for MSAs.

A combination of existent open standards for access control is relied on and security checks are proposed to detect activities of malicious insiders and help contain a compromise in a microservice from spreading across the application and to other domains. The proposed design includes configurable gateways at the level of each microservice and enables fine-grained access control in MSAs. Details of the approach for microservices access control are discussed and other findings are presented in the rest of the chapter.

3.1 Introduction

As discussed in the previous chapter, microservices-based applications require numerous security considerations. In addition to having a wider attack surface compared to old monolithic applications, microservices very often communicate over an insecure network within their domain (Otterstad and Yarygina 2017). This makes traffic monitoring, interception and replay attacks possible for any malicious node on the same network (Sun et al. 2015). Microservices do introduce coordination complexity which, in turn, creates new security risks. This brings forward trust challenges as, effectively, every microservice is an independent party that, in the extreme case, cannot be trusted (Nehme et al. 2019*b*). After discussing the specifications of the microservices architectural paradigm and the security challenges of microservices-based applications in the previous chapter, two vulnerabilities found in common practices of access control in the literature were identified; in particular, microservices communication paradigm creates new opportunities *confused deputy* attacks and the manipulation and theft of *powerful tokens*. A *confused deputy*, referred to as the 'vulnerability du jour' (Härtig et al. 2017), is a privilege escalation attack in which a service that is trusted by other services is

compromised; this results in the trustees responding to requests from a compromised microservice, without knowing that it is acting on behalf of the attacker (Rajani et al. 2016). *Powerful tokens*, in turn, result from the fact that, typically one valid authorisation token is enough to have access to every microservice in the application since requests pass through a gateway (the orchestrator) that can access all the system services with that access token. These are normally Open Authorization tokens that are created through one OAuth client, and their theft leads to an exposure at the level of every microservice (Sun and Beznosov 2012, GDS 2016); following OAuth protocol, the possession of this token grants access to its bearer for the access rights granted by this token (Sun and Beznosov 2012). The context of this chapter is user-centred services that are multiparty and inter-domain. In particular, scenarios are considered where microservices from multiple domains are acting on behalf of the user by requesting access to personal data or assets; the data exchange process should be transparent to and controlled by the data owners. One example of such systems is a digital government portal: multiple administrations, that are nevertheless independent and segregated, have to coordinate to provide services for citizens, and citizens need to ensure that their data is safe, while being aware of how this data is being used, and what is being processed; on the other hand, each administration is responsible for protecting the citizens' data, and of correctly performing its role. The requirement of fine-grained access control for microservices and giving users control over their data was inspired by discussions with the team leading the digital transformation project in Lebanon. Microservices-based architecture has been agreed on as the development paradigm of the new government services, and the proposal for a fine-grained access control was presented at the Lebanese Digital Transformation Conference in 2018[1]. Gonzalez et al. (González et al. 2016) proposed an approach based on eXtensible Access Control Markup Language (XACML) enabling users to control access to their personal data in an e-government context; their approach, however, is designed for monolothic applications. This chapter presents an approach to target these requirements that applies to microservices-based systems with access to sensitive data in different administrative domains.

The proposed approach for microservices-based applications enables fine-grained access-control, thus mitigating several security challenges specific to microservices while giving the user control over their requests. Beyond globally validating a token at the entry-level (the Gateway interfacing the user or another external application), each service is proposed to have its own local Gateway that validates highly-descriptive and fine-grained tokens. These tokens are centrally generated, short-lived and have a narrow access scope. Additionally, these gateways include security checks that reveal and mitigate potential malicious activities, like data theft from government departments,

[1] `https://digitaltransformation.gov.lb/wp-content/uploads/2018/08/antonio-nehme-session3-video.mp4`

unauthorised requests for actions on behalf of the user, or tampering with the logic of execution of government digital services through a compromised microservice in one department. Furthermore, to enable scalability and reusability, the gateways are configurable and therefore reusable to enable their integration with microservices outside their core functionalities; new gateways would be deployed with the microservices that they protect when needed. In a nutshell, this architecture requires a user to explicitly allow actions from the multiple services engaged and belonging to different parties, while confining permissions of the services with pre-defined policies that all parties agree on. A prototype implementation was done as part of this contribution, and the proposed approach and findings were published in (Nehme et al. 2018).

3.2 Practices for Access Control

Many approaches, found in the literature, rely on powerful tokens strategy, i.e. one access token giving access to all the system's components, for access control. This results from using one OAuth client for a microservices-based application: Pantanjali et al. (Patanjali et al. 2015) present an example of an implementation where powerful tokens are being used, and (Yarygina and Bagge 2018, Gao and Uehara 2017, Geisriegler et al. 2017, Suryotrisongko et al. 2017, Sun and Beznosov 2012, Jander et al. 2018) also point out to using similar approaches in their work. OAuth token theft has been approached in literature. Sun et al. (Sun and Beznosov 2012) conducted a large scale empirical security examination of real world implementations of OAuth 2 access delegation with Single Sign-On (SSO) focusing on web applications vulnerabilities, as well as vulnerabilities in web browsers. On their sample of 96 cases, Sun et al. revealed vulnerabilities enabling token theft and session swapping attacks and suggest guidelines and mitigation techniques for applications and identity providers to reduce these threats. While this study helps to mitigate token theft and session manipulation attacks, their focus is on vulnerabilities at the application layer of monoliths, which does not cover the specifications of microservices-based applications. Azeem et al. (Ahmad et al. 2014) used Identity and OAuth tokens to minimise the possibility of token theft; however, the combination only reduces the chances of a successful attack and does not protect against powerful tokens theft in the service-to-service communication. Approaches based on OAuth 2 and XACML open standards are commonly proposed in the literature to protect web services interfaces (Tang et al. 2015, Yarygina and Bagge 2018, Nehme et al. 2019*b*, González et al. 2016, Gorski et al. 2014*a*). XACML and OAuth 2 are discussed separately in (Ilhan et al. 2015, Samlinson and Usha 2013), and Bojan (Suzic 2016*b*) mentioned the possibility of combining the two standards; however, the combination was not detailed or applied by any of them. Hui et al. (Zhang et al. 2012) based their implementation on this combination; however, their solution targets a specific use case

that is not applicable to microservices. Finally, work on a new OAuth grant type, Token Exchange (Jones et al. 2019), still in progress, tackles similar trust related problem in access control as this paper. It is equally tailored for microservices in which the authorization server is in charge of policy decisions based on the identity of users, calling and called services, predefined action and access rules.

In short, the chapter presents the first attempt for designing a reusable and user-centric Identity and Access Management (IAM) security component for primitive (only implementing functional requirements) microservices that mitigates powerful token theft and the confused deputy problem. The reusability and configurability of the proposed security components renders this approach scalable with microservices and adaptable to their requirements.

3.3 Microservices Access Control: Problem Statement

This section introduced a representative scenario, presents the threat model for the access control of microservices-based applications, gives an overview of the design principles and security requirements that this chapter abides to, and shows the inadequacy of most used approaches for access control and the common vulnerabilities they lead to in microservices-based applications.

The digital government scenario, presented next, is derived from a case study focused on by the digital transformation team in Lebanon in 2018; this sets a basis that reflects common transparency and access control requirements in an e-government scenario and makes a reference to other similar scenarios. The scenario is applying for a passport issued by the Department of State. The applicant needs to be a citizen to be eligible to apply for the passport service. The user logs in to a central portal, and selects the passport service; by logging in, the portal fetches the required information for access control: the citizenship status in this example. Information about this citizen, available in other government departments, are required by the department of state to process the request: the address and marital status of the user are required from the Department of Interior Affairs, as well as an attestation of a clean criminal record from the Department of Justice; these attributes are already agreed on between the departments. The user needs to approve the personal data attributes that will be shared between departments, and an access token will be produced reflecting each consent. Each token only serves to access one specific service of one department. The process of applying for a passport with the interactions described above was the only one identified during the course of this research. The focus of this part of the research is limited to digital government related scenarios due to time restrictions and to the importance of the security and reliability in e-government applications.

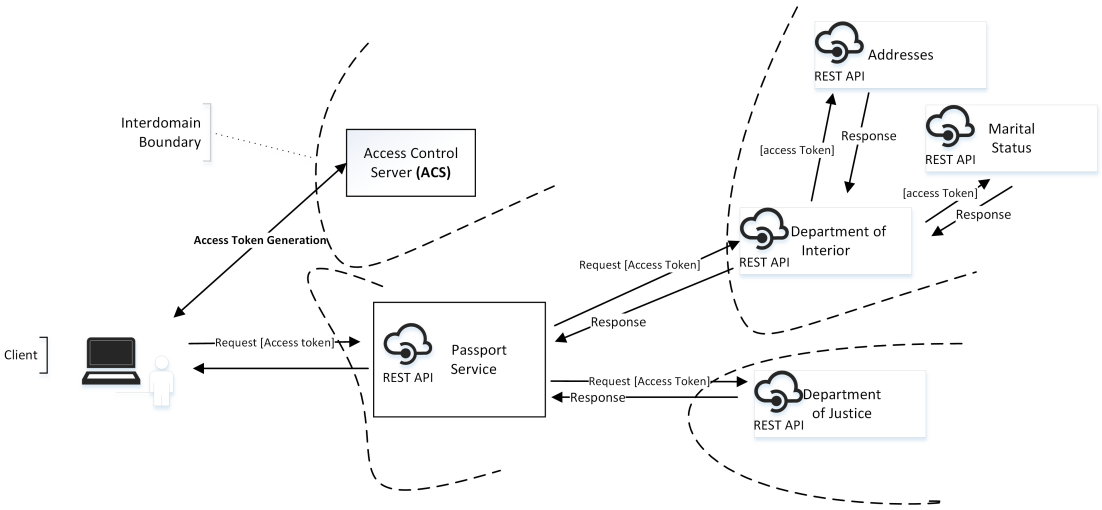

Fig. 3.1 Applying for a Passport

3.3.1 Threat Model for Microservices Access Control

This model assumes that traditional inter-domain security mechanisms, including intrusion detection and prevention systems, firewalls, input validation, mutual TLS authentication and encryption are placed between different security domains. These security mechanisms as well as the authentication and authorisation servers are trusted are not compromised, but the application microservices are not.

These microservices, and the virtual machines (containers) which they run on, can be under the control of an attacker, or even abused by a privileged insider. This gives the adversary the ability to intercept requests and responses, steal and manipulate tokens by replacing a token belonging to a user with another, and send requests from the compromised microservice. A compromised microservice cannot generate a new access token without the user's consent on the list of scopes: generating tokens is only possible through a redirection to the OAuth server following the OAuth protocol. Access Token theft can happen at the level of any compromised microservice, or by an insider monitoring local traffic.

3.3.2 Security Requirements for Microservices Access Control

Considering the scenario detailed in Section 3.3, this chapter aims to fulfil the following requirements deduced from discussions with the Lebanese digital transformation team:

- R1: Access policies are needed to control which services a user can access.
- R2: Every personal data attribute, at each department, needs user consent to be shared with another department.
- R3: Departments only share data following predefined and verifiable agreements with other services.

– R4: An access token should only serve to perform actions on behalf of a user and access the assets of this user exposed by a single service in one department.

Where the corresponding security goals are:

– R1 requires fine-grained access policies, that must relate to the (micro)service itself

– R2 separates control between user and service providers by allowing administrative policies on a per-service basis

– R3 verifies the authenticity of consumers and limits the impact of insiders malicious activities

– R4 protects against *Powerful Token* and *Confused Deputy* attacks.

3.3.3 Decoupling Security from Functional Requirements

A further requirement is to decouple the control of the microservice from the service itself. This was done by designing our architecture using reusable and configurable gateways at the level of each microservice. These components can be added to secure primitive services, and modified to meet different policies. Figure 3.2 shows Resource Microservice (RMS), a primitive microservice exposing assets, protected by a local Micro-Gateway (GW). In order for a request to reach the RMS, security policies enforced by GW have to be met by the requesting service or party- the Consumer Microservice (CMS); note that the consumer microservice should have another gateway to enforce access control policies. The Resource Microservice, which encapsulates only the primitive functionality, is thus released from the verification logic and only manages the assets themselves (such as personal data).

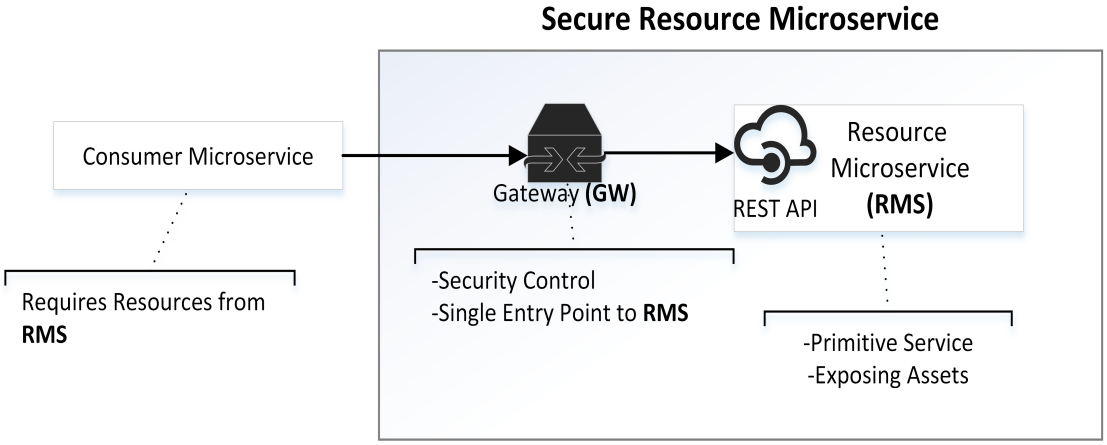

Fig. 3.2 Gateway to Secure Primitive Services

A reusable security component placed around services enables consistency, simplicity, and portability (Linthicum 2016); adaptability and flexibility are essential requirements to follow. For different scenarios, a variety of attributes have to be considered when designing security solutions, and a trade-off has to be made between multiple variables including performance, security tightness, user-friendliness, and ease and flexibility of management.

3.3.4 Inadequacy of Current Practices with Microservices

Open Authorization 2 (OAuth 2) is one of the most commonly used mechanisms with microservices-based applications. The protocol aims to enable access delegation by giving the end user the option of selecting a list of scopes prior to the generation of an access token (Microsoft 2012). Following this protocol, when a relying party (a client) requires access to the data of end-users or the permission to perform actions on their behalf, a user gets redirected to the OAuth 2 authorisation server which, after authenticating the client application, presents a list of predefined scopes for this user to choose from (Microsoft 2012, Sun and Beznosov 2012). After that, an access token gets generated with the scopes and sent back to a predefined end-point of the client application (Microsoft 2012). OAuth 2 access scopes are used to define the token holder's access rights. However, the standard only gives the ability to define static, normally coarse-grained scopes, and does not provide any support for auditing and flexible policy enforcement (Suzic 2016*b*,*a*). OAuth 2 is also used as a layer underlying authentication protocols and Single Sign-On systems (Yarygina and Bagge 2018). OpenID Connect is an example of these protocols and it is commonly used for authentication with MSA (Patanjali et al. 2015); it is an enabler for identity federation by producing an ID token with end-user information, and a practice of the separation of concerns principle. Nevertheless, these approaches are not particularly suitable for MSA due to their large attack surface in such a fine-grained architecture (Dragoni et al. 2017); they normally rely on a single token that is used to access all parts of the system resulting in several problems, *powerful token* theft being the most obvious (Ahmad et al. 2014, GDS 2016, Jander et al. 2018). Following an OAuth 2 based approach with *powerful tokens* for access control, any service having access to a session with a valid token can make requests to other components on behalf of the user (Patanjali et al. 2015). Sun et al. revealed that common vulnerabilities in web applications, including Cross Site Scripting, Cross Site Request Forgery, TLS misconfiguration, enable the theft of access tokens in old style monolithic applications (Sun and Beznosov 2012). The diversity of hosting components (the containers), the large number of exposed APIs, and the busy communication often over an insecure network (Otterstad and Yarygina 2017, Dragoni et al. 2017, Sun et al. 2015) give a larger window of opportunity for token theft. Even

with end-to-end encryption of the communication between microservices suggested by Yarygina and Bagge (Yarygina and Bagge 2018), compromising or having privileged access to any microservice can be enough to steal an access token.

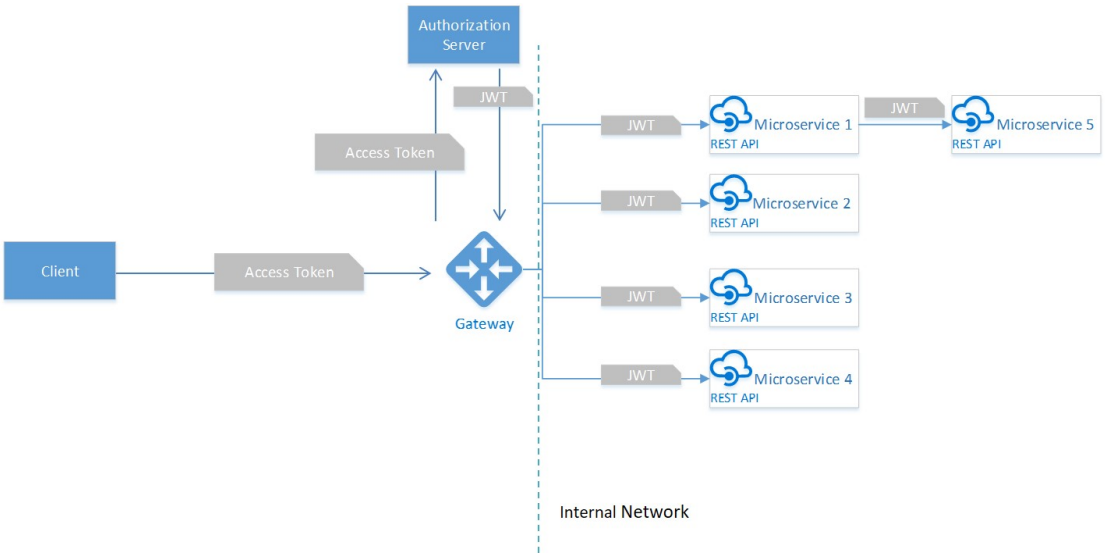

Fig. 3.3 A Representation of the Common Integration of OAuth with Microservices

On the other hand, there is the *confused deputy* problem. Härtig et al. (Härtig et al. 2017) emphasise on the popularity of this vulnerability in microservices-based applications and call for tools to detect it. As explained, a *confused deputy* is a component that has access to sensitive resources, and which can be manipulated by an adversary to have an indirect access to these resources (Rajani et al. 2016). In essence, the *confused deputy* attack arises from trusting a component based on mere identity information such as the component's IP address or an ID token (Newman 2015, Yarygina and Bagge 2018). Figure 3.3 represents the common integration of OAuth 2 and OpenID Connect with MSA: an access token, generated by the user with the authorizarion server, gets sent with the user's request to a gateway; this token, being a reference to information that identifies the requester, gets exchanged at the gateway level with a data structure, the JSON Web Token (JWT), which the gateway and microservice can use (Preuveneers and Joosen 2017). The JWT reflects only identity information with OpenID Connect, and access scopes reflecting the consent of the user for the application to act on his/her behalf when standard OAuth 2 is used; in both cases, it is a single token (Patanjali et al. 2015). In Figure 3.3, the gateway used to call all the microservices becomes a confused deputy if compromised and in the scenario presented in Section 3.3, the passport service is a candidate for a confused deputy. The key point to prevent this is to have the resource services, the Department of Justice and of Interior Affairs microservices in our scenario, verify that the calling microservice is acting truthfully on behalf of the user. This requires, for example, tokens to be individual to each component, and have finer granularity reflecting user's consents on access rules.

3.4 An Approach for Microservices Access Control

Figure 3.4 represents the proposed approach for access control between consumer and resource microservices. This approach is built on a combination of XACML for administrative and OAuth 2 for user-defined policies. XACML is an open standard enabling the definition of complex access control policies based on the identity and role of an end-user, and it is seen as the de-facto standard for policy description (Suzic and Reiter 2016, Fernández et al. 2017). This standard covers the enforcement of access control policies and ensures the decoupling of the application from the definition and enforcement of access control policies (Suzic and Reiter 2016). XACML specifications includes a Policy Administration Point (PAP) to define, store and and administer security policies and a Policy Decision Point (PDP) to evaluate access requests based on the defined policies and to provide a decision based on this policy (Suzic and Reiter 2016). In practice, PAP and PDP both implemented at the level of the authorisation server (Pereira et al. 2017). A Policy Enforcement Point (PEP) is another component that intercepts a request and acts based on the decision of the PDP to grant or deny access to the protected resources (Suzic and Reiter 2016, Pereira et al. 2017). The architecture involves an Access Control Server (ACS) acting as an OAuth 2 and XACML server, consumer microservices (CMS) holding OAuth 2 client credentials and requiring access to resources, Resource Microservices (RMS) hosting and exposing assets, and a Gateway (GW) to secure each microservice.

A request to CMS requires an ID token, generated by ACS when the user logs in, from the authentication session to verify the access rights of the user; to request resources from RMS, it also needs to generate an OAuth 2 token by having the user consent on the access scopes. As shown in Figure 3.4, the Resource Gateways (RGWs) RGW1 and RGW2 are gateways to the resource microservices RMS1 and RMS2; consumer micro-services also have a Consumer Gateway (CGW) each, CGW1 and CGW2, to enforce administrative access control policies. Typically, a central gateway in MSA sits in front of all services and can take different roles ranging from a simple address forwarder to an orchestrator. In the proposed architecture, each microservice has its own gateway that includes security and control functions and is fairly independent of the microservice; this makes it reusable as a configurable component across different microservices. Note that a central gateway is still present as a single entry point to the administrative and security domain to provide conventional network security services such as intrusion detection and prevention, firewalls, input validation, mutual TLS authentication or encryption. The key functionality of a gateway per microservice is becoming a single entry point to each microservice that, while being fairly agnostic to the service itself, is able to validate the authenticity of the incoming requests. Following the specifications in this thesis, these gateways include other mechanisms for security

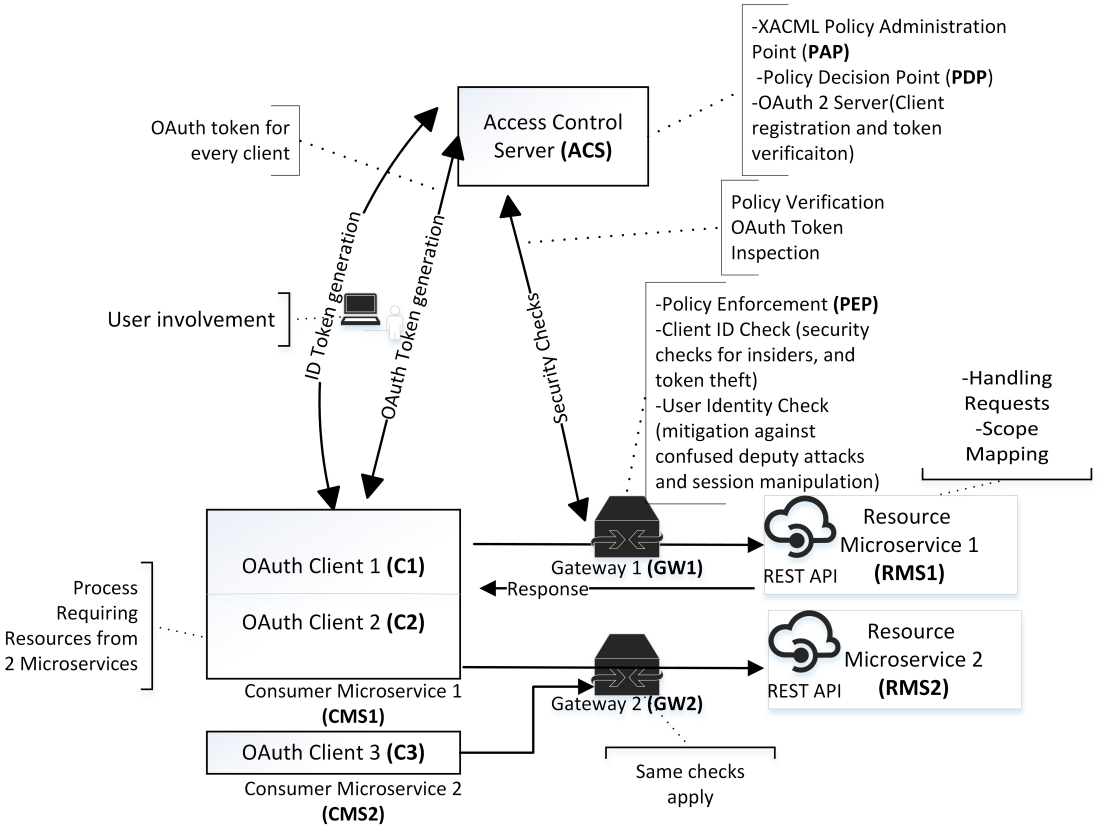

Fig. 3.4 Overview of the Security Architecture: Gateways for Security Enforcement, and an OAuth Client per Consumer-Resource

assurance, policy enforcement, token theft detection, auditing and incident reporting; these measures serve to minimise blind trust between services, and therefore limit the effect of a successful *confused deputy* attack. The details for these checks and the requests flow of requests are explained in the next parts of this section.

3.4.1 A Fine-Grained Access Control

XACML is used to create access control policies that define if a user can interface a particular microservice. Policies are directly enforced by the GWs, each acting as a PEP. The PEP component of the GW checks the user's identifier by inspecting the user's ID token in the authentication session. The ACS is the PDP and determines if this user is authorised to access a microservice endpoint to make a particular request. In the case of resource microservices, the request goes through other security checks discussed in Section 3.4.2. OAuth 2 is used for users to delegate access to part of their protected data, residing at a resource microservice, to a consumer microservice. OAuth 2 produces a token that maps to access scopes; these scopes reflect the actions that the holder of the token is permitted to perform on behalf of the issuer, in this case accessing their data. Being part of the token, scopes are used by RMS to share only the data that the owner has given consent for. The proposed approach includes creating an OAuth client

for every pair of consumer-resource microservices allowing the generation of verifiable tokens with access scopes tailored for the combination. Consider Figure 3.4: OAuth clients C1 and C2 are used to send requests from the consumer microservice CMS1 to two different resource services, RMS1 and RMS2 respectively, exposing user's data. Although this approach gives the flexibility of using one OAuth client for multiple microservices, one OAuth client is recommended per pair of consumer-resource microservices to limit the power of access tokens. Also, a microservice can receive requests from more than one consumer service as shown with RMS2 receiving requests from both CMS1 and CMS2. OAuth client creation is always done at the ACS level following the OAuth 2 common practice. Scopes are defined during creation, and client credentials (a unique identifier and a password) are generated to be used by consumer micro-services for access tokens production.

3.4.2 Proposed Security Checks

For each request from microservices to access resources of another, an OAuth access token needs to be provided, alongside the ID token in the authentication session, by the sender of the request. The ID token is inspected by the GW of every microservice (Consumer or Resource) to verify the eligibility of access of the user, and the OAuth token is to be inspected by the RGW of the resource microservice before the request gets through to the RMS. This gateway sends the access token to an endpoint of the ACS to verify its authenticity and retrieve the information mapped to it. To illustrate with Figure 3.4, if a service CMS1 needs some of the user's personal information from RMS1, CMS1 uses C1's client credentials to produce an OAuth access token following OAuth 2 common practice. The user is required to choose the access scopes and confirm access for the OAuth token to be produced for CMS1. An access request is sent from service CMS1 to RMS1. RMS1, through its gateway RGW1, uses the token inspection endpoint of ACS to verify the authenticity of the access token and to decode it. The token would have a reference to the OAuth client ID, token scopes, the subject (user) identifier, and an expiry date. Given that the token is authentic and valid, RGW1 would perform the following security checks:

1. **'User Identity Check'** by verifying that the user in the ID Token (the user that authenticated to the portal) is the same as the subject of the OAuth token
2. **'Client ID Check'**, by checking C1's OAuth client ID against a set of authorized client IDs to access the service

The first check reveals tokens' theft and manipulation attempts, and the second diminishes a token's power and limits blind trust between components. If these checks pass, the gateway forwards the token information to the microservice; otherwise, the request

is denied and the incidence gets reported. If the gateway lets the request through to the resource microservice, the latter returns the attributes of the user mapped to the scopes of the access token.

