TU WIEN Informatics

Design and Implementation of a Blockchain-based Zero Trust Architecture on the Edge

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Cem Bicer, BSc.
Matrikelnummer 01425692

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.-Prof.Dr. Schahram Dustdar
Mitwirkung: Dr. Ilir Murturi

Wien, 2. Mai 2023

Cem Bicer

Schahram Dustdar

TU WIEN Informatics

Design and Implementation of a Blockchain-based Zero Trust Architecture on the Edge

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Cem Bicer, BSc.
Registration Number 01425692

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof.Dr. Schahram Dustdar
Assistance: Dr. Ilir Murturi

Vienna, 2nd May, 2023

Cem Bicer

Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Cem Bicer, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2. Mai 2023

Cem Bicer

Danksagung

An dieser Stelle möchte ich mich bei meinem Professor, Dr. Schahram Dustdar, bedanken, der mir die Chance gegeben hat, diese Diplomarbeit zu schreiben. Des Weiteren möchte ich mich bei meinem Mitbetreuer Dr. Ilir Murturi bedanken, der mir während meiner Arbeit stets zur Seite gestanden hat und mir wertvolle Ratschläge gegeben hat.

Ein großer Dank gilt meinen Eltern und Geschwistern, die mich in meiner gesamten Ausbildungslaufbahn unterstützt haben. Ohne ihre Ermutigung und Unterstützung wäre ich nicht hier, wo ich heute stehe.

Insbesondere möchte ich meiner Frau Eda danken, die mich nicht nur emotional, sondern auch fachlich unterstützt hat. Ihre Geduld, Unterstützung und Ermutigung haben mich während dieser intensiven Zeit sehr gestärkt.

Ich bin dankbar für all die Menschen in meinem Leben, die mich auf diesem Weg begleitet haben.

Acknowledgements

At this point, I would like to express my gratitude to my professor, Dr. Schahram Dustdar, who gave me the opportunity to write this thesis. Additionally, I would like to thank my co-advisor Dr. Ilir Murturi, who supported me throughout my work and provided valuable advice.

A big thanks goes to my parents and siblings who have supported me throughout my education. Without their encouragement and support, I would not be where I am today.

In particular, I would like to thank my wife Eda, who has supported me not only emotionally but also professionally. Her patience, support, and encouragement have greatly strengthened me during this intense time.

I am grateful for all the people in my life who have accompanied me on this journey.

Kurzfassung

Internet der Dinge (IoT) Netzwerke setzen sich üblicherweise aus einer enormen Anzahl an verbundenen Geräten zusammen, die alle verschiedene Software- und Hardware-Eigentschaften aufweisen. Die Sicherheit in solch einem heterogenen System aufrechtzuerhalten, kann zu einer Herausforderung werden, denn die Angriffsfläche dieses Systems ist hier sehr groß und die Diversität der Knoten gibt Angreifern die Möglichkeit, eine Vielzahl an verschiedenen Angriffsvektoren zu verwenden um in das System einzubrechen. Konventionalle perimeterbasierende Systeme verwenden Zugangsdaten (Benutzername/Password, Zertifikate, etc.), um die Authentizität der Akteure zu bestimmen und um zu entscheiden, welchen Akteuren der Zugriff ins Netzwerk gewährleistet wird. Jedoch wird das nur beim erstmaligen Eintritt ins System überprüft, was bedeutet, dass ein Angreifer nur die Zugangsdaten eines systembekannten Akteurs abgreifen muss, um Zugriff ins System zu erlangen. Da die meisten IoT-Geräte nur mit schwachen Chips ausgestattet sind und auch kaum Software oder Hardware zum Bekämpfen von böswilligen Aktivitäten (z.B. AntiVirus Software) zur Verfügung stehen, ist das Risiko relativ hoch, dass Geräte innerhalb des Systems gekapert werden. Um diese Gefahr zu verringern, könnte eine andere Sicherheitsarchitektur verwenden werden: die Zero Trust Architektur. Zero Trust Systeme vertrauen keinen Akteuren implizit. Die Vertrauenswürdigkeit jedes Akteurs wird kontinuierlich überprüft und die Richtlinien zur Gewährung des Zugangs zum System ändern sich dynamisch auf der Grundlage unterschiedlicher Eigenschaften dieses Akteurs.

Diese Arbeit schlägt eine neuartige Cybersecurity-Architektur vor, die eine Zero Trust Architektur verwendet, wie sie vom National Institute of Standards and Technology (NIST) beschrieben wird. Zusätzlich integriert die neue Architektur ein Blockchain-Netzwerk, um die Sicherheitslage des Systems noch weiter zu verbessern. Die Blockchain-Komponente dient als unveränderliche Datenbank zur Speicherung von Anfragehistorien, die zur Verifizierung der Vertrauenswürdigkeit von Akteuren verwendet werden. Wir stellen das Design dieses Systems bereit und geben eine umfassende Beschreibung der Verantwortlichkeiten jeder Komponente.

Schließlich bewerten wir das Design, indem wir es tatsächlich implementieren und auf einer Testumgebung bereitstellen, um dann einige Testfälle auszuführen. Wir haben außerdem verschiedene Varianten dieses Systems implementiert und dieselben Tests auf diesen durchgeführt. Die Ergebnisse werden dann zwischen den Systemen verglichen, um zu demonstrieren, wie sich einige Designentscheidungen auf nichtfunktionale Ei-

genschaften der von dieser Arbeit entworfenen Architektur auswirken. Die Bewertung konzentriert sich auf nichtfunktionale Eigenschaften wie Performance, Skalierbarkeit, Implementierungsaufwand und Komplexität. Die Ergbenisse unserer Evaluierung zeigen, dass es möglich ist, ein IoT-System zu implementieren, das die Zero Trust Architektur mit integrierter Blockchain umsetzt.

Abstract

Internet of Things (IoT) networks usually contain a huge number of connected devices, which are very diverse in hard- and software. Maintaining security in such a heterogeneous system can be very challenging, as the attack surface of that system is very large and the diversity of the nodes allows attackers to use a variety of different attack vectors to breach into the system. Today's conventional perimeter-based systems use credential-based authenticity (username/password, certificates, etc.) to decide if an actor is allowed to access the network. However, this is only verified once on the system's perimeter, which means that gaining access as an attacker is as hard as getting access to the credentials of a device already known to the system. As most IoT devices are equipped with low-performance chips and often lack software or hardware to block any breaches (e.g. antivirus software), the risk of devices getting hijacked is relatively high. To reduce this risk, a different security architecture could be used: the Zero Trust Architecture. Zero trust systems do not trust any actors implicitly. The trustworthiness of each actor is constantly verified and the policies for giving access to the system change dynamically based on the different properties of that actor.

This thesis proposes a novel cybersecurity architecture that uses a Zero Trust Architecture as described by the National Institute of Standards and Technology (NIST) and integrates a blockchain network into it to further enhance the security posture of the system. The blockchain component serves as an immutable database for saving request history, which is used for verifying the trustworthiness of actors. We provide the design of the said system and give a comprehensive description of each component's responsibilities.

We provide an actual implementation of the designed system and evaluate non-functional properties by executing some test cases against it. We also implemented different variants of that system and executed the same tests against those as well. The results are then compared between the systems to demonstrate how some design decisions affect the non-functional properties of the architecture designed by this thesis. The evaluation focuses on non-functional properties like performance, scalability, implementation effort, and complexity. Our evaluation results show that it is feasible to design and implement an IoT system implementing the Zero Trust Architecture with an integrated blockchain.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Motivation and Problem Statement 1
1.2 Aim of the work 2
1.3 Running example 3
1.4 Research questions 4
1.5 Structure 4

2 Background **7**
2.1 Edge-to-Cloud Continuum 7
2.2 Zero-Trust Architecture 8
2.3 Blockchain 13

3 Related work **17**
3.1 Zero Trust 17
3.2 Blockchain-based Zero Trust 19

4 System Design **21**
4.1 Components 21
4.2 Security measures 28
4.3 Technology stack 31
4.4 Communication 33

5 Evaluation **41**
5.1 Implementation effort 41
5.2 Performance 42
5.3 Scalability 48
5.4 Complexity 49
5.5 Advantages and Challenges 49

6 Conclusion **51**
6.1 Research Questions Revisited 52
6.2 Limitations 53
6.3 Future work 54

List of Figures **55**

List of Tables **57**

List of Algorithms **59**

Acronyms **61**

Bibliography **63**

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

Most current cybersecurity methods for computer networks use perimeter-based security models which focus on protecting resources with identity-checking mechanisms using cryptography to ensure that only authorized entities can access a secure area. The identity of an actor - be it a user or a thing - is proven by providing login credentials - like username and password - or by providing a certificate which contains details about the actor. In both cases, there is a logical component in the network, which checks the authenticity of a peer before they can access the requested resource. After a successful check, the actor is considered trustworthy and they get access to the requested resource. This authentication strategy worked very well in the past and is still actively used in many secured networks.

However, with the rise of the Internet of Things (IoT), new challenges emerged regarding securing resources. IoT networks typically have many heterogeneous devices which all communicate with each other or with servers within the network. Those devices usually send data to or read data from the servers. Ideally, all "things" in the network are authenticated somehow and the network only accepts data from those authenticated and trusted devices to prevent unauthorized access. This cybersecurity characteristic rejects requests from external, untrusted and maybe even malicious entities by design. However, as the number of connected devices in an IoT network is often very high, the risk of data leaks is also high due to more potential points of failure. If trusted devices are hijacked by attackers they could access resources from the secured network without being detected, as the network already authenticated the compromised devices on the first connection and assumes that it is still trustworthy.

An example of an IoT network with a large number of heterogeneous devices is a smart city. The things in a smart city can have different types of sensors to sense the environment,

such as cameras for monitoring pedestrian crossings or traffic jams, temperature and humidity sensors for monitoring weather conditions in public parks or sensors for detecting technical defects of elevators in malls. All those devices communicate either directly with the cloud or with an edge server which then sends the data to the cloud. In a smart city, there could be sub-networks that group some devices or servers together to function as a standalone IoT network, or all public devices could be part of a single huge IoT network. With such a large number of devices operating in a smart city, the attack surface for potential attacks is huge and the risk of the system being attacked is more likely. Furthermore, if an attacker is successful in compromising a device and gaining access to the internal infrastructure of the smart city, it could have a significant impact on the smart city's security.

In such a scenario, the Zero-Trust (ZT) approach can help to reduce the above-mentioned security risks. The principles of zero-trust focus on protecting *resources* such as data or services, rather than protecting an entire network or domain. With the ZT approach, no implicit trust is assumed and all entities are treated as untrustworthy at any point in time by default, whether they are internal or external. On each incoming request, the ZT system verifies some properties of the requester to decide if the access is granted or rejected, and if the access is granted the given access rights are always as strict and atomic as possible to only allow executing this specific request. This approach assumes that a connected peer could be compromised at any time at any transaction and therefore checks its privileges, access rights and previous behaviour on every transaction [RBMC20]. The philosophy of a zero trust system is "never trust, always verify" [SD18].

In this sense, it is very critical that such a system does not allow tampering with the request history of actors and that the request validation component, which is responsible for granting or rejection incoming requests, cannot be manipulated or controlled by attackers. To overcome the risk of manipulating persisted data, a distributed ledger can be utilized, and for preventing hijacked or dead validation nodes to decide on granting or rejecting requests, a consensus mechanism between all validating nodes can be utilized. The request history of actors can be logged in a blockchain for immutability of the request history and the request validation can be distributed to multiple components and the final decision can be made using a consensus algorithm.

1.2 Aim of the work

The goal of this thesis is to showcase a Proof-of-Concept (PoC) system which uses the ZT architecture - as defined in the publication of the National Institute of Standards and Technology (NIST) "Zero Trust Architecture" [RBMC20] - in the context of IoT networks, backed by a private blockchain and extended by a consensus-based request validation process. The PoC is designed to meet the requirements of a ZT architecture (Zero Trust

Architecture (ZTA)) according to NIST and implemented as an almost[1]-production-ready system to show the applicability of such systems in an edge-to-cloud continuum environment. The implementation meets the requirements of all the mentioned basic tenets within the NIST's definition of ZT and ZTA - either fully or partly (see Section 2.2 for a description of the tenets) [RBMC20, pp. 6-7]. The PoC also sheds a light on challenges and advantages that arise when opting for using the ZT paradigm as the security mechanism for an IoT network. The evaluation and discussion specifically focus on the scalability, general performance and complexity - not in the sense of computational complexity, but rather level of difficulty in understanding, designing, or managing - of the system.

1.3 Running example

The running example will be in the context of smart cities, as already hinted in the previous sections. A smart city can be a huge network of devices all connected to the internet. To not go beyond the scope of this thesis, we only implemented a small portion of a smart city in the PoC. This way we can focus on the core concept of the ZTA and how it fits into the concept of IoT.

The PoC concretely implements the public park monitoring system part of a smart city, more precisely the basic components of such a monitoring system. There are stationary actors and users which interact with the system. Stationary actors can be temperature or humidity sensing IoT devices which are mounted somewhere in a public park - hence the term "stationary" - and which send temperature or humidity data to the system periodically. Users, on the other hand, are humans which read those data for monitoring purposes, like gardeners from the public service. Those two components are the only "external" components, meaning that they are not part of the components of the ZTA itself.

To further help understanding the way the system works, we will focus on some use cases which will be mentioned throughout this thesis and which will demonstrate some properties of the system. Consider situations as below:

- A gardener of the public service wants to know the current humidity level of the soil in a park without having to be on-site.
- The system should notice compromised devices and should not let them access resources or send data anymore
- The system should notice malicious transactions, like requesting access to data which is not meant to be accessed by the requester.

[1]The implemented PoC cannot be used in production as-is. To make the system ready for production, one has to think of typical non-functional requirements like availability, fault-tolerance, etc. The PoC shall only be considered as a proof that the concept is realizable as a running system.

- It should be possible to add new IoT devices to the system and they should be able to send or read data, depending on their roles.

Some of those use cases will be demonstrated more theoretically, some more practically - by showing how the PoC behaves in certain scenarios.