3.4.3 Operational Flow of the Proposed Approach

The sequence diagram in Figure 3.5 shows a representative example of the proposal. This shows the dataflow of an operation between a CMS and RMS of Figure 3.4; it also reflects an access request between the passport microservice and one of the resource microservices in the scenario presented in Section 3.3. One service, the CMS, is to retrieve resources from another service, the RMS. A central ACS is used as an OAuth 2 authorization server, as well as an XACML server with policy administration and decision points. The ACS can include or be linked to an authentication server that produces and keeps track of authentication sessions with ID tokens. Each gateway (CGW and RGW) functions as a PEP, which inspects the ID token, and uses the ACS PDP to check the policy rules. Access rules can be defined as a set of URLs and actions mapped to a group of users (i.e. Role-based); however, more complex policies can be defined following any policy definition criteria.

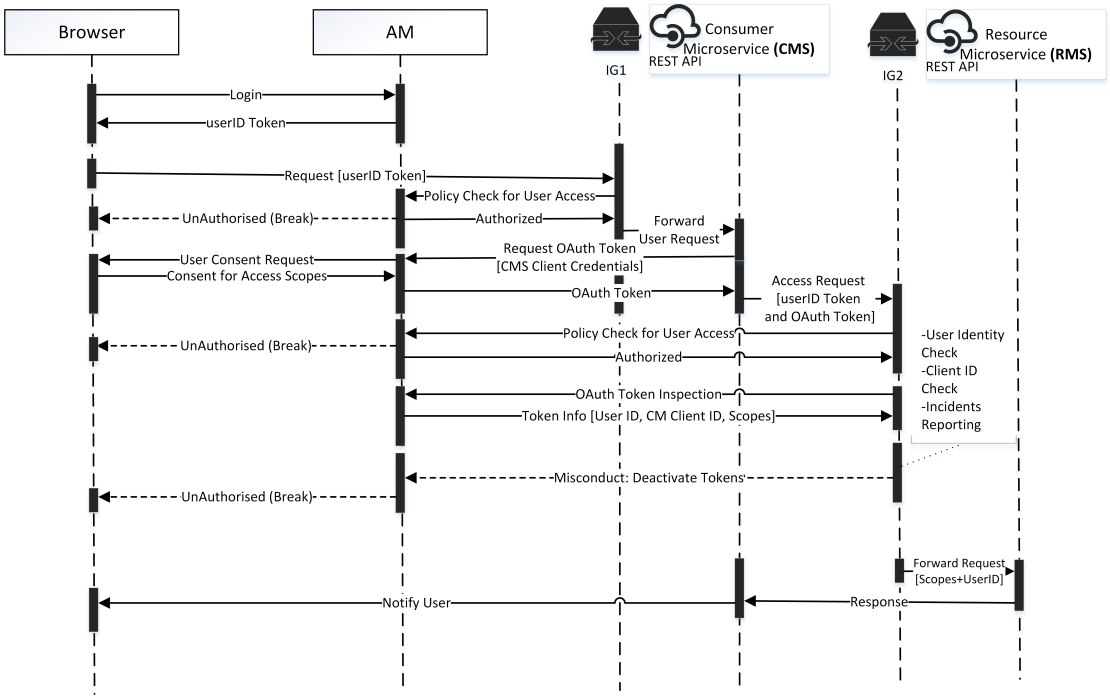

Fig. 3.5 Sequence Diagram Representing a Service-to-Service Interaction

Before any attempt to access CMS, the user has to have an active session with an ID token. When the user sends a request, the PEP at CGW inspects the user's ID token with the Policy Decision Point of ACS, and if the action with CMS is allowed, this user is able to initiate a request with the service. When CMS requires data/service from an

external resource (RMS), it first needs to request an OAuth access token. CMS uses its OAuth credentials, specific for RMS, to initiate the token production request with ACS. In turn, ACS requires the user to be authenticated and to choose the access scopes. At the level of ACS, an Intrusion Detection System can detect session manipulation attempts between the last two interactions with it. The produced access token is sent to CMS, and a request with the ID and OAuth tokens in the header is sent to RGW. RGW, protecting the resource microservice, checks if the user is authorized to access the service that it protects and, if so, the OAuth token is sent to the OAuth token inspection endpoint of ACS. This token gets verified, decoded, and sent back to RGW to perform the User Identity and Client ID Checks described in 3.4.2. If any of the previous checks fails, an appropriate alert will be sent to the system administration and the user session and access token get deactivated. If all conditions are met, RGW sends the request with the user ID and the access scopes to RMS. This service has now the data attributes and/or methods mapped to the token scopes and the data of the user will now be sent to CMS.

3.5 Analysis of the Proposed Approach

This section revisits the early requirements listed in Section 3.3.2 and discusses how the proposed access control approach addresses them.

3.5.1 Fine-Grained Access Control

With PEPs used at each microservice gateway level, access policies allow defining access roles for users to particular services (R1). Gateways here keep any unnecessary potential load off the microservices and act as a further defence layer. Since XACML allows to define complex policies, one can further add contextual access rules such as time and location. Having multiple OAuth 2 clients helps to enforce transparency in the system by requiring users' consent for each access operation to their personal data and giving them the option to choose what they want to share. Scopes are defined during OAuth 2 client creation following agreements between the resource and consumer microservices departments (R3), and having an OAuth client per consumer-resource microservice enables a fine-grained user-centred access control at the level of microservices (R2). Scope to resource mapping is done at the RMS level, and having scopes tailored to each service gives the transparency needed for systems in which privacy is key to users' trust.

3.5.2 Token Theft Mitigation

Having multiple OAuth 2 clients, for different consumer-resource combinations, limits the power of access tokens. With one OAuth 2 token per access task, a stolen token would only be a threat to the data of a particular person in one microservice only. These tokens can have a short lifespan since they are meant to be used once and for one particular request. Also, due to the User Identity Check at the gateway level, access to information from a stolen access token is not possible without access to the ID token of the same user. Any attempt from a conflicting user session would result in deactivating the tokens and reporting the incidence; even session hijacking can be rendered ineffective with a stolen token's short lifespan. Also, the Client ID Check diminishes the token's power by limiting the services that accept the token. This partially fulfils the security goal of R4.

3.5.3 Confused Deputy Mitigation

Going back to Figure 3.4, a token produced with C1, belonging to CMS1 and valid for RMS1, would not be valid for RMS2. This is also valid if service CMS1 is allowed to access both services RMS1 and RMS2, and even if RMS1 and RMS2 belong to the same department (R4). The combination of the User Identity Check, the Client ID Check, and requiring user consent for every service to service data access is a mitigation against the confused deputy attack. These security checks and practices minimise trust between services and give an assurance that a service is acting faithfully on behalf of the user. Therefore, this approach achieves the security goal of R4.

On a related note, the proposed approach mitigates some malicious insiders' activities. According to an IBM report in 2015, 60% of attacks are due to an insider (IBM 2016). If an insider manages to create an OAuth client on ACS to be used by a malicious node, the resource microservices would not accept any access token from this new client since its ID is not in the list of trusted clients of any RGW. This approach minimises the possibility of having a service confused with a rogue/fake client (R3).

3.5.4 Manageability and Reusability

To help with the manageability of the large number of microservices in MSA, categorising services into groups according to their security requirements is likely necessary. These requirements are decided based on the functionality of the microservices, the trust context, and the criticality of the assets that it handles (BS 2015). This is a common approach for large enterprise software to protect their resources (Yu et al. 2016). In this thesis, consumer are separated from resource microservices and each requires a different gateway; another categorisation may separate microservices protecting very sensitive data and only responding to trusted addresses from microservices hosting less

sensitive data. Having reusable security components helps to define configurations with security functions to meet different requirements; this facilitates securing new primitive microservices by plugging in these predefined gateways. Security gateways are extensible and can include other security functionalities including, but not limited to, logging and auditing, cryptographic roles, and throttling. Figure 3.6 represents the extendibility of the approach with gateways, tailored for different security requirements, used to protect different groups of services; The focus of the chapter, however, is limited to access control.

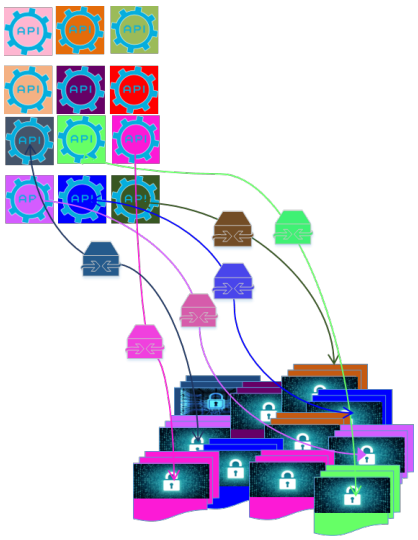

Fig. 3.6 A Representation of Security Enforcement for Different Security Requirements with Gateways

3.6 Implementation of Fine-grained Access For Microservices

A proof of concept was implemented using ForgeRock open source components. ForgeRock Access Management (AM[2]) is used as the central access control server (ACS) for its ability to manage authentication, OAuth access delegation and XACML policies. As for microservices local gateways, ForgeRock Identity-Gateway (IG[3]) is used due to its Policy Enforcement and OAuth 2 token validation filters, and the flexibility that it provides to extend its functionality. This approach is feasible to implement using any technological stack supporting OAuth 2 and XACML; a gateway can be written with any programming language that supports XACML, HTTPS calls, and the implementation of the proposed security checks. For the sake of clarity of this demonstration, Postman[4] is used to play the role of a consumer microservice with an ID token, accessed by the authentication cookie, and an OAuth 2 token, sending an access request

[1] https://www.forgerock.com/platform/
[2] https://www.forgerock.com/platform/access-management
[3] https://www.forgerock.com/platform/identity-gateway
[4] https://www.getpostman.com/

to an RMS protected behind an IG. This shows the same behaviour of a consumer-to-resource microservice call, with the resource microservice protected by RGW.

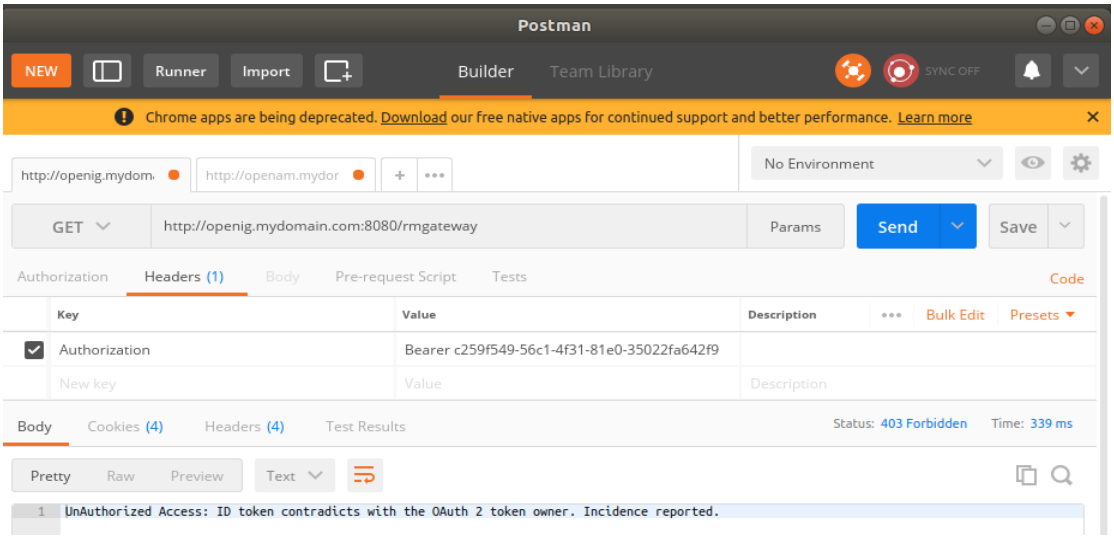

Fig. 3.7 Token Theft Detection

3.6.1 Prototype of the Proposed Security Checks

This section presents the prototype of the presented approach for access control reflecting the applications of the security checks to detect token manipulation attempts. Figure 3.7 shows the response of an RGW on a failed User Identification Check. This is one approach to detect session hijacking and OAuth token theft. Both tokens would be deactivated in this case.

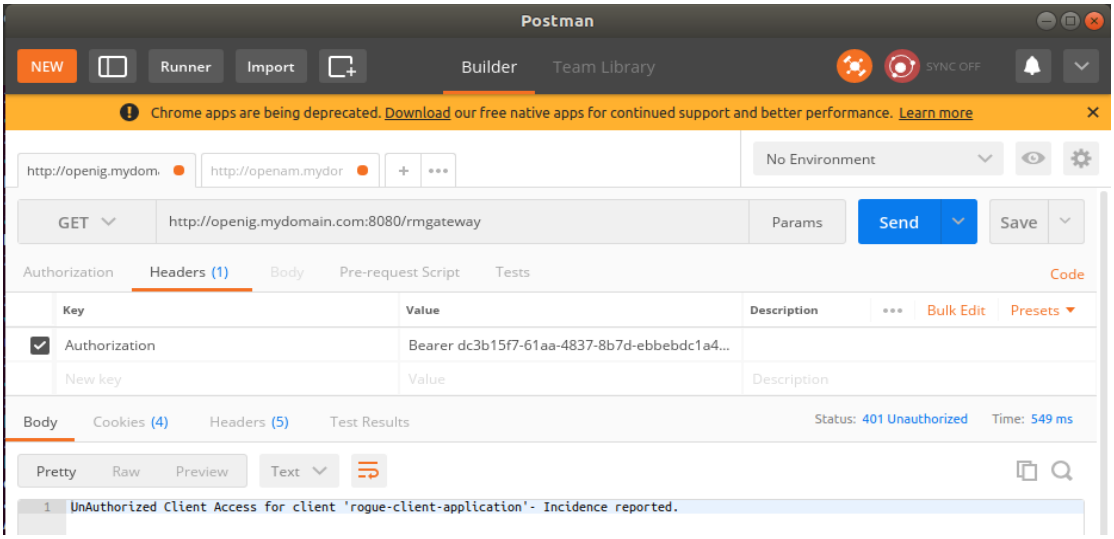

Fig. 3.8 Unauthorized Client Detection

Figure 3.8 shows a request sent to RMS from an unauthorized OAuth client; this reflects the response of using an access token for a different consumer-resource combination, even if this resource and RMS are part of the same department. Client ID Check weakens the power of tokens, limits the trust between services to minimize the effect of a successful *confused deputy* attack, and mitigates creating fake OAuth clients by an insider.

Figure 3.9 shows a successful malicious request caused by the absence of the proposed security checks. In this case, an unauthorized client, potentially created by an insider, is used to send the request, and the resource microservice responded with the data. Due to the absence of the Client ID Check, a malicious microservice with a fake OAuth client can be a threat, leading to data exfiltration from RMS. Having an OAuth client per consumer-resource combination alongside the Client ID Check mitigates this threat. It also minimises trust between microservices by only allowing essential communications between them and requiring the access control server involvement for token production and verification rather than blindly trusting a microservice or its domain. This practice minimises the impact of a *confused deputy* attack by limiting what can be done with a potentially compromised microservice.

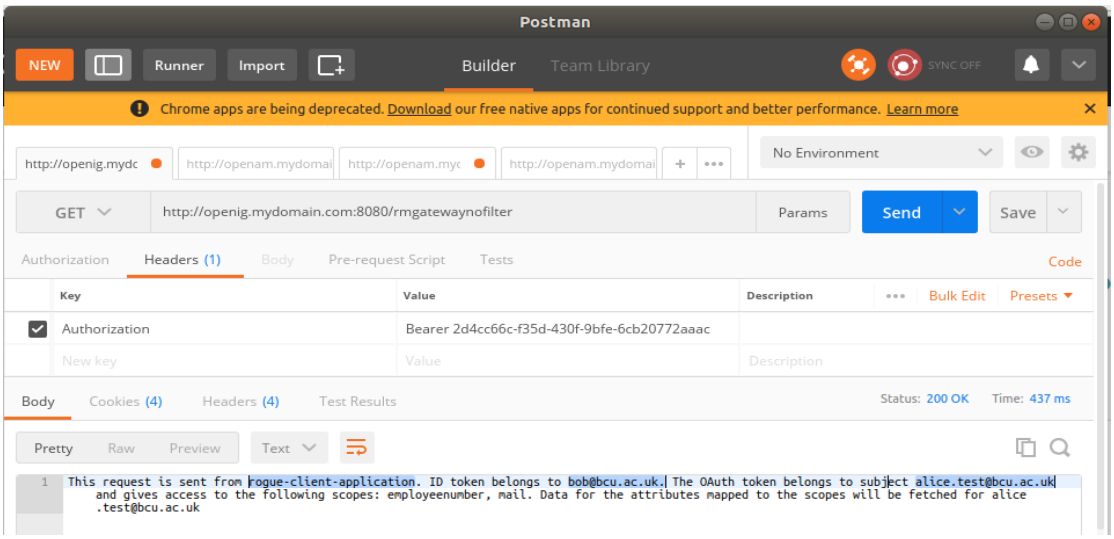

Fig. 3.9 Malicious Request Without our Security Checks

Moreover, the user of the session and the OAuth token subject are not the same, which suggests using a stolen token for the request. Without the User Identity Check, token theft and manipulation would not be detected. This gives this malicious user the ability to apply to services using another user's information. The User Identity Check mitigates these attacks.

3.6.2 Performance Evaluation of the Prototype

This section shows the overhead resulting from the proposed approach. This experiment was conducted on an Ubuntu 17.10 running on a machine with 2.6 GHz Core i7 processor and 12 GB of RAM; the aim is to show the overhead caused by adding gateways configured for consumer microservices (CGW) and resource microservices (RGW). The line chart in Figure 3.10 visualises the response time of 250 service calls for the same microservice without any gateways, with CGW, and with RGW. ACS is placed in a separate Linux container, on the same machine, to isolate the effect of data propagation over the internet. The lines show that the response time is the highest for microservices protected by RGW; the numbers confirm that, on average, an overhead of 23% results from adding a CGW, and of 32% occurs from adding RGW to a microservice. This means that a mixture of gateways protecting consumer and resource microservices should lead to an overhead of less than 32% on average over the time to access a certain endpoint. The overhead of User Identity Check and Client ID Check

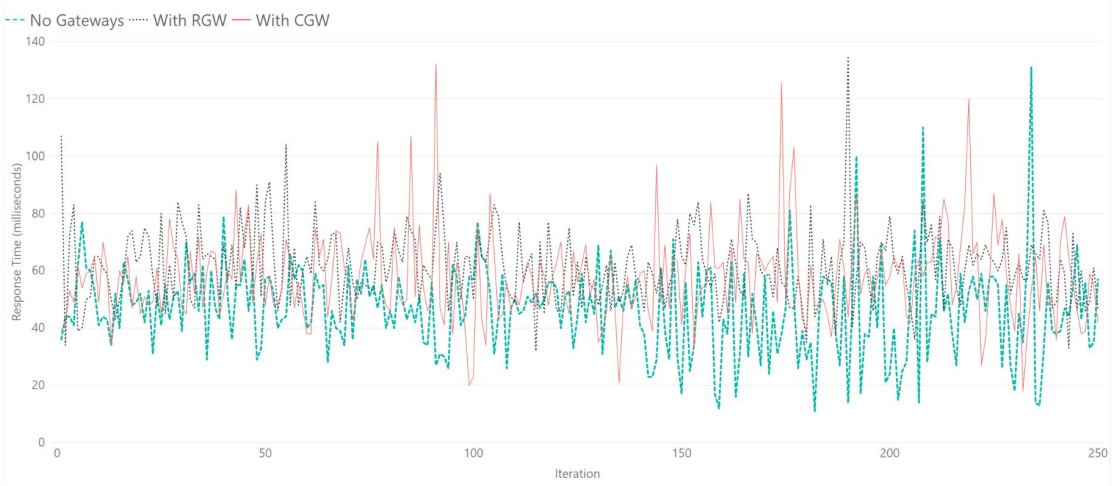

Fig. 3.10 Line Chart Showing Our Experimental Results

is minimal, given that these are simple checks performed locally and not requiring any additional interactions with the ACS. As for ACS, the load factor is mostly affected by the number of exposed resource microservices due to the extra checks of OAuth 2 tokens; this is relatively easy to overcome with cheap cloud elastic scaling.

3.7 Conclusion

This chapter highlights some security challenges that microservices-based applications are prone to in connection to access control and authorisation, both when the user is the trust anchor and when microservices work in conjunction. A security design was presented that allows fine-grained access control with access gateways at the microservice level. The concept was demonstrated by implementing a proof of concept combining XACML and OAuth 2, two leading open standards that are designed without considering the specifications of microservices-based applications. The proposed security checks help to minimise implicit trust in microservices-based applications enabling to contain a security breach caused by a compromised microservices or by a malicious privileged insider. Although digital government scenarios have been the focus of this chapter, the discussed approach applies to any microservices-based applications includes private data or enables acting on behalf of users.

The work in this chapter contributes to the big picture of the research that looks into the chain of trust in distributed multi-party systems with connectivity, verifiability of requests, user control over personal data and accountability as central requirements. Several challenges are kept open: the dependence on trusting key elements, for example, Access Control Servers pose a risk and are able to compromise the whole system if they get compromised. On the other hand, from a user perspective, user repudiation is still an open challenge. These challenges are left open to the literature and for future work given the time restriction of this research. Finally, usability testing will help evaluate the impact of the proposed approach on end-users. This requires the resources of a large institution to design a pilot and test it on a large enough population to produce trustworthy results.

The fine-grained access control model concludes the contribution of this research for the security of microservices-based applications. As highlighted in this chapter, applications from different administrative domains collaborate to fulfil a business function. Collaborations should be audited following a robust approach to hold participants accountable for their actions. Next chapter explores auditing of distributed collaborations and presents a novel and confidentiality friendly approach to record evidence of transactions without trusting a single entity to generate, verify or store audit records.

CHAPTER 4

AuDiC: Auditing of Distributed Workflows Collaborations

Workflows involve actions and decision making at the level of each participant. Trusted generation, collection and storage of evidence is fundamental for these systems to assert accountability in case of disputes. Ensuring the security of audit systems requires reliable protection of evidence in order to cope with its confidentiality, its integrity at generation and storage phases, as well as its availability. Collusion with an audit authority is a threat that can affect all these security aspects, and there is room for improvement in existent approaches that target this problem.

After focusing on the security of applications in the previous chapters, this chapter presents the contribution of this research for auditing to assign accountability in workflow collaborations combining different applications. Without relying on a single point of trust to generate, verify or store audit records, the proposed auditing approach improves the confidentiality, integrity and availability of audit trails while offering auditing capability to any threshold of participants. The adequacy of the approach to produce reliable evidence despite possible collusion to destroy, tamper with, or hide evidence is discussed in this part of the thesis.

4.1 Introduction

A virtual organisation, defined as a collaboration of independent organisations to fulfil a business requirement (Nami and Malekpour 2008), requires tasks and decision making to be spread among the different administrative and security domains of participants. These collaborations necessitate a way to keep track of transactions between the involved parties, and evidence of actions (i.e. audit trail) needs to be available to assign accountability in case of a dispute. Evidence should be reliable and its processing should not disclose confidential information to any party. Different administrations do not trust each other to generate, store or protect the confidentiality of audit trails reflecting actions, decisions and data exchanges between participants; relying on an audit service, a trusted third party, to generate audit trails is also a risk on the confidentiality of transactions and the authenticity of the generated evidence if this party is compromised.

To illustrate the challenge, a hypothetical scenario is presented: car insurance companies rely on data about the driver's behaviour and habits to calculate their annual fees. When an individual applies for renewal, the insurance company sends a request to assess the applicant to the Ministry of Transport. This ministry requests the car mileage from garages that perform vehicles annual checkups as well as any arrest warrant for the applicant from the police department, and checks the databases of traffic cameras for the neighbourhood in which the car is most frequently used. Insurance companies do not get all the details about the driver due to the confidentiality of this data; they receive a report with an assessment from the Ministry of Transport and use it to determine the insurance fees. Audit records for every request should be kept: each organisation maintains a log, and an audit service is used to build audit trails. A privileged insider at the Ministry of Transport, colluding with an insurance company, modifies the mileage received from the garages to increase insurance fees. This insider also modifies the local logs, and colludes with the audit service to modify the audit records. As seen in this scenario, a collusion with an entity managing a centralised audit service renders its data unreliable. Moreover, audit records in workflows include sensitive information for all participants. In this scenario, audit records stored at the Ministry of Transport or with a central audit system expose personal and confidential data.

For a workflow audit architecture to be reliable, one participant should not have the option of colluding with the authority that manages the audit system to breach the confidentiality of, tamper with, or destroy evidence (Zawoad et al. 2013). A central approach in which a single system is trusted to collect, verify, and store logs is a single point of trust in this case (Ahmad et al. 2018, Wombacher et al. 2005). In practice, workflow engines, commonly used to coordinate interactions between workflow participants and to provide auditing services, are a representation of a central approach for audit (Rudolph et al. 2009, Lim et al. 2012, Vahi et al. 2013). Some approaches, (Wouters et al. 2008) for example, propose holding each participating administration accountable to store its own audit logs. Although this practice protects the confidentiality of participants, it does not provide protection against tampering with and destroying evidence.

Therefore, a *decentralised* and *trustless* approach is followed to produce reliable evidence. In this thesis, *Decentralised* is defined as not depending on a single system to generate audit trails, and *trustless* audit as not giving an organisation the ability to produce digital evidence without verification of its authenticity by another organisation. Following the architecture presented in this chapter, audit records are retrieved and verified in a distributed way immediately after generation by participants, while protecting the confidentiality, integrity, and availability of this data from malicious entities working individually or colluding with each other. Following the proposed approach for auditing, malicious behaviour of tampering with, deleting, or false reporting of audit records is detected by honest participants. The contribution in this chapter is an archi-

tecture that offers mitigation from collusion-related threats to tamper with or destroy multi-party workflow audit data. This depends on the collaboration of participants to verify and obtain a copy of encrypted audit records reported to an audit server, as well as to ensure the authenticity of this server. Secret sharing mechanisms (Brickell 1989, Shamir 1979) are relied on to minimise the risk of confidential data exposure.

In a multi-party workflow, this architecture:

- Offers a means to verify reported audit records.
- Supports distributed storage of audit data at any degree of details.
- Introduces a data structure for audit trails covering arbitrary topology.
- Offers audit capability to any K out of N participants.

This chapter presents AuDiC, an approach for Auditing of Distributed workflow Collaborations, and discusses an implementation of this approach that includes an untrusted centralised server; for clarity, the discussion in this chapter is limited to auditing approaches relying on a comparable technology stack to this implementation, and the discussion of blockchain and the blockchain-based implementation of this approach are left to the next chapter. This chapter is organised as follows. Section 4.2 presents a review summarising common approaches for auditing discussed in the literature; Section 4.3 presents the problem statement and the threat model, and key concepts in AuDiC are introduced in Section 4.4. The specifications of the first implementation are presented in Section 4.5, followed by an analysis of its adequacy to fulfil our goals in Section 4.6. Implementation and evaluation are covered in Sections 4.7, and Section 4.8 concludes the chapter.

4.2 Auditing Approaches: State of the Art

Many audit frameworks proposed in the literature log events as evidence to verify abiding to security policies and regulatory requirements. Rudolph et al. (Rudolph et al. 2009) designed an audit trail with a summary of participation exchanged between participants to show fulfilment of tasks and enforce behavioural policies during workflow execution; they extended their work in (Velikova et al. 2009) to cover the anonymity of participating entities, but did not discuss the security of audit trails at storage and their protection from destruction. Hale et al. (Hale et al. 2013) present a design and verification framework for services interactions across different clouds to verify abiding to information sharing policies. Other solutions (Lim et al. 2012, Bates et al. 2017, Sundareswaran et al. 2012, Aravind and Sandeep 2015, Yao et al. 2010, Flores 2014, Gajanayake et al. 2011, Pulls et al. 2013, Accorsi 2011, Vahi et al. 2013, Marty 2011, Redfield and Date 2014, Kieseberg et al. 2016) cover a variety of audit data including

hashes and signatures, users' consents for access control, data provenance, service level agreements related logs, compliance and operations, and records of database access. However, a trusted central system is used to process logs in (Sundareswaran et al. 2012, Pulls et al. 2013, Vahi et al. 2013) and to store audit records in the other frameworks. Austria and Uruguay follow a central approach for audit for their e-government systems (Hartmann and Steup 2015, González et al. 2012). This raises trust and confidentiality problems since collusion with the central point in any of these cases makes tampering with or destroying evidence possible, as well as breaching the secrecy of audit data.