1.4 Research questions

The main research questions the thesis is going to focus on and answer are as follows.

RQ1: How can the zero-trust paradigm be applied to secure smart city systems with heterogeneous IoT devices using blockchain technology?

In particular, what components are needed for implementing such a system to meet the requirements of a ZTA? How do those components communicate with actors or with each other and how do they validate the trustworthiness of actors on incoming requests? A PoC will be used to demonstrate concrete implementations of the ZT concepts.

RQ2: Which implementation advantages and challenges are implied when deciding to apply the zero-trust architecture in general?

Here we concentrate on the advantages and challenges of *implementing* a zero-trust system, not on the zero-trust concept itself. This should give future technical architects and system designers some hints on what they should consider when either switching from a conventional cybersecurity system to a ZT system or when starting to develop a system from scratch and needing to decide which security concept they want to use.

RQ3: How scalable/performant is the system compared to today's conventional systems?

This question, on the other hand, concentrates on the *concept design* used in the PoC. It clarifies how the chosen system design is either more or less scalable or performant in comparison to conventional security systems. This is evaluated in a theoretical manner, as well as with actual performance tests on a test environment.

1.5 Structure

The remaining sections of this thesis are organized as follows: In Chapter 2, we discuss the background of the thesis and the current state-of-the-art in the context of IoT networks, ZTA systems and blockchains. Chapter 3 presents the design of the implemented PoC, by listing its components with their responsibilities and the security measures taken to have an almost-production-ready system. In Chapter 4 we will present some technical details about the PoC implementation. In Chapter 5 we present the evaluation results of

some important non-functional characteristics of the PoC. Last but not least, Chapter 6 offers a conclusion and a discussion of the findings and opportunities for future work.

CHAPTER 2

Background

This chapter provides the background and context for the key concepts used in this thesis: the edge-to-cloud continuum, the zero-trust architecture, and blockchain technology. It briefly explains the relevance of the edge-to-cloud continuum in today's world. It also introduces the concept of ZTA with its tenets and core components and explains its significance in today's increasingly connected and complex digital landscape, and how blockchain technology can play a crucial role in securing it.

2.1 Edge-to-Cloud Continuum

More and more devices get connected to the internet as time passes. According to IoT Analytics, there were around 10bn connected IoT devices in 2019 and they forecast that there will be more than 3x the number in 2025 (30.9bn) [Lue20]. The more devices communicate via the internet, the more data they will produce. This can lead to performance problems when the data is directly sent to cloud servers for real-time processing, as those servers are usually located far away from the device sending or receiving the data. Not only is the latency an issue, but also the processing power of cloud servers is usually not strong enough to handle huge loads of data at once.

To overcome these issues, another layer of servers can be employed: the Edge or Fog [BMZA12]. Edge computing - or Fog computing as it is called by Cisco - is a relatively new computing strategy that essentially moves the computational power of a network closer to the end user - i.e. the computation is done "on the edge of the network". By doing this, data does not have to travel the long way to the cloud server to be processed, but the closer-located edge servers undertake the processing, reducing latency. When those edge devices are equipped with powerful processing units, the performance of the network can also increase in comparison to usual cloud computing. Like the actors and users within an IoT network, the edge servers in the edge-to-cloud continuum can also be very heterogeneous.

Although there is no standardized definition for a "smart city" [YXC+15] and there are also no instructions on how to build such a city, the edge-to-cloud continuum makes perfect sense to be used within a smart city. In fact, Dapeng et al. [LLT+21] were even able to show in a practical way that the edge-to-cloud continuum - which they refer to as fog-cloud continuum in their paper - is applicable to smart cities. Applying this to a whole city which is basically a network of sub-networks, where a sub-network is an IoT network on its own, we get a *huge* number of dissimilar devices all being connected to the smart city network.

2.2 Zero-Trust Architecture

Using a conventional perimeter-based approach in a smart city, by having a perimeter in which authenticated users are trusted, may not be the most secure security model. The huge number of heterogeneous devices connected to the network opens up a large surface for potential cyber attacks against the network. Additionally, most IoT devices are only equipped with processing units with limited processing power to save energy and cost, which results in those devices not being able to appropriately defend themselves against cyber attacks. This increases the risk of attackers breaching the perimeter of a network through an already authenticated connected device. Once entered, an attacker can move freely within the network as the authentication process had already been finished and will not be repeated again. Unfortunately, it is almost impossible or at least very hard to detect if a device requesting resources from the network is still trustworthy or if it had been hijacked by a malicious entity. The authenticity of that device may be validated successfully, but this does not rule out the possibility, that someone else is trying to hide behind the device's identity. Thus, trusting only the authenticity of actors is not sufficient anymore.

The term "zero-trust" had been used by John Kindervag for the first time in 2010 [KB+10]. In the same year, the NIST introduced guidelines for a zero-trust architecture [RBMC20]. In a network implementing the ZTA, no entity is trusted by default and each and every request to the system is checked for possible malicious activities. The zero-trust architecture tries to combat the above-mentioned issue with the high number of potential entry points for attackers, by introducing additional security checks during the process of validating the trustworthiness of actors. Identity checks are not enough to guarantee that an actor is trustworthy.

2.2.1 The 7 Tenets

The NIST suggests extending the security checks and using a more dynamic policy. A dynamic policy not only checks for identity, but also for the application or service the actor used for making their request, the resource they want to access, the environment the actor is in, how the actor requested the resource, and how they behaved in the past. All those checks together form a policy that is not as static as checking user credentials, but rather adapts to changing actors, environments, and behavior. Based on those criteria,

the actor is allowed to execute their request or the request will be rejected. Using a dynamic policy to determine access to resources is one of the tenets the NIST listed (**tenet no. 4**) [RBMC20, p. 6].

In a ZTA, the target to be secured is not the whole network or a domain but rather single resources. A resource can be a file, a service (e.g., an API), or a database, meaning that even if an actor wants to read something from the system and then - depending on the read value - wants to save something in a database, the ZT mechanism may not allow those actions to be executed in a single transaction but could rather split them into two requests, where each request is validated on its own. Those two characteristics are another two tenets defined by the NIST (**tenets no. 1 and 3**): (1) every data and service is a resource and (2) access to resources is granted on a per-session basis, which means that it may not be allowed to request two independent resources in a single transaction (or session).

There is also no secure area within the network which automatically implies trust. Network location alone is not enough to tell if an actor is trusted (**tenet no. 2**). Applied to the running example this means that even if actors request resources from within a government-owned private network, they are not given access without validating the request first. There is, however, an "implicit trust zone" which is an area where entities that have been given access to a resource can stay in. This area is ideally as small as possible and the duration an actor stays in this zone is very small so that they only have the chance to execute their initial request and nothing more.

As already mentioned, in a ZTA all requests are validated before access is given to the requested resource. The whole authentication and authorization process is a constant cycle of evaluating the trustworthiness of the requester with each incoming request (**tenet no. 6**). This implies that no permission is inherited from previous requests. Depending on the granularity of the policy, a system could grant access to multiple resources at once to finish a transaction without validating each access to a resource, or it could reauthenticate and reauthorize each resource access separately. A ZTA system could also reevaluate the trustworthiness of actors on a time-based approach - e.g. an actor could be allowed to access a resource a couple of times within a given time period - or it could trigger the evaluation only when resources are modified or written. Different variants are allowed here, as long as authentication and authorization happen regularly.

Another important criterion in a ZTA is monitoring (**tenet no. 5**). It is always a good idea to monitor activities in a system to be able to understand how the system behaves in certain situations and how the system is used by actors. In a ZTA system, it is even more crucial to have good monitoring mechanisms in place, because there it's not only important for understanding specific flows of the system, but also to be able to validate the trustworthiness of actors more accurately. The more details the system has about an incoming request, the better it can identify any malicious actions. Not only that, but with good monitoring capabilities a system is able to find vulnerabilities and subverted entities more efficiently and can adapt policies more dynamically according to that.

Linking to the previous tenet, a zero-trust system should not only monitor activities, but also collect as much information as possible (**tenet no. 7.**). For instance, information about the last interactions of actors and requests to specific resources can increase the quality of decision-making. ZTA system could include those information when enforcing their policies. The more a system collects about actors, the more it can verify the authenticity and intentions of them, and the better it can predict the next actions. Being able to predict subsequent interactions of actors can be very valuable. Deviations in the predicted behavior pattern could be a sign of hijacked devices.

It is important to note that, while it is recommended that a ZTA network complies to those basic tenets, the NIST emphasizes that it is not necessary to implement all of them in their purest form into the ZT system [RBMC20, p. 6]. They see it more as an *ideal goal* to have all seven incorporated into the security mechanism.

2.2.2 Assumptions for network connectivity

Enterprises implementing the ZTA should not only apply the above-mentioned tenets, but should also make the following assumptions about the ZTA network according to NIST [RBMC20, p. 8].

1. **There is no implicit trust zone in the network, not even the enterprise-owned private network should be assumed to be an implicit trust zone.** This assumption is more or less implied by tenet 2 (see above). Every request could come from a malicious attacker and should also be handled as such. All communication should therefore be authenticated and all traffic should be encrypted.
2. **Not all connected devices and actors may be owned or configurable by the enterprise.** This needs to be considered when designing the policies. Enterprises may define policies for enterprise-known actors trying to access enterprise-owned resource, but from nonenterprise-owned devices.
3. **No resources are inherently trusted.** This is similar to tenet 6 but applies to resources.
4. **Enterprise-owned resources could be located in nonenterprise-owned infrastructure.** Those resources could, for instance, be located in external cloud servers.
5. **Enterprise subjects and assets not located in the enterprise-owned infrastructure cannot fully trust their local network connection.** Remote enterprise subjects and assets always have to assume that the cloud server they are located in, is potentially malicious or hostile.
6. **Assets being passed between enterprise- and nonenterprise-owned components need to retain their security posture.** The communication workflow between enterprise and nonenterprise components and the assets transferred in

between need to consistently retain their security posture and not leak any vulnerabilities.

2.2.3 Logical components

Apart from the above-mentioned tenets and assumptions, NIST also define components of a ZTA. There are core components that are crucial for every ZTA and components that can vary according to the system's intentions and use cases. However, the core components do not have to be present as physical (hardware or software) components but should rather be seen as logical components, meaning that a component could be split into multiple actual components which together form the logical components, or multiple logical components could be merged into a single physical component, respectively. All components mentioned in this section are depicted in Figure 2.1. The following list describes the core logical components of a ZTA according to NIST [RBMC20, p. 9].

- **Policy Engine (PE):** The PE component is responsible for the actual decision-making. It gathers input from multiple data sources and based on that calculates if access to the requested resource should be granted or not. The set of instructions for the decision calculation is called the *trust algorithm* (see Section 2.2.4). The trust algorithm is fed with the incoming request and inputs from the data sources and outputs a yes/no decision, where "yes" means "the request is allowed to be executed and the access to the requested resource shall be granted" and "no" means the opposite. The policy engine component together with the Policy Administrator (PA) component forms the "Policy Decision Point (PDP)" logical component. As mentioned previously, the PE and PA components could also be merged into a single PDP component.
- **Policy administrator (PA):** As the name suggests, this component administers the granting/rejecting of requests. The PA is closely tied to the PE and relies on its decision result. When the PE calculates a yes decision, the PA starts establishing the connection to the requested resource. When the PE calculates a no decision, the PA shuts down the connection and the request is rejected. The PA is, however, not responsible for actually shutting down the connection or for providing the requested resource, but it rather instructs the Policy Enforcement Point (PEP) component (see description of this component below) to shutdown the connection or it configures it for establishing a connection to the requested resource. This configuration could be an access token that can be used to access the requested resource. The access token should only be valid within a predefined time-period, which depends on the policies of the system. It could, for instance, be valid for only this single request or it could be a time-based token that expires after a given time.
- **Policy enforcement point (PEP):** This component is the entry point to the system. Requests to resources are sent to the PEP and the PEP either tell the requester, that the request is rejected or gives access to the requested resource.

As explained before, the PEP gets instructions from the PA to decide whether to grant or deny access. Here again, the PEP does not have to be a single physical component, but could also be split into a client-side and a resource-side component. The client-side component would be installed on the actor device (e.g. as an agent on their laptop) and the resource-side component would be installed in front of the actual resource (e.g. as a gateway controlling access to it). As can be seen in Figure 2.1, the PEP is the gateway between the untrusted and the trusted zone.

Aside from the core components, there are also additional components with various different responsibilities. Those components are mainly used for feeding the trust algorithm with input to decide if an incoming request should be granted or rejected. The additional components may not be located within the enterprises internal network and may even not be controlled by them. Some of them are listed and described below. Not all of the listed components have to be implemented in a ZTA, and on the other hand, a ZTA could introduce other additional components as well. The NIST describes the following additional components [SD18].

- **Continuous diagnostics and mitigation (CDM) system:** This component is responsible for checking properties of the requesting actor's agent. This includes, but is not limited to, checking the agent's operating system for vulnerabilities and its integrity. CDM systems are also responsible for validating nonenterprise-owned devices, i.e. agents not controlled by the enterprise.
- **Industry compliance system:** This component contains all policies to comply to any regulations of a regime. This includes, for instance, being compliant with the GDPR (General Data Protection Regulation) of the European Union.
- **Threat intelligence feed(s):** Any newly discovered threats, attacks, vulnerabilities, or any kind of flaws in software are reported by this component to the policy engine. Those information could be gathered from internal or external sources.
- **Network and system activity logs:** This component saves network activities within the ZT system. Such activities include network traffic, access requests of actors and other events which can be useful when enforcing the policies.
- **Data access policies:** The actual access policies to resources are provided to the PE by this component. It contains access privileges needed to access resources, among other data access properties.
- **Enterprise public key infrastructure (PKI):** This component is responsible for generating public keys and logging issued certificate within the enterprise.
- **ID management system:** Details about known actors are saved in this component. It contains subject information like username, email address and access rights and roles of actors. This component could also use external services like lightweight directory access protocol (LDAP) servers to identify subjects.

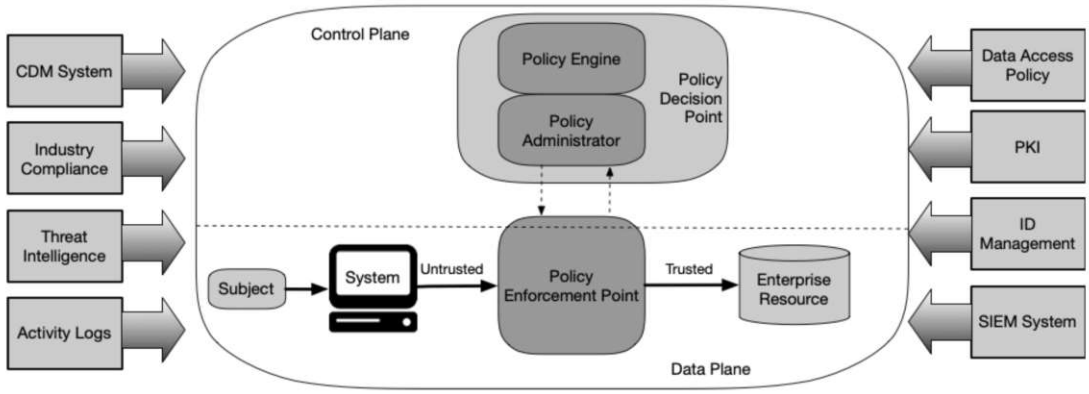


Figure 2.1: ZTA core components (within the Control Plane and Data Plane) and additional components used as data sources to enforce the policies (outside the Control/Data plan area) [RBMC20, p. 9]

- **Security information and event management (SIEM) system:** This component collects security-relevant information. With this data, the ZT system's policy can later be refined.

2.2.4 Trust algorithm (TA)

As mentioned before, the PE runs a trust algorithm which consists of instructions to calculate a decision for an incoming request. Each incoming request is passed to the trust algorithm and with the input of various data sources (i.e. the additional components as mentioned above), a decision is made whether the incoming request should be trusted or not. The actual input depends on the system's use cases and security posture, but in general the input types can be grouped as the following: information about the requester (*identity, authentication*), information about the request itself (*usage*), information about the agent the actor requested with (*environment*), the history of the requesting actor (*behaviour*) and general threats observed within the system or "around" it - i.e. threats that could affect the system, but have not been observed within yet.

The TA does not have to output a binary decision. It is also feasible that the trust algorithm returns a score-based output which can be used to compare two or more requests for confidentiality. This way the ZT system could, for instance, discover when actors get less confidential with each request and could react accordingly, e.g. by adapting the policies for that actor or for the whole system.

2.3 Blockchain

The term "blockchain" had been first introduced by Satoshi Nakamoto [Nak08] in the context of the first ever cryptocurrency Bitcoin, and the technology gained popularity

ever since. The blockchain technology is, however, not limited to the Bitcoin and not even to cryptocurrencies in general. Many other applications arose, making use of the characteristics and properties of the blockchain technology. One of them being the concept of smart contracts - which will be briefly explained in Section 2.3.2.

Blockchain is a technology that allows for the creation of a digital, decentralized and distributed ledger. It is a peer-to-peer network where each node holds a copy of the digital ledger, and new entries added to it can be verified by all the peers. This property of the blockchain technology allows for a consented network without requiring any centralized authority to verify the correctness of transactions. The ledger in a blockchain is - as the name suggests - a chain of blocks, where each block holds a reference to the previous block. A block contains transactions that have been done within the network. One very important property of a blockchain is the immutabily of the ledger. Although the ledger is distributed - with each peer holding a copy of it - peers cannot simply change their copy and use this as the correct ledger. Whenever a new block is added to the chain, all the peers within the network validate the correctness of it and try to reach consensus.

2.3.1 Consensus algorithm

There are many different types of consensus algorithms used for blockchains. The very first consensus algorithm type used in the context of blockchains was Proof-of-Work (PoW) [Nak08], namely in the Bitcoin network. PoW is a Byzantine Fault Tolerant (BFT) consensus algorithm, meaning that even if there are malicious or broken nodes within the blockchain network, the system is still able to reach consensus. This characteristic is analogues to the Byzantine Generals Problem [LSP82]. Since the introduction of the Bitcoin network, we have had other consensus algorithms used in blockchain networks as well, e.g. Proof-of-Stake (PoS) and Raft [OO14].

In general, in a **proof-of-work** consensus algorithm, a block contains a (hash) value, which can only be obtained by doing an expensive calculation - this value is the *proof* that the peer adding the new block did some *work* to generate this block. Consensus is reached when all peers verified that the calculation is correct. Verifying the correctness of that value, on the other hand, is very easy and fast (*one-way function*).

Proof-of-stake consensus algorithms try to get rid of the expensive calculation - which costs a lot of energy [KÖ19] - by introducing pseudo-randomness. When there are two or more block candidates, the block which comes from the peer biding the most coins will be accepted more likely then the others. Coins are the rewards for adding a block to the chain, so the more blocks a peer already added to the blockchain, the more it can bid for the next block and the more likely the block of this peer is selected. Proof-of-stake had been first introduced in the Ppcoin blockchain [KN12]. Proof-of-stake is also a BFT algorithm.

The **Raft** consensus algorithm is not a BFT algorithm, but it is a Crash Fault Tolerant (CFT) algorithm. *Raft* cannot prevent malicious nodes from being detected and overruled, but it can handle crashes and unavailability of nodes very well - hence the term **crash**

fault tolerant. Raft is mainly used when the nodes in the network are known to be benign, e.g. in a permissioned (= private) blockchain network, where the using organization has control over the nodes. In general, nodes in the network are chosen randomly to be *leader candidates*. All the candidates then send requests to all nodes in the network to vote for it to be the leader - which node votes for which candidate is defined by an algorithm that is not explained here, as this would go beyond the scope of this thesis. If a leader candidate got the majority of votes, it sends a heartbeat to all other nodes, stating that it is the leader now. Now this node is officially the *leader* and all other nodes are the *followers*.

The Raft algorithm is crash fault-tolerant because it still can reach a consensus when some followers crashed or are not available. As long as there are more than half of the nodes available, consensus can be reached. Moreover, if no leader candidate got the majority of votes or all leader candidates crashed for some reason, the election times out, and a new round of leader candidates is selected [OO14]. Raft is the recommended consensus algorithm in Hyperledger Fabric (HLF) (v2.4), a framework for permissioned blockchains [ABB+18].

2.3.2 Smart contract

Although the term "smart contract" had been introduced in 1997 by Nick Szabo [Sza97], it gained special attention with the rise in popularity of blockchain networks. A smart contract is - in blockchain terminology - a software program containing a set of rules (= instructions) that have been agreed on by some parties and which is saved and executed in a blockchain. Due to the fact that a blockchain is immutable, smart contracts are also immutable. This guarantees that all agreed parties have the same copy of the contract and adhere to it without needing a third-party entity to ensure that. The HLF blockchain framework uses the term "chaincode" for a group of smart contracts. In a HLF blockchain, *chaincodes* are deployed and the contained *smart contracts* are executed [ABB+18].

A crowdfunding system could, for instance, be implemented using smart contracts. There could be a smart contract collecting funds from donors until a certain amount is reached, to then transfer the money to the peer who initiated the crowdfunding campaign. Technically speaking, the donors would call a function with the amount as parameter, which would accumulate all donations, and as soon as a given amount is reached, the smart contract would automatically transfer the money to the crowdfunding initiator.

CHAPTER 3

Related work

As zero-trust architecture is a relatively new research area, there are many papers within the last couple of years to this topic. In this chapter, we list and analyse some related research papers in the context of the Internet of Things, Zero Trust Architecture, and Blockchains. The focus is on papers employing the zero-trust security mechanism (see Section 3.1) and systems utilizing a blockchain in a zero-trust system (see Section 3.2). We briefly summarize the papers in the next couple of paragraphs.

3.1 Zero Trust

Xiaojian et al. proposed zero-trust security for a power IoT network [XLJ+21]. The designed system checks multiple properties of actors, calculates a trust level based on those attributes and compares this level with the security level of the requested resource. The security level of a resource (e.g. a service) is defined according to the importance of that resource. The trust level of actors are calculated on a per-session basis and the access is granted or denied dynamically. There is a central rule base which contains rules (=policies) for accessing a resource. This rule base is taken into consideration when calculating the trust of actors and it is adapted dynamically based on the actor's behaviour and other properties, e.g. version, vulnerability patch, and identity. This calculation is done by a central service, called the *trust analysis engine* - this is similar to the *Policy Engine* component mentioned in the NIST definition.

In the paper from DeCusatis et al. [DLSP16] the zero-trust security approach is applied in a cloud computing context in a different and interesting way. They proposed a novel architecture which uses steganography to embed authentication data in TCP packets. This approach eliminates the ability for attackers to fingerprint the network, as a connection request fails silently if an actor is not trustworthy. This is due to the fact that the authentication data is embedded in the transport layer. This means that the TCP connection between an actor and the system will not even be established if the actor

is not trustworthy. From a technical point of view, the system consists of two gateways which are responsible for enforcing policies: the first gateway embeds an authentication token in the first packet of the TCP request, and the second gateway reads that token and enforces the actual policies. If the actor is trusted, the second gateway allows the TCP connection. If the actor is not trusted, e.g. it tries to access a resource which it is not allowed to access, then the second gateway denies the connection by not allowing the TCP connection request to finish. In NISTs ZTA terminology, the second gateway is the Policy Enforcement Point (PEP) component. With this approach, the latency for policy enforcement is low and the bandwidth is high, as the actual packet content is not inspected during the trust-check.

One of the most sensitive data of human beings is their medical details. These data needs to be secured very thoroughly and access should only be given to permitted people, which are usually the patient the medical data is for or the doctor who created or issued it. Since the rise of the IoT and the 5G network, this medical data is not only available in paper form anymore but more and more medical institutions allow online downloads of the patient's medical documents. This comes with a big security risk as it allows malicious people to spoof their identity to get sensitive data of others, or to get access to servers holding the data through other malicious ways. Chen et al. presented a medical system using a zero-trust architecture which tries to combat the above-mentioned security issues [CQZ+20]. Instead of only checking the identity of the actor requesting data from the system, they introduced a four dimensional (4D) access control framework which checks the *subject*, *object*, *environment* and *behaviour* of actors. The subject is the identity of the actor, the object is the resource the actor wants to access, the environment is the network the actor requested data in or the system the actor requests with, and the behaviour dimension is the access history of the actor. The system performs risk judgment based on the the two dimensions *subject* and *environment* to calculate scores for potential risks associated with different combinations of subject and environment. The risk report is then used in the trust assessment process to decide whether the requesting actor is given access to the requested resource or not.

In a system employing zero-trust security principals the amount and quality of data collected from actors is crucial for a strong security posture. It is important to not only rely on identity data but also collect other types of data, such as data about the current system state the actor operates in. Garcia-Teodoro et al. [GTCMF+22] introduced a zero trust access control mechanism that creates security profiles of actors and based on that decide whether the actor is given access to the requested data or not. The mentioned security profile is built by collecting lots of different information about the actor, e.g. installed OS, MAC address of the used device, current status of the OS like RAM usage or level of battery, recent network traffic of the used device, and many more (see page 7 in [GTCMF+22] for a full list of all collected data). This strategy has the huge advantage that attacks would need to spoof lots of OS or device related data to fully whitewash the hijacked device. Even if there is no malicious person involved, this approach is also able to detect malfunctions of devices more accurately and thus it can react faster in

such a scenario. On top of the sheer amount of collected data the proposed system also makes use of machine learning to improve the analysis quality with time. The more data is collected and validated the more dynamic and accurate the system can operate.

3.2 Blockchain-based Zero Trust

Samaniego et al. introduced a blockchain-based middleware for managing access to resources inside an IoT network, using the zero trust paradigm [SD18] and called it Amatista. The unique selling point of Amatista is that it uses a two-level hierarchical mining process. There are first-level miners which get the sensor data of IoT devices, validate the identity of the sender and then forward the data to the second layer. The second-level miners then validate the access rights of the sending device and check if it has the privileges to execute the request, e.g. sending temperature measurements to the weather station or fetching location data of another IoT device. The second layer uses a consensus algorithm to agree on the validity of transactions before a block for a transaction is stored in the blockchain. However, Amatista does not implement the zero-trust part of its system as recommended by the NIST [RBMC20]. The zero-trust part in Amatista is the embedding of a blockchain with miners which use a consensus algorithm to decide whether to trust or not trust incoming requests.

Another zero-trust blockchain approach in the context of IoT was introduced by Sultana et al. [SHL+20]. This paper focuses on a medical IoT network where patients and doctors or medical technologists share medical test results like MRI scans or X-ray files. The medical technologist, e.g. radiologist, sends the X-ray files to the system which creates a block in the blockchain. This block contains some typical block information like the sender and receiver of the X-ray files and also a reference to the location where the files are saved - the files are persisted in a separate database. After the block has been written to the blockchain, the receiver, e.g. the patient, can fetch the X-ray files. This system has its focus on transferring large data and therefore uses the approach with the separate database next to the blockchain. It also describes how the workflow of sending and receiving data looks like and how the peers authenticate. Although the authors mention the ”zero-trust” principle a lot and the paper states to define a system with zero trust, it does not fully implement the actual zero-trust paradigm as defined by NIST. The senders of data, e.g. the medical technologists, have to login with their username/password and the sender’s device’s health is checked to make sure that it is not hacked or compromised - which totally conforms with the zero trust paradigm - but the device of the receiver is not checked when trying to fetch the data. The receiver, e.g. the patient, only has to authenticate with a username/password pair and Two-Factor-Authentication and then gets access to their test results. What is also missing in this conceptual system is the validation of the peer’s behaviour before granting access to resources and also dynamically adjusting privileges based on the observations of the behaviour - which is tenet no. 4 in the NIST definition of a zero trust architecture [RBMC20].