To protect the confidentiality of audit data, especially when relying on a third party service as proposed in (Rajalakshmi et al. 2014) and (Ray et al. 2013), a common practice is to encrypt audit data prior to its submission to the outsourced storage. (Waters et al. 2004, Accorsi 2010, 2013) are proposals that require a trusted party to protect encryption keys for audit records, and Wouters et al. (Wouters et al. 2008) rely on the user to protect the key in their approach. Entities with encryption keys can breach the confidentiality and in some cases the integrity of audit data; also, refusing to share these keys when an audit is required leads to withholding evidence. The framework proposed by Ray et al. (Ray et al. 2013) uses a secret sharing scheme to split encryption keys to multiple logging hosts; however, their approach covers logging for single applications and does not consider multi-party interactions.

For forensic investigations, a high degree of assurance for the integrity of digital evidence is required at the collection, generation, and storage phases (Zawoad et al. 2013, Alqahtani and Gamble 2014). A variety of hash-based approaches, relying on hash chains and hash trees, are proposed in the literature (Tian et al. 2017, Zawoad et al. 2013, 2016, Ma and Tsudik 2009, Kieseberg et al. 2016) to assure the integrity of audit records. Kieseberg et al. (Kieseberg et al. 2016) use a hash chain in a health-related context including multiple stakeholders to produce logs aimed to verify the authenticity of transactions reflecting decision making at each stage of a workflow; confidentiality and availability of records are out of the scope of their work. Ma and Tsudik (Ma and Tsudik 2009) assume the benevolence of the entity managing the logging system and use hash chains to verify that it has not been compromised. They aim to detect attempts to modify logs generated prior to a compromise of the logging entity, and rely on trusted storage for their approach; their work does not consider collusion-related attacks and does not discuss protecting the confidentiality of their data. Tian's work (Tian 2017) relies on a hash chain structure, inspected by a trusted third party, to assure the tamper resistance and integrity of audit data covering users' operational behaviour with a cloud provider; a collusion with the cloud provider to falsify records prior to the hashing process can, however, cover a malicious behaviour of falsifying audit data. Zawoard et al. (Zawoad et al. 2016) mitigate possible tampering with evidence that can result from collusion between users, investigators, and cloud service providers.

They encrypt the log files with a public key belonging to a law enforcement agency to protect users' privacy, and adopt a hash-chain scheme referred to as 'Proof of Past Log (PPL)', covering the entire log history for every user on a daily basis. However, these approaches trust the logging entity, being the cloud provider, to report correct audit data and to assure its availability. A proposal by Ahsan et al. (Ahsan et al. 2018) is an extension that covers some of the limitations of (Zawoad et al. 2016). In their work, log records generated by cloud service providers are not trusted, and users are required, within a specific timing, to verify their logged activity with their cloud provider and to file a complaint in case of any activity that needs to be denied. Logs attributed to a user are then stored on the cloud service provider servers and encrypted with this user's private key. To mitigate withholding evidence by users, a secret sharing mechanism is proposed to distribute shares of every user's keys to different cloud providers that are assumed not to collude. This approach assumes continuous human cooperation for its verification phase of the audit records. Moreover, approaches discussed in (Tian 2017, Zawoad et al. 2016, Ahsan et al. 2018) cover auditing users' activities in a single cloud only, and do not consider destruction of evidence by a cloud provider or multi-party workflows.

Blockchain is another approach that uses hash chains and the consensus of a large network of nodes to protect its data from tampering (Weber et al. 2016). Uses of blockchain include verifying the integrity of files, and running programs in a transparent and trusted way while producing an audit trail of these runs; however, this technology does not scale to store large transactions, and therefore only hashes of these transactions are normally stored on the blockchain (Tian 2017, Xiong and Du 2019). Blockchain based approaches are discussed in details in the next chapter.

The auditing approach presented in this chapter improves the availability, confidentiality, and integrity of audit records in distributed workflow collaborations. Practices in the literature for collective building of audit trails are combined with forensic approaches to protect digital evidence, and verification mechanisms are introduced in this architecture to assert accountability of participants in multi-party workflows.

4.3 Auditing of Workflows: Problem Statement

Collusion with third parties, tampering with, and hiding records are among the most used techniques to hide a fraudulent activity (Abreu et al. 2018). This work argues the need for improvement in audit approaches used for workflows covering multiple administrative domains. Every participant should be accountable for a contribution reflecting an action or a decision in a workflow. Existent audit approaches, discussed in the previous section, depend on trusted parties to record or store audit records; this renders the integrity and availability of these records questionable. Moreover, traditional practices

of recording digital evidence pose a privacy risk on the involved parties whose interactions can be sensitive and confidential. Also, trusting a single entity with encryption keys leaves room for withholding evidence. When following approaches discussed in the literature, the problems below can be faced:

- Destroying or withholding evidence
- Breaching the confidentiality of transactions
- Tampering with audit data at storage
- Reporting incorrect audit data

The contribution of this research for auditing of inter-domain interactions and data exchange focuses on tackling these challenges. This work aims to offer a reliable audit approach that protects the confidentiality, integrity at generation and storage phases, as well as the availability of evidence. In order to achieve that, the proposed approach avoids relying on or trusting a single entity to record, store, or protect the confidentiality of audit records.

4.3.1 Threat Model With a Centralised Audit Server

This model excludes compromised elements being under the control of an attacker. Actions are considered to reflect the intentions of the administration managing an element, rather than a fault or compromise in the system. Certificate authorities and key issuers are honest; they distribute correct cryptographic keys and do not expose participants' secrets. Participants are also trusted to protect their cryptography keys. Communication between the audit server and workflow participants is over a secure channel.

The adversary vector includes participants that aim to rig an audit reporting process in order to avoid non-repudiation. Adversary's goal can be achieved by destroying or tampering with audit records, or by using incorrect cryptographic keys for encryption or signatures to jeopardise the records' usability. The entity managing the audit server can also be malicious, and may attempt to tamper with, destroy, or hide audit records from some participants. A number of malicious entities (including workflow participants and the audit server) can collude to falsify the audit process; this includes skipping the verification of reported audit records, hiding audit data from honest participants, cooperating to distribute different versions of records, and delaying the workflow with false alerts. Breaching the confidentiality of workflow transactions using audit records is another malicious goal for an adversary.

Under this model, this work aims to verify the authenticity of reported audit records as they are generated as well as their integrity during storage or distribution, and to

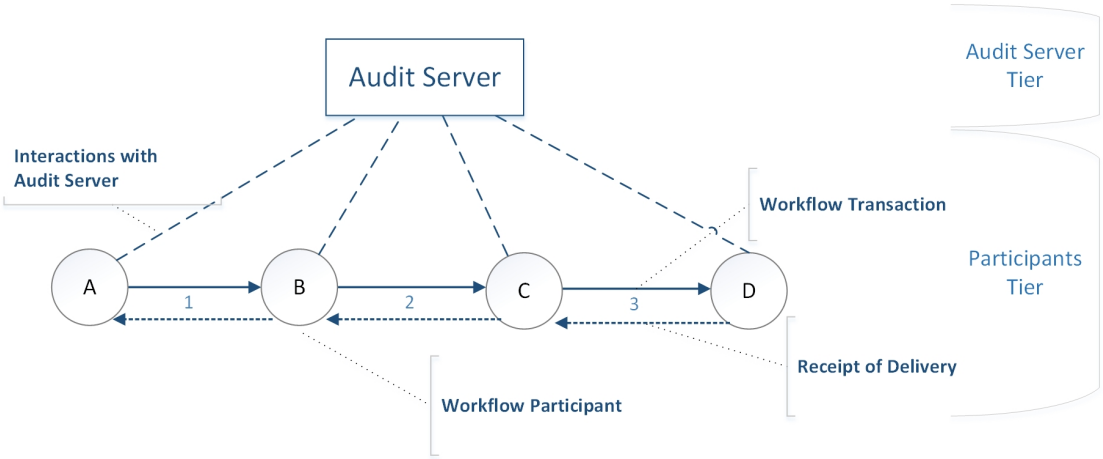

Fig. 4.1 A Simplistic and Technology Independent Representation of AuDiC

protect their confidentiality at every stage of the audit process. The availability of evidence is also achieved when a definable number of participants are honest. Audit data reported by colluding entities is considered unreliable, but any malicious behaviour resulting from collusion is proven by using reliable audit records reported by honest participants.

4.4 Trustless and Collaborative Auditing

After studying the existent approaches for auditing discussed in Section 4.2, this work adopted practices including hashes to protect the integrity and threshold cryptography to safeguard the confidentiality of audit records at storage; to tackle the identified problems in existent auditing systems listed in Section 4.3, this thesis proposes AuDiC: a confidentiality-friendly, application-agnostic, collaborative and trustless approach to audit workflows. When designing AuDiC, the focus was on not trusting a single entity to record, store, or protect the confidentiality of audit records. Encrypted audit trails, covering arbitrary workflows, are checked for authenticity as they are built, and are distributed to participants to protect from tampering with or destroying evidence. To safeguard the confidentiality of this data, a threshold of key shares is required to reconstruct the decryption key. This also minimises the chance of withholding evidence by an entity that is holding the key. Audit trails can be constructed with any degree of details ranging from a summary of interactions to full transactions. A high level representation of AuDiC is shown in Figure 4.1. As shown in the figure, AuDiC includes an audit server for the purpose of displaying data to workflow participants. Workflow participants participate in the generation and verification of audit records, and exchange receipts with every transaction. When relying on a centralised server, the audit server is not trusted and different measures are taken to verify that it cannot behave maliciously.

In this chapter, the focus of the discussion is on a centralised implementation of the audit server, which is replaced with a blockchain-based implementation in Chapter 5.

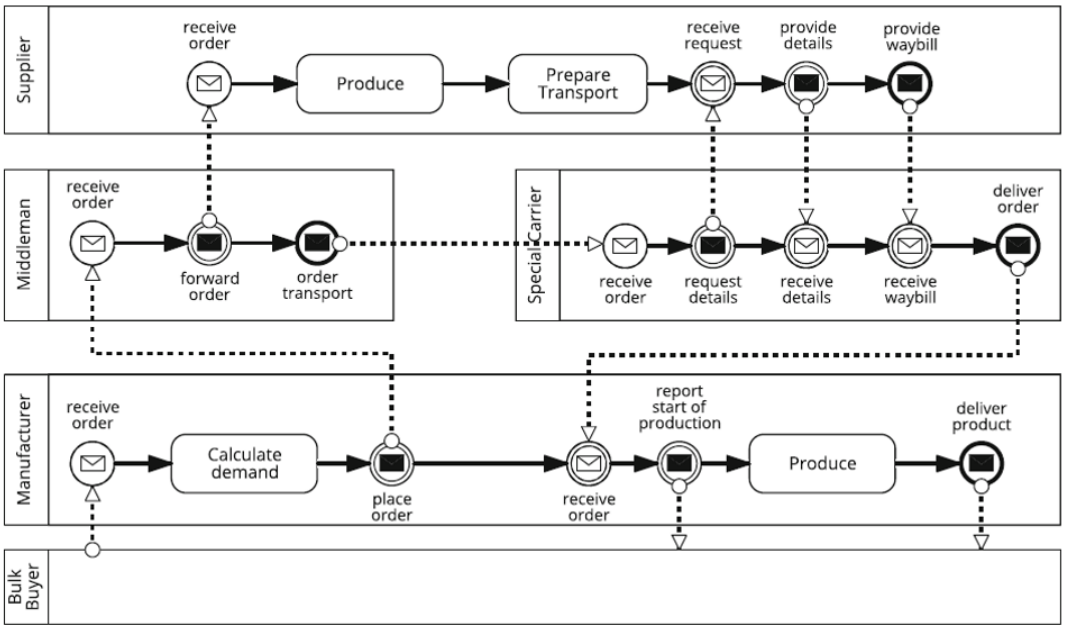

Fig. 4.2 BPMN Representation of a Supply Chain Workflow. Credit Goes to Weber et al (Weber et al. 2016)

4.4.1 Notation for Auditing Operations of Workflows

A graph is used to represent workflows, in which nodes (vertices) represent organisations in the workflow, and edges with weights represent the order of their interactions. Figure 4.2 is a BPMN representation of a workflow, and Figure 4.3 is the equivalent representation following the adopted notation in this thesis. In Figure 4.2, Bulk Buyer **BB** places an order with the Manufacturer **M**, and the latter calculates the demand and orders materials via a Middleman **MM**. This Middleman sends the order to a Supplier **S**, and delegates the delivery to a Special Carrier **C**. When the materials are produced, the supplier arranges for the delivery with the Carrier after identifying the latter, and the Carrier delivers the materials to the Manufacturer. The Manufacturer reports the start of production to the Bulk Buyer, and delivers the goods to the latter when they are produced.

In Figure 4.3, $edge(BB, M)$ is the first interaction in the workflow manifested by the request from the Bulk Buyer to the Manufacturer. Workflows covered in this thesis are pre-established, meaning that the participating entities and their order of interaction are known by participants at the beginning of the workflow execution and do not change until it finishes.

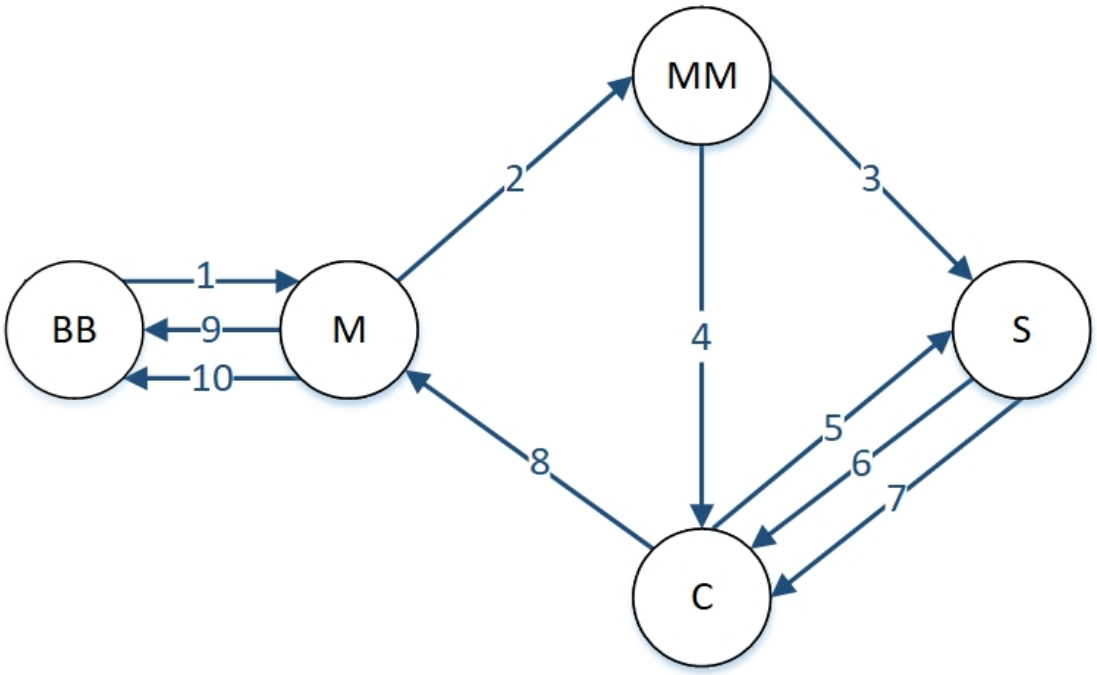

Fig. 4.3 Our Representation for the Supply Chain Workflow

For ease of representation, the notation below is used to describe message processing and cryptography operations:

- Edge e_i represents a transaction where i reflects the order of execution in the workflow.
- Concatenation of string X with string Y is represented by: $X|Y$.
- Shares of a workflow private key wk, split with a secret sharing mechanism to N shares, are denoted as $K_1...K_N$. The equivalent public key is wpk.
- An encrypted message referred to as $message$, sent from A to B, containing P_{AB} and the signature of the sender over the $payload$, is represented by

$$P_{AB}(S_A, E_B) = E_B[(P_{AB}) + S_A(P_{AB})].$$

- $Data$ sent from A to B as part of P_{AB} is represented as D_{AB}.
- Topology data sent from A to B as part of P_{AB} is represented by T_{AB}.
- An additional signature over the $message$, S'_{sender}, can be sent alongside the cipher text. The $message$, in this case, is represented as:

$$P_{AB}(S_A, E_B, S'_A) =$$

$$P_{AB}(S_A, E_B) + S_A[P_{AB}(S_A, E_B)]$$

– An $audit$ record of a $message$ sent from A to B is represented as:

$$A_{AB} = P_{AB}(S_A, E_{wpk}, S'_A).$$

– A unique reference to an audit record sent from A to B is represented as:

$$a_{AB} = Hash(A_{AB})$$

where $Hash$ is a one-way hash function with a strong collision resistance.

– D_i, P_i, A_i, a_i represent the $data$, $payload$, $audit$ record, and the $reference$ to an $audit$ record of e_i respectively.

– $Node$ and *participant* are used interchangeably to indicate an organisation in a workflow.

4.4.2 Coverage of Arbitrary Topology

The representation of workflows in this thesis is application-agnostic and covers arbitrary topologies. Following this representation, three communication patterns are distinguished:

– Sequential with no backpropagation (Figure 4.4.1): transactions travel forward through workflow participants.

– Sequential with backpropagation (Figure 4.4.2): Nodes expect a response for their requests to complete their role in a workflow.

– Parallel Paradigm (Figure 4.4.3): this includes a node sending requests to multiple other nodes (e_1 and e_2), and multiple nodes sending requests to the same participant (e_3 and e_4); these are comparable to fork and join actions respectively (Albert et al. 2005).

AuDiC covers any arbitrary topology resulting from a combination of these communication patterns. Similar representations of workflows and comparable communication patterns are adopted and discussed in the literature (Bilal et al. 2005, Wang et al. 2017, Albert et al. 2005, Azarmi et al. 2012).

To enable audit trails to cover any topology, *topology* data is added to the $payload$ to identify the position of participants in the workflow and help deduce their order of interactions and communication pattern. *Topology* data includes the *issuer* and *recipient*

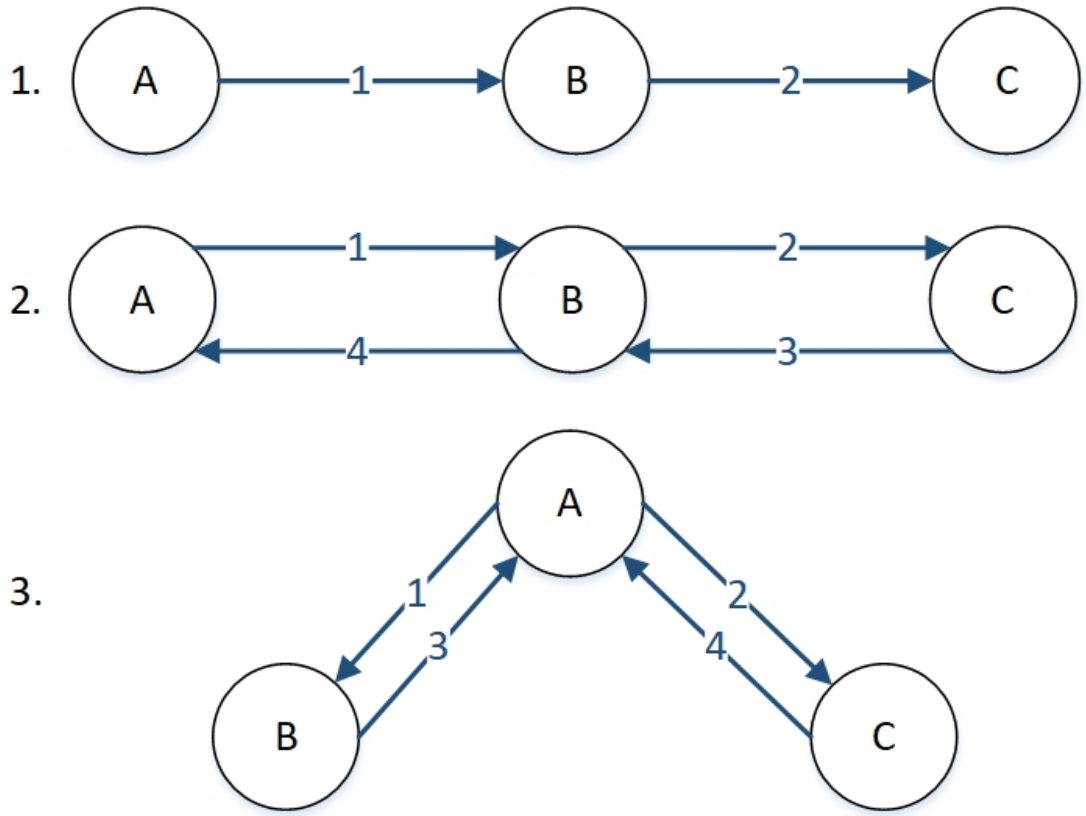

Fig. 4.4 Communication Patterns in a Graph Based Representation of Workflows

of a transaction, and a *Label* set to *ini* if the issuer is the initializer of the workflow and *parallel* when requests are sent in parallel. A reference to previous audit records is also added and categorised into groups:

1. $Prev$: refers to the $audit$ records covering the previous linear transactions that the requester received
2. $ParaPrev$: refers to the $audit$ records resulting from responses to requests a participant sent in parallel
3. $Notifications$: refers to $audit$ records reflecting a participant's notifications of actions to other participants.

Workflows are initiated and terminated by a single participant (Albert et al. 2005); workflow initiators and terminators can, but do not need to, be the same entity. To understand how the constructed audit trails cover complex and arbitrary topologies, this section shows $payloads$ exchanged between participants in Figure 4.5 topology, which combines these three communication patterns. To avoid the complexity of details irrelevant to the purpose of this section, the signature and encryption associated with each $payload$ and $audit$ record are not shown in the list below.

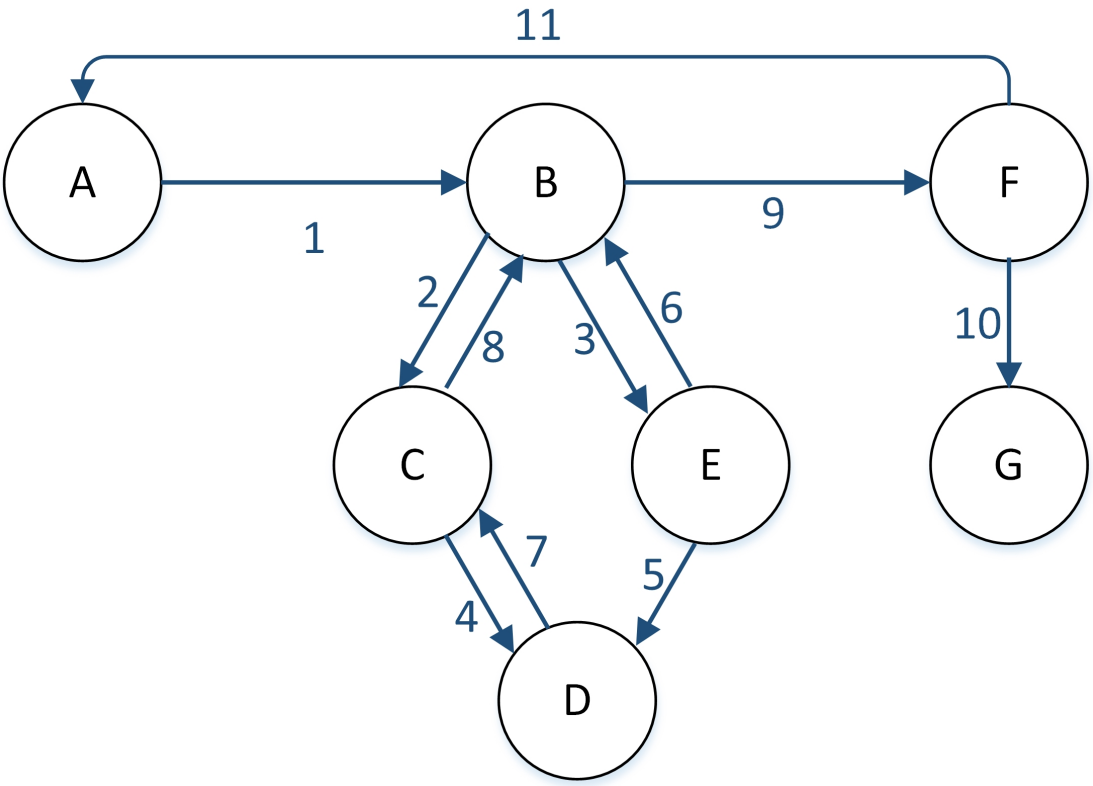

Fig. 4.5 Our Representation of Workflows

$$P_1 : [D_1, Label = ini]$$

$$P_2 : [D_2, Label = parallel, Prev = a_1]$$

$$P_3 : [D_3, Label = parallel, Prev = a_1]$$

$$P_4 : [D_4, Prev = a_2]$$

$$P_5 : [D_5, Prev = a_3]$$

$$P_6 : [D_6, Prev = a_3, Notifications = a_5)]$$

$$P_7 : [D_7, Prev = [a_4, a_5]]$$

$$P_8 : [D_8, Prev = a_7]$$

$$P_9 : [D_9, ParaPrev = [a_6, a_8]]$$

$$P_{10} : [D_{10}, Prev = a_9]$$

$$P_{11} : [D_{11}, Prev = a_9, Notifications = a_{10}]$$

$$P_{final} : [final, Prev = a_{11}]$$

Following AuDiC for constructing audit records, an $audit$ record always includes exchanged $data$ (or a summary of it), unless it is published by the final participant as a sign of completion of the workflow; hence, the flag "$final$" can replace the reported $data$ in a record as is the case for P_{final}. An $audit$ record always points to previous ones, unless it reflects the initiation of a workflow with a single or parallel request, referred to with $Label =$ "ini" and $Label =$ "$ini, parallel$" respectively; in the example discussed in this section, the workflow is initiated with a single request sent from $node\ A$ to B, hence the use of $Label =$ "ini" in P_1. $Prev$ contains the audit records from requests that a participant received from one or multiple participants, $ParaPrev$ includes the audit records equivalent to responses resulting from requests a participant sent in parallel, and $Notifications$, when applicable, includes records of a participant notifying another of an action. A participant sending parallel requests ($node\ B$), having the $Label =$ "$parallel$", adds a reference of the equivalent $audit$ records of the requests it received in the $Prev$ parameter of the request sent to its callees (see P_2 and P_3); it then passes reference to records of the received responses from the parallel callees within the $ParaPrev$ parameter of the transaction sent to the next participant in the workflow (P_9). Following the protocol of AuDiC, $audit$ records belonging to the $Notifications$ category are always reported by the notifier: $node\ E$ notifies $node\ D$ of the state of the workflow, therefore it is expected to report an $audit$ record (P_5) that is verified by the recipient of the notification and to have a reference to the notification $audit$ record in the transaction sent to the next participant ($node\ B$ receiving P_6). The same case applies to $node\ F$ sending a notification to $node\ G$. The final record is published by the $node$ which receives the last $message$; this record announces the end of the workflow and points to the previous $audit$ records following the same criteria for any other transaction.

Algorithm 1 shows how audit trails covering a workflow execution are decoded. The workflow topology reflecting the order of participants interactions is deduced, and the exchanged data in each of these interactions is revealed following the degree of details in the reporting strategy that participants agree on; this can be as comprehensive as recording the full data exchange. In addition to the reported data reflecting what was sent in a transaction between participants, each $audit$ record includes a pointer to the previous $audit$ records. The final record is the root that has connections to all previous records, and therefore it is used as the initial point to decode the audit trails. This algorithm is recursive, with the final $audit$ record being the initiating point and the initial record being the base case; the list of $audit$ records, being the audit trail, and an index of a record are the input of this algorithm. The $data$ part of the audit record at the index position of the audit trail gets revealed, and previous records pointed to in this record are next to be processed by the algorithm; an iteration of the algorithm ends when it hits the base case, and the next record in the stack gets processed.