Dorri et al. [DKJG17] utilised a blockchain to secure the communication between IoT

devices within a smart home. The network consists of IoT devices which communicate with each other or with the cloud, a private local blockchain for saving transactions, several local storages to save IoT data and a miner which handles all transactions inside the network or incoming and outgoing transactions from or to the smart home network. Each smart home only has a single miner which is the central processor for adding blocks to the blockchain and for granting permissions for resources inside the network. This approach, however, is missing the zero trust property. If a device wants to access a resource or wants to communicate with another device in the network, it asks the network's miner for permission. The miner checks the access policies, and if the requester has the needed privileges, both the requester and the requestee get a shared key for secure communication. The requester is not checked for trustworthiness, but only checked for identity. This check is, however, one of the main properties of a zero-trust architecture [RBMC20].

CHAPTER 4

System Design

The core of this thesis is the design and implementation of a Proof-of-Concept combining all above mentioned technologies in a self-contained system. The design of the system complies with the ZTA tenets and implements the core components mentioned in the ZTA definition of the NIST [RBMC20]. Furthermore, the blockchain components are implemented using the Hyperledger Fabric framework, building a permissioned blockchain. The PoC is centered around a use case for a smart city, which means that it considers properties of such a network, i.e. allowing distribution of components to multiple edge servers or considering IoT devices with low computational power. By giving insights into the PoC's inner components and presenting the intercommunication of those components, the reader shall be able to evaluate non-functional properties of the system, such as the effort needed to implement it, the complexity, and the usability of it.

4.1 Components

The system consists of three groups of components: ZTA components, Blockchain components, and IoT components. The ZTA components are the NIST core components and additional components for ensuring zero-trust in the system. Together they enforce the defined policies and grant or reject incoming requests. For logging the incoming requests and their access decisions a distributed ledger is used, which is implemented as a permissioned blockchain. The system is designed to be accessible by users as well as stationary IoT devices equipped with environmental sensors. The big picture of the system with all its physical components can be seen in Figure 4.1.

4.1.1 ZTA Components

The core components are listed above in Section 2.2.3. However, the components are only described logically and are very high-level. In addition to the high-level definition, we

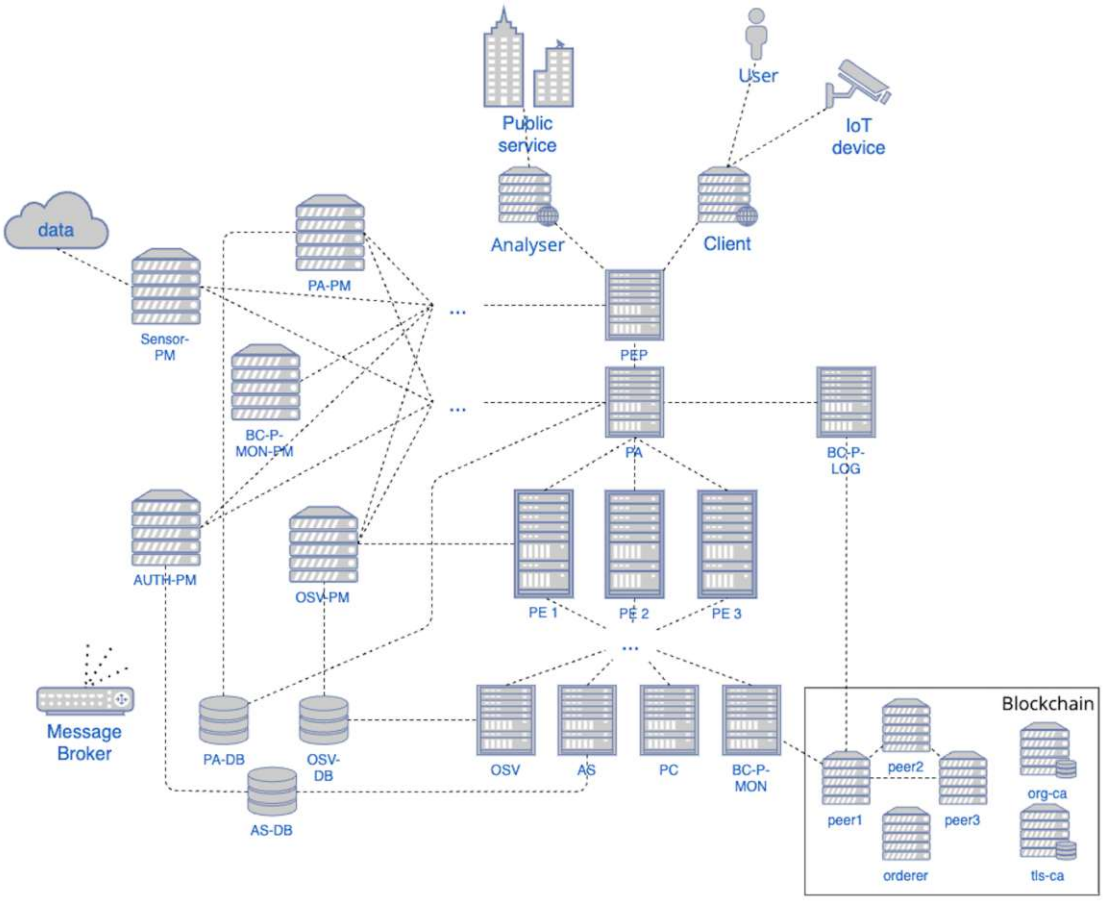


Figure 4.1: Network diagram of the PoC

provide their concrete responsibilities and their concrete technical parts within the PoC in the next paragraphs. Apart from the core components, there are also some additional components for building and evaluating policies.

Core components:

Policy Enforcement Point: The PEP is a single logical component, but can be broken into two different physical components, as mentioned in the NIST definition [RBMC20] and above in Section 2.2.3. In the PoC system, the Policy Enforcement Point consists of a component on server-side - called **PEP** - and two client-side components for connecting to the server - called **Analyser** and **Client**. Users and stationary IoT devices communicate via the Client component and system maintainers interact with the Analyser component. The Client component is implemented as a simple gateway which enriches incoming requests by some important details about the actor and passes them to the PEP component. The PEP component is only accessible via the two client-side components. The Analyser component, on the other hand, is restricted to users

with administration rights and it only fetches maintenance data like connected policy engines, known actors and their request history. The PEP component takes incoming requests from the client-side components and passes them to the Policy Administrator for validation. After the request is validated and the access is granted, the PEP will fetch or send the requested data from or to the corresponding Persistence Managers. If, for instance, the government installed a new stationary device in a public park for measuring the outside temperature and wants to connect this device to the smart city network, an administrator would use the Client component to send the creation request to the PEP, which further passes it to the PA. If the PA grants the access, the PEP sends the new stationary device's details to the Authentication Persistence Manager (AUTH-PM) to persist and from now on, the new stationary device is recognised by the system. If the access gets rejected, the PEP does not communicate with any Persistence Manager (PM), but it tells the requester that the request had been rejected.

Policy Administrator: The PA is responsible for initiating the validation of incoming requests and for creating access tokens for the PEP which are needed for accessing one of the PMs. The Policy Administrator does not do the actual validation, but it sends a validation request to all known PEs, which do the actual validation. The validation process is executed using a simple consensus algorithm: all PEs are triggered to start validating the incoming request, the PA is notified by each PE when the validation finished, and if more than the half of all PEs resulted in the same decision, this decision is taken as the correct one. Depending on the type of request, the PA either waits for the PEs to finish validating or sends the validation request to all and immediately returns to the PEP with the message that the validation had been triggered. Requests for saving (non-administrative) data are executed asynchronously and all administrative requests as well as all GET requests are executed synchronously. This is due to the fact that actors are not always interested in the response to their request, e.g. when sending their sensor data to the system. Additionally, the IoT devices send sensor data regularly, thus blocking the system for those kinds of requests affects performance. If the request is executed asynchronously, the PA notifies the PEP via a message broker. Regardless of whether the request had been executed synchronously or asynchronously, the PA sends an access token to the PEP and to the corresponding PM if the access is granted. The PM then only accepts requests from this access token. An access token consists of a unique secret (string), a time-to-live value representing the validity duration of this access token (in seconds), and a list of access rights for telling the PM what type of requests the caller is allowed to execute with this access token. The PEP needs to attach this token to the request when accessing the PM.

Policy Engine: The PoC consists of a couple of PEs. All of them have the same source code so they do exactly the same. Each PE runs an instance of the Trust Algorithm (TA), which contains the policies and rules for granting access to a resource, hence the TA is the brain of the whole validation system. The TA is executed as soon as the PE gets a validation request from the PA. All PEs are triggered simultaneously and the TA is

executed synchronously. The Trust Algorithm fetches various data and details about the requester - user or stationary actor - and the incoming request from different components. This data is then evaluated step-by-step and a decision is generated out of them. The TA checks the authenticity of the requester with data from the Authentication Service (AS) (= identity checks), the operating system of the requester for vulnerabilities with data from the Operating System Vulnerabilities (OSV) component (= environment checks), the parameters of the incoming request with the help of the Parameter Checker (PC) (= usage checks), and the history of the requester for any suspicious activities from the Blockchain Peer Monitoring (BC-P-MON) component (= behaviour checks). As soon as the validation finishes, the PE sends the decision to the PA.

Additional components:

Authentication Service: The AS is one of the components which provide input data for the Trust Algorithm. Details about all known users and stationary actors are persisted in the AS's database, such as their IDs, access rights and IP/MAC addresses - IP and MAC addresses are only saved for stationary actors. Checks against the AS's data are: (a) for users, only the access rights are checked, and (b) for stationary actors, in addition to the access rights, the IP and MAC addresses from the incoming request are compared with the addresses in the database. The TA fetches those details and checks, for instance, that the requester is allowed to execute the incoming request according to their access rights. The AS only reads data from the database and is not able to modify it. For adding, updating or deleting authentication details about actors, the Authentication Persistence Manager (AUTH-PM) has to be used (see the description of Persistence Managers (PMs) below for more details about the responsibilities of PMs).

Operating System Vulnerability: The OSV is another component used for validation. It saves details about known vulnerabilities of operating systems. The source of this information could be an external service like the Common Vulnerabilities and Exposures (CVE) service[1] or the known vulnerabilities could be added manually via an API. For simplicity, some hardcoded vulnerabilities are inserted on startup and new vulnerabilities can be added via the OSV API in the PoC. Details about the requester's operating system are included in the incoming request and this information is checked against the data in the OSV's database.

Parameter Checker: The PC is responsible for checking parameters of incoming requests, more precisely syntactic and semantic correctness of values, e.g. if an IP address is syntactically correct or if the value of a temperature reading is semantically valid.

Blockchain Peer Monitoring (BC-P-MON): The BC-P-MON component has the identity - i.e. certificate and private key - of a peer from within the blockchain network. It is able to fetch the historical data of actors from the blockchain through the installed

[1] https://www.cve.org/

chaincode (smart contract). The fetched data is checked for malicious or suspicious activities by the TA. For instance, if the last X requests had been rejected because the actor tried to fetch data from a restricted resource, and the next incoming request tries to fetch the same data again, the TA is able to recognise this suspicious activity and can therefore block the actor temporarily. There could be more sophisticated techniques when analysing the history of actors in place. The TA could also check for specific patterns in the history to identify malicious or hijacked actors.

Persistence Manager (PM): There are many different PM components in the PoC. Each type of resource has a dedicated PM in front of it and accessing it is only possible through the dedicated PM. Every incoming request has to go through the whole ZT chain, starting with the client-side PEP component through the PA, PE, the validation components, and at the end to the PM. No resource is allowed to take a shortcut - see ZTA tenet no. 6 in Section 2.2.1. This implies that every request type has to have a PM to handle it. For instance, if the system wants to support reading the electricity consumption of public buildings, it has to implement a PM which can access the electricity data of those buildings. As can be seen in the big picture of the PoC (see Figure 4.1), some PMs access databases which are also accessed by validation components. For instance, the AUTH-PM component and the AS component are both connected to the AS-DB. It is, however, not possible to modify data in the AS-DB from the AS component. The AS component only supports fetching data needed for validating incoming requests. It does not have methods to e.g. delete or modify data, whereas the AUTH-PM component has full access (read and write) to the database. Additionally, as already mentioned above, the PM can only be accessed via a valid access token, which needs to be registered by the PA first.

Blockchain Peer Logging (BC-P-LOG): This component is used for logging incoming actor requests in the blockchain. The PA sends a log request to this component after the PEs consented to a validation decision. The BC-P-LOG component also has the identity - again, certificate and private key - of a peer from the blockchain. It takes the incoming request and the decision outcome as input from the PA and sends it to the chaincode to persist it in the blockchain.

4.1.2 Blockchain Components

In addition to the above mentioned components for ensuring zero-trust in the PoC, there are also dedicated components which are needed to build and run a permissioned blockchain within the system[2]. The only connection between the ZT components and the blockchain components are the BC-P-* components. This connection is established by allowing those components to use the certificate and private key of the same peer

[2]The blockchain network is used as another additional component which provides input for the trust algorithm.

from within the blockchain network to interact with the installed chaincode. All other ZT components do not know the blockchain components and vice versa.

The blockchain technology of choice is the HLF framework[3]. HLF is an open source framework for building permissioned blockchains which provides pre-built Docker images[4] and good documentation [Hyp22b] to build a blockchain network. Before listing and describing the components in the PoC, we first provide a brief and high-level introduction to the Hyperledger Fabric infrastructure.

Hyperledger Fabric Blockchain Infrastructure

Hyperledger Fabric allows creating multi-organizational permissioned blockchain networks [ABB+18]. Each organization can have one or more nodes collaborating in the blockchain. An organization's nodes are identified by the organization-owned certificate authority, which issue certificates for them. There are also peers which are the fundamental elements in a HLF network. They are basically nodes, that host the distributed ledger and chaincodes. Organizations can contribute to the blockchain by deploying own peers. They are also the entrypoints to the blockchain, as organizations can use those peers to interact with the network, e.g. by sending or fetching data to or from it. Within a HLF blockchain network, there can be one or more channels which peers can join to create a blockchain. Each channel has its own ledger, and peers can join multiple channels at once, leading to a very flexible network where organization could have some peers in many channels and some peers only serving one ledger, i.e. being joined in only one channel. This also means that peers could host multiple different smart contracts, modifying multiple ledgers.

Smart contracts are the main tools to interact with ledgers. They are deployed by peers and are only valid within the channel they had been deployed in. However, a channel can have multiple smart contracts and chaincodes, respectively. There is a slight difference between a smart contract and a chaincode: smart contracts are functions to interact with the ledger and chaincodes are containers of one or more smart contracts. The ledger consists of two pieces: the blockchain, which packages all transactions within the network into chunks (*blocks*), chains them together and logs them in an immutable way, and the world state, which is basically a data type containing business objects that can be inserted, updated, deleted or fetched with the help of the installed chaincode.

In comparison to HLF blockchains, other blockchain networks like Ethereum[5] or Bitcoin[6] are not permissioned, meaning that any node can participate in the consensus process, during which transactions are ordered and bundled to blocks. As there is no control over all nodes, the consensus algorithms in such a network work with probabilities, which will eventually guarantee consistency within the ledger. This is not the case

[3] https://www.hyperledger.org/use/fabric

[4] A list of all Docker images of the Hyperledger Team can be found in https://hub.docker.com/u/hyperledger

[5] https://ethereum.org/de/

[6] https://bitcoin.org/

for permissioned blockchains: all nodes within the network are controlled by one or a couple of organizations. In HLF, this property of permissioned blockchains is used to eliminate the probabilistic part of the consensus algorithm. To make the consensus process deterministic, HLF networks contain so-called *orderer* nodes - which together form the *ordering service* [ABB+18]. These nodes are responsible for ordering the transactions within a blockchain.

The consensus algorithm of the HLF framework from chaincode invocation to adding a block to the blockchain works like follows [ABB+18]:

1. When a smart contract is called, the calling application sends a transaction proposal to one or more endorsers in the network, according to the endorsement policies of the channel the transaction is executed in. The concrete framework which does this in our PoC implementation is the Fabric gateway service. This service is the entry point to interact with a blockchain, which implements a simple API for submitting transaction proposals.

2. These endorsers execute the transaction proposal against the blockchain, but without really modifying it - i.e. they only simulate the execution. They sign the transaction afterward, which tells the other components that the signing endorser actually handled the transaction.

3. When enough endorsers simulated the transaction to satisfy the endorsement policy - this requires that all endorsers returned the same simulation result - the transaction is created and sent to the ordering service.

4. This and other transactions are then packaged into blocks and distributed to all peers within the channel to update their local copy of the distributed ledger. The packaging is done by the ordering service.

5. Lastly, all peers independently validate the received block. This is done by e.g. checking that all required signatures are present (according to the endorsement policy)

PoC's blockchain components:

Organizations: For simplicity, the PoC only consists of a single organization (called *org*), with its own certificate authority (called *org-ca*).

Certificate authorities: Apart from the organizations certificate authority, there is also a certificate authority for creating TLS certificates (called *tls-ca*). This TLS CA creates certificates for the organization itself and the organization creates certificates for its peers.

Peers: In the PoC, there are three peers (*peer1*, *peer2* and *peer3*), but only one peer (*peer1*) is used for submitting transaction proposals. This could be extended, such that all peers are used for submitting transaction proposals so that the system can still continue to work when one peer is down.

Orderer: Again for simplicity, there is only one orderer node, which is also the only component in the ordering service.

Chaincode: There is only one channel and it only contains one chaincode. This chaincode contains smart contracts for fetching history data from actors - which are saved in the world state - and for persisting incoming requests of actors in the world state.

4.2 Security measures

Zero-trust is not the only cybersecurity property of the PoC. Having a system implement a ZT architecture alone, does not make it safe against cyberattacks or malicious actors by any means. Nowadays, encryption is used in many high-quality production software applications as a security measure. Encryption can take place in, e.g. the login process, during communication between actors, or when saving credentials in the persistence layer. The PoC also makes use of encryption to strengthen the system against some types of cyberattacks. There are some additional security measures the PoC uses, which are listed and explained in the next paragraphs.

Zero-Trust

The most obvious security measure within the PoC is the zero-trust property. Within a zero-trust system, even known-to-be-good devices and actors are checked for trustworthiness continuously, in the case that they one day could be untrustworthy because they got hijacked or manipulated in a malicious way. Zero-trust also implies that every interaction with the system and within the system is logged and policies are changed dynamically according to - but not limited to - these logs. Dynamic policies also make sure that the system is flexible and can adapt to new vulnerabilities more efficiently.

Encryption

Like most secure production systems, the PoC also employs encryption. All traffic within the system is encrypted using Transport Layer Security (TLS). Each component has its own certificate and private key and communication is encrypted (e.g. using HTTPS) between components. The blockchain peers have certificates signed by the organization's CA and they only communicate with nodes which are known to the system - this is guaranteed by HLF's Membership Service Provider (MSP) concept. In summary, the MSP[ABB+18] is a directory which contains a list of known identities. Each component has its own MSP, which it can use to verify the identity of nodes within the network.

The ZTA components, on the other hand, have certificates signed by the system's root Certificate Authority (CA) and all ZTA components only communicate with components with certificates signed by this root CA.

Blockchain

As mentioned above, blockchain technology also comes with some implicit security measures. The fact that blocks within a blockchain cannot be tampered with easily[7], also introduces a security measure for such systems. In the PoC, an attacker would only need to hijack two out of three peers to have control over more than 51% of the nodes, but in a real production system, there would be more than only three peers to operate the blockchain.

Another security measure which comes with a blockchain is the redundancy of data. As the blockchain is distributed over all peers within the network, data can not get lost that easily. As long as one peer is still running, the blockchain state can always be recovered from that peer. The blockchain within the PoC is a permissioned blockchain, which means that the ledger data is always in the hands of known peers.

Consensus

In addition to the consensus algorithm of the blockchain, our Proof-of-Concept implementation uses another consensus algorithm in a different place as well. As mentioned in Section 4.1.1 in the context of the PA component, the validation process is also implemented to use a consensus algorithm. The consensus algorithm of choice is Practical Byzantine Fault Tolerance (PBFT) [CL+99]. In the PoC, the PA lets all PEs validate the incoming request and when the majority resulted in the same decision outcome, this result is taken as the correct decision. The Policy Administrator acts as the moderator, which initiates the validation process and then waits for the results to accept the one with the most occurrences (*Majority Voting* system). Figure 4.2 shows how the validation process with a compromised PE would look like. Again, for an attacker to control the validation process, it needs to have control over at least half of all PEs.

Network segmentation

As already mentioned before, the ZTA components do not know the blockchain components - except for the BC-P-* components. Additionally, some components within the network are only accessible from specific components. In the PoC this is guaranteed with Docker's networking feature and by restricting access to some critical communication endpoints of components to only specific identities via their certificates (see Section 4.4 for more details). For instance, the PEP cannot connect to the PEs, the PA cannot

[7] *Easily* does not mean, that it impossible to manipulate the blockchain. There are many attack vectors against blockchains, but in general, the attacker needs lots of resources or power to do that, e.g. for the 51% attack, an attacker needs control over more than half of all nodes within a blockchain [SMG19]

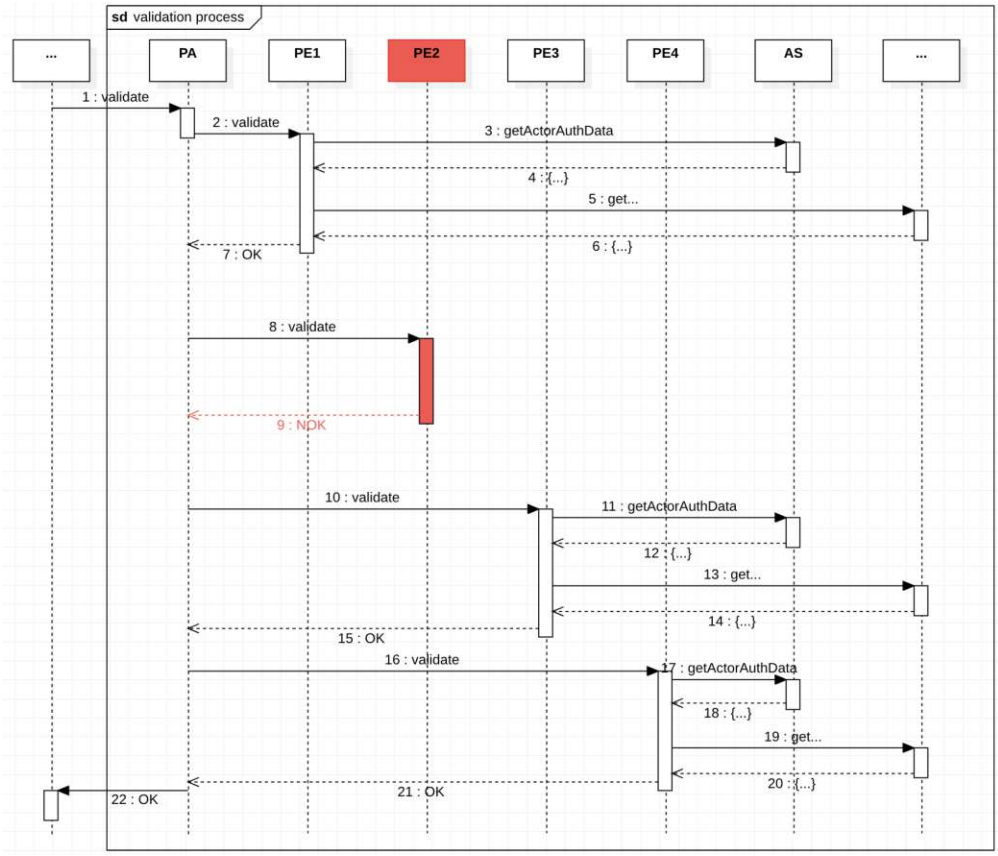


Figure 4.2: Validation process starting from the PA. PE2 is compromised and acts maliciously (it returns "not OK" for a valid request). The PA still returns OK to the caller, as three out of four PEs answered with OK. For readability, only the AS is shown from the validation components and the rest is represented by the "..." lifeline

connect to any of the additional validation components (e.g. AS, OSV, etc.), and a validation component cannot register an access token in a PM, because the endpoint for doing that is restricted to some components only. This has the advantage that the whole network cannot be compromised if only one component is controlled by an attacker. In addition to that, we can make sure that the system is only accessible by the provided clients, by only allowing interactions with the server-side PEP component with a valid client certificate.

Resource separation

As far as resources are concerned, they are somehow secured by all above security measures (e.g. encryption, consensus, zero-trust) and in addition to them, the PoC also separates the physical resource (e.g. a database) as follows: when authentication data like an IP address is needed for validating the IP address provided by a stationary actor

in an incoming request, the AS component fetches it from the authentication database (read-operation). However, when an administrator wants to change the IP address of a stationary actor, the actual update-operation is done from the dedicated PM, hence the PoC separates validation-relevant resource access from actual actor request operations.

4.3 Technology stack

Each component in the PoC, i.e. in Figure 4.1, runs in a Docker container - except for the actual actors and users (e.g. public service) - and they are all orchestrated with Docker Compose[8]. All blockchain components use Docker images provided by HLF, all ZTA core components, validation components, PM components, and the Blockchain (BC) logging component use Spring Boot Docker images. The HLF blockchain components had to be set up and initialised correctly and the ZTA components had been implemented from scratch in a microservice manner[9], meaning that the system is split up into its smallest possible units (microservices) and the communication between those units happen via REST APIs (synchronously). The client-side components of the PEP are implemented with Angular (*Analyser* component) and Python (*Client* component). For asynchronous communication, Redis[10] is used as the message broker of choice.

4.3.1 ZTA components

The ZTA-specific components are all Spring Boot Applications[11] which are implemented in Java[12]. Spring Boot is specifically designed for implementing microservices and it has great support for many technologies used in microservices, such as JSON serialization, asynchronous communication via messages brokers and API encryption, to name some. The Spring Boot applications are implemented in a 3-tier architecture, i.e. there is a presentation layer (the REST API), an application layer (the services containing the business logic) and a persistence layer (the database connection). Not all ZTA components need a persistence layer, but those who do (e.g. the AS component for reading authentication data of actors), use the SQL database PostgreSQL[13] and/or the NoSQL key-value database Redis.

4.3.2 HLF components

As mentioned before, the HLF components are Docker containers built from Docker images provided by Hyperledger themselves. There are images for peers, orderers, CAs,

[8] https://docs.docker.com/compose/

[9] The proof-of-concept implementation does not employ a *pure* microservice approach. All validation components connect to databases which are also used by the dedicated Persistence Managers. In a microservice system, two microservices should not share the same database with each other. To overcome this, there could be two separate databases which are synchronized regularly.

[10] https://redis.io/

[11] https://spring.io/

[12] https://www.java.com/

[13] https://www.postgresql.org/

and chaincode environments. These images are very powerful in terms of customization and configuration possibilities. It is possible to configure an image almost completely from within a Docker compose file with the help of environment variables. HLF also provides binaries to build a blockchain network natively on a Linux, Windows or OSX machine. For the connection between the blockchain network and the ZTA components, we used the Hyperledger Fabric SDK for Java[14]. This is a convenient framework for connecting to a HLF permissioned blockchain from a Java application - in this case from a Java Spring Boot application. As far as the chaincode implementation is concerned, Hyperledger Fabric provides interfaces for Java, Go and Node.js [Hyp22c] for implementing chaincode logic. The proof-of-concept implements a chaincode in Java, so the chaincode is packaged into a JAR file and deployed in a separate Docker container.