Algorithm 1 Audit Trail Decoding Algorithm

Requirement: Represents Workflow Interactions from an Audit Trail

Input: $Audit\_Recs[]$, $index$ ▷ Array with the decrypted audit records, and an index of a record in the array

Output: Workflow Interactions

```
1: if Audit_Recs[index].getdata()!=Ø then
2:     display(Audit_Recs[index])
3: if Audit_Recs[index].getLabel().contains(ini) then
4:     break()
5: if Audit_Recs[index].Prev!=Ø then
6:     for each P ∈ Audit_Recs[index].Prev() do
7:         Index = find_index(Audit_Recs[], P)
8:         Run(Audit_Recs[], index)
9: if Audit_Recs[index].ParaPrev!=Ø then
10:    for each PP ∈ Audit_Recs[index].ParaPrev() do
11:        Index = find_index(Audit_Recs[], PP)
12:        Run(Audit_Recs[], index)
13: if Audit_Recs[index].Notifications!=Ø then
14:    for each N ∈ Audit_Recs[index].Noticications do
15:        Index = find_index(Audit_Recs[], N)
16:        Run(Audit_Recs[], index)
```

Next, Algorithm 1 is traced with the audit trail of the workflow of Figure 4.5. A $payload$ reflects an $audit$ record given that, as discussed in the previous section, an $audit$ record is an encryption with the workflow public key wpk of the $payload$ with signatures of participants. When transactions are recorded in full, the part that is shown of the exchanged $payloads$ in the list presented earlier in this section is the same for P_i and A_i of each transaction of $edge\ e_i$ in Figure 4.5; the list of $payloads$, reflecting the list of decrypted $audit$ records, is the first input for Algorithm 1 and the $index$ of the final record is the second input. The first iteration of the recursion finds the $index$ of A_{11} in $Audit\_Recs[]$ and an iteration of Algorithm 1 with $Audit\_Recs[]$ and the $index$ of A_{11} is added to the stack. A_{11} gets displayed and the algorithm finds the $index$ of A_9, pointed to in $Prev$ of A_{11}, and adds an iteration with $Audit\_Recs[]$ and the $index$ of A_9 to the stack. Then, A_9 gets displayed and the algorithm finds the $index$ of A_6, pointed to in $ParaPrev$ of A_9 and adds an iteration with $Audit\_Recs[]$ and the $index$ of A_6 to the stack. Next, A_6 gets displayed, the algorithm finds the $index$ of A_3 and another iteration of the algorithm with $Audit\_Recs[]$ and the $index$ of A_3 gets added to the stack. A_3 gets displayed, and the $index$ of A_1 is fetched this time. Running Algorithm 1 with $Audit\_Recs[]$ and the $index$ of A_1 hits the base case with $Label = ini$ for A_1 after the latter gets displayed, then the last iteration in the stack terminates; the iteration preceding the last, covering A_3, also terminates at this stage given that A_3 only points to A_1 as a previous $audit$ record. Next, the algorithm moves to cover A_5, pointed to in

$Notifications$ of $audit$ record A_6 still being processed with an iteration in the stack. The same process carries on, and all the records get displayed for auditing at the end of the algorithm.

A_6 gets displayed, and an iteration of the algorithm with $Audit\_Recs[]$ and the $index$ of A_3 gets added to the stack; the latter terminates when hitting the base case. The algorithm now goes back to the iteration covering A_9, already in the stack, and the next $audit$ record to be processed is the second in $ParaPrev$, A_8. An iteration of the algorithm with $Audit\_Recs[]$ and the $index$ of A_8 as arguments gets added to the stack, A_8 gets displayed, and an iteration of the algorithm for A_7 gets added to the stack. A_7 now gets displayed and an iteration with A_4, pointed to in $Prev$ of A_7 gets added to the stack. Similar to other iterations, A_4 gets displayed and an iteration for A_2 followed by and iteration for A_1 (hitting the base case) get added to the stack. The next record in $Prev$ of the iteration in the stack covering A_7 is A_5; an iteration gets added to the stack with $Audit\_Recs[]$ and the $index$ of A_5 and it terminates when the base case it hit. This brings the run back to the iteration covering A_{11}, already in the stack, and the algorithm continues to cover A_{10} pointed to in $Notifications$ of A_{11}. Finally, A_{10} gets displayed, and an iteration covering A_9 gets added to the stack. The stack gets emptied following the termination of each iteration hitting the base case, and the run terminated after covering all the records in the trail.

4.4.3 Key Management

Certificates for every participant are assumed to be managed with public key infrastructure (PKI). While certificates can be used in multiple workflows, workflow keys are only used in a single workflow topology.

Workflow key management is handled by an entity that generates a key pair (wpk, wk), splits (wk) following a verifiable secret sharing mechanism (Stadler 1996, Brickell 1989, Shamir 1979), and securely distributes (wpk, K_i) to participants; a threshold K of shares (K_i) can reconstruct wk. Workflow key distribution can either be done through direct messages to each participant over a secure channel or by encrypting each share of the key with the corresponding participant's private key and posting them to the audit server.

This critical role should be given to the participant that has the least incentive to expose wk. This is generally the first or last participant depending on the workflow: a first participant in one workflow can be a travel agency required to keep track of bookings for its customers, and the last participant in another workflow can be a car manufacturing industry that needs to keep track of where parts of customised vehicles, ordered by customers, are from. The adoption of a Publicly Verifiable Secret Sharing (PVSS) scheme (Chor et al. 1985, Stadler 1996, Schoenmakers 1999) enables any participant to

verify that other participants have received authentic shares of the same secret without revealing this secret. PVSS was first proposed by Stadler (Stadler 1996) and used by Schoenmakers (Schoenmakers 1999) for an electronic voting application. For their approach to support key recovery, D'Souza et al (D'Souza et al. 2011) adopted PVSS and explicitly required the secret shared among participants to be the legitimate private key equivalent for a public key. In the context of this thesis, this enables any entity knowing the equivalent public key wpk of the shared secret wk to verify that the key generator was honest with the key distribution, and that the secret key can be reconstructed with a threshold of the shares. Cryptography proofs and details are outside the scope of this thesis. This work only argues the feasibility of this approach in the context of this research.

4.5 System Overview with a Centralised Audit Server

This section discussed the details of AuDiC when using a traditional centralised server. The 'Audit Server' can be hosted by any $node$ or managed by a separate entity, but is not trusted by participants. It is only used to display $audit$ records to participants during the workflow, and not to permanently store any data. For every action, an encrypted $audit$ record is published on the audit server by a participant, and verified by another. Participants check the authenticity of what is displayed on the server, and update their local storage of $audit$ data on a regular basis throughout the workflow.

As shown in Figure 4.6, the proposed approach includes two types of security verifications.

- The *Audit Record Verification* requires every $node$ that receives a $message$ to check it against its equivalent audit record published on the audit server.
- The *Audit Server Verification* requires every $node$ to check for any inconsistency suggesting malicious behaviour from the audit server.

Before going through the details of these verification mechanisms, the data structure used to build the audit trails is presented in the following section.

4.5.1 Audit Data Structure

AuDiC covers logging for every interaction between participants in a workflow. A $message$, sent from a participant to another, contains a $payload$ that includes the $data$ for the intended recipient. Cryptographic operations are performed to assure the authenticity and confidentiality of the transaction. For the simplicity of the discussion, the linear topology shown in Figure 4.6 is assumed. A $message$ sent from C to D in the

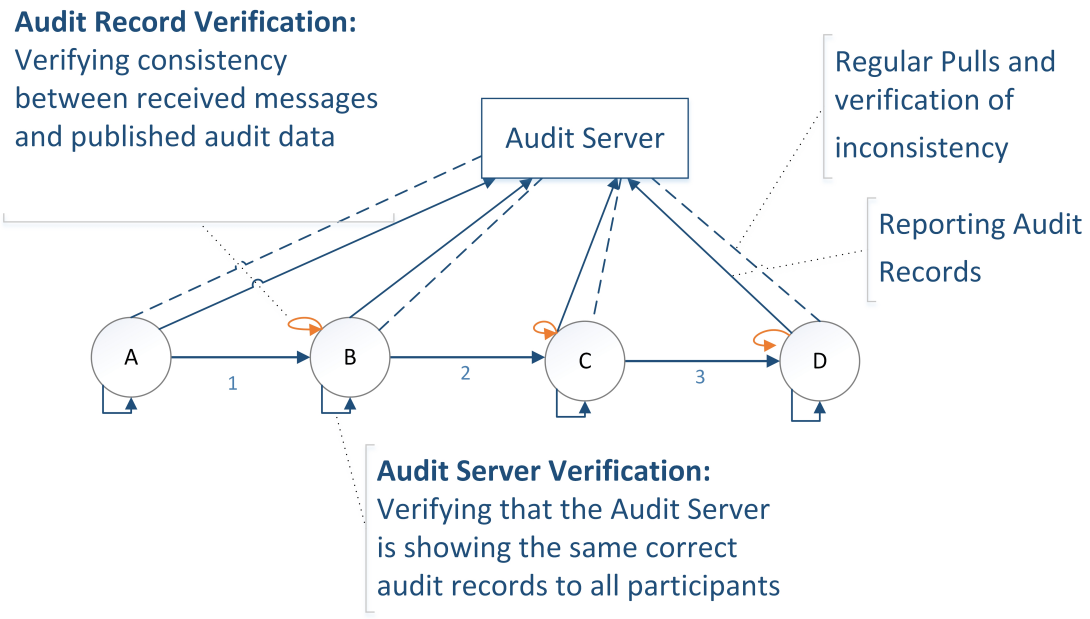

Fig. 4.6 Overview of the Implementation of AuDiC With a Centralised Server

workflow would be of the form

$$M_{CD} = P_{CD}(S_C, E_D)$$

In turn, $audit$ record published by C on the audit server is the same $payload$ of the $message$, encrypted with the workflow public key and signed after encryption:

$$A_{CD} = P_{CD}(S_C, E_{wpk}, S'_C).$$

The second signature helps to verify that the audit server has not tampered with the $audit$ record. Each $payload$ includes $data$ to the receiver, $audit$ records of the previous transactions of the workflow when applicable, as well as *topology* data. In this case

$$P_{CD} = D_{CD} + a_{BC} + T_{CD}.$$

As explained in Section 4.4.2, decryption of an audit record with wk, recovered from a threshold K out of N parts (K_i) of wk, leads to another encrypted one, which in turn gets decrypted with the same key.

4.5.2 Audit Record Verification

Throughout the workflow, participants are required to verify the correctness of the *audit* records equivalent to the *message* that they receive.

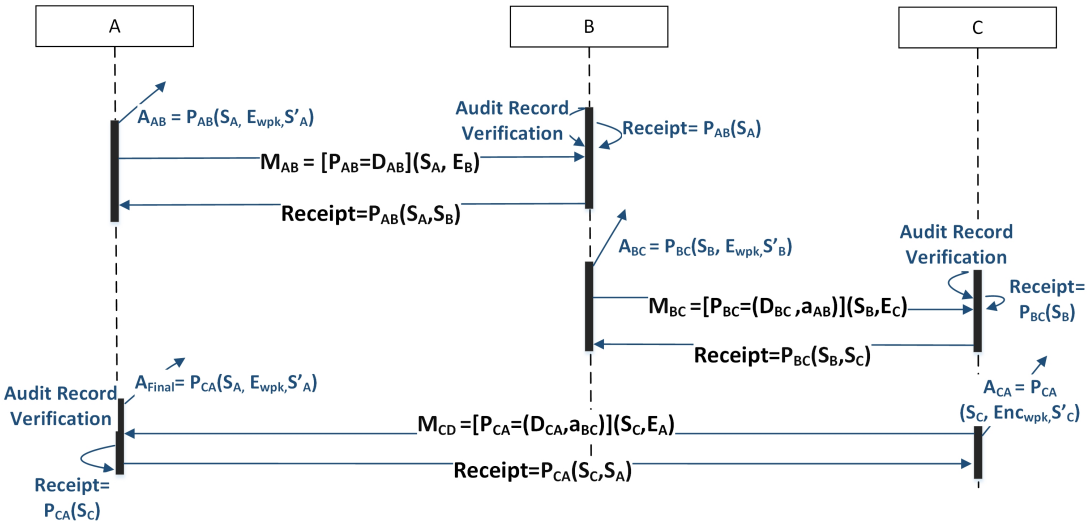

Fig. 4.7 Sequence Diagram for AuDiC on a Workflow that Starts and Ends with Participant A: Upwards Arrows Represent Reporting to the Audit Server

As shown in Figure 4.7, an *audit* record is published by every participant after sending a *message* as well as the last participant after receiving the final *message*. The content of the *payload* is only shown in the exchanged *message* for the clarity of the figure.

A recipient decrypts the *message* with its private key, and verifies the sender's signature. It then performs *Audit Record Verification* shown in Algorithm 2: the recipient updates its storage of audit records from the audit server, and checks if a record of the *message* that it received is displayed; this is done by re-encrypting the *message* with *wpk*, and checking if the cipher version of it is among the pulled records. It checks if the audit records of previous transactions referred to in the *message* are displayed on the audit server. The recipient also verifies that the sender of the *message* did not omit any reference of audit records that should be included, and that these audit records are for the correct previous transactions and signed by the right previous participants in the workflow; this is feasible since every participant knows the topology of the pre-established workflow.

Algorithm 2 Audit Record Verification Algorithm

Requirement: Verify that a received message is correctly reported to the Audit Server

Input: Msg ▷ Message received by participant

Output: Boolean indicating that the audit protocol is being followed

1: $Server\_Recs[] \leftarrow Pull\_Recs()$ ▷ Pull audit records from the server
2: $Msg\_Enc \leftarrow Enc_{wpk}(Msg)$ ▷ Enc is an encryption function
3: **if** Msg_Enc exists in Server_Recs[] **then**
4: Continue
5: **else**
6: Return **False**
7: **if** $Msg.Label = ini \ \ \& \ \ Server\_Recs.size() == 1$ **then**
8: Return **True** ▷ Initialiser of the workflow publishing the first audit record
9: **if** $Msg.Label = [ini, parallel] \ \ \& \ \ VerifyParallelism(Server\_Recs)$ **then**
10: Return **True**
▷ $VerifyParallelism()$ checks if records are published in parallel by checking time between the first and last published audit record
11: **if** $Msg.Para$==$\emptyset$ & $Msg.ParaPrev$==$\emptyset$ **then**
12: Return **False**
13: **else**
14: **if** $Msg.Para$!=$\emptyset$ **then**
15: **for each** $R \in$ M$sg.Para$ **do**
16: **if** $R \notin Server\_Recs[]$ **then**
17: return **False**
18: **if** $Msg.ParaPrev$!=$\emptyset$ **then**
19: **for each** $R \in$ M$sg.ParaPrev$ **do**
20: **if** $R \notin Server\_Recs[]$ **then**
21: return **False**
22: **if** $Msg.Notifications$!=$\emptyset$ **then**
23: **for each** $R \in$ M$sg.Notifications$ **do**
24: **if** $R \notin Server\_Recs[]$ **then**
25: return **False**
26: Return **True**

Back to Figure 4.7, after a successful *Audit Record Verification* the recipient of a $message$ keeps the $payload$ and the sender's signature as a receipt. The recipient then signs over the first receipt and sends this back as a receipt to the sender; this is a proof of delivery. In case any of these steps goes wrong, protocol is to raise concern for malicious behaviour. Another test to verify the authenticity of the audit server is *Audit Server Verification*. Details about this mechanism are covered in the next section.

4.5.3 Audit Server Verification

Audit Server Verification algorithm (Algorithm 3 shown below) is executed by every participant right after sending or receiving a $message$, and on a recurrent basis during the workflow. The frequency of execution is configured depending on the average execution time of a workflow. The aim is to ensure that the audit server is displaying the same authentic workflow $audit$ records to every participant and to distribute these records to participants.

Algorithm 3 Audit Server Verification Algorithm

Requirement: Verify the authenticity of the Audit Server
Update local storage of audit records for participants
Input: $Reported\_Recs$[] ▷ Audit records reported by a participant
$Stored\_Recs$[] ▷ Audit records stored Locally with a participant
Output: Boolean indicating if Audit Server is honest

```
1:  Server_Recs[] ← Pull_Recs()                    ▷ Pull audit records from the server
2:  if Reported_Recs !=∅ then
3:      for each R∈ Reported_Recs[] do
4:          if R ∉ Server_Recs[] then
5:              return False
6:  if Stored_Recs !=∅ then
7:      for each R∈ Reported_Recs[] do
8:          if R ∉ Server_Recs[] then
9:              Return False
10: Local_Recs[] ←Server_Recs[]                    ▷ Updating Stored Records
11: H← Hash(Local_Recs[])                          ▷ Comparing Digests
12: if H==Last Published Digest then
13:     Return True
14: else
15:     Return False
```

Tailoring responses to participants is possible if the Audit Server is malicious. The following mechanisms are designed to target this challenge:

Digest on the fly: when publishing $audit$ records, each participant is required to add a digest of its local audit records storage, including the one that is being published, signed with its private key. Back to Figure 4.7, A_{BC}, in this case, would be published alongside the following

$$Hash(A_{AB}|A_{BC}) + S_C[Hash(A_{AB}|A_{BC})].$$

Referring to Algorithm 3, participants verify that the $audit$ records that they published on and pulled from the audit server are still displayed; they then update their storage of $audit$ records. After that, they compare the $Hash$ value of their stored audit records with the signed digest reported by the last publisher. Different digest values suggest

that the audit server is showing different records to participants. Detecting a malicious activity from the audit server requires an honest participant to publish an $audit$ record after this activity was committed.

Verify at the End: alternatively, the same steps of the Algorithm are followed, except that the hash comparison is only done at the end of the workflow. The final $node$ publishes a signed digest of the records it has, and other participants follow and do the same. Not having identical digest values suggests a malicious behaviour from the audit server.

Combo: This combines both of the previous two approaches.

4.5.4 Protocol of AuDiC with a Centralised Audit Server

This section lists the actions required to reach the security goals of AuDiC:

- Participants keep their receipts, and are required to alert others if a signed receipt of delivery is not received after sending a $message$.
- *Audit Record Verification* is always performed by recipients on $message$ delivery. Failure of verification alerts all participants.
- *Audit Server Verification* is performed by a sender after publishing a record, a receiver when getting a $message$, and by every participant on a recurrent basis during the workflow. Failure of verification alerts all participants.
- Participant keep the $audit$ records that they receive for every transaction in the workflow.

Figure 4.8 is a representation of the state of worflow participants and the audit server in terms of $audit$ records storage at the end of the workflow execution. Each participant receives a copy of the encrypted audit trail displayed on the audit server after verifying the authenticity of each record after generation through *Audit Record Verification*, and challenging the audit server throughout the worflow with *Audit Server Verification*. The encrypted audit trail displayed at the server, possible to decrypt with a threshold of key shares held by participants, is deleted at this stage to reduce unnecessary resource allocation on the audit server.

To follow this protocol is essential in order to maximise the security protection offered through our audit approach, and to reveal malicious activities by entities working on their own or colluding. Potential attack scenarios are discussed in the next section.

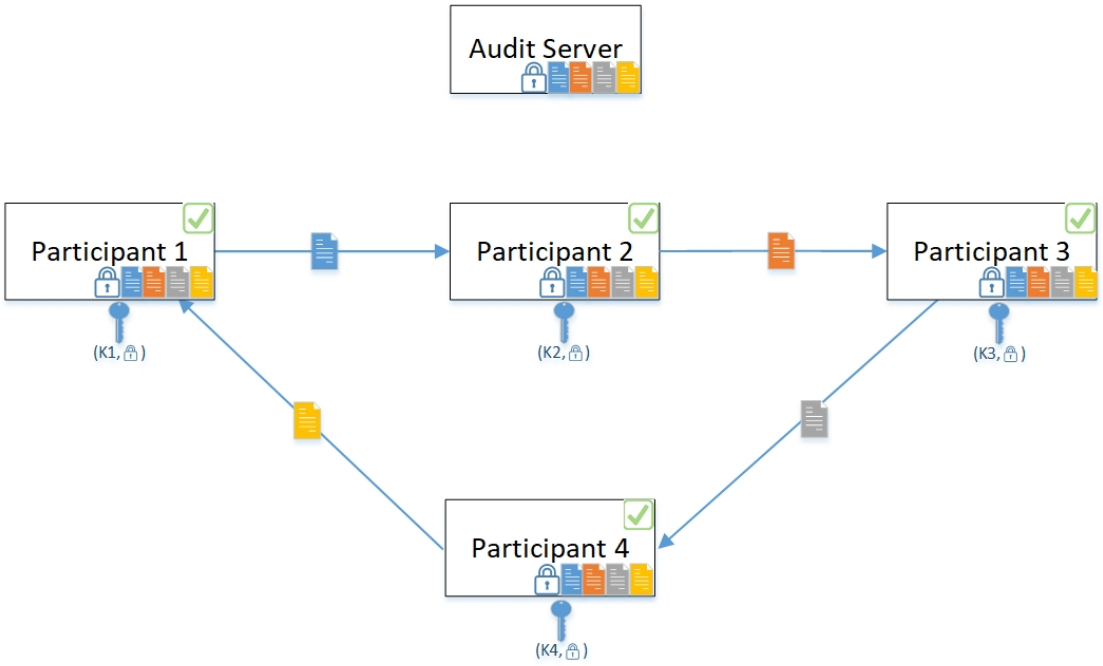

Fig. 4.8 An Overview of the Audit Trails in AuDiC

4.6 Analysis of AuDiC with a Centralised Audit Server

This section presents the attack scenarios caused by a malicious $node$ or a malicious audit server working individually, collusion among participants, and collusion between the malicious participants and the audit server. The way each scenario is handled following AuDiC is analysed and discussed next.

4.6.1 Malicious Participant

Working individually, a malicious participant can attempt to **truncate audit trails**: such participant attempts to publish incorrect $audit$ data for the $message$ it sent, or to send incorrect $audit$ records of previous transactions in a $message$. This is caught by the *Audit Record Verification* performed by the honest recipient of the $message$. Moreover, a malicious sender of a $message$ can publish a false $audit$ record of a $message$ in an attempt to avoid repudiation. This action is detectable by any participant of the workflow given that the topology, and therefore the expected number of exchanged $messages$ are known; receipts with the honest recipient in this case are used to determine the correct $audit$ record.

4.6.2 Malicious Audit Server

A malicious audit server may attempt to hide $audit$ records from all or some participants, or to tamper with these records.

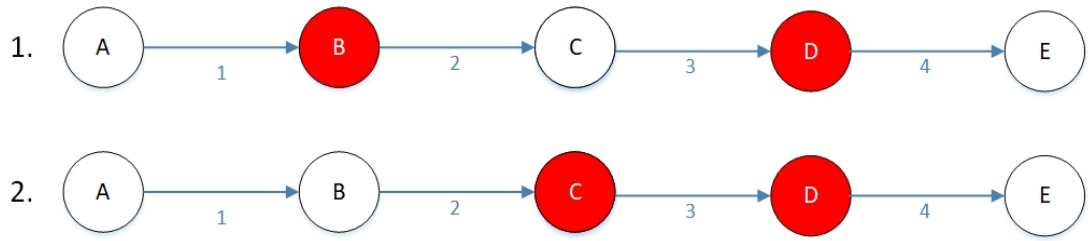

Fig. 4.9 Collusion Between Participants with a Coloured Background

Hiding Audit Records: a malicious audit server can attempt to hide $audit$ records from some participants to limit the distribution of $audit$ data and facilitate the destruction of evidence. This is detected by digest verification of any adopted mechanism.
Tampering with Audit Records: this is detected as soon as an honest node reports and audit record with **Digest on the Fly** or **Combo** mechanisms, or at the end of the workflow when **Digest at the End** is followed.

4.6.3 Collusion Between Nodes

What colluding participants can achieve and the impact of their collusion vary according to their positioning in a workflow. Possible malicious behaviour of consecutive and non-consecutive colluders are discussed.
Non-Consecutive Colluders: One malicious node (**B** for example) in Figure 4.9.1 can report an $audit$ record to the audit server for the other (malicious participant **D**) to use as part of the $message$ sent to the next participant. This attempt to **truncate the audit trail** is detected on the fly by the honest participant performing *Audit Record Verification*. Moreover, the absence of receipts of delivery (from **C** in this case) is another proof of the inauthenticity of the rogue $audit$ record.

Consecutive Colluders: one $node$ can cover for the other by skipping the *Audit Record Verification* phase. This leads to one of the cases below:

- Truncating the audit trail if **C** in Figure 4.9.2 publishes a corrupted $audit$ record or a record not including reference to previous $audit$ data
- Having audit trail with a false record, if **C** reports different data for audit than what it sent to **D**.

$Audit$ records posted or verified by $nodes$ not following AuDiC protocol are unreliable; in this case, $nodes$ **C** and **D** can be exchanging data over unmonitored channels, which eliminates the assumption of A_3 reflecting the entire communication between these two participants. $Audit$ data published by **B** is reliable since **B** has a receipt of delivery signed by **C** to prove its honesty. Honest $node$ **E** also has a receipt, and it makes sure **D** publishes the equivalent $audit$ record of the $message$ it sent on the audit

server. Moreover, following AuDiC protocol, truncation of the audit trail does not lead to loss of $audit$ data covering previous transactions. If there are more than two consecutive colluding $nodes$, the $nodes$ that have followed the protocol and that surround the colluding participants will have credible $audit$ data. Consecutive colluders may appear to be following protocol, but, in some cases, transaction between participants reflect on the ones that follow; a workflow ensuring a unanimous vote on a decision is an example: considering the same linear topology of Figure 4.9, a supporting vote from all members enables **E** to execute a certain decision. Malicious colluders **C** and **D** submit a record for a vote against the decision, while **D** misleads **E** with a claim that all members voted with the decision; this is revealed when decrypting and analysing the audit trail and both **C** and **D** would be held accountable for cheating. In other cases where transactions do not directly and clearly reflect on the next ones, colluding participants will be held accountable for the data they report. A scenario illustrating this case is discussed in Section 4.6.5.1.

4.6.4 Collusion Between Participants and the Audit Server

This section illustrates cases of collusion between participants and the audit server and discusses the adequacy of each verification mechanism. In Figure 4.10.1, malicious node **C** can sign two different versions of an audit record; it relies on the audit server to display the correct record for only **D** and **E**, since they require it for the *Audit Record Verification*, and to display the faulty one to **A** and **B**. This is detected with any *Audit Server Verification* mechanism that is used:

1. If signed digests are published with $audit$ records (**Digest on the Fly**), this malicious activity is caught when node **D**, the first honest $node$ following the malicious activity, publishes its audit data.

2. With **Verify at the End**, the malicious behaviour gets detected when comparing hash values at the end. When comparing audit records, two versions of a record incriminate both the audit server and the publishing $node$.

3. The combination of methods (**Combo**) also detects this malicious behaviour.

In Figure 4.10.2, **Digest on the Fly** does not detect the collusion with the Audit Server since there is not an honest $node$ that follows the malicious one. However, it does not lead to any data loss since all transactions have been covered in previous $audit$ records.

The combination of methods (**Combo**) combines the advantages of the two verification mechanisms; Figure 4.10.3 is a scenario where **Combo** method is a good option:

– Attempts from **C** and the audit server to show **A** and **B** different versions of A_{CD} than the one sent to **D** is detected on the fly.

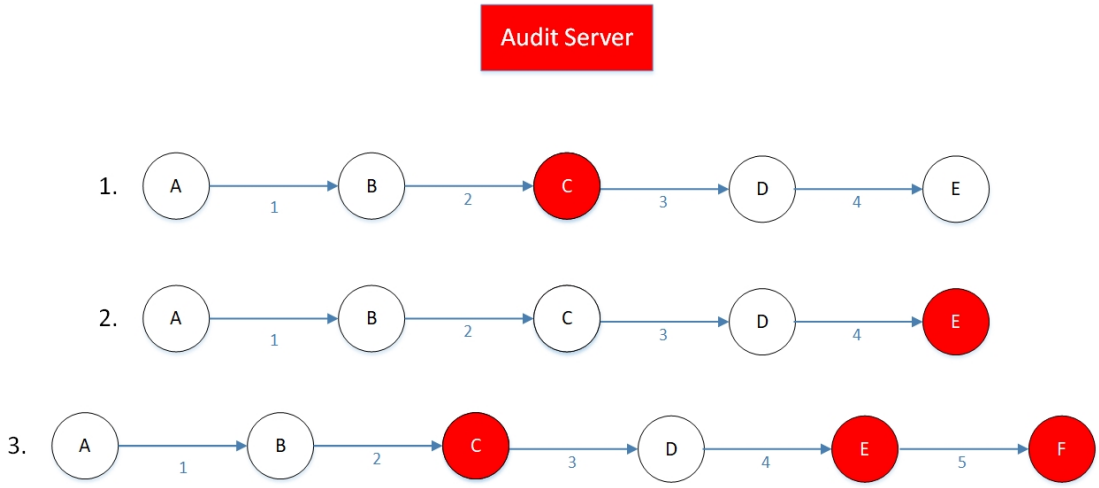

Fig. 4.10 Malicious Audit Server Colluding with Participants

– **E** and **F** cannot collude with the audit server to show honest participants different $audit$ data.