4.3.3 Client components

There are two different PEP client-side components which have to be used to interact with the system: the Analyser component and the Client component. The former is implemented as an Angular application and can be used for monitoring the system's state, e.g. active policy engines, known actors, and actors' request history. The Analyser client component has its own certificate which is the entry key to the whole system. The Client component, on the other hand, is some kind of proxy server for interacting with the system. It is implemented in Python and provides a REST API which basically mirrors the API of the server-side PEP component. However, incoming requests will not be passed to the PEP as they come in, but the Client component intercepts the request and adds system specific details to the request, like installed OS, MAC address, etc. The Client component needs to be installed on the actor's device directly or on an edge device receiving requests from an IoT device, if the IoT device does not have enough resources to run a Python application. This component has a client certificate as well to allow interactions with the backend system. The system does not allow connections to the API from other sources because the two client components have to inject environment-specific information like OS version and MAC address. An incoming request without environmental details like the two mentioned device information are rejected, as the zero-trust system can only work as expected, if these kind of information is given - remember that the PoC checks identity, usage, behaviour and *environment* properties of the actor. Restricting the system to only the known clients is done by only allowing access to the server-side Policy Enforcement Point for entities with a root-CA-signed client certificate[15]. Access for entities, which cannot provide such a valid client certificate, is rejected.

4.3.4 Other components

The big picture in Figure 4.1 shows a cloud component with the text "data". In the PoC, we did not really use a cloud for sensor data, but simply used a SQL database - running in a Docker container defined in the Docker compose file - to store it. However, in a

[14]https://hyperledger.github.io/fabric-gateway-java/

[15]The PoC uses X509 certificates and identifies "client certificates" with the attribute *OU=client*

production environment, this simple database would be a data center which could be located far away from the smart city and is reachable via a cloud service.

The remaining components (*User*, *IoT device* and *Public service*) are actual users and devices with sensors, respectively. A user could be an employee working for the government (e.g. gardener) who needs to read data from the smart city (e.g. humidity level in a park). This employee would use a device like a smartphone or computer which has the Python client component running. On top of that, a user-friendly UI application (e.g. a smartphone app or a website) could be implemented which is connected to the Python client. The public service office would use the Analyser UI directly for monitoring purposes.

4.4 Communication

All components within the system communicate over the internet - in the PoC, the host machine acts as the internet. We used a mix of synchronous and asynchronous communication in our implementation. Synchronous calls are executed against the REST APIs when the result of the request contains important data. For instance, when creating a new user, the request needs to be done synchronously, as the system returns an API key for that user. On the other hand, if an IoT device sends temperature data, the system does not need to create a response and can silently save the incoming temperature data in the background. In general, all GET requests are executed synchronously.

Although some requests are handled synchronously and others asynchronously, all incoming requests go the same route (see Figure 4.3 for the workflow of an example request). As mentioned before, the first stop for every request to the system is one of the known client components. Here the request is extended by details about the device's environment. The extended request is sent to a Policy Enforcement Point[16] which is the gateway from the outside world to the trust zone. From now on, the remaining communication needed for validating and finally executing this request happens within the so-called *implicit trust zone* [RBMC20, p. 5]. In this zone, components always trust each other, i.e. the receiver of a message does not validate the sender's trustworthiness and assumes that it can trust the sender and the incoming message. However, this does not really affect the system's security as the request from the not-yet-trusted actor is only passed as payload between the internal components until it got validated for trustworthiness, only then it gets executed - or ignored, if the request is not trusted. Additionally, the communication within this trusted zone is encrypted and even internal components are limited in their capabilities, as they are only allowed to communicate with components they need to communicate with for handling the incoming request. This makes the implicit trust zone very small - actually, the trust zone is only a linear path within the system's workflow.

[16] Although our proof-of-concept only implements a single Policy Enforcement Point, it is possible to have many server-side PEP components. There could be one per sub-network - e.g. each building in the smart city could have its own PEP - or there could be many distributed components which are only accessible through a load balancer.

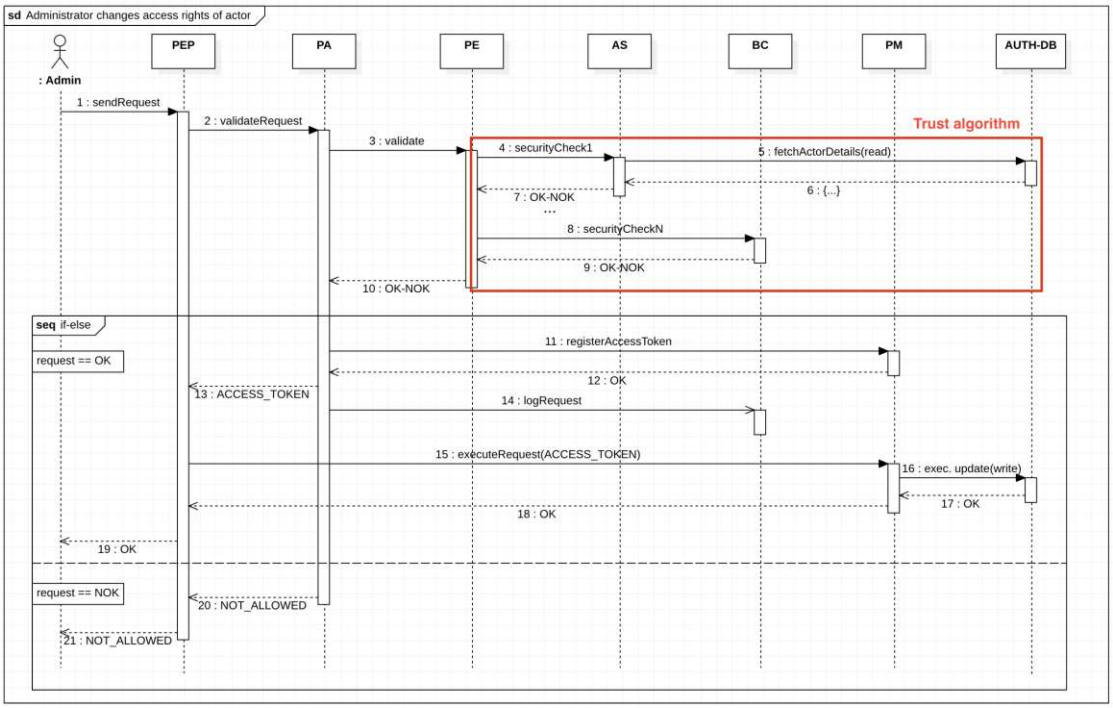


Figure 4.3: Sequence diagram of a use case: An administrator updates the access rights of an actor. The marked area (red) is where the Trust algorithm is executed. *Note:* for readability only one PE and only two validation components are shown, but it should be easy to imagine all remaining PEs and validation components in this diagram.

The PEP component forwards the incoming request and the actor's identity (see Section 4.4.1 for details) to the Policy Administrator. This component initiates the validation request by instructing all active Policy Engines to validate the trustworthiness of the actor and the validity of its request. As already mentioned in Section 4.2, here is where the majority voting starts. Depending on the type of request, the PA either waits for all PEs to finish their task (synchronous requests) or the PA immediately returns to the PEP stating that the validation started (asynchronous requests). The former type of requests is realised very straightforwardly: the PA iterates through all PEs, sends a validation request to each and blocks until the PE's trust algorithms terminated and returned a decision outcome before it continues with the next PE. If all PEs returned a decision, the PA picks the most occurrent decision and returns it in the ongoing validation request from the PEP. The latter type of requests is implemented with the help of a message broker: the PA sends the API requests to all PEs, which start the execution of their trust algorithm (see Section 4.4.2) in a background task and return immediately. Meanwhile, the Policy Administrator started listening for messages within a specific topic of the message broker. After the trust algorithm of a Policy Engine terminated, this Policy Engine instance sends the result of the validation to this specific topic. The PA then counts this validation result as one vote and continues listening for the other PEs to

finish. In total, the Policy Administrator only waits X seconds for the votes - in the PoC implementation, this time is set to five seconds - to not be slowed down by any Policy Engine which runs very slow or does not even respond at all. Also, the Policy Administrator does not wait for all Policy Engines to finish, but immediately finishes the validation process if a decision outcome got more than 51% of the votes. This redundancy is a security measure to overrule malicious or dead PEs. As long as more than the half of all PEs are running (or are benevolent), the system can correctly validate incoming requests. When consensus is reached, the Policy Administrator notifies the PEP about the validation result by calling the API method for executing incoming requests. The API call either tells the PEP that the incoming request shall be rejected, which will animate the PEP to return to the requesting actor with a message saying the the request got rejected, or it will contain an access token for the Persistence Manager responsible for executing the actual request. In both the synchronous and asynchronous type of requests, the access token had been registered on the responsible PM by the PA before and is only valid for a given amount of time - in our PoC the time-to-live of access tokens is set to 60 seconds. The PEP then has time to call the PM with this access token to actually execute the request, e.g. access or modify a resource, until the token expires. Depending on the type of request, the result of the PM is passed to the original requester via the PEP or the PEP ignores the result. In both cases, the workflow of this request finishes here.

4.4.1 The X-Requester Header

The identity of an actor is given in the HTTP request's "X-Requester" header. This information is in JSON format and has the following structure:

```
X-Requester:
{
    "agent": "...",
    "actor": "...",
    "ip_address": "...",
    "mac_address": "...",
    "os_id": "...",
    "os_version": "...",
    "auth_token": "..."
}
```

Actors have to add their actor ID and their authentication token (API key) to it when accessing one of the client components. The Analyser component guarantees this by forcing the user to provide their credentials (i.e. actor ID and API key) before opening the actual dashboard page. The Client component, on the other hand, has to be called with the X-Requester header already present in the HTTP header, but only with the credentials filled out, which will then simply be passes to the PEP component. Both

the Analyser and Client component will, however, populate the remaining properties of the requester header - they don't have to and even cannot be provided explicitly by the actors themselves.

The X-Requester object is then moved from the header to the payload in the remaining API calls within the trust zone, together with a description of the original incoming request of the actor. During the whole validation process, this payload is not modified.

4.4.2 Trust algorithm

The trust algorithm is the brain of the validation process. It defines how an incoming request is validated and decides which actor is trustworthy and which access request should be granted or rejected. The TA collects relevant data from different validation components to then calculate a decision based on that data. In the PoC implementation, the trust algorithm is static in the order of security checks, which means that each incoming request from any actor is always validated the same way. The trust algorithm's instructions for validating incoming requests are shown in Algorithm 4.1, 4.2 and 4.3, respectively. The actual logic for the different validation types are not present in the pseudo-code algorithms, but we provide a brief textual description of them in the next paragraphs.

Algorithm 4.1: Trust algorithm pseudo-code

Input: The X-Requester header *requester* and the incoming request of this actor *incomingRequest*

Output: A validation decision containing the decision (OK/NOK) and a list of validation failures

```
1 validationFailures = executeSecurityChecks(requester, incomingRequest);
2 decision = evaluateValidationFailures(result);
3 return decision;
```

Algorithm 4.2: executeSecurityChecks

Input: The X-Requester header *requester* and the incoming request of this actor *incomingRequest*

Output: A list of validation failures

```
1 validationFailures = [ ];
2 validationFailures.add(validateAuthenticity(requester, incomingRequest);
3 validationFailures.add(validateOperatingSystem(requester);
4 validationFailures.add(validateParameters(requester, incomingRequest);
5 validationFailures.add(validateBehaviour(requester);
6 return validationFailures;
```

Algorithm 4.3: evaluateValidationFailures

Input: The list of validation failures from the security checks *validationFailures*
Output: A decision for the list of validation failures

```
1 decisionOutcome = {};
2 if validationFailures has critical or behavioural failures then
3 |   decisionOutcome.decision = NOK;
4 else
5 |   decisionOutcome.decision = OK;
6 end
7 decisionOutcome.decidedAt = now();
8 return decisionOutcome;
```

validateAuthenticity(requester,incomingRequest): In this validation method, the Authentication Service is asked for details about the requesting actor. This identity data contains the actor's ID, the actor's IP and MAC address (in the case of stationary actors) and a list of access rights assigned to this actor. The API key of the actor is implicitly validated by providing it to the AS as a parameter when fetching the actor's details from the database. The AS fetches the data with the actor's ID and its API key. The persisted IP and MAC addresses are compared with the ones provided in the X-Requester header (for stationary actors) and the incoming request is compared with the access rights of the actor.

validateOperatingSystem(requester): This method checks the operating system for any vulnerabilities by requesting all known vulnerabilities of the requester's OS from the OSV component and then comparing the requester's OS version with the versions in which these vulnerabilities had been found.

validateParameters(requester,incomingRequest): Here the request itself is checked for syntactical or semantical issues, like format of IP address and value of the humidity reading (e.g. 0 <= value <= 100).

validateBehaviour(requester): This is the validation method with the most complex logic, as it does not only fetch and compare data from another component, but it also asserts some special conditions (see Algorithm 4.4). The history of the requester is retrieved from the BC-P-MON component, which calls a chaincode method with the help of the Fabric Gateway API. This call runs through the whole Hyperledger Fabric consensus process, i.e. endorsing, ordering and validating of the transaction. After consensus is reached, the last 50 requests of the actor is fetched from the ledger's world state and afterwards returned to the TA. The trust algorithm then analysis the history and enforces the behavioural policies (again, see Algorithm 4.4).

The above mentioned validation checks are all static, but the system could be extended to have a self-adapting trust algorithm which can change the security checks or priorities them differently depending on the requester or the incoming request (see Section 6.3 for how this could be done).