4.6.5 Representative Scenarios

AuDiC is application-agnostic, and can be applied on any workflow. In this section, scenarios that reflect on the previous analysis are presented.

4.6.5.1 Supply Chain Scenario

The scenario is inspired from Weber et al. (Weber et al. 2016) and Figure 4.2 shows their representation of the workflow; this was presented earlier in Section 4.4.1 of this chapter. The following conflict example is also presented in their work: the Carrier delivers eight pallets instead of ten to the Manufacturer three days after the expected delivery date. The Manufacturer complains to the Supplier who argues that this is what was ordered by the Middleman, and the Middleman claims that the fault to be on the side of the Supplier. The Manufacturer refuses to accept the delivery and asks the carrier for another trip to the supplier. The Carrier is now eligible for a compensation for the additional journey; the Manufacturer also requires a compensation, part of which goes to the Buyer for the delay in the order. Either the Middleman or the Supplier should issue these compensations, and neither of them admits responsibility for the fault. AuDiC is suitable to resolve this conflict and to prevent the responsible entity from tampering with their logs, or from colluding with a centralised audit service to frame the other innocent party.

Figure 4.11 represents scenarios in which AuDiC is adopted with malicious participants trying to avoid repudiation. Figure 4.11.1 consider the middleman **MM** to be the malicious participant responsible for the conflict described above. During the workflow,

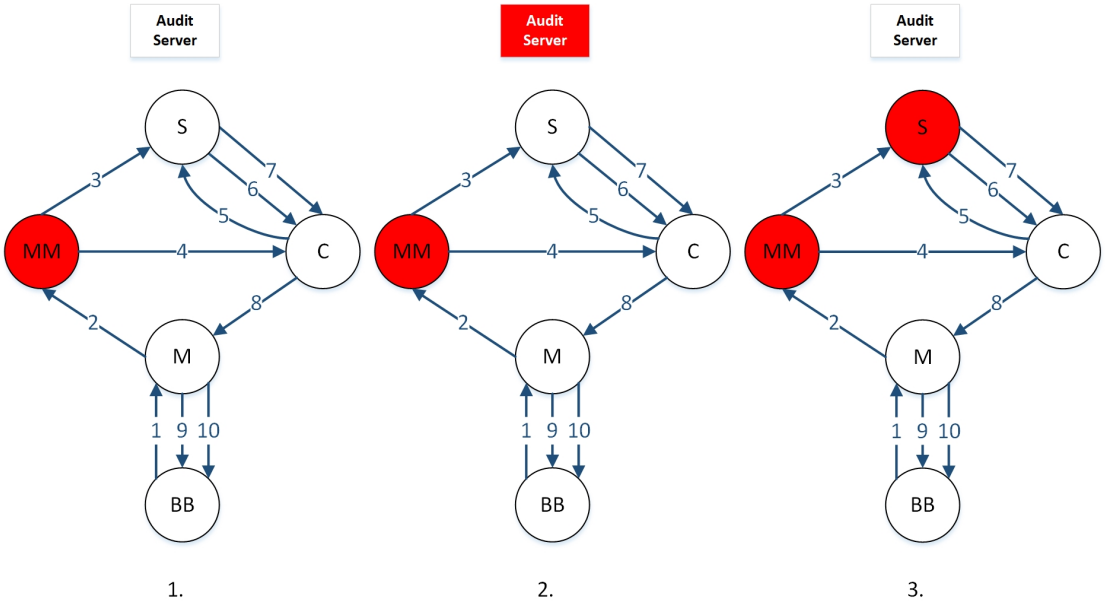

Fig. 4.11 Malicious Activities When AuDiC is Followed. Case 1 Shows a Single Malicious Entity, Case 2 Shows a Collusion Between an Entity and the Audit Server, and Case 3 Shows a Collusion Between two Participants

MM can try to report an inauthentic $audit$ record in an attempt to frame the supplier **S**. **S** will detect this attempt on the fly as discussed in Section 4.6.1. **MM** can also try to blame the manufacturer **M** by deleting the records it has after the workflow execution; **M**, as well as other $nodes$, would have a copy of the $audit$ records in this case. Figure 4.11.2 considers a case in which **MM** colludes with the audit server in an attempt to display the correct $audit$ record to **S**, but not to other participants; this attempt is detected by any honest $node$ regardless of the adopted server verification mechanism as discussed in Section 4.6.4. Figure 4.11.3 represents the only scenario in which reporting an incorrect $audit$ record goes undetected following AuDiC approach. In this case, accountability is assigned following the data in the $audit$ record A_3; this is a limitation in roach, as well as other approaches for audit that cannot guarantee capturing all communication between colluding $nodes$ able to communicate through unmonitored channels. Colluding participants may attempt not to publish an $audit$ record for transaction 3; a missing record is detected by honest participants on the fly given that our workflows are pre-established. An extreme case is when **MM**, **S** and the Audit Server collude with each other. Colluding participants can agree to produce two versions of the A_3 in an attempt to distribute different audit trails to different participants. Following any verification mechanism for the audit server, any two honest participants can reveal if they have different versions of an $audit$ record. A collusion between **MM** and any other participant is also detectable on the fly.

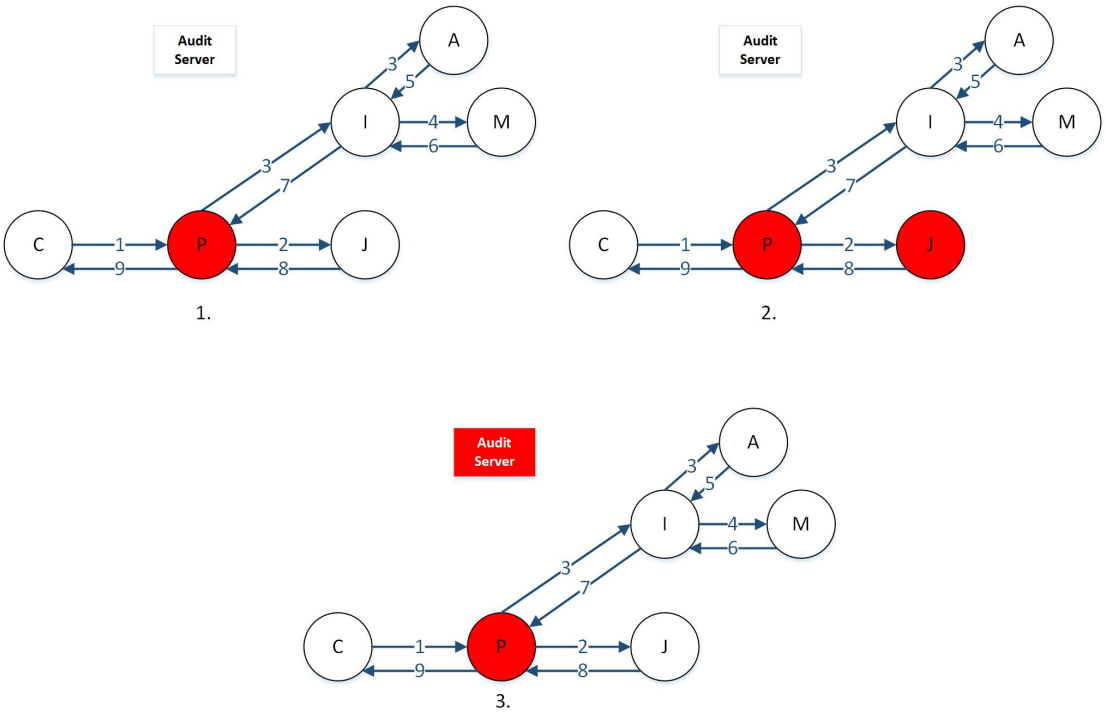

Fig. 4.12 A Graph Based Representation of the Scenario of Applying for a Password

4.6.5.2 Applying for a Passport

Revisiting the scenario of applying for a passport, presented in Chapter 3. A citizen authenticates through a central government portal **C** and applies for a passport through a service **P** in the Department of State. **P** requires identity information from the Department of Interior **I**, and an attestation of a clean criminal record from the Department of Justice **J**. In Figure 4.12, **A** and **M** are (micro)services within the Department of Interior administrative and security domain. A conflict scenario occurs if a citizen with a criminal record manages to get a passport; in this case, **P** can claim to have received an inauthentic electronic record from **J**, and the Department of Justice denies any wrongdoing. It is the word of one department against the other here. Note that while the contribution of this research in Chapter 3 mitigates the successful manipulation of requests by a malicious insider, this contribution does not give any assurance that organisations are honest and acting in good faith. Also, while the approach for fine-grained access control verifies the authenticity of requests at the level of each service, the authenticity of the response of these services cannot be verified. Let's suppose that a malicious employee in the passport department is behind manipulating the request of service **P** to share the wrong record. Following traditional methods for auditing, the latter can collude with the entity managing the logging system to tamper with the logs and cover the tracks of that employee, or with an insider in the Department of Justice that can tamper with their logs. In contrast to traditional means to record digital evidence, AuDiC offers protec-

tion against collusion to tamper with evidence after the fact, and produces reliable audit records to assign accountability.

The same adversarial scenarios, discussed in Section 4.6.5.1 and represented in Figure 4.11 apply in this case. This scenario is revisited and the details of the adversarial model are discussed in the next chapter. Note that a collusion between the citizen using **C** to initiate the passport application request and the malicious employee in **P** does not imply that **C** is part of the collusion. The reason being that the rogue citizen only interacts with the client through an interface, and cannot therefore affect the functionality of **C**.

4.7 Implementation and Evaluation with a Centralised Audit Server

A proof-of-concept that includes all the functionalities and verification mechanisms proposed in this chapter was implemented. Referring to Figure 4.1, the code for the participants as well as the audit server tier were developed as part of the contribution of this research.

The audit server, as well as the workflow participants were developed with Java. For the audit server tier, the server exposes an API enabling participants to upload $audit$ records alongside the $Hash$ of their local records. Following the implementation in this research, the $Hash$ value of the local records is always published alongside the $audit$ record, allowing to fulfil the **Combo** verification mechanism discussed in Section 4.5.3. The server exposes another endpoint enabling participants to pull the records displayed on the server.

As for the workflow participants tier, the implementation included the construction of the data structure described in Section 4.4.2 to build $payload\ P_i$, $message\ M_i$ and $audit$ records A_i following the specifications in Section 4.5.1 for every $edge\ e_i$ of an arbitrary graph representing a workflow topology. *Audit Record Verification* and *Audit Server Verification* are also implemented at the level of workflow participants. Workflow participants encapsulate the same logic but run on separate Java virtual machines, and they expose APIs enabling their interactions with each other.

4.7.1 Load Emulation of the Audit Server

In this section, a number of workflows relying on the same audit server were simulated. For this evaluation, a linear topology was used for $nodes$ exchanging $messages$ while following the protocol of AuDiC. The traffic of requests to the Audit Server is considered to be log-normally distributed (Paxson 1993, Goseva-Popstojanova et al. 2006). To simulate server load, a log-normally distributed delay of the form $e^{\mu+\sigma Z}$ was introduced to the Audit Server's response time. The figure below shows the average processing time for each case. While the audit server processing power is inversely proportional to

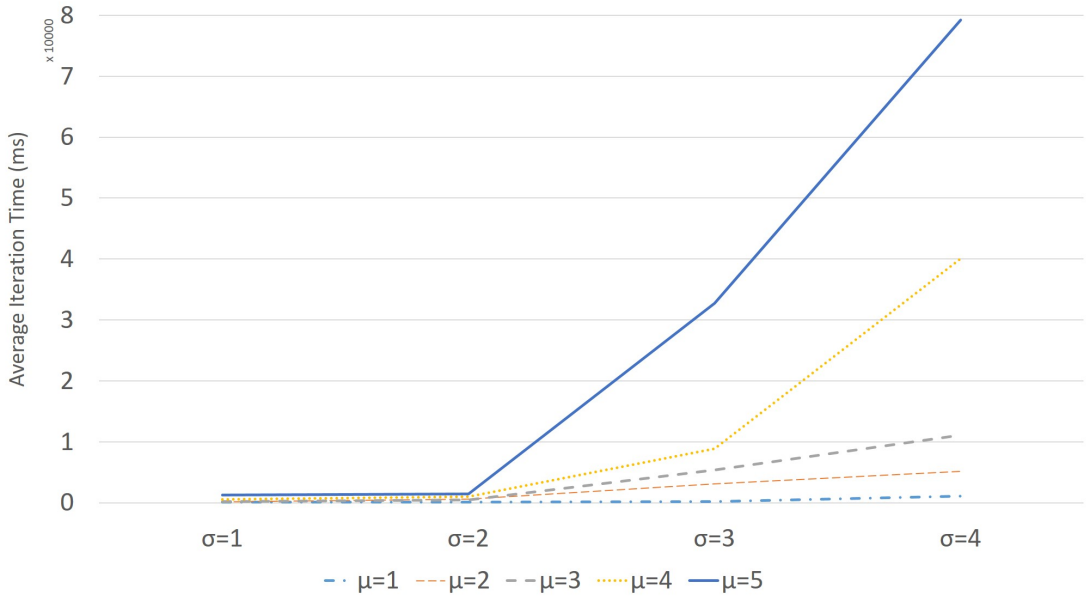

Fig. 4.13 Average Processing Time for Different Log-normally Distributed Delays

μ, σ reflects the multitude of workflows running simultaneously. Following this evaluation method, changing the number of $nodes$ and the size of the exchanged $payload$ resulted in graphs with different scales for the Average Iteration Time, but with similar slopes for the lines. Comparing the slope of curves in Figure 4.13, servers with high computational power show stability and reliability at scale. Less powerful servers can still be used for systems that can afford latency at busy times.

4.7.2 Performance Evaluation with the Centralised Audit Server

AuDiC offers the flexibility of logging evidence to any degree of details. When data exchanged in transactions is too large to be recorded in full in an $audit$ record, participants have the option of agreeing on an audit recording approach that covers enough information to ensure non-repudiation; this flexibility of recording evidence helps to optimise the disc space required for storage of audit trail with every participant.

This section discusses the evaluation of the implementation with different sizes of $Data$ in participants' transactions; for this evaluation, $Data$ is recorded in full in each equivalent $audit$ record of a transaction. The processing time of a transaction, which includes cryptography operations on the $payload$ at the sender and recipient side, data propagation time, reporting and verifying the equivalent audit record of the $message$ following AuDiC protocol is affected by a number of attributes. The size of the $payload$ to be sent, the size data pulled from the audit server and the load on the audit server are factors affecting the processing time. After emulating a busy server in the previous section, this evaluation shows the processing time of transactions following AuDiC

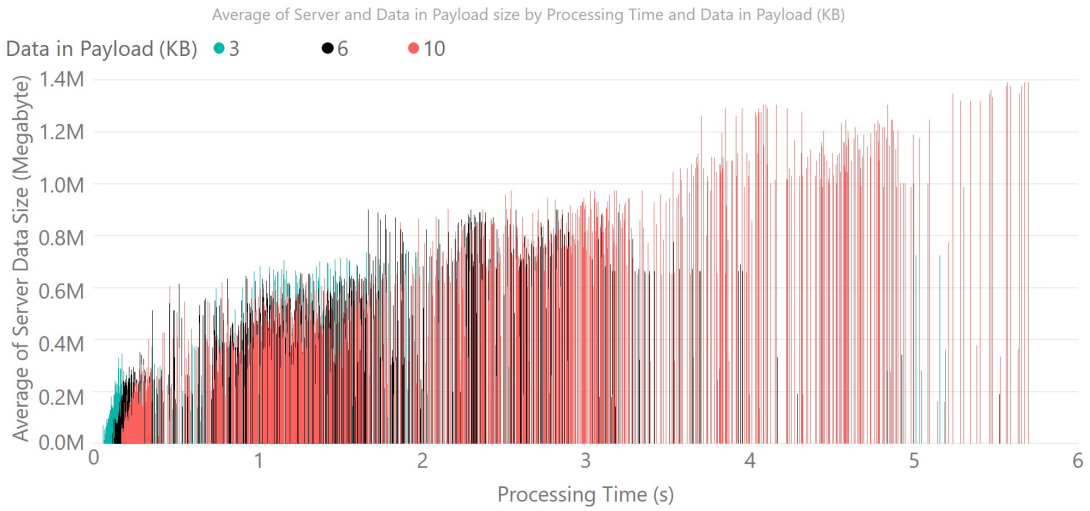

Fig. 4.14 Processing Time of AuDiC with Respect to the Size of the Payload and the Server Data Size on Topologies of 15 Nodes

protocol on different workflows topologies and $payload$ sizes.

To give a realistic evaluation of the application-agnostic approach proposed in this chapter, the implementation was tested on topologies generated with a topology generation tool. Brite[1], a topology generation tool developed at Boston University, was used to generate workflow topologies of 15 and 20 participants. To keep some randomness in the topologies by avoiding a fully connected graph, a limit for the number of $edges$ was set to 95 with 15 participants and 145 for 20.

For a fair and consistent evaluation, the results are separated according to the number of participants; this gives comparable computational resource allocation for each iteration of the experiment after dedicating the resources required to bootstrap participants and the audit server. On average, the time required to process a $message$ with 15 $nodes$ is 346, 954, and 1784 milliseconds for transactions containing 3, 6 and 10 Kilobytes of $data$ respectively; equivalent values for 20 $nodes$ are 650, 2693 and 4635 milliseconds. Figure 4.14 and 4.15 illustrate the results with 15 and 20 participants. The size of a $data$ evidently reflects on the $message$ processing time as well as the memory required by the audit server to display $audit$ records.

In each of these figures, the high density of the colour representing a $data$ size suggests frequent occurrences of records within a range of processing time values; the three evaluated data sizes with both topologies show a cluster on the first 500 ms when the pulled data from the server is of a small size (less than 400 KB). Processing times of 3 KB transactions with 15 and 20 $nodes$ spread between 100 and 2000 ms; with 6 KB, most frequent response time values are within the range of 1 and 4 seconds with 15 participants, and 1 and 8 seconds with 20. More distributed results show with 10 KB

[1] `https://www.cs.bu.edu/brite`

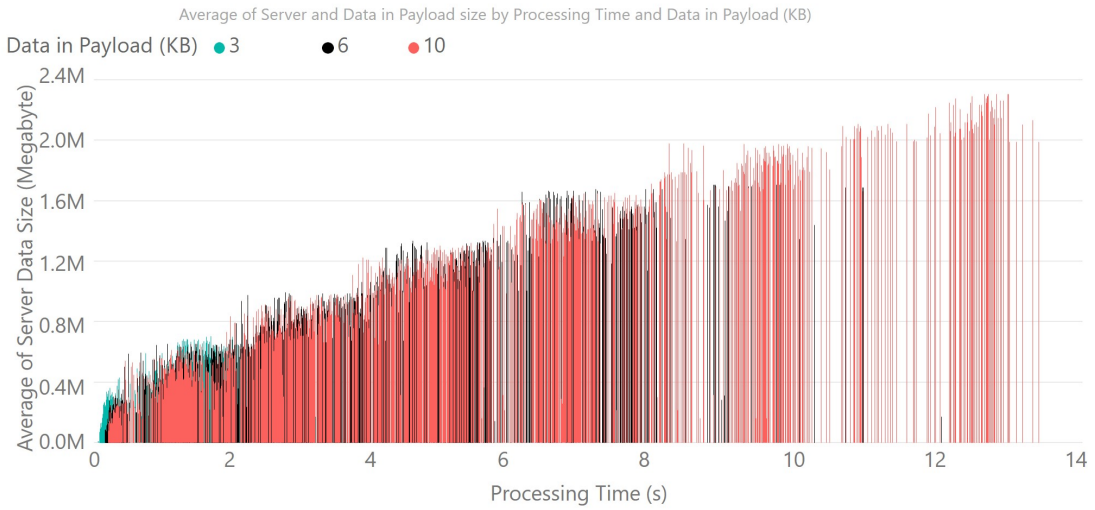

Fig. 4.15 Processing Time of AuDiC with Respect to the Size of the Payload and the Server Data Size on Topologies of 20 Nodes

with the majority of values ranging between 1 and 6 seconds with 15 $nodes$, and take up to 13 seconds with 20 $nodes$. The figures shows a linear relationship between the processing time and the server data size. The trend is more visible in Figure 4.15, which reflects more records than Figure 4.14 due to the feasibility of having more connections between $nodes$ compared to smaller topologies.

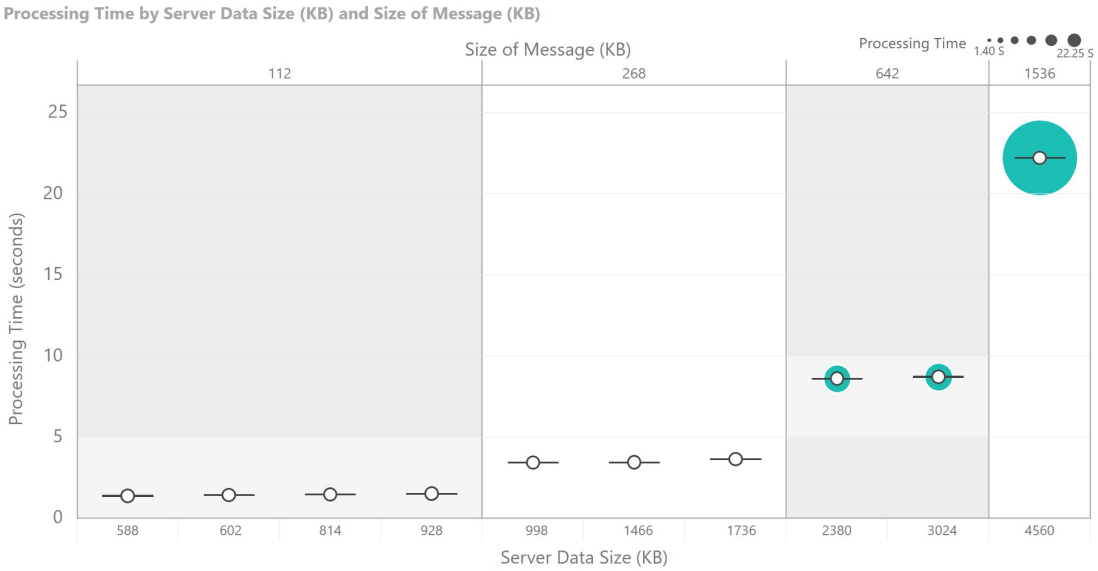

Fig. 4.16 Processing Time with Respect to the Size of Messages and Server Data Size

Workflow transactions and audit trails are normally in terms of bytes and kilobytes respectively (Rimba et al. 2017, Pulls et al. 2013); even though files (PDFs for example) are useful to be exchanged in some workflow cases, the data included in audit records can be limited to logs similar to those recorded by workflow engines. Nevertheless, if needs be, files can be serialised and included in the audit as part of the audit trail. The implementation of this approach was also tested on larger $payloads$, as visualised in Figure 4.16; this figure shows part of an evaluation of a topology of 15 participants, and is obtained through an incremental increase of the size of transactions throughout the workflow execution. These results are specific to the implementation that was done during this research and they can be enhanced by optimisation of the code.

4.8 Conclusion

This chapter of the thesis covered AuDiC, a collaborative and trustless approach for auditing of workflows. Compared to common auditing practices, this approach helps to enhance the availability, as well as the integrity and confidentiality of digital evidence during generation and at storage phases. Collusion to hide or tamper with audit data is detectable, and the chance of withholding evidence is reduced due to the audit capability of any K out of N participants.

AuDiC is application-agnostic and enables capturing digital evidence to any degree of details. While the first aspect increases the practicality and applicability of this approach to different domains that require a collaboration of multiple parties, the second one gives assurance for non-repudiation. Captured data covering interactions between participants can include, but are not limited to, suggestions and decisions, requests for actions and data access requests, allocation of tasks and approvals over budgets, prescriptions, documents and studies, etc. In addition to that, AuDiC enables recording any other information deemed useful for a specific application domain, including access tokens, IP addresses, timestamps, etc. A number of scenarios were used to represent examples of application domains that require the collaboration of different organisations. While some applications have more relaxed security requirements than others, the tight measures that are offered help with the applicability of the approach in domains in which the confidentiality of digital evidence is critical to preserve the privacy of participants, and the availability and integrity of audit records is required to assign accountability for wrongdoing.

The centralised audit server is a single point of failure in the implementation covered in this chapter. While this server is only used to display $audit$ records during a workflow execution and recommend wiping its content after the end of a workflow, this server remains a bottleneck; this can lead the workflow to hang when it is overloaded. To overcome this limitation, blockchain is explored as an alternative for the implementation of the audit server in AuDiC. Although its highly distributed nature makes it robust and highly available, blockchain requires special considerations due to its method of operation. The adoption of blockchain for an alternative implementation of the audit server is discussed in the next chapter.

CHAPTER 5

A Blockchain-Based Implementation of AuDiC

The previous chapter presented AuDiC and discussed an implementation that includes a centralised audit server. This chapter elabirated on the discussion of auditing approaches offered in the literature to cover blockchain related approaches and present an alternative implementation of AuDiC using blockchain to develop a decentralised audit server. A protocol that makes use of the security features of blockchain while considering its limitations is discussed, and the findings are presented in the remainder of this chapter.

5.1 Introduction

Blockchain, as the name suggests, is a timestamped list of data blocks containing a defined number of transactions each and chained together with cryptographic ties (Weber et al. 2016); this list is hosted and maintained in a distributed way with a large network of machines, referred to as nodes, which record, share and aggregate data about transactions (Weber et al. 2016, Calvaresi et al. 2018). Blockchains are designed to offer an immutable append-only register possible through the verification of the blockchain nodes of every transaction or block added to the chain (Calvaresi et al. 2018); in other words, a new transaction can only be added to a block and a new block can only be added to the chain through the consensus of all or a majority of the nodes being part of the blockchain network (Tian 2017, Nofer et al. 2017); also, a node cannot try to change existent data of the blockchain without being flagged as a potential threat (Tian 2017). Some blockchains support smart contracts, which are user-defined programs written in a programming language specific to the blockchain technology (Calvaresi et al. 2018, Weber et al. 2016). Similar to blocks and transactions, these contracts cannot be modified once added to the blockchain. Smart contracts are executed on the blockchain, making its outcome the subject of consensus between the blockchain nodes (Tian 2017, Weber et al. 2016). Blockchains improve the transparency and security in decentralised systems, but are not designed to store large amounts of data (Tian 2017); this is due to having identical copies of the full ledger stored, maintained and validated by different members of the blockchain network (Ahmad et al. 2018, Sullivan and Burger 2017).

The implementation of AuDiC, described in the previous chapter, relies on a centralised server acting as the audit server. This makes the scheme prone to halt or fail with the failure of this single point. Failure of nodes is addressed in blockchain given that the network is resilient and remains functional even when a number of nodes fail (Nofer et al. 2017). In contrast with approaches built around blockchain technology, the optional use of blockchain with AuDiC offers an elegant solution to replace the centralised implementation of the audit server. To overcome the limitations of relying on a centralised server, this chapter investigates blockchain as a distributed infrastructure to base the implementation of the audit server on. Blockchain, having its own limitations, requires some changes in the protocol proposed in the previous chapter. Section 4.4 is referred to for the common aspects of our auditing approach between the two implementations, as well as the notation that is used in this chapter; new practices and notations are also introduced and discussed. This chapter follows a similar scheme of the previous one to facilitate spotting the differences between the two implementations. The goal of the auditing scheme remains to minimise the effect of collusion related threats to tamper with, breach the confidentiality of, withhold or destroy $audit$ records.