Algorithm 4.4: Identify behavioural issues

Input: The X-Requester header *requester*, the incoming request of this actor *incomingRequest*, the maximum number of allowed critical issues within the last X requests *maxAllowedCrit* and the amount of seconds after which a critical issue is not considered anymore *critValidity*

Output: A list of validation failures

```
1  validationFailures = [ ];
2  history = get history of actor with ID requester.actor;
3  criticalCount = count critical failures in history;
4  lastCriticalIssue = get last critical issue in history;
5  if criticalCount ≥ maxAllowedCrit then
6  |  if lastCriticalIssue is within last critValidity seconds then
7  |  |  validationFailures.add(POSSIBLY_MALICIOUS_ACTOR);
8  |  end
9  else if criticalCount > 0 then
10 |  if incomingRequest has issues then
11 |  |  if lastCriticalIssue is within last duration seconds then
12 |  |  |  validationFailures.add(ZERO_TOLERANCE);
13 |  |  end
14 |  else
15 |  |  no behavioural issues found
16 else
17 |  no behavioural issues found
18 return validationFailures;
```

The TA itself runs synchronously and is deterministic. All security checks are executed and the result of all of them are taken into consideration when building the validation decision. The caller is also informed about all security check failures and their severity. In our PoC implementation, for instance, we use the severity levels *LOW*, *MODERATE*, *HIGH* and *CRITICAL*. All critical failures result in the rejection of the validated request. The other failures are considered more as informative failures - except for two specific behavioural failures: (1) if the actor has more than a defined threshold of critical issues within the last X requests and the last critical issue is within the last Y seconds, the system creates a high severity behavioural failure saying that the actor could possibly be malicious, and (2) if the actor has some critical issues in its history, but still less than the given threshold, and the currently validated request has any kind of validation failure, the trust algorithm also creates a high severity behavioural failure saying that zero tolerance is applied on that actor. Both cases will set the decision to NOK, i.e. the

incoming request is rejected. Algorithm 4.4 contains pseudo-code of the relevant part of the trust algorithm showing the handling in both mentioned cases.

Some input parameters of the trust algorithm are hardcoded in the PoC, but making them editable on runtime is also possible by either adding a new API endpoint to the system for managing such rules, if actual users need to be able to adapt them or by adding a mechanism into the system's internals that changes the rules depending on some criteria. Such hardcoded rules are, for instance, the number of last history requests to consider for validation (= 50 in the PoC), the maximum number of allowed critical issues within the history (= 5 in the PoC) and the duration for how long the last critical issue is considered as a behavioural issue (= 2 seconds in the PoC), among others.

CHAPTER 5

Evaluation

In this chapter, we present the evaluation results of the non-functional properties of the proof-of-concept system. We do not discuss or rate actual computational logic like the trust algorithm or low-level system functionality, but rather consider the system as a whole during the evaluation. For testing performance and scalability, we defined some test cases and executed them against the PoC implementation on a test environment. To be able to compare the PoC with other security approaches, we implemented additional systems in different variants and executed the same test cases against them, on the same testbed.

The ZTA system design with all its core and additional components can get very complex. All the ZTA-specific components introduce an overhead compared to conventional perimeter-based systems, which increases the number of active components by a non-negligible amount. On top of that, implementing a whole self-contained permissioned blockchain adds even more complexity to the system. This not only makes the system complex in terms of orchestration but also the implementation effort increases. The complexity and implementation effort is discussed in more detail in this chapter.

In the following sections, we will discuss the evaluation process and results of the non-functional properties *performance*, *scalability*, *implementation effort*, and *complexity* in more detail.

5.1 Implementation effort

This section discusses the additional effort needed to implement a zero-trust system backed by a blockchain, compared to perimeter-based systems. It is obvious that the zero-trust architecture defined by the National Institute of Standards and Technology comes with an additional implementation effort to guarantee zero-trust. Assuming the functionality is the same in a zero-trust and a non-zero-trust (= perimeter-based) system

and that there is no additional implementation effort required for the actual business logic, the zero-trust architecture needs to implement ZTA-specific components to verify the trustworthiness of requesters, in addition to the base components which serve the system's functionality. This includes implementing the trust algorithm, the policy engines, the validation components like OSV and AS, and all the persistence managers in front of each resource. Increasing the number of requestable resources in a zero-trust system also increases the number of components linearly. For instance, if we want to extend the PoC implementation by a smart meter resource for persisting and reading power consumption of smart homes, we would need to implement a persistence manager for that resource as well and connect it to the system.

In our PoC system the trust algorithm is kept very basic. Each request goes through the same validation process, i.e. the same set of instructions to verify the trustworthiness of requesters, and there is no mechanism to dynamically self-align policy rules for requesters on specific events - those properties would ideally be implemented in a fully production-ready system. Even with these limitations the system still has a significant implementation overhead compared to conventional systems. In a perimeter-based system actors also have to be authenticated somehow, but usually not on each and every request and the number of validated properties are also less compared to a zero-trust setup.

Another aspect that increases the implementation effort is the integration of a permissioned blockchain. Not only do technical architects need to implement, connect, and startup components that host the blockchain, but if this blockchain is also used within the trust validation process as a ledger containing actors' requests history, there also needs to be a connection between the validation component and the blockchain nodes. In the PoC, we decided to implement a blockchain client in the ZTA part of the system which uses the HLF gateway API to interact with chaincode, by using a blockchain peer's identity. As mentioned in Section 4.1.2, the implemented system could also be extended to connect each blockchain peer with a dedicated ZTA-side component to have a more fail-safe validation process. In this case, the blockchain clients would also grow linearly in relation to the available blockchain peers, which in turn would also come with an extra implementation cost. In front of those blockchain clients, there would ideally also be a load balancer to split the system's load to multiple peers.

5.2 Performance

The performance of such a PoC system is best evaluated by having another system in place with which we can compare the PoC with. We implemented other systems with the exact same functionality as the PoC system, but with architectural differences. These systems and the PoC system were deployed on a test environment and some test cases were defined to be executed against them. In the following sections, we demonstrate the test environment, the implemented variants of the PoC, the test cases, and the evaluation results of those tests.

Table 5.1: Software and hardware specifications of the test environment

Operating system	CPU	RAM
Linux Ubuntu v22.04.2 (headless)	22 vcores	32 GB

Table 5.2: Five different variants were used for testing. The variants are described relative to the originally implemented PoC system, i.e. the "excludes" column in this table means that the listed components are missing compared to the PoC system.

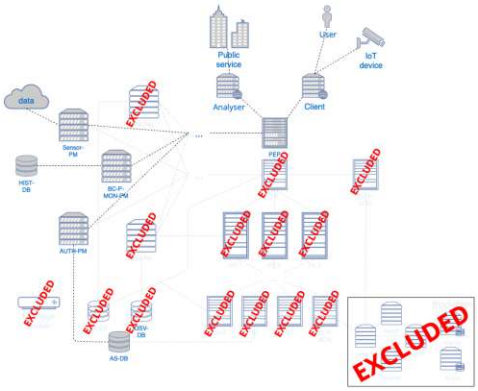
Variant	Name	Description	Excludes
1	Conventional	Basic security mechanism where only authenticity and access rights are checked	All BC components, PE, PA, all validation components
2	No BC	ZTA system but without a blockchain	All BC components
3	No BC (x4)	Same as Variant 2, but with 4x more PEs (12 in total)	All BC components
4	ZTA-BC	The original PoC	Nothing
5	ZTA-BC (x4)	The original PoC, but with 4x more PEs (12 in total)	Nothing

5.2.1 Testbed

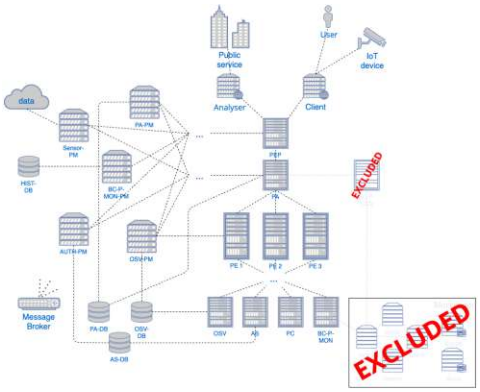
The proof-of-concept system consists of 30+ components. In a real-world scenario, the components would be split into multiple edge servers and devices, but for testing the performance and scalability of the system, we decided to start everything up on a single, yet powerful server. The hardware and software details about the server are presented in Table 5.1. The components were executed inside Docker containers which had the whole power of the host available.

5.2.2 Variants

To have a comparable evaluation result, we implemented different variants of the PoC system and we even implemented a basic system that does not have any zero-trust properties and no blockchain in place. The concrete variants are listed and described in Table 5.2 and the big picture of two of them is depicted in Figure 5.1. The variants are defined in such a way that the main technologies of the PoC can be evaluated for performance impact, e.g. there are variants with zero-trust properties, but no blockchain properties, or variants with less and variants with more policy engine instances. This way we are able to evaluate which part of the system slows down the execution time more than others. All variants produce the same output when fed with the same inputs. This means that the functional properties are the same for all variants and we can fully concentrate on evaluating the non-functional properties.



(a) The big picture of the "conventional" system. It excludes all blockchain components, the policy administrator, the policy engines and all validation components.



(b) The big picture of the "no-blockchain" system. It excludes all blockchain components and persists the request history of actors in a simple SQL database

Figure 5.1: The two additional systems implemented for the performance evaluation of the proof-of-concept system. The "no-blockchain" variant (b) has yet another variant with 12 policy engines.

5.2.3 Test cases

The actual test cases were implemented as Python scripts which were executed against the different variants. Every test script starts in the Client component, which means that each request goes through the whole process beginning from the Client component down to the database. We define five test cases which all have a different focus. There are tests checking the performance impact of synchronous requests in comparison to asynchronous requests, or measuring how the number of policy engines affects the overall performance, giving insides into the scalability of a system variant, etc. The test cases and the actual requests executed in them are described in the following list. It needs to be mentioned that each test case sets itself up, meaning that needed users or stationary actors are initialized within the test case's process. The initialization steps are, however, excluded from the execution time.

- **A user requests data that it does not have access to (TC1):** This is a very simple test case to demonstrate which system is faster in recognizing that a user does not have permission to the requested resource. It creates a user with insufficient access rights and this user then executes a request to a resource it does not have the access right for. The forbidden request is executed 5 times in a row to eliminate any outliers because of server hiccups or other unexpected things. The execution time of this test case is the time between the sending of the first request and the response to the last request.
- **A stationary actor sends 20 temperature readings (POST) and a user**

reads them afterward (TC2): In this test case a stationary actor is created and this actor then sends 20 temperature readings to the system in a row - i.e. without halting in between the requests. Immediately after, a user is created which reads all readings of this actor in a single read request. The execution time of this test case contains the 20 write requests only.

- **A stationary actor sends 1000 temperature readings (POST) and a user reads them afterward (TC3):** This test case is similar to the previous one (TC2), with the only difference that the stationary actor sends 1000 temperature readings to the system. This test case can be compared with the next test case (TC4) to evaluate the performance difference between synchronous and asynchronous requests. The execution time in this test case only contains the execution time of the 1000 requests, i.e. the synchronous read request is not taken into consideration.
- **A user sends 1000 requests (GET) containing some temperature data (TC4):** This test case creates a stationary actor and a user. The stationary actor sends five temperature readings to persist and the user reads the persisted data 1000 times (synchronous GET requests). Similar to the above test case, the execution time only contains 1000 requests.
- **Four stationary actors send data simultaneously (TC5):** This test case demonstrates how the system handles a high load from 4 simultaneously running threads. Each thread sends and reads data and all threads execute the same requests. Here the execution time contains the whole execution from start to end, i.e. it includes the creation of the required actors and it also waits until all threads are finished.

5.2.4 Performance evaluation

The above-listed test cases had been executed against each above-mentioned system variant. The system had been pre-filled with some data before the tests were executed. Each test case had been executed five times in succession and each system variant had been built and deployed from scratch before each test run - not before each test *case*, but each test *run*, i.e. before running the script for filling the system with initial data. This way we can guarantee that every test is executed under the same conditions on each variant. The test cases are executed five times in a row without resetting the system in between, because we also wanted to consider any increases in execution time with increasing data in the system, and as a running system is usually not empty, this also makes the test cases more realistic. For the evaluation of the performance, the average execution time of the 5 executions is taken and compared between variants. The evaluation process is drawn in Figure 5.2.

The results of this performance evaluation are depicted in Figure 5.3. There are some key takeaways that can be extracted from this figure. We will discuss some important key observations in the following paragraphs.

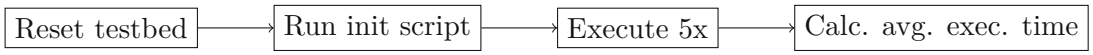


Figure 5.2: Process for executing a test case against a system variant. This process is done for each pair of test cases and variants, i.e. 25 times in total

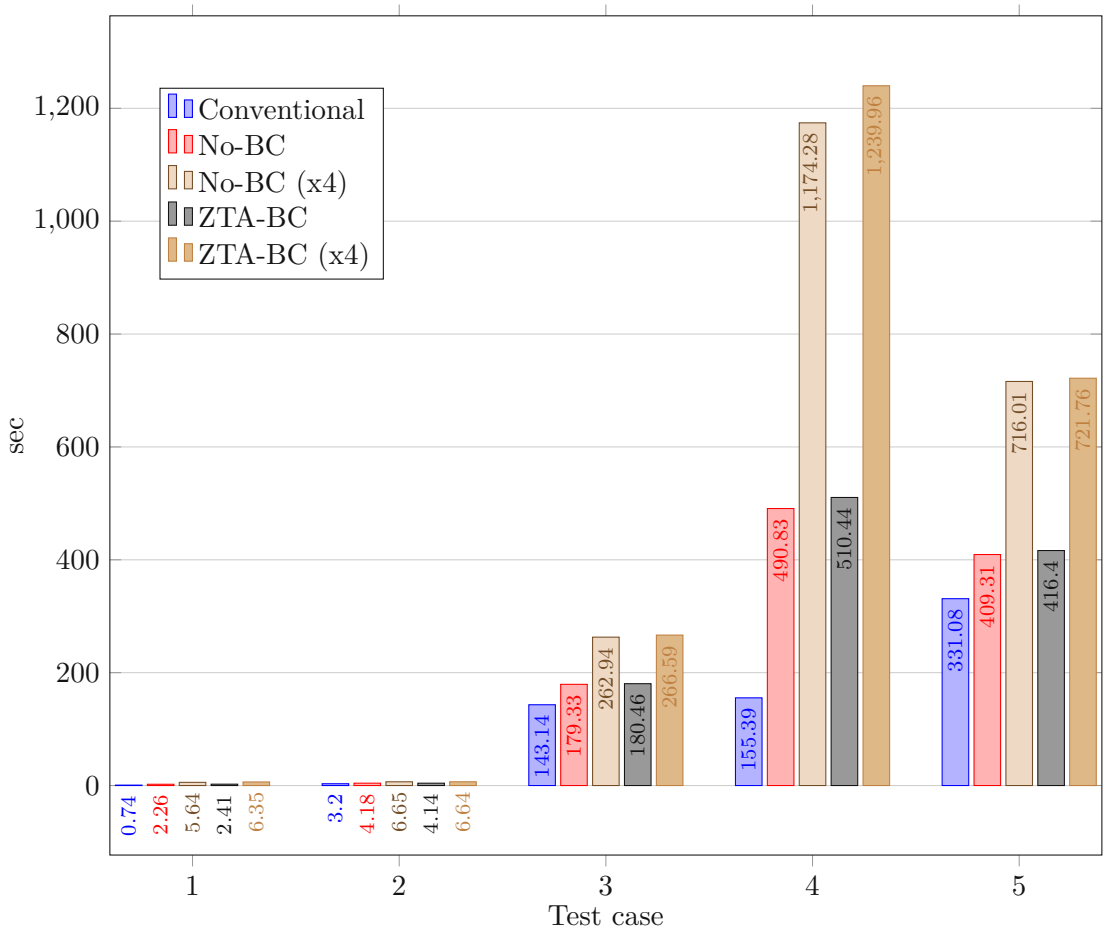


Figure 5.3: Execution times (in seconds) of the test cases against five different system setups: "Conventional" system (blue), ZTA without BC (red), ZTA without BC with 4x more PEs (lightbrown), ZTA with BC (grey), ZTA with BC with 4x more PEs (gold)