5.2 Blockchain for Auditing: State of the Art

The discussion of related work in this chapter is limited to approaches that require the use of blockchain for auditing. Other auditing related approaches that use conventional centralised servers are discussed in Section 4.2 of the previous chapter. A number of blockchain-based approaches assume trust for the entity generating audit records. Cucurull and Puiggalí (Cucurull and Puiggalí 2016) challenge the storage entity with checkpoints published on a Bitcoin blockchain reflecting the integrity of the logs prior to the time each checkpoint is recorded; however, tampering with logs is possible between the checkpoint intervals. Putz et al. (Putz et al. 2019) target this limitation by enabling the verification of the integrity of each log entry through hashes published on a permissioned blockchain. They verify that individual log records, collected from different organisations, have not been modified since generation. They also replicate their audit data to ensure its availability, but trust the entity storing their logs with the confidentiality and privacy of their audit records. Tian (Tian 2017) uses blockchain with distributed databases to track a food supply chain process. Each participant in the supply chain generates and maintains audit records of its part of the process, and submits a proof of authenticity of the records they have to the blockchain. Lu and Xu (Lu and Xu 2017) present another application of blockchain enabling the verification of the originality of products in a supply chain. Ahmad et. Al (Ahmad et al. 2018) use blockchain to verify audit logs generated by an Online Transaction Processing Systems in a cen-

tralised database. Tang et. al. (Tang et al. 2018) propose uploading a digest of their logs, stored on the cloud, to a blockchain to enable the verification of the integrity of their audit records after generation. The approaches presented above reflect the common security assumption of trusting the entity producing digital evidence and using blockchain to verify that records have not been modified after generation. The verification of the correctness of the produced evidence is not covered in these approaches.

Other approaches do not assume the same trust level for the logging entity; they either rely on blockchain's smart contracts to produce their audit trails, or verify the authenticity of audit records during generation through the blockchain. Tapas et al. (Tapas et al. 2019) do not assume trust during their generation of logs, and rely on mutual challenges between two parties to verify the authenticity of evidence reported to cover interactions between them. They follow a blockchain-based approach that supports the verification of basic operations between a client and a cloud service provider storing data, and do not consider workflows including multiple administrative domains. Weber et al. (Weber et al. 2016) proposed using blockchain to solve the lack of trust in the execution of business processes. Smart contracts are used to check if interactions are conforming to the choreography model and to control the logic of their business processes, and audit trails are generated from contracts execution; however, their model requires the data to be in plaintext at the level of their smart contracts. In their approach, Suzuki et al. (Suzuki and Murai 2017) suggested that interactions between two entities happen through a blockchain, and that the data area of blockchain transactions is used for data exchange between participants; although this approach enables encrypted communication, cost and scalability are obvious limitations. A lightweight and confidentiality friendly approach has not been covered in the literature to verify the integrity of audit trails in a workflow combining different domains.

The most prominent e-government framework using blockchain to support auditing is X-Road. This framework was developed by Cybernetica for the Estonian e-government, and is fully or partially used in other countries including Sweden and Azrebijan (Hartmann and Steup 2015), Finland (Priisalu and Ottis 2017), Haiti, Nambia and the United Kingdom (Freudenthal and Willemson 2017). X-Road architecture relies on distributed security servers installed at the level of each party that is approved to participate in data exchange; these security servers connect to a central component that authenticates participants and plays the role of an exchange layer (Pappel et al. 2017). X-Road aims to keep centralisation at a minimum, yet exchanged data goes through a central middleware referred to as the Document Exchange Centre (Pappel et al. 2017). Logging for data exchange is performed centrally (Hartmann and Steup 2015, Kütt and Priisalu 2014), in addition to logging messages at the level of participants with security servers (Freudenthal and Willemson 2017). Therefore, X-Road presents a combination of a central and distributed approach to record audit data. To protect the integrity of

audit records in Estonia, X-Road reports hashes of activity logs to a private blockchain, developed by Guardtime (Priisalu and Ottis 2017, Robinson and Martin 2017, Calvaresi et al. 2018, Martinovic et al. 2017). This blockchain is hosted in Estonian government network and adding blocks is done without a mining process; trust is assumed for blockchain nodes in the Estonian government context (Calvaresi et al. 2018). However, this audit approach leaves room for participants in the data exchange to collude with the entity that manages X-Road to tamper with logs of transactions prior to reporting the hash to the blockchain (Martinovic et al. 2017).

The proposed auditing approach in this thesis is an improvement over existent auditing systems that workflows spreading across multiple parties rely on. Compared to blockchain-based approaches that depend on a trusted entity to generate audit records for workflows, AuDiC enhances the privacy protection of workflow participants by not exposing their transactions at the level of this entity, and eliminates the need to trust a single entity with the audit records generation. Compared to frameworks that rely on each participant to generate audit records for their contribution in the workflow, AuDiC offers a means to verify these records. Moreover, compared to existent audit systems, AuDiC minimises the chance of successful collusion with the entity generating the audit trails to avoid repudiation and is an improvement for the protection of the availability as well as the confidentiality of audit records at storage.

5.3 Blockchain for Auditing of Workflows: Problem Statement

Compared to common practices that use blockchain for auditing of workflow collaborations spreading across multiple organisations, the blockchain-based implementation of AuDiC offers mitigation for one or multiple problems of the following:

- Destroying or Withholding Evidence
- Breaching the Confidentiality of Transactions
- Reporting Inauthentic Audit Records

In addition to that, this chapter aims to adjust a known limitation of the previous implementation: the centralised server is a single point of failure. Replacing this centralised with a blockchain-based implementation solves this limitation, but introduces new ones related to the use of blockchain:

- Blockchain transactions are public and permanently stored.
- Blockchain is not designed to host large amounts of data.

Therefore, in addition to the list of problems covered in the previous chapter, privacy challenges imposed by blockchains' methods of operation are considered while taking

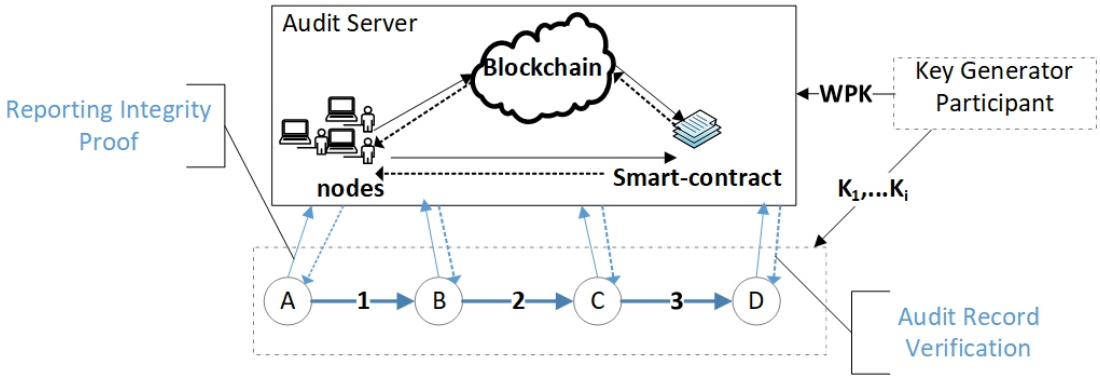

Fig. 5.1 Overview of Blockchain as an Audit Server in AuDiC

into account their data storage limitations. This contribution also offers a reliable audit approach that protects the confidentiality and integrity at generation and storage phases, as well as the availability of evidence.

5.3.1 Threat model with a Blockchain-Based Audit Server

Similarly to the threat model of the previous chapter, actions of participants are assumed to reflect the intentions of each organisation, and not to be the result of a compromise making it under the control of an attacker. Certificate authorities are honest, and participants are entrusted with their keys. The adversary model includes workflow participants colluding with each other in an attempt to avoid repudiation. Blockchain nodes are assumed to be distributed, operated by different organisations and spread across multiple locations, which eliminates the option of controlling a majority of blockchain nodes. The smart contracts are also assumed to be free of vulnerabilities that can alter their correct behaviour. Communication between participants and the blockchain is conducted over a secure channel. Cryptography keys, including participants' and workflow keys, are assumed to be generated following best security practices to minimise the chance of reversing the encryption.

5.4 System Overview with Blockchain-Based Audit Server

This chapter presents a blockchain-based implementation for the audit server of AuDiC. Same as the previous approach, an *audit* record covering any degree of details of a transaction in a workflow is generated by the sender, and the recipient verifies that this record corresponds to the message that it received. However, blockchain's method of operation requires different considerations than a traditional server. Considering blockchain limitations, some modifications are introduced to the protocol of the previous chapter. A comparison between the two implementations is presented in the next section.

5.4.1 Key Differences With the Centralised Audit Server

Figure 5.1 represents an overview of AuDiC with a blockchain-based implementation of the audit server. In contrast with the implementation relying on a centralised server, the functions of the two types of audit servers are different from the perspectives below:

- Blockchain is trusted following the threat model, which eliminates the need for *Audit Server Verification*: while the entity managing a centralised audit server can tamper with the records or display different content to different participants, the distribution of blockchain nodes across multiple physical locations and administrative domains gives a high level of assurance of the authenticity of what is displayed on a blockchain; in other words, the content that one participant gets from the blockchain can be trusted to be the same as the others get.

- Blockchain is immutable, and a copy of its data is replicated across its network of nodes: a $Hash$ of $audit$ records, referred to as *integrity proof*, is only displayed rather than the encrypted records on the Blockchain. This affects the role of the audit server as a means to distribute $audit$ records to all workflow participants, which requires assigning this task to the recipients after verifying the equivalent $audit$ record of the messages they received. Even though a $Hash$ value does not reveal any information about its equivalent $audit$ record, the address of the participant publishing this record shows on the Blockchain.

- Blockchain is trusted to execute *Audit Record Verification*: given that a centralised server is managed by a single entity, the execution of the proposed security checks on the server cannot be trusted. Through its smart contracts, represented in Figure 5.1, Blockchain gives a higher level of assurance for the authenticity of code execution than traditional servers due to the transparency of blockchain and the distributed execution of these contracts by its nodes.

- Blockchain is used to display participants and workflow public keys: given that any data displayed on a centralised audit server, including cryptography keys, can be challenged following the same approach used for audit records with *Audit Server Verification*, a centralised server can be relied on to display workflow and participants public keys during workflow execution. However, this server is only used to display data during a workflow execution, and storing cryptography keys on the server to be displayed for different workflow executions is not recommended. In contrast to a centralised server for which wiping the content of a specific workflow execution is recommended after it terminates, content cannot be deleted from a blockchain. A positive aspect of the immutability of blockchain is the option of displaying public keys of participants making the blockchain act

as a certificate authority. Similar to the implementation with a centralised server, workflow key generation is assigned to the entity that has the least incentive to expose the workflow private key; the same options for key distribution apply here, with direct messaging being a preferred option to minimise unnecessary transactions and data storage on the blockchain.

5.4.2 Audit Data Structure

Considering the linear topology of Figure 5.1, exchanged messages between participants follow the same structure used in the centralised implementation approach where:

$$M_{CD} = P_{CD}(S_C, E_D)$$

and

$$P_{CD} = D_{CD} + a_{BC} + T_{CD}.$$

Same as the previous implementation approach, $Payload$ includes the $data$ to be sent in addition to a reference of previous $audit$ records and *topology* data.

Compared to the centralised audit server implementation, and following best practices that suggest not to store large data on blockchain, *integrity proofs* of every $audit$ record are used as payload to be uploaded to the blockchain with every transaction instead of the full $audit$ record. In this sense, and considering the linear topology of Figure 5.1, an audit record A_{CD} is generated and stored by the sender of the message of the form

$$A_{CD} = P_{CD}(S_C, E_{wpk}).$$

This sender also produces an *integrity proof* of the message of the form

$$I_{CD} = Hash(A_{CD})$$

and submits this record to the blockchain through a smart contract. Blockchain keeps a record of entities triggering smart contracts; in this sense, the participant that uploads *integrity proofs* to the blockchain is identifiable, which eliminates the need for the external signature suggested for the previous implementation of the approach to keep track of the state of the workflow.

5.4.3 Audit Record Verification

Figure 5.2 shows the steps followed to verify an $audit$ record with a blockchain-based implementation of the audit server. The sender of the $message$ generates the equivalent audit record of this message and uploads its *integrity proof* to the blockchain. When receiving the $message$, participant B stores the signature as a receipt, decrypts the

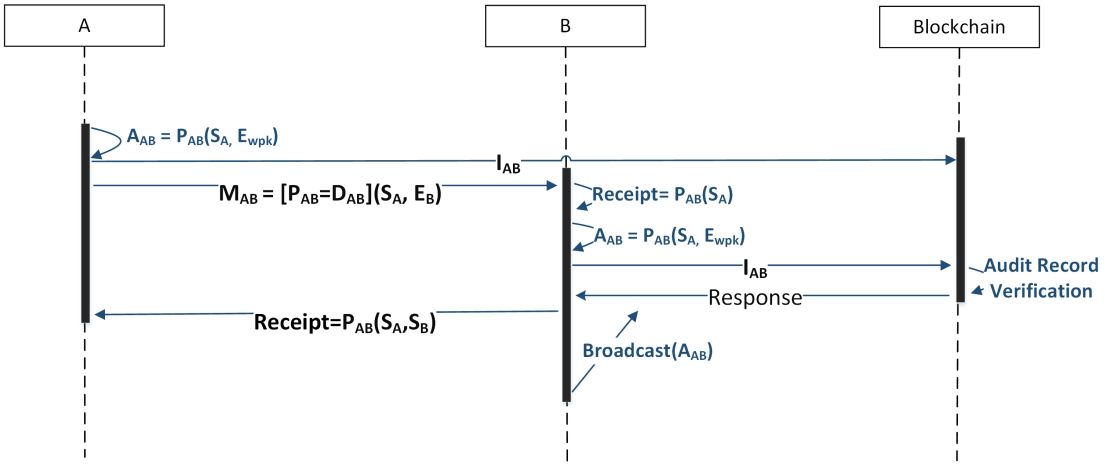

Fig. 5.2 Sequence Diagram Representing AuDiC Protocol with a Blockchain-Based Implementation of the Audit Server

$message$ to obtain the $payload$, and regenerates the $audit$ record of the $message$ by re-encrypting the $payload$ that it received with wpk. B stores A_{AB}, and looks for the *integrity proof* of the $audit$ record on the blockchain. Finding I_{AB} on the blockchain is an assurance for B that participant A can be held accountable for the $message$ it sent. Moreover, the recipient verifies that the reference of the previous $audit$ records, sent as part of the $payload$, against the records on the blockchain and checks if it was published by the correct participant in the workflow. If any of these checks fail, participants in the workflow get alerted; this is the *Audit Record Verification*. *Integrity Proofs* are pushed on the blockchain and verified through a smart contract. A receipt of delivery is then sent to the sender with the signature of the receiver. Receipts are useful for revealing transactions between two participants without requiring the reconstruction of wk. After that, the receiver of the $message$ sends A_{AB} to the other participants in the workflow who in turn verify the equivalent *integrity proof* on the blockchain.

5.4.4 Protocol

Compared to the centralised implementation of the audit server discussed in the previous chapter, a blockchain-based implementation eliminates the need to challenge this server to prove its authenticity. It also enables the execution of *Audit Record Verification* on the server, rather than the participants end. However, the limitation of blockchain requires shifting the task of sharing $audit$ records with workflow participants to the recipient of a $message$.

- Participants keep their receipts and alert all participants if a receipt of delivery is not received.
- *Audit Record Verification* is always performed by recipients on $message$ delivery,

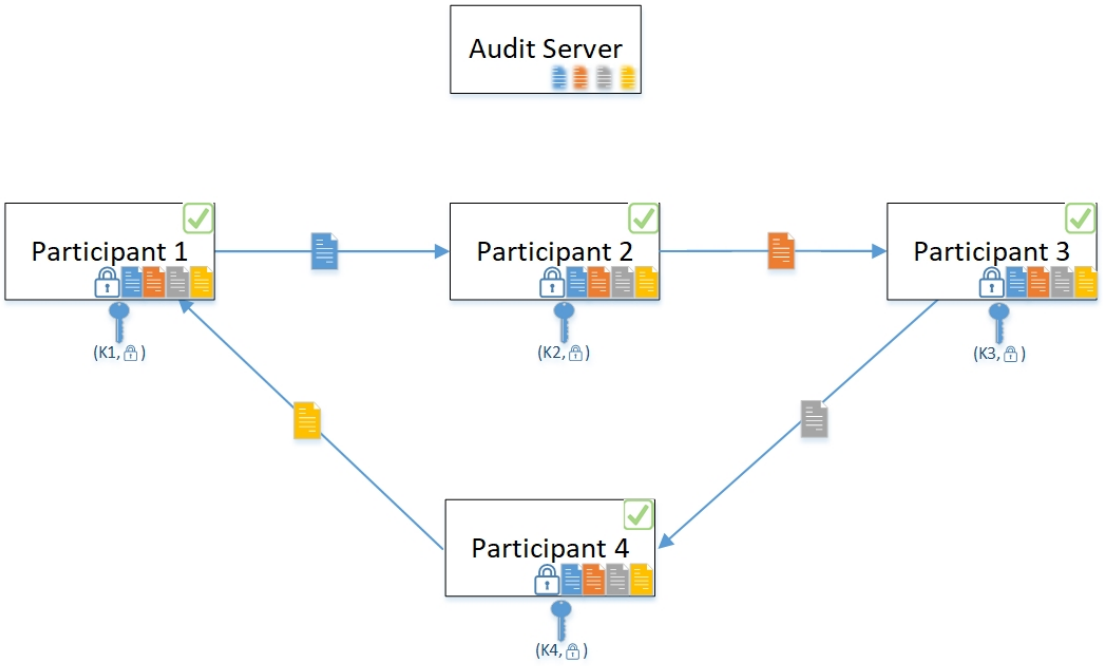

Fig. 5.3 Overview of AuDiC with a Blockchain-Based Implementation of the Audit Server. Faded Messages on the Blockchain Represent *Integrity Proofs* of the Actual $Audit$ Records

and by each participant when receiving an $audit$ record. Failure of this verification alerts all participants.

– Recipient broadcasts the equivalent $audit$ record of a $message$ they received to all the other participants, subject to a successful *Audit Record Verification*.

– Participants keep the $audit$ records of the $messages$ they receive, as well as the $audit$ records they receive for other transactions in the workflow.

Following this protocol, Figure 5.3 represents the state of each workflow participant as well as the blockchain-based audit server in terms of storage of audit data at the end of a workflow execution. The audit server in this case permanently stores *integrity proofs* of the $audit$ records, while participants keep an encrypted copy of the audit trail. Following each step of this protocol is required to maximise the level of protection of the approach. While the third step of the protocol can be omitted, skipping this step limits the $audit$ records availability to the sender and receiver of a $message$; this makes the destruction of evidence less challenging for participants attempting to escape repudiation. Additionally, skipping this step leads to withdrawing the ability for any K out of N participants to audit the full trail of transactions. Potential attacks on this implementation of the approach are presented in the next section.

5.5 Analysis of AuDiC with Blockchain-Based Audit Server

This section discusses activities resulting from a malicious participant in the workflow, or from collusion between two or more participants. Collusion with the audit server is not an option following the threat model in this chapter. The effect of a successful collusion depends on the positions of colluding $nodes$ in the topology.

5.5.1 Malicious Participant

Working individually, a participant can attempt to avoid repudiation by not submitting the *integrity proof* of a $message$ it sent to the blockchain. This is detected on the fly by the next honest participant running *Audit Record Verification*. This malicious participant can also attempt to **truncate the audit trail** by using an incorrect reference to a previous $audit$ record, possibly by using a reference to a self published record, in the $message$ it sends. This is detectable with blockchain given that the publishing entity of a record is identifiable through its address on the blockchain. Truncation in the context of AuDiC only leads to the loss of a pointer to a previous record but does not affect the availability of the latter. Moreover, a malicious sender of a $message$ can produce a false $audit$ record and submit its *integrity proof* to the blockchain in an attempt to avoid repudiation. This attempt is detected through the absence of receipt of delivery containing the signature of the receiver at the level of the malicious sender. Due to the role of broadcasting $audit$ records assigned to the receiver of a $message$, a malicious participant can send different $audit$ records to different participants; as an attempt to avoid detection, malicious participant sends $audit$ records that have *integrity proofs* published on the blockchain. This is detected on the fly by any honest participant due to the transparency of blockchain revealing the publisher of the *integrity proof* as well as the time the transaction was performed on the blockchain.

5.5.2 Non-consecutive Colluders

The blockchain-based implementation of AuDiC offers the same resistance for non-consecutive colluders as the implementation with the centralised server. With the colluding participant in Figure 5.4.1, an attempt by participant **D** to truncate the audit trail by using an $audit$ record published by **B** as part of the $message$ it sends to **E** is detected on the fly by the honest participant through *Audit Record Verification*; also, honest participant **C** can prove the $message$ it sent through the blockchain records and the receipt of delivery.

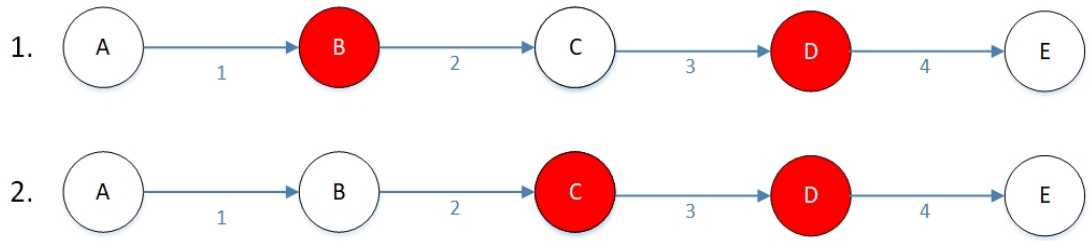

Fig. 5.4 Collusion Between Participants with a Coloured Background

5.5.3 Consecutive Colluders

The same malicious attempts exercised by consecutive colluding participants with the centralised server implementation apply in the case of the blockchain. Referring to Figure 5.4.2, participant **D** can skip the *Audit Record Verification* phase leading to one of the following cases:

- truncating the audit trail through publishing a corrupted A_3 or a false reference of a previous $audit$ record (a_2) in A_3.
- having an audit trail with a false $audit$ record A_3

The same analysis of Section 4.6.3 of the previous chapter applies in this case; for the first case, truncating the audit trail does not lead to any data loss, and colluding participants will be held accountable for what is reflected in A_3 for the second case.

In addition to that, and due to the role of broadcasting A_3 that participant **D** has, the latter can send different versions of this record to different participants; participant **C** in this can publish an *integrity proof* for each version of the record. Due to the transparency of blockchain, honest participants can suspect this malicious activity when spotting multiple records on the blockchain. Moreover, decrypting and analysing $audit$ records held by different participants reveal this collusion, and accountability is assigned to **C** and **D** for this malicious activity.

If step 3 of the protocol is omitted, consecutive participants would only have the $audit$ records of transactions exchanged between them. In the collusion represented in Figure 5.4.2, **C** and **D** can decide not to cooperate in an audit, and to destroy A_3 to repudiate a malicious action. Other participants would not have the option of performing a full audit in this case, but the collusion between **C** and **D** would be revealed.

5.5.4 Scenarios

One of the scenarios presented in Section 4.6.5.2 of the last chapter is revisited in this section. The applicability of the blockchain-based implementation of AuDiC is analysed on this scenario and the tolerance of the approach for collusion between partici-

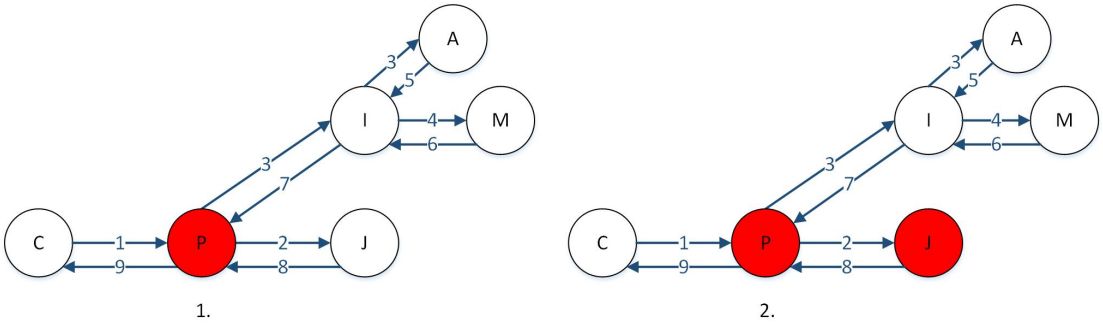

Fig. 5.5 Malicious Behaviour of Nodes when Auditing the Passport Scenario

pants is discussed. Malicious attempts to produce a passport from the passport department for a citizen with a criminal record are considered. Figure 5.5 shows a malicious participant working individually and attempting to avoid repudiation, and another scenario with a collusion between two consecutive $nodes$. Considering Figure 5.5.1, and assuming that a malicious employee in the passport department ignores the criminal record sent from the honest service **J**. **J** publishes I_8 on the blockchain, keeps a copy of A_8, and expects a receipt of delivery from **P**. Failure of **P** to send the receipt of delivery alerts **J**, which reports suspicion of malicious behaviour. Malicious employee can delete A_8 stored at the level of **P**; however, the proof of delivery receipt held by **J** and its copy of A_8 reveals that the wrongdoing is at the level of **P**. Also, **P** is expected to broadcast A_8 to the other participants in the workflow; failure to do so or an attempts to share inauthentic $audit$ records is detected by other honest participants following AuDiC protocol. Considering a collusion, represented in Figure 5.5.2, between **P** and **J** through malicious employees in both departments, and assuming that these malicious employees managed to skip publishing I_8 on the audit server; this gets detected on the fly by any honest participant performing *Audit Record Verification* upon receiving A_8. If the malicious employee in **P** managed to skip the broadcasting phase of A_8, honest participants, aware of the workflow topology, suspect a malicious behaviour and alert each other. Moreover, in an attempt to mislead participants, colluding $nodes$ may produce different versions of A_8, with or without an equivalent I_8 for each version, for **P** to distribute each version to a different participant; this is detected by honest participants on the fly due to the transparency of blockchain, and an investigation reveals this collusion. **P** and **J** can produce evidence that does not reflect the payload in M_8. The blame is assigned to one of the participants in accordance with the $audit$ record in this case: the blame is assigned to **P** if A_8 reflects a criminal record, and **J** takes the blame if A_8 shows an inauthentic record produced on its end. Producing a corrupted record is also possible through this collusion, but it reflects the collusion between the two participants.

Omitting step 3 of the protocol, requiring **P** to share A_8 with other participants, gives malicious colluders the option to destroy A_8. Given that AuDiC gives the option of

recording $audit$ data to any degree of details, A_8 can include information leading back to the colluding employees in these departments. Destroying A_8 reveals a collusion between the two departments, but leads to uncertainty in terms of locating the insiders in this case.

Collusion between participants is analysed in a similar way for the supply chain scenario presented in Section 4.6.5.1 of the last chapter. AuDiC offers tolerance against collusion when the audited transactions are not between colluding $nodes$, and gives evidence that reveals collusion when these transactions are.

5.6 Implementation and Evaluation with Blockchain-Based Audit Server

This section compares the blockchain-based implementation of the audit server and the implementation with a centralised server while following the corresponding protocol for each case. Then, the effect of sharing $audit$ records with workflow participants on the performance of the blockchain-based implementation is analysed. This is followed with a highlight on the effect of integrating the approach proposed in chapter 3 with AuDiC.

Compared to the implementation of the centralised audit server discussed in Section 4.7 of the last chapter, changes were required in the audit server as well as the workflow participants tier in Figure 4.1. For the audit server tier of the blockchain-based implementation, Ethereum blockchain is used with Solidity to develop smart contracts. A blockchain node is deployed on a local machine with Geth client and Web3j library is used to enable the interaction with the blockchain through Java. Methods were developed to upload *integrity proofs*, verify $audit$ records, upload and retrieve keys from the blockchain. To enable interactions between the workflow participants and the audit server, an API with multiple endpoints is developed to enable calling the different functionalities of the server. As for the workflow participants tier, while part of the code including constructing $messages$ and $audit$ records remained the same as the previous implementation, *Audit Record Verification* was changed to be triggered with an API call to the audit server. The functionality of sharing $audit$ records was also implemented following the protocol in Section 5.4.4.

5.6.1 Comparing Performance of Both Implementations

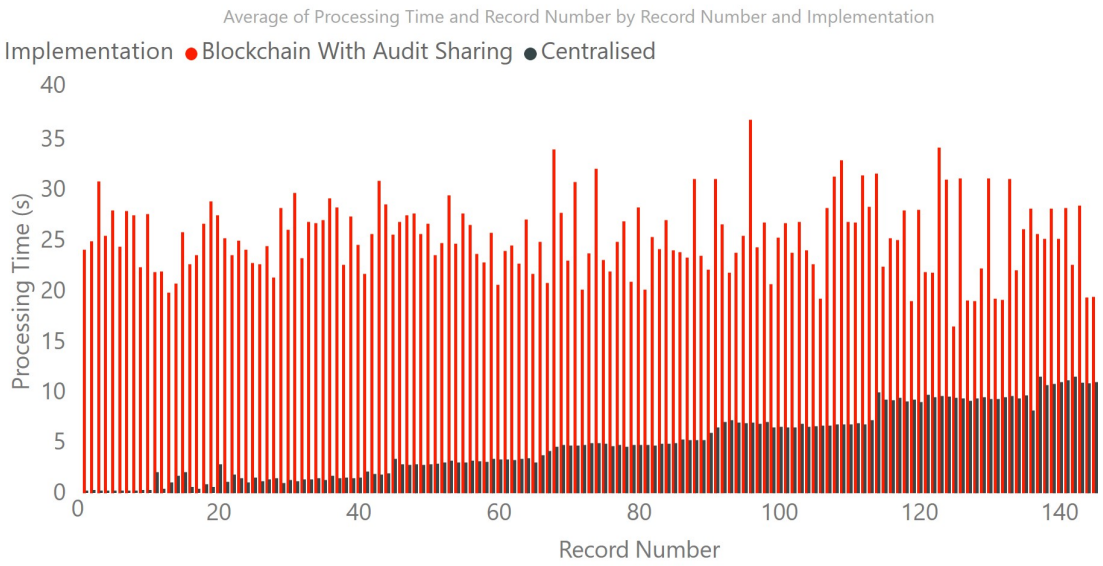

Fig. 5.6 Blockchain and Centralised Audit Server Performance Comparison

This section compares the performance of each implementation of the audit server following the protocols described in chapter 4 for the centralised implementation and in this chapter for the blockchain implementation of the audit server. In each case, the audit server and participants are hosted on the same machine with an Intel (R) Core i7-6700HQ CPU 32 GB of RAM running Windows 10 operating system. Both implementations were evaluated on 25 different topologies of 20 participants with up to 145 edges generated with Brite. Record Number in the figure reflects the number of records already reported to the audit server in each case before the audit record of a particular transaction gets reported. While *integrity proofs* of $audit$ records are submitted to the audit server in the first case, full $audit$ records are sent to be displayed on the audit server in the second case. For both cases, the processing time is shown for a $message$ of a fixed size, containing 10 KB of $data$, exchanged between participants following AuDiC protocol to record and verify $audit$ records for each implementation. At the end of each workflow execution, every participant ends up with the audit trail for all the interactions in the workflow. As reflected in the figure, while the blockchain's performance is stable, a delay proportional to the number of records already on the server is experienced with the centralised implementation. While a centralised server, being a single point of failure, is the weak link in the implementation of chapter four, a decentralised network of blockchain nodes offers stability and reliability at scale. Blockchain, however, introduces a delay as seen in the figure caused by its mining and consensus process.

5.6.2 Effect of Sharing Records

After reflecting on the importance of the sharing step of $audit$ records with workflow participants from the security perspective in Section 5.5, this section evaluates the effect of this step on the performance of the blockchain-based implementation of AuDiC. Using the same setup described in 5.6.1, the same performance evaluation is repeated while disabling sharing of $audit$ records by the recipient in the workflow (step 3 in the protocol). Note that this reflects on step 2 and step 4 of the protocol, given that recipients of $audit$ records are required to store and verify these records against the blockchain. Figure 5.7 reflects the average processing time of transactions by their order in the workflow topology when sharing $audit$ records is enabled and when it is disabled. A clear analysis can not be derived from the graph, but a difference in performance is noticed through the higher green bars than the reds. As for the averages processing time of transactions of all the topologies, the sharing step results in 3.2 seconds as a difference between 25.4 and 22.2 seconds.

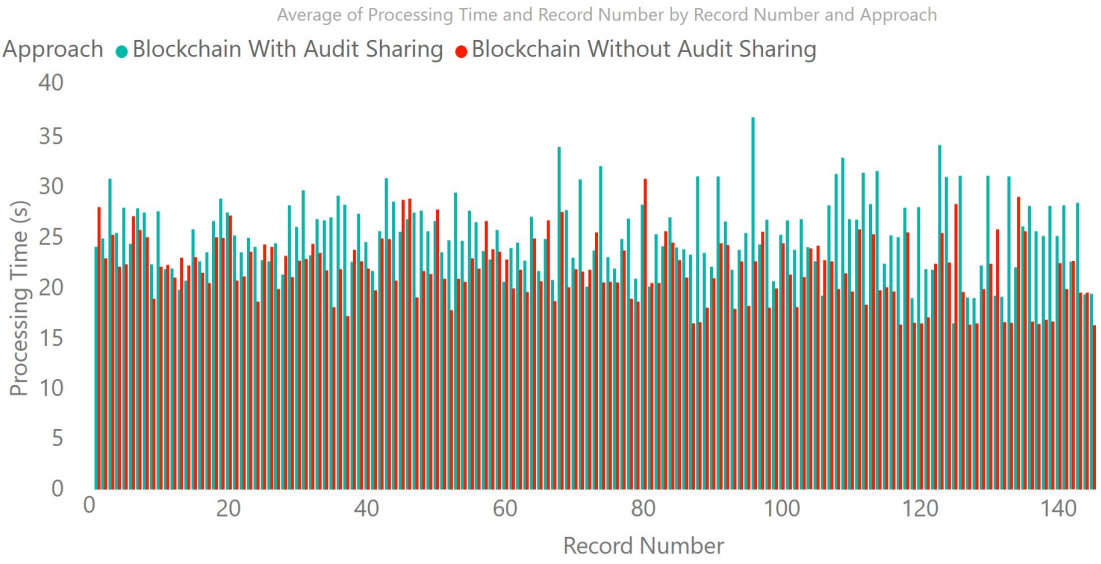

Fig. 5.7 Performance Comparison of the Blockchain-Based Implementation of AuDiC From the Perspective of $Audit$ Data Sharing

Attributes to consider when making a decision of which implementation works better for a specific business context includes the number of transactions in the workflow and the expected load on the audit server.

5.7 An Application Agnostic Evaluation of our Contributions

Throughout this thesis, a number of representative scenarios were used for the mere purpose of illustrating the applicability of an approach. However, given that the approaches can work on any application domain, the evaluations of the implementations are application agnostic, and a topology generation tool was used for a fair and objective representation of distributed workflow collaborations.

AuDiC enables capturing any degree of details on the transactions between participants, including the tokens used for user identification and access control. To represent the overhead resulting from the security checks for auditing, the same 25 topologies involving 20 participants used in Section 5.6 were reused to record the data propagation time of a $payload$ with 10 KB of $data$ without any interaction with an audit server or audit-related security checks. Without the overhead caused by the interaction with an audit server and the equivalent protocol for each implementation, the processing time of a message was fairly steady with an average of 210 ms. This average can be compared with the average processing time of 4.6 seconds when AuDiC protocol is followed with a centralised audit server implementation and of 25 seconds with the blockchain-based implementation of the audit server. Going back to the evaluation of chapter 3 showing a 23% and 32% overhead resulting from the integration of CGW and RGW respectively with their security checks, the negligibly of this overhead (applicable to the overage of 210 ms) on the overall performance is deduced when the proposed approach for fine-grained access control of microservices is integrated with either implementation of AuDiC.

5.8 Conclusion

This chapter presented a blockchain-based implementation for AuDiC. While blockchain offers security and reliability that centralised servers do not, blockchain has scalability and privacy limitations imposed by its method of operation. Following the criteria with which blockchain is used, the security features it offers are made use of while overcoming its limitations to fulfil the same goals that were achieved with a centralised server implementation. Revisiting these goals, AuDiC offers a means to produce, verify, and store audit records in a trustless, collaborative and confidentiality friendly way; it also offers auditing capability to any K out of N participants in a workflow. The aim is to protect audit records from collusion related threats enabling destroying or withholding evidence, as well as tampering with or breaching the confidentiality of different participants during the generation of audit records or at storage phases.

While the content and code execution on the audit server can be trusted with a blockchain-based implementation due to its distributed nature, an implementation that

includes a centralised server requires workflow participants to challenge the audit server; this helps participants verify that the entity managing the server is not acting maliciously by eliminating, tampering with, or showing different versions of audit records to different participants. On the other hand, while a centralised audit server can serve as a means to display encrypted audit records to different workflow participants, a blockchain-based implementation requires shifting the duty of sharing audit records to participants to ensure high availability of evidence. Each implementation of AuDiC has its advantages and downsides, and the aim of the security analysis and the evaluations performed in chapters 4 and 5 is to give an overview that helps adopters make an informed decision on which implementation is a better fit for their business context.

The security aspects offered by either implementation of AuDiC is an improvement over existent approaches that rely on traditional centralised servers or blockchain technology to offer auditing capabilities for distributed workflows. Although workflows that use smart contracts obtain reliable audit trails, the transparency of blockchain makes the approach of AuDiC to challenge the generation of audit records in a confidentiality friendly way novel for auditing distributed collaborations. Also, the distribution of audit records during the workflow execution while verifying that all participants obtain the same audit trail gives additional assurances for the authenticity and availability of digital evidence compared to other approaches that rely on backing up databases, or that use blockchain to verify the integrity of their content. In addition to that, the audit data structure that enables recording digital evidence to any degree of details, and the strategy of linking records to create an audit trail that can cover arbitrary workflow topology are features that help this approach to reach a wide audience. Although AuDiC is designed with high confidentiality, integrity and availability requirements in mind, its tight security measures can be relaxed for applications that do not require, for example, strict confidentiality requirements between participants; moreover, offering both implementations enable the adoption of the approach for organisations that have legal, political, or corporate culture restrictions on the technology that should be used. AuDiC is a step forward towards achieving highly dependable auditing and it demonstrates the importance of verification as opposed to trust in any security context.

CHAPTER 6

Conclusion and Future Work

6.1 The Big Picture

This research made a contribution to the security of microservices-based application with a focus on access control and to the auditing of workflow based collaborations between different organisations. Each of these contributions is applicable to any application domain, and can be adopted separately; however, adopting the combination helps to enhance the robustness of security for distributed applications. This section discusses how the contributions in this thesis connect and present the rationale for the order that was followed throughout this research. This analysis shows how the security reference architecture for microservices helps to provide the security environment upon which the fine-grained access control approach can fulfil its security goals, and how the latter contribution enable the environment required for the collaborative auditing approach.

This research started with a review of existent security practices that are applicable to microservices-based applications and resulted in a holistic security reference architecture; this covers microservices as components and their interactions to form a microservices-based application starting from the design phases and throughout the application lifecycle. This review layed the foundations for the main contribution at the application level focused on the access control of microservices-based applications where security measures were proposed to integrate standards for access control and delegation with microservices in a more secure way than common practices.

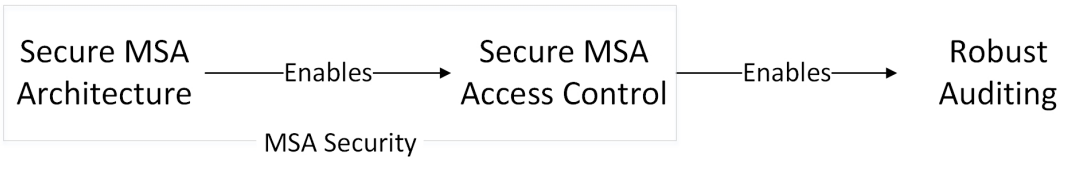

Fig. 6.1 Research Contributions in Order

As shown in Figure 6.1, the first contribution represents the focus of this thesis on the security of microservices-based applications. This research then moved to covering interactions between these applications with a robust and confidentiality friendly

auditing approach for inter-domain interactions to mitigate collusion-related threats to tamper with, destroy or breach the confidentiality of evidence. While each of these contributions stands on its own, the combination gives a better security outcome for distributed applications that are built through the integration of microservices-based applications. Figure 6.1 shows the contributions of this research in perspective and further details are discussed in the following sections.

6.1.1 Secure Architecture Enabling Securing Access Control

Access Control is a fundamental practice for the security of any application requiring the protection of certain assets. However, a secure access control placed to protect these assets is not effective when these are exposed through other vectors like injection attacks and poor handling of security configurations and credentials. While the contribution of this research for microservices access control introduces mitigation measures against specific vulnerabilities to the architecture of microservices-based applications, the proposed protocol and the trusted execution of its security checks are only enabled with secure components that were introduced in the security reference architecture for MSA.

To begin with, the proposed approach for access control relies on having OAuth 2 clients tailored for up to each microservice protecting certain assets (the Resource Microservices). Following OAuth 2 protocol and as discussed in Chapter 3, each OAuth client requires credentials to issue access tokens. Without a secure provisioning of these credentials for microservices at the bootstrap phase, and without the secure storage of these credentials enabled with the key vault (root of trust) of the security reference architecture, the approach for access control is weakened with the possibility of tricking users to issue access tokens for microservices they did not request to access. Moreover, the scalability of the proposed gateways, embedding the logic of the security checks for access control, with the microservices cannot be trusted without the verification against the root of trust.

Also, the security considerations in the application design reflect on the effectiveness of access control for data protection. Using a single database for all the system microservices enables an attacker to access all the application data through a single vulnerable microservice. Having weak security configurations, including easy passwords for databases and running containers with root users, is another enabler to penetrate a system and access its data. Moreover, traditional interface attacks, including SQL and JavaScript injections and their derivatives are other enablers for inauthentic data manipulations and exfiltration.

In addition to that, availability of applications precedes the usefulness of their access control system. Microservices require special attention due to their limited resources; a

denial of service on a single microservice can jeopardise the availability of the application, which justifies the recommendation for gateway at the level of the microservices-based application (Neri et al. 2019). Other protection measures for availability in the reference architecture in this thesis include the internal monitoring and scaling system, the firewall and intrusion detection system at the network entry point of the application.

6.1.2 Secure Applications Enabling Robust Collaborative Auditing

AuDiC, the proposed approach in this thesis for auditing of distributed collaborations assumes each transaction to reflect the intention of the administration that manages an application participating in the workflow. This assumption is not reasonable without robust security measures within the application security domains, and for its interactions with the other domains; the security guidelines and reference architecture in Chapter 2 aim to offer this level of security. In addition to that, AuDiC imposes security verification mechanisms on participants as part of the audit protocol, as well as requirements to store audit records and receipts. While unintentional failure to follow these measures due to a vulnerability weakens the assurances that AuDiC provides, intentional skipping of steps in the protocol can help participants to escape repudiation if they claim benevolence. Moreover, the security of AuDiC depends on the protection of cryptography keys: leaking a participant share of wk is a step closer for an adversary to breach the confidentiality of workflow participants, and leaking the participant private key enables framing the vulnerable participant through a fabrication of evidence. The key vault component of the secure reference architecture helps participants to protect their cryptography keys. Moreover, the availability of applications being part of a workflow is a prerequisite for auditing.

On the other hand, a robust access control system within an application domain is of critical importance to ensure the correct behaviour within an application. With a vulnerable access control in a microservices-based application, a successful confused deputy or token manipulation attack enables, for example, making wrongful actions on behalf of an honest employee. While accountability assurance is a major goal of AuDiC, similar vulnerabilities enabling evidence manipulation within a domain defeat the purpose of this auditing scheme.

6.2 Conclusion and Future Work

Going back to the structure of distributed collaborations of microservices-based applications across domains introduced in Chapter 1 (Figure 1.1), this research made contributions that cover different layers. Having this perspective as a basis, this research followed an inner-to-outer security strategy following which a contribution covering an outer layer is enabled with another covering the inner ones. While there is still room for

improvement in each of these layers, this work helps to lay the foundations of security for microservices-based applications, and offers an approach for auditing for reliable auditing of distributed workflows. For each contributions, verification, as opposed to trust, is followed as a strategy to enhance the security of applications within a single domain as well as their interactions between domains. This section summarises the contributions of this researchand shed light on future work and research directions.

6.2.1 Summary of the Research Contributions

This section gives a recap on each of the contributions of this research including a highlight what was achieved in each, and the added value compared to common practices.

This research started with a study of microservices architectural paradigm covering motivations for the shift away from the old monolithic paradigm, the resemblance of practices yet the difference of scopes and goals between microservices-based applications and the Service-Oriented Architecture, as well as the principles and enablers of microservices. A representation of microservices-based applications was modelled, the security challenges focusing on microservices specifications were analysed, and security hardening measures following common security practices throughout the software development lifecycle were presented in Chapter 2. While this approach is not meant to present an exhaustive list covering the details of security measures applicable to MSAs, the aim is to give a comprehensive view covering fundamental security practices for microservices as independently deployable components, their interactions within and outside their domain, the underlying infrastrusture and the architecture of the application. Practices for secure scaling of microservices and for handling of cryptography materials are also highlighted as part of this study. These practices were represented with a security reference architecture for microservices-based applications to help practitioners build security by design.

Research on microservices architectural paradigm has a short history, and the literature is poor on security and architectural guidance for it. Compared to existent work in the literature, this study introduces a more comprehensive view of the security of microservices-based applications. The shift towards microservices and the growing adoption of this paradigm by industries raises concerns of vulnerabilities resulting from security design flaws. Embedding security practices since the design stages of applications helps avoid expensive software refactoring, as well as liabilities and reputation damage caused by a successful compromise. The common integration of standards for access control with microservices makes them vulnerable to *powerful token* theft and manipulation, in addition to a vulnerability caused by microservices trust model: the *confused deputy* problem. After laying the foundations with the security architecture, this thesis moved to main focus of the contribution of this research for microservices-

based applications revolving around access control.

This research presents an approach for access control that, although based on commonly used open standards, takes into account the design specifications of microservices that make common practices for access control of microservices-based applications vulnerable. This approach is based on the integration of OAuth 2 and XACML open standards with a couple of security checks that were designed to detect requests manipulations. Following this approach, successful token thefts, once enabling the token holder to act on behalf of the user and have access to their resources, are rendered ineffective with the narrow scopes and short lifetimes of tokens; also, requests manipulations are detected on the fly, and the *confused deputy* attack is mitigated with the necessity of users' engagement to produce access tokens specific to requests enabling actions on the assets. To enable the easy adoption and integration of this approach with existent systems, this research categorised microservices according to their roles into consumers and resources, and designed configurable and reusable micro-gateways including the security checks through which requests need to go before reaching primitive microservices. In the context of this thesis, micro-gateways perform service-to-service authentication, check access policies to verify if a user is authorised to access a service and if a service is acting faithfully on behalf of a user, and enable the generation of short-lived access tokens with fine-grained access scopes reflecting the delegation of a user for a service to act on its behalf. Although the proposed approach for access control is designed to mitigate vulnerabilities caused by microservices specifications and exploited within a single application domain, organisations very often collaborate through their applications. The impact of an exploit of any of these vulnerabilities in a single application spreads to other domains interacting with the compromised application. This applies to cases in which users' assets are spread across different administrations like digital government or e-health applications, or to cases in which users decisions take effect in multiple organisations like supply chain and engineering.

When thinking of collaborations of organisations, financial gains and avoiding reputation damage are among numerous reasons that may motivate an organisation to avoid repudiation of some actions or decisions. Being part of different administrative domains, organisations need audit trails of their collaborations to be able to assign accountability to an organisation responsible for wrong-doing. In extreme cases, participants should not trust each other or any single entity to produce or store audit records of transactions. Assurances should be given showing that a malicious entity, working individually or colluding with an audit system, does not have the option of tampering with, destroying or withhold digital evidence. Moreover, a single entity should not be trusted with confidential data belonging to different organisations. The inadequacy of approaches in the literature to fulfil these requirements was a motivation to design and implement AuDiC, an application-agnostic approach to audit distributed collaborations

to any degree of details while targeting the challenges above. This is another contribution of this research.

Part of this contribution is designing audit records that allow recording any degree of details of transactions and that, when linked together to form an audit trail, can cover any workflow topology. This is detailed in Section 4.4.1, which introduces a representation of this research for workflows as well as the data structure of audit records, and discusses how these records connect to cover arbitrary workflow topologies. The other part is enabling the verifiability of each of these records without trusting a single entity, and ensuring the high availability of the audit trail without breaching the confidentiality of any of the workflow participants or giving the option to any of them to avoid repudiation by withholding evidence and refusing to decrypt a record. The task of verifying each audit record is assigned to the recipient of its equivalent message, and an untrusted centralised audit server is used for the mere purpose of facilitating the distribution of encrypted audit trails to participants during execution. This server is challenged by every participant to verify that it is not tampering with or eliminating encrypted records, and that it is not showing different versions of the audit trail to different participants. The proposed approach relies on verifiable secret sharing, which enables participants to verify that the entity generating and splitting a key is honest with the key distribution, and that a threshold of participants is needed to decyrpt the audit trail. The functionalities above were implemented and a fully functional prototype implementation was produced combining components that enable the simulation of the proposal for auditing of distributed systems. As a preliminary evaluation for the scalability of the centralised server, a busy server was emulated by adding a log-normally distributed delay to the response time of the audit server reflecting the expected behaviour of the server under load and when multiple workflows are simultaneously using the same server. Then, to offer a fair and objective evaluation of the implementation, a topology generation tool was used to produce arbitrary topologies, and the performance of the implementation was evaluated on different topologies and different sizes of data in audit records. Being aware of the limitation of this implementation due to reliance on a server that may fail, the applicability of the approach was investigated with a blockchain-based implementation for the audit server. Although the method of operation and the highly distributed nature of blockchain offers reliability and stability as well as trust assurances, the limitations and specifications of blockchains lead to making amends to AuDiC. This is another part of the contribution of this research to the auditing of distributed workflow based collaborations.

Although blockchain related approaches to produce audit trails or verify the integrity of audit records have been extensively discussed in the literature, the assurances offered by AuDiC and its applicability to arbitrary workflows have not been covered in any previous work. While blockchain types, fees, consensus mechanisms and tech-

nologies are out of scope of this work, this research considers blockchain scalability limitations while making use of its transparency and security features when using this technology to implement the audit server of AuDiC. Smart contracts with APIs were developed and the required modifications in the previous code were implemented to achieve the same security goals reached with the centralised audit server. The same evaluation approach on arbitrary topologies was followed to compare the steady performance of the blockchain-based implementation and the initial one including a centralised server. Given that blockchain method of operation and scalability limitation lead to assigning the distribution of audit records to recipients of every transaction, another evaluation of the blockchain-based implementation that omits the audit records distribution step was performed; this leaves the sender and recipient of a transaction, rather than all participants, with the equivalent audit records of transactions exchanged between them. While this thesis recommends a full distribution of records to all participants, limiting the availability of equivalent audit records of transactions to the sender and receiver may make sense in contexts in which locating collusion leading to destruction of evidence is enough to assign accountability.

As discussed in the previous section, following the order with which the contributions were presented, each one serves as an enabler for the other. Measures taken in one contribution are relied upon in other later ones to ensure that the security assumptions can be met. Cyber attacks are getting more sophisticated by the day, and security measures can never guarantee enough protection against attack vectors that are not discovered at the time they were put in place. With this in mind, the security strategy followed in this work suggests the inner layers of Figure 1.1 not to be trusted by the outer ones as a measure to contain a security breach when it happens in a domain rather than enabling it to spread to other domains. Due to the time restrictions and other resource limitations of this study, a number of possible improvement have been left for future work. The next section highlights the limitations of our work and sheds light on future directions of research.

6.2.2 Limitations

Each of the proposed approaches in this thesis has limitations; some of the limitations result from the protocol that is proposed, and others are due to the available cryptography approaches.

The proposed approach for MSA access control results in an additional load on the authorisation server, which requires allocating more computational resources at the level of the latter to ensure its high availability. The authorisation server is also a single point of trust that can be dishonest with the legitimate production of access tokens; this can either be through the production of tokens without the consent of an end user

or by not abiding to the access scopes that the user selects. Also, the maintenance and management of access control lists and of the access scopes at the level of microservices results in an additional complexity at the level of the applications.

AuDiC also has a limitation related to the the production of the workflow key. Following this approach, the entity that has the least incentive to expose the workflow key can breach the privacy of all participants in the workflow. This limitation is inherited from the currently available approaches for the generation of keys, and a distributed generation of cryptographic keys is still an open problem.

6.2.3 Future Research Directions

Chapter 2 of this thesis offered a high level reference architecture highlighting existent approaches and measures to help avoid vulnerabilities caused by microservices specifications. The large number of containers and exposed interfaces on the same machine, the implicit trust between microservices, and the common reliance on an insecure network to communicate are the top reasons for the special attention required for the security of MSAs. While this thesis suggested protection measures to consolidate security by reducing the exposure to certain vulnerabilities, detecting intrusion when possible, and handling of security configurations and cryptography materials throughout the life cycle, many aspects in the reference architecture can be elaborated on. These include:

- Security configurations and attestations of microservices throughout the life cycle with the secure hardware suggested in the architecture.
- Internal security monitoring and packet inspection on the network to detect intrusion
- Cloud container orchestration services and their support for microservices

Collaborations between academics and the industry are required to cover the breadth of these aspects and reach the needed security maturity, and to develop specialised security elements tailored for microservices specifications.

With regards to the contribution for microservices access control, free and available at the time components were used to produce a simple proof of concept. The functionalities and the support for the free versions of the components that were used were restrictive and very limited. In August 2019, Forgerock released their first version of microgateways as well as a token validation component[1]. This proves the validity of this research, and suggests that more work needs to be done by academics and the industry to secure microservices access control.

As for the contribution of this thesis for auditing of distributed collaborations, although the recommended security measures have been implemented with a centralised

[1] `https://backstage.forgerock.com/docs/mg/1`

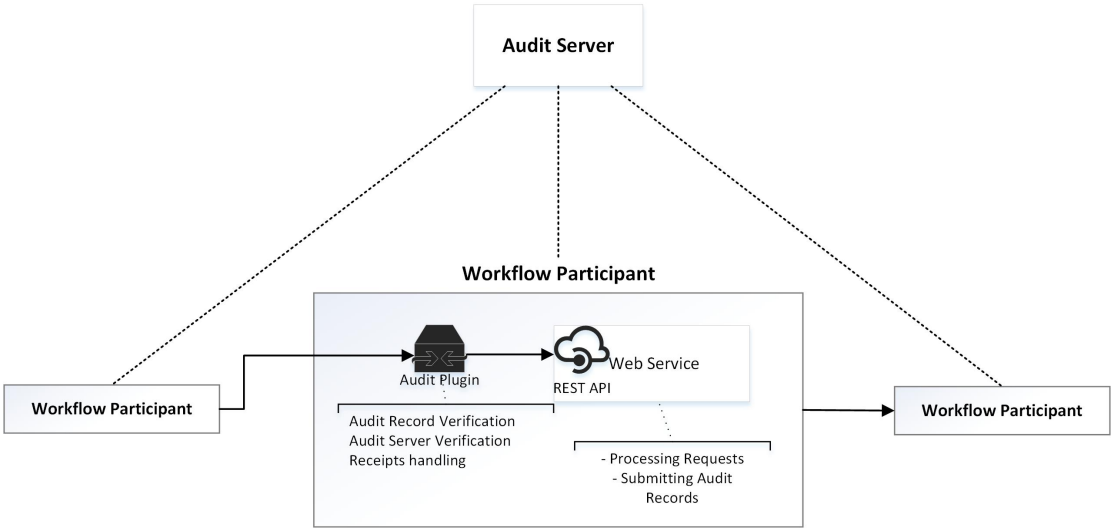

Fig. 6.2 Implementing Re-usable Plugins to Audit Workflows

and blockchain-based audit server, sharpening and optimising the code will improve the performance of the implementation. Moreover, to enhance the portability and ease of re-usability of AuDiC with existent systems, the proposed security measures can be segregated from the functionality of existent services and implemented with reusable and configurable gateways; Figure 6.2 represents the role that a reusable gateway, the audit plugin, can play to minimise the need to modify web services in existent systems.

Moreover, a future plan is to install and evaluate AuDiC in a real life setup given that this has not been achieved during this research time frame. Another possible research direction is a more comprehensive investigation of the use different types of blockchain with AuDiC, including cost, performance, attacks by malicious nodes of the blockchain network and vulnerabilities in smart contracts.

REFERENCES

Abreu, P. W., Aparicio, M. and Costa, C. J. (2018), Blockchain technology in the auditing environment, *in* '2018 13th Iberian Conference on Information Systems and Technologies (CISTI)', IEEE, pp. 1–6.

Accorsi, R. (2010), Bbox: A distributed secure log architecture, *in* 'European Public Key Infrastructure Workshop', Springer, pp. 109–124.

Accorsi, R. (2011), Business process as a service: Chances for remote auditing, *in* '2011 IEEE 35th Annual Computer Software and Applications Conference Workshops', IEEE, pp. 398–403.

Accorsi, R. (2013), 'A secure log architecture to support remote auditing', *Mathematical and Computer Modelling* **57**(7), 1578 – 1591.

Ahmad, A., Hassan, M. M. and Aziz, A. (2014), A multi-token authorization strategy for secure mobile cloud computing, *in* '2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering', IEEE, pp. 136–141.

Ahmad, A., Saad, M., Bassiouni, M. and Mohaisen, A. (2018), Towards blockchain-driven, secure and transparent audit logs, *in* 'Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services', ACM, pp. 443–448.

Ahsan, M. M., Wahab, A. W. A., Idris, M. Y. I., Khan, S., Bachura, E. and Choo, K.-K. R. (2018), 'Class: Cloud log assuring soundness and secrecy scheme for cloud forensics', *IEEE Transactions on Sustainable Computing* .

Albert, P., Henocque, L. and Kleiner, M. (2005), Configuration based workflow composition, *in* 'IEEE International Conference on Web Services (ICWS'05)', IEEE, pp. 285–292.

Alqahtani, S. and Gamble, R. (2014), Embedding a distributed auditing mechanism in the service cloud, *in* '2014 IEEE World Congress on Services', pp. 69–76.

Alshuqayran, N., Ali, N. and Evans, R. (2016), A systematic mapping study in microservice architecture, *in* 'Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on', IEEE, pp. 44–51.

Aravind, A. and Sandeep, A. (2015), Workflow signature for business process domain: A new solution using ibmkd, *in* 'Communication Technologies (GCCT), 2015 Global Conference on', IEEE, pp. 619–622.

Argyriou, M., Dragoni, N. and Spognardi, A. (2017), Security flows in oauth 2.0 framework: a case study, *in* 'International Conference on Computer Safety, Reliability, and Security', Springer, pp. 396–406.

Azarmi, M., Bhargava, B., Angin, P., Ranchal, R., Ahmed, N., Sinclair, A., Linderman, M. and Othmane, L. B. (2012), An end-to-end security auditing approach for service oriented architectures, *in* 'Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on', IEEE, pp. 279–284.

Bates, A., Hassan, W. U., Butler, K., Dobra, A., Reaves, B., Cable, P., Moyer, T. and Schear, N. (2017), Transparent web service auditing via network provenance functions, *in* 'Proceedings of the 26th International Conference on World Wide Web', International World Wide Web Conferences Steering Committee, pp. 887–895.

Bilal, M., Thomas, J. P., Thomas, M. and Abraham, S. (2005), Fair bpel processes transaction using non-repudiation protocols, *in* 'Services Computing, 2005 IEEE International Conference on', Vol. 1, IEEE, pp. 337–340.

Bradley, J., Sakimura, N. and Jones, M. (2015), 'Json web signature (jws)', `https://tools.ietf.org/html/rfc7515`.

Brickell, E. F. (1989), Some ideal secret sharing schemes, *in* 'Workshop on the Theory and Application of of Cryptographic Techniques', Springer, pp. 468–475.

BS (2015), Information technology – security techniques – information security management for inter-sector and inter-organizational communications., Standard, British Standards Institution.

Butzin, B., Golatowski, F. and Timmermann, D. (2016), Microservices approach for the internet of things, *in* 'Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on', IEEE, pp. 1–6.

Calvaresi, D., Dubovitskaya, A., Calbimonte, J. P., Taveter, K. and Schumacher, M. (2018), Multi-agent systems and blockchain: Results from a systematic literature review, *in* 'International Conference on Practical Applications of Agents and Multi-Agent Systems', Springer, pp. 110–126.

Chor, B., Goldwasser, S., Micali, S. and Awerbuch, B. (1985), Verifiable secret sharing and achieving simultaneity in the presence of faults, *in* '26th Annual Symposium on Foundations of Computer Science (sfcs 1985)', IEEE, pp. 383–395.

Ciuffoletti, A. (2015), 'Automated deployment of a microservice-based monitoring infrastructure', *Procedia Computer Science* **68**, 163–172.

CloudPassage (2017), 'Verify, don't trust: Best practices for reducing vulnerability exposure in docker environments', `https://pages.cloudpassage.com/part-2-best-practices-reducing-vulnerability-ug.html`. Accessed: 2017-12-30.

Combe, T., Martin, A. and Di Pietro, R. (2016), 'To docker or not to docker: A security perspective.', *IEEE Cloud Computing* **3**(5), 54–62.

Cucurull, J. and Puiggalí, J. (2016), Distributed immutabilization of secure logs, *in* 'International Workshop on Security and Trust Management', Springer, pp. 122–137.

de Vrieze, P. and Xu, L. (2018), 'Resilience analysis of service-oriented collaboration process management systems', *Service Oriented Computing and Applications* **12**(1), 25–39.

Di Francesco, P., Malavolta, I. and Lago, P. (2017), Research on architecting microservices: Trends, focus, and potential for industrial adoption, *in* '2017 IEEE International Conference on Software Architecture (ICSA)', IEEE, pp. 21–30.

Dragoni N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L. (2017), Microservices: yesterday, today, and tomorrow, *in* 'Present and ulterior software engineering', Springer, pp. 195–216.

Dragoni, N., Dustdar, S., Larsen, S. T. and Mazzara, M. (2017), 'Microservices: Migration of a mission critical system', *arXiv preprint arXiv:1704.04173* .

D'Souza, R., Jao, D., Mironov, I. and Pandey, O. (2011), Publicly verifiable secret sharing for cloud-based key management, *in* 'International Conference on Cryptology in India', Springer, pp. 290–309.

Ebert, C., Gallardo, G., Hernantes, J. and Serrano, N. (2016), 'Devops', *IEEE Software* **33**(3), 94–100.

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

Fernández, F., Alonso, Á., Marco, L. and Salvachúa, J. (2017), A model to enable application-scoped access control as a service for iot using oauth 2.0, *in* 'Innovations in Clouds, Internet and Networks (ICIN), 2017 20th Conference on', IEEE, pp. 322–324.

Fetzer, C. (2016), 'Building critical applications using microservices', *IEEE Security and Privacy* **14**(6), 86–89.

Flores, D. A. (2014), An authentication and auditing architecture for enhancing security on egovernment services, *in* '2014 First International Conference on eDemocracy eGovernment (ICEDEG)', pp. 73–76.

Fowler, M. (2014), 'Microservices prerequisites', `https://martinfowler.com/bliki/MicroservicePrerequisites.html`.

Fowler, M. and Lewis, J. (2014), 'Microservices', *ThoughtWorks* .

Freudenthal, M. and Willemson, J. (2017), Challenges of federating national data access infrastructures, *in* 'International Conference for Information Technology and Communications', Springer, pp. 104–114.

Gajanayake, R., Iannella, R. and Sahama, T. (2011), 'Sharing with care: An information accountability perspective', *IEEE Internet Computing* **15**(4), 31–38.

Gao, X. and Uehara, M. (2017), Design of a sports mental cloud, *in* 'Advanced Information Networking and Applications Workshops (WAINA), 2017 31st International Conference on', IEEE, pp. 443–448.

GDS (2016), 'Our approach to api authentication.', `https://gdstechnology.blog.gov.uk/2016/11/14/our-approach-to-authentication` note = Accessed: 2018-05-20.

Geisriegler, M., Kolodiy, M., Stani, S. and Singer, R. (2017), Actor based business process modeling and execution: A reference implementation based on ontology models and microservices, *in* 'Software Engineering and Advanced Applications (SEAA), 2017 43rd Euromicro Conference on', IEEE, pp. 359–362.

González, L., Echevarría, A., Morales, D. and Ruggia, R. (2016), 'An e-government interoperability platform supporting personal data protection regulations', *CLEI electronic journal* **19**(2), 8–8.

González, L., Ruggia, R., Abin, J., Llambías, G., Sosa, R., Rienzi, B., Bello, D. and Álvarez, F. (2012), A service-oriented integration platform to support a joined-up e-government approach: the uruguayan experience, *in* 'International Conference on Electronic Government and the Information Systems Perspective', Springer, pp. 140–154.

Gorski, P. L., Iacono, L. L., Nguyen, H. V. and Torkian, D. B. (2014*a*), Service security revisited, *in* '2014 IEEE International Conference on Services Computing', IEEE, pp. 464–471.

Gorski, P. L., Iacono, L. L., Nguyen, H. V. and Torkian, D. B. (2014*b*), Service security revisited, *in* '2014 IEEE International Conference on Services Computing', pp. 464–471.

Goseva-Popstojanova, K., Li, F., Wang, X. and Sangle, A. (2006), A contribution towards solving the web workload puzzle, *in* 'International Conference on Dependable Systems and Networks (DSN'06)', IEEE, pp. 505–516.

Gummaraju, J., Desikan, T. and Turner, Y. (2015), Over 30% of official images in docker hub contain high priority security vulnerabilities, Technical report, Technical report, BanyanOps.

Hale, M. L., Gamble, M. T. and Gamble, R. F. (2013), A design and verification framework for service composition in the cloud, *in* '2013 IEEE Ninth World Congress on Services', pp. 317–324.

Härtig, H., Roitzsch, M., Weinhold, C. and Lackorzynski, A. (2017), Lateral thinking for trustworthy apps, *in* '2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)', IEEE, pp. 1890–1899.

Hartmann, K. and Steup, C. (2015), 'On the security of international data exchange services for e-governance systems', *Datenschutz und Datensicherheit-DuD* **39**(7), 472–476.

Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L. E., Pahl, C., Schulte, S. and Wettinger, J. (2017), Performance engineering for microservices: Research challenges and directions, *in* 'Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion', ACM, pp. 223–226.

Hildebrand, J. and Jones, M. (2015), 'Json web encryption (jwe)', `https://tools.ietf.org/html/rfc7516`.

IBM (2016), 'An integrated approach to insider threat protection.', `https://www-05.ibm.com/services/europe/digital-whitepaper/security/growing_threats.html`. Accessed: 2018-05-15.

Ilhan, Ö. M., Thatmann, D. and Küpper, A. (2015), A performance analysis of the xacml decision process and the impact of caching, *in* '2015 11th International Conference on Signal-Image Technology and Internet-Based Systems (SITIS)', IEEE, pp. 216–223.

Jander, K., Braubach, L. and Pokahr, A. (2018), 'Defense-in-depth and role authentication for microservice systems', *Procedia computer science* **130**, 456–463.

Jarwar, M. A., Ali, S., Kibria, M. G., Kumar, S. and Chong, I. (2017), Exploiting interoperable microservices in web objects enabled internet of things, *in* 'Ubiquitous and Future Networks (ICUFN), 2017 Ninth International Conference on', IEEE, pp. 49–54.

Jones, M. (2015*a*), 'Json web algorithms (jwa)', https://tools.ietf.org/html/rfc7518.

Jones, M. (2015*b*), 'Json web key (jwk)', https://tools.ietf.org/html/rfc7517.

Jones, M., Nadalin, A., Campbell, B., Bradley, J. and Mortimore, C. (2019), 'Token exchange', https://tools.ietf.org/html/draft-ietf-oauth-token-exchange-19.

Jones, M., Sakimura, N. and Bradley, J. (2015), 'Json web token (jwt)', https://www.rfc-editor.org/rfc/rfc7519.txt.

Karmel, A., Chadromouli, R. and Iorga, M. (2016), 'Nist definition of microservices, application containers and system virtual machines', *Nat'l Inst. of Standards and Technology (NIST) Special Publication* pp. 800–180.

Kieseberg, P., Malle, B., Frühwirt, P., Weippl, E. and Holzinger, A. (2016), 'A tamper-proof audit and control system for the doctor in the loop', *Brain informatics* **3**(4), 269–279.

Kuntze, N. and Rudolph, C. (2011), Secure digital chains of evidence, *in* 'Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on', IEEE, pp. 1–8.

Kütt, A. and Priisalu, J. (2014), Framework of e-government technical infrastructure. case of estonia, *in* 'Proceedings of the International Conference on e-Learning, e-Business, Enterprise Information Systems, and e-Government (EEE)', The Steering Committee of The World Congress in Computer Science, Computer . . . , p. 1.

Lightstep, D. (2018), 'Global microservices trends a survey of development professionals', https://go.lightstep.com/global-microservices-trends-report-2018 note = Accessed: 2019-10-20.

Lim, H. W., Kerschbaum, F. and Wang, H. (2012), 'Workflow signatures for business process compliance', *IEEE Transactions on Dependable and Secure Computing* **9**(5), 756–769.

Linthicum, D. S. (2016), 'Practical use of microservices in moving workloads to the cloud', *IEEE Cloud Computing* **3**(5), 6–9.

Lu, D., Huang, D., Walenstein, A. and Medhi, D. (2017), A secure microservice framework for iot, *in* 'Service-Oriented System Engineering (SOSE), 2017 IEEE Symposium on', IEEE, pp. 9–18.

Lu, Q. and Xu, X. (2017), 'Adaptable blockchain-based systems: A case study for product traceability', *IEEE Software* **34**(6), 21–27.

Ma, D. and Tsudik, G. (2009), 'A new approach to secure logging', *ACM Transactions on Storage (TOS)* **5**(1), 2.

Maleshkova, M. (2015), Towards Open Services on the Web-A Semantic Approach, PhD thesis, The Open University.

Martinovic, I., Kello, L. and Sluganovic, I. (2017), 'Blockchains for governmental services: Design principles, applications, and case studies', *Centre for Technology and Global Affairs— University of Oxford* .

Marty, R. (2011), Cloud application logging for forensics, *in* 'proceedings of the 2011 ACM Symposium on Applied Computing', ACM, pp. 178–184.

Masood, A. and Java, J. (2015), Static analysis for web service security-tools and techniques for a secure development life cycle, *in* '2015 IEEE International Symposium on Technologies for Homeland Security (HST)', IEEE, pp. 1–6.

Meier, J., Farre, C., Taylor, J., Bansode, P., Gregersen, S., Sundararajan, M. and Boucher, R. (2009), 'Improving web services security: Scenarios and implementation guidance for wcf', *Microsoft Developer Network* .

Microsoft (2012), 'The oauth 2.0 authorization framework', `https://tools.ietf.org/pdf/rfc6749.pdf`.

Nami, M. R. and Malekpour, A. (2008), Application of self-managing properties in virtual organizations, *in* 'Computer Science and its Applications, 2008. CSA'08. International Symposium on', IEEE, pp. 13–16.

Namiot, D. and Sneps-Sneppe, M. (2014), 'On micro-services architecture', *International Journal of Open Information Technologies* **2**(9), 24–27.

Nehme, A., Jesus, V., Mahbub, K. and Abdallah, A. (2018), Fine-grained access control for microservices, *in* 'International Symposium on Foundations and Practice of Security', Springer, pp. 285–300.

Nehme, A., Jesus, V., Mahbub, K. and Abdallah, A. (2019*a*), Decentralised and collaborative auditing of workflows, *in* 'International Conference on Trust and Privacy in Digital Business', Springer, pp. 129–144.

Nehme, A., Jesus, V., Mahbub, K. and Abdallah, A. (2019*b*), 'Securing microservices', *IT Professional* **21**(1), 42–49.

Neri, D., Soldani, J., Zimmermann, O. and Brogi, A. (2019), 'Design principles, architectural smells and refactorings for microservices: a multivocal review', *SICS Software-Intensive Cyber-Physical Systems* pp. 1–13.

Newman, S. (2015), *Building microservices: designing fine-grained systems*, " O'Reilly Media, Inc.".

NGINX, I. (2015), 'The future of application development and delivery is now', `https://www.nginx.com/resources/library/app-dev-survey/` note = Accessed: 2018-12-20.

Nofer, M., Gomber, P., Hinz, O. and Schiereck, D. (2017), 'Blockchain', *Business and Information Systems Engineering* **59**(3), 183–187.

Odat, A. M. (2012), E-government in developing countries: Framework of challenges and opportunities, *in* '2012 International Conference for Internet Technology and Secured Transactions', IEEE, pp. 578–582.

Otterstad, C. and Yarygina, T. (2017), Low-level exploitation mitigation by diverse microservices, *in* 'European Conference on Service-Oriented and Cloud Computing', Springer, pp. 49–56.

Pappel, I., Pappel, I., Tepandi, J. and Draheim, D. (2017), Systematic digital signing in estonian e-government processes, *in* 'Transactions on large-scale data-and knowledge-centered systems XXXVI', Springer, pp. 31–51.

Patanjali, S., Truninger, B., Harsh, P. and Bohnert, T. M. (2015), Cyclops: a micro service based approach for dynamic rating, charging and billing for cloud, *in* 'Telecommunications (ConTEL), 2015 13th International Conference on', IEEE, pp. 1–8.

Paxson, V. (1993), 'Empirically-derived analytic models of wide-area tcp connections'.

Pereira, Ó. M., Semenski, V., Regateiro, D. D. and Aguiar, R. L. (2017), 'The xacml standard: addressing architectural and security aspects'.

Preuveneers, D. and Joosen, W. (2017), 'Access control with delegated authorization policy evaluation for data-driven microservice workflows', *Future Internet* **9**(4), 58.

Priisalu, J. and Ottis, R. (2017), 'Personal control of privacy and data: Estonian experience', *Health and technology* **7**(4), 441–451.

Pulls, T., Peeters, R. and Wouters, K. (2013), Distributed privacy-preserving transparency logging, *in* 'Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society', ACM, pp. 83–94.

Putz, B., Menges, F. and Pernul, G. (2019), 'A secure and auditable logging infrastructure based on a permissioned blockchain', *Computers and Security* p. 101602.

Rahman, M. and Gao, J. (2015), A reusable automated acceptance testing architecture for microservices in behavior-driven development, *in* 'Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on', IEEE, pp. 321–325.

Rajalakshmi, J. R., Rathinraj, M. and Braveen, M. (2014), Anonymizing log management process for secure logging in the cloud, *in* '2014 International Conference on Circuits, Power and Computing Technologies [ICCPCT-2014]', pp. 1559–1564.

Rajani, V., Garg, D. and Rezk, T. (2016), On access control, capabilities, their equivalence, and confused deputy attacks, *in* '2016 IEEE 29th Computer Security Foundations Symposium (CSF)', IEEE, pp. 150–163.

Ray, I., Belyaev, K., Strizhov, M., Mulamba, D. and Rajaram, M. (2013), 'Secure logging as a service-delegating log management to the cloud', *IEEE Systems Journal* **7**(2), 323–334.

Redfield, C. M. and Date, H. (2014), Gringotts: securing data for digital evidence, *in* '2014 IEEE Security and Privacy Workshops', IEEE, pp. 10–17.

Richardson, C. and Smith, F. (2016), 'Microservices from design to deployment', `https://www.nginx.com/blog/microservices-from-design-to-deployment-ebook-nginx.pdf`.

Rimba, P., Tran, A. B., Weber, I., Staples, M., Ponomarev, A. and Xu, X. (2017), Comparing blockchain and cloud services for business process execution, *in* '2017 IEEE International Conference on Software Architecture (ICSA)', IEEE, pp. 257–260.

Robinson, N. and Martin, K. (2017), 'Distributed denial of government: The estonian data embassy initiative', *Network Security* **2017**(9), 13–16.

Rudolph, C., Kuntze, N. and Velikova, Z. (2009), 'Secure web service workflow execution', *Electronic Notes in Theoretical Computer Science* **236**, 33–46.

Saito, T., Tsunoda, Y., Miyata, D., Watanabe, R. and Chen, Y. (2016), An authorization scheme concealing client's access from authentication server, *in* 'Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2016 10th International Conference on', IEEE, pp. 593–598.

Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M. and Al-Hammadi, Y. (2016), The evolution of distributed systems towards microservices architecture, *in* 'Internet Technology and Secured Transactions (ICITST), 2016 11th International Conference for', IEEE, pp. 318–325.

Samlinson, E. and Usha, M. (2013), User-centric trust based identity as a service for federated cloud environment, *in* '2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)', IEEE, pp. 1–5.

Savchenko, D. I., Radchenko, G. I. and Taipale, O. (2015), Microservices validation: Mjolnirr platform case study, *in* 'Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on', IEEE, pp. 235–240.

Schoenmakers, B. (1999), A simple publicly verifiable secret sharing scheme and its application to electronic voting, *in* 'Annual International Cryptology Conference', Springer, pp. 148–164.

Shamir, A. (1979), 'How to share a secret', *Communications of the ACM* **22**(11), 612–613.

Sill, A. (2016), 'The design and architecture of microservices', *IEEE Cloud Computing* **3**(5), 76–80.

Siriwardena, P. (2014), *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*, Apress.

Slater, N. (2015), 'Using containers to build a microservices architecture', https://medium.com/aws-activate-startup-blog/using-containers-to-build-a-microservices-architecture-6e1b8bacb7d1.

Solinski, A. and Petersen, K. (2016), 'Prioritizing agile benefits and limitations in relation to practice usage', *Software quality journal* **24**(2), 447–482.

Souppaya, M., Morello, J. and Scarfone, K. (2017), 'Application container security guide', *NIST Special Publication* **800**, 190.

Stadler, M. (1996), Publicly verifiable secret sharing, *in* 'International Conference on the Theory and Applications of Cryptographic Techniques', Springer, pp. 190–199.

Sullivan, C. and Burger, E. (2017), 'E-residency and blockchain', *computer law and security review* **33**(4), 470–481.

Sun, S.-T. and Beznosov, K. (2012), The devil is in the (implementation) details: an empirical analysis of oauth sso systems, *in* 'Proceedings of the 2012 ACM conference on Computer and communications security', ACM, pp. 378–390.

Sun, Y., Nanda, S. and Jaeger, T. (2015), Security-as-a-service for microservices-based cloud applications, *in* 'Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on', IEEE, pp. 50–57.

Sundareswaran, S., Squicciarini, A. C. and Lin, D. (2012), 'Ensuring distributed accountability for data sharing in the cloud', *IEEE Transactions on Dependable and Secure Computing* **9**(4), 556–568.

Suryotrisongko, H., Jayanto, D. P. and Tjahyanto, A. (2017), 'Design and development of backend application for public complaint systems using microservice spring boot', *Procedia Computer Science* **124**, 736–743.

Suzic, B. (2016*a*), Securing integration of cloud services in cross-domain distributed environments, *in* 'Proceedings of the 31st Annual ACM Symposium on Applied Computing', ACM, pp. 398–405.

Suzic, B. (2016*b*), User-centered security management of api-based data integration workflows, *in* 'NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium', IEEE, pp. 1233–1238.

Suzic, B. and Reiter, A. (2016), Towards secure collaboration in federated cloud environments, *in* 'Availability, Reliability and Security (ARES), 2016 11th International Conference on', IEEE, pp. 750–759.

Suzuki, S. and Murai, J. (2017), Blockchain as an audit-able communication channel, *in* '2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)', Vol. 2, IEEE, pp. 516–522.

Tang, L., Ouyang, L. and Tsai, W.-T. (2015), Multi-factor web api security for securing mobile cloud, *in* 'Fuzzy Systems and Knowledge Discovery (FSKD), 2015 12th International Conference on', IEEE, pp. 2163–2168.

Tang, Y. R., Xing, Z., Xu, C., Chen, J. and Xu, J. (2018), Lightweight blockchain logging for data-intensive applications, *in* 'International Conference on Financial Cryptography and Data Security', Springer, pp. 308–324.

Tapas, N., Merlino, G., Longo, F. and Puliafito, A. (2019), Blockchain-based publicly verifiable cloud storage, *in* '2019 IEEE International Conference on Smart Computing (SMARTCOMP)', IEEE, pp. 381–386.

Thompson, N., Ravindran, R. and Nicosia, S. (2015), 'Government data does not mean data governance: Lessons learned from a public sector application audit', *Government information quarterly* **32**(3), 316–322.

Thönes, J. (2015), 'Microservices', *IEEE Software* **32**(1), 116–116.

Tian, F. (2017), A supply chain traceability system for food safety based on haccp, blockchain and internet of things, *in* 'Service Systems and Service Management (ICSSSM), 2017 International Conference on', IEEE, pp. 1–6.

Tian, H., Chen, Z., Chang, C.-C., Kuribayashi, M., Huang, Y., Cai, Y., Chen, Y. and Wang, T. (2017), 'Enabling public auditability for operation behaviors in cloud storage', *Soft Computing* **21**(8), 2175–2187.

Twining, P., Raffaghelli, J., Albion, P. and Knezek, D. (2013), 'Moving education into the digital age: The contribution of teachers' professional development', *Journal of computer assisted learning* **29**(5), 426–437.

Vahi, K., Harvey, I., Samak, T., Gunter, D., Evans, K., Rogers, D., Taylor, I., Goode, M., Silva, F., Al-Shakarchi, E. et al. (2013), 'A case study into using common real-time workflow monitoring infrastructure for scientific workflows', *Journal of grid computing* **11**(3), 381–406.

Vandikas, K. and Tsiatsis, V. (2016), Microservices in iot clouds, *in* 'Cloudification of the Internet of Things (CIoT)', IEEE, pp. 1–6.

Velikova, Z., Schütte, J. and Kuntze, N. (2009), Towards security in decentralized workflows, *in* 'Ultra Modern Telecommunications and Workshops, 2009. ICUMT'09. International Conference on', IEEE, pp. 1–6.

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R. and Gil, S. (2015), Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud, *in* 'Computing Colombian Conference (10CCC), 2015 10th', IEEE, pp. 583–590.

Wang, L., Du, Y. and Liu, W. (2017), 'Aligning observed and modelled behaviour based on workflow decomposition', *Enterprise Information Systems* **11**(8), 1207–1227.

Waters, B. R., Balfanz, D., Durfee, G. and Smetters, D. K. (2004), Building an encrypted and searchable audit log., *in* 'NDSS', Vol. 4, pp. 5–6.

Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A. and Mendling, J. (2016), Untrusted business process monitoring and execution using blockchain, *in* 'International Conference on Business Process Management', Springer, pp. 329–347.

Werner, M. and Gehrke, N. (2015), 'Multilevel process mining for financial audits', *IEEE Transactions on Services Computing* **8**(6), 820–832.

Wombacher, A., Wieringa, R., Jonker, W., Knežević, P. and Pokraev, S. (2005), Requirements for secure logging of decentralized cross-organizational workflow executions, *in* 'OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"', Springer, pp. 526–536.

Wouters, K., Simoens, K., Lathouwers, D. and Preneel, B. (2008), Secure and privacy-friendly logging for egovernment services, *in* '2008 Third International Conference on Availability, Reliability and Security', pp. 1091–1096.

Xiong, Y. and Du, J. (2019), Electronic evidence preservation model based on blockchain, *in* 'Proceedings of the 3rd International Conference on Cryptography, Security and Privacy', ACM, pp. 1–5.

Yao, J., Chen, S., Wang, C., Levy, D. and Zic, J. (2010), Accountability as a service for the cloud: From concept to implementation with bpel, *in* 'Services (SERVICES-1), 2010 6th World Congress on', IEEE, pp. 91–98.

Yarygina, T. (2017), Restful is not secure, *in* 'International Conference on Applications and Techniques in Information Security', Springer, pp. 141–153.

Yarygina, T. and Bagge, A. H. (2018), Overcoming security challenges in microservice architectures, *in* '2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)', IEEE, pp. 11–20.

Yu, Y., Silveira, H. and Sundaram, M. (2016), A microservice based reference architecture model in the context of enterprise architecture, *in* 'Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), 2016 IEEE', IEEE, pp. 1856–1860.

Zawoad, S., Dutta, A. and Hasan, R. (2016), 'Towards building forensics enabled cloud through secure logging-as-a-service', *IEEE Transactions on Dependable and Secure Computing* (1), 1–1.

Zawoad, S., Dutta, A. K. and Hasan, R. (2013), Seclaas: secure logging-as-a-service for cloud forensics, *in* 'Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security', ACM, pp. 219–230.

Zhang, H., Li, Z. and Wu, W. (2012), Open social and xacml based group authorization framework, *in* '2012 Second International Conference on Cloud and Green Computing', IEEE, pp. 655–659.

Zimmermann, O. (2016), 'Microservices tenets: agile approach to service development and deployment', *Computer Science-Research and Development* **32**(3), 301–310.