Read operations take much longer than write operations in our zero-trust system. Comparing test cases 3 and 4 between different variants, we see that in all zero-trust variants TC3 (1000 *read* requests) takes around **2x** the execution time compared to TC4 (1000 *write* requests), only in the "conventional" system, the time is almost the same. The reason for the huge time difference is obvious: read requests are validated synchronously and write requests are asynchronous. For all zero-trust variants, this means that each read request waits for the PEs to reach consensus, whereas write requests only trigger the background validation task and immediately after return to the requester. The conventional system does not encounter this difference, because there is no validation process involved, i.e. read requests do not have to wait for consensus between policy engines, because there are none.

More policy engines means longer execution time - especially for read requests. As the validation process includes waiting for consensus between the policy engines, the execution time of a request validation increases the more policy engines are added to the system. The difference is, however, surprisingly not very big in all cases. For write requests, where the validation process is executed asynchronously, the execution time with 4 times more policy engines increases only around **2,4x** (TC1, ZTA-BC), **1,5x** (TC2, ZTA-BC), **1,5x** (TC3, ZTA-BC) and **1,7x** (TC5, ZTA-BC), respectively. This effect is more significant for read requests (synchronous validation): the increase in execution time in TC3 on the "ZTA-BC (x4)" in relation to TC4 on the "ZTA-BC (x4)" system is higher (approx. **+4.7x**) than in TC3 on the "ZTA-BC" in relation to TC4 on the "ZTA-BC" (approx. **+2.8x**)

The presence of a blockchain does not affect the performance significantly in our zero-trust system. When reading data from the blockchain, the query is executed *without* consensus between the blockchain peers. This is the default behaviour in the Hyperledger Fabric framework [Hyp22a]. As the logging of actors' requests in the blockchain is executed asynchronously and fetching actors' request history from the blockchain does not go through a consensus process, the presence of a blockchain does not decrease the zero-trust system's performance significantly. For instance, the difference in execution time between the No-BC and ZTA-BC systems is around **+1.05x** (approx. +5%) in all test cases.

The validation process takes the most time in our zero-trust system. This is verified empirically by checking the increase in execution time when extending the number of policy engines by multiple. For instance, increasing the number of policy engines from 3 to 12 (4x) almost **triples** the execution time. This means that the increase in execution time is almost linear to the increase in policy engines. Recall that the consensus algorithm in the request validation process asks all PEs for validation and only notifies the PEP about the validation result when more than half of the policy engines returned the same decision (see Section 4.2). This has the consequence that the more PEs are added to the system, the more decisions of PEs are needed to reach consensus,

the longer it takes for the policy administrator to accept a decision and the longer the overall execution time of the request takes.

5.3 Scalability

We already showed above that the system can theoretically be extended by as many policy engines as desired and can still serve the system's functionalities. We also discussed that in a real-world scenario, there would be more policy enforcement points, e.g. one per smart home (see the footnote in Section 4.4), which means that the system is technically also able to handle multiple PEPs. Although the system is extendable by those components, the actual PoC implementation slows down noticeably when doing so (see Section 5.2.4). However, this does not necessarily mean that the *design* of the system is not scalable at all. Many factors could cause the proof-of-concept to slow down when confronted with a huge load of requests or lots of policy engines. For instance, as the PoC spawns new threads for each policy engine when the incoming request is validated asynchronously (i.e. write requests) and the test environment is limited to 22 CPU cores, the system will lose performance on high-load because of the number of context switches needed between the threads.

It is also possible to add more resources to the system and the validation process could also be extended by more validation properties by adapting the trust algorithm. This, however, requires some implementation effort (see Section 5.1) and is not feasible during runtime with the system design presented in this thesis. Adding new resources does not affect the overall execution time of requests as the resource is only touched when an actor actually requests it, but adding more policies to the trust algorithm does, as this affects *all* incoming requests and slows down the overall validation process.

Scaling up the actors is done by installing the client-side part of the policy enforcement point on the actor's device. The client software could be provided as a download to install it on that device or it could be pre-installed on proprietary devices which are used by the actors. Another option is to install it on an edge server which acts as the gateway to the system. In this case, increasing the number of actors does not require any further tasks except for registering new users or actors. The scalability in this context depends on the actual use cases. In the smart city example used in this thesis, all actors and users are known to the system, which means that they will be provided with devices with pre-installed client software.

In the end, the PoC should only show that the designed system concept is indeed implementable and that it is applicable in the context of smart cities. It should not be used in production as-is, because it lacks non-functional properties which typical real-world systems should take care of (see Section 6.3).

5.4 Complexity

The designed PoC system contains lots of components. There are 8 components that are sole needed for enabling zero-trust - PA, PE 1-3, OSV, AS, PC and BC-P-MON - and 6 with are needed for enabling and hosting the permissioned blockchain. Additionally, there is one PM per available resource. Incoming requests are passed through a lot of components until they are actually executed, more precisely an incoming request touches the PEP, PA, each PE, all validation components a couple of times (once per PE), all blockchain peers and the blockchain orderer component, the PM and then finally the database, i.e. the actual resource - if the requester is trustworthy. When counting the components in the big picture (see 4.1), we get 30 components, i.e. each successfully validated and trusted request touches around 60% of the system's components (19 components). This fact shows that the PoC design which just implements CRUD operations, is very complex in relation to its functional capabilities. Not only does it require implementing a relatively high number of components, but the orchestration of them and establishment of secure and robust communication between them can also get very complex - especially when there is a mix of synchronous (REST APIs) and asynchronous communication (Message Broker - Pub/Sub).

Setting up the blockchain is a relatively complex task on its own. Although the Hyperledger Fabric framework offers extensive documentation of each and every aspect of their permissioned blockchain components, it still needs a lot of steps to go through the needed documentation pages. Starting from setting up the organization, the certificate authorities, the peers, and the orderer components, to setting up encrypted communication between them (TLS), to implementing and setting up a chaincode, to then implement clients to connect to the peers.

There are other aspects which further increase the complexity: the consensus algorithm for validation and the validation algorithm (trust algorithm) itself. Although the employed consensus algorithm for the validation process itself is kept fairly simple, it still adds extra computational complexity to the system. Additionally, the current implementation of the trust algorithm is very linear and static. If it would be extended to be more flexible and dynamic in enforcing policies, it would also increase the complexity significantly and the comprehensibility of validation decisions would also suffer.

5.5 Advantages and Challenges

The main goal of introducing zero-trust aspects to an already running system or implementing a new system with zero-trust properties in contrast to a perimeter-based system, is usually to increase the security of that system. With a zero-trust architecture, the system is more flexible in trusting actors and policies can be adapted more dynamically. Especially in the context of the Internet of Things where a huge number of actors are involved in a network, the security of all of them cannot be guaranteed at any time. Even if all of those actors are enterprise-owned, being able to maintain the security of all

the heterogeneous devices is a very hard task. In such a case the zero-trust architecture comes in handy, as it allows for defining dynamic policies and enforcing them on *each request*. So with the correct policies in place, it is possible for a zero-trust system to identify if an actor had potentially been compromised and to reject all its requests. This can be a very useful property of a network depending on the use cases of the system.

Additionally, with the integrated permissioned blockchain another level of trustworthiness is added. The actors' request history is very valuable in such a zero-trust system as it allows for recognizing behavioural patterns of actors and based on that rejecting requests of them, hence enabling some sort of early warning system. If this history data is manipulated, the system would not be able to decide efficiently and correctly whether to reject a request or not, as an attacker could always whitewash the history of the actor it hijacked to not get rejected by the system. Because of this, it is a good idea to save history in a distributed ledger. Having the history distributed to many peers eliminates the risk of being tricked by a malicious blockchain peer and thus taking a manipulated history for validation.

All the security advantages of a blockchain-based zero-trust system mentioned in this thesis come with the cost of having a more complex system and losing performance - in terms of execution time. Not only is the implementation effort higher but also the maintenance of a running zero-trust system is higher compared to a perimeter-based system with the same functional requirements. However, as this PoC is within the context of smart cities and such networks anyway consist of a huge number of nodes that need to be implemented, connected to the network and maintained, the additional effort for implementing zero-trust in the back-end of that system should be tolerable.

CHAPTER 6

Conclusion

In an Internet of Things network like a smart city, there is usually a huge number of interconnected and mostly heterogeneous devices. Even if all connected devices are enterprise-owned and no third party is allowed to connect to them or talk with them, the risk of a device getting hijacked is relatively high, as the attack surface is large due to the huge number of connected devices. The heterogeneity of those devices makes it even more likely that one of them got compromised, as attackers are given more options for different attack vectors. On top of that, IoT devices usually have low computational power and are in most cases not able to defend against many attacks.

In this thesis, we introduced a system design for such a huge smart city network that implements the zero-trust architecture as defined by NIST. A system implementing the ZTA does not trust any actors in its network by default and validates their trustworthiness of them on each request. With such a system in place, even if an IoT device gets compromised the system is still able to recognize it and take measures against it, e.g. reject requests coming from that device. The better the policies are defined the more efficient the system is able to do that. In our designed system, the policies are heavily focused on actors' request history to decide whether or not to trust actors. The request history of the connected devices is very valuable in our designed system, as it allows for more accurate decision-making when it comes to trusting a device or not. For that reason, the history needs to be stored in a way such that it is impossible or very hard to tamper with it. We connected a permissioned blockchain to the zero-trust system to persist the history of actors, which is hosted by multiple blockchain nodes - in the PoC there are three peers hosting the blockchain.

We focused on the design and implementation of a system with the zero-trust architecture which is backed by a blockchain. This thesis demonstrated that such a system is indeed implementable and also applicable in the context of the Internet of Things. We also showed some non-functional properties of it by theoretically discussing them and comparing them with conventional perimeter-based systems, and we also implemented

test cases and executed them against a running system on a test environment to show other non-functional properties of the system. We came to the conclusion that such a system is very dynamic in terms of policy enforcement and can react on a per-request basis. With the trust algorithm being the central component in the validation process, system designers can find the policies in a single place, making policy management a lot easier. However, the initial implementation effort is higher compared to perimeter-based systems, and the overall system can get very complex as it has a lot more components to maintain. In fact, the system consists of two sub-networks with many components in them: the zero-trust network and the blockchain network. Also, from our performance and scalability experiment, we saw that the system is scalable on some levels but that the performance suffers when the system is scaled up too high. Nevertheless, the designed system is still able to maintain its security posture even on high load, i.e. it is still able to validate each incoming request separately.

All in all, by combining zero-trust with a permissioned blockchain, the system adds many useful layers of security for smart cities. The heterogeneity and low computational power of IoT devices no longer pose a major security threat to the system as compromising a device and using its identity is not enough for an attacker to gain access to the system anymore. An attacker would need to spoof many more properties to gain access without being noticed as an attacker by the system.

6.1 Research Questions Revisited

RQ1: How can the zero-trust paradigm be applied to secure smart city systems with heterogeneous IoT devices using blockchain technology?

We showed that a zero-trust system is applicable to smart cities with the zero-trust architecture definition provided by the *National Institute of Standards and Technology* [SD18] and the permissioned blockchain technology provided by the *Hyperledger Fabric* framework [Hyp22b]. We designed the system in Chapter 4 by complying to the zero-trust tenets listed in Section 2.2.1 (defined by NIST [SD18, p. 6ff]), and described each components in detail. The blockchain is used as an immutable database for the history of actors' requests. This history is a crucial component in the zero-trust system which help deciding if an actor is trustworthy or not, by analyzing it and identifying (potentially) malicious activity.

RQ2: Which implementation advantages and challenges are implied when deciding to apply the zero-trust architecture in general?

The main advantage of using the designed system compared to perimeter-based security models is the possibility of dynamically enforcing policies. An actor is checked for trustworthiness on **each request** and the decision is made by checking some properties of that actor. The policy enforcement is able to reject requests as soon as it recognizes the malicious behaviour of the requester. The better the policies are defined and

implemented, the more efficiently the system is able to block attackers or broken actors. The algorithm logic for validating the trustworthiness of actors is the most important part when implementing such a system. This algorithm - also called "trust algorithm" [SD18, p. 17] - can be as static as having a set of rules which are always verified the same for each request, or it can be as dynamic as deciding which policies to enforce for which actors under which conditions - e.g. when a user requests a sensitive resource very often during a time where the user usually does not request anything, the system could apply stricter policies than having a user which requests usual resource during a work day.

The more dynamic the trust algorithm is and the more properties it includes in its validation process, the more complex the system gets. The complexity can be very challenging when implementing such a system with very complex trust algorithms. Compared to conventional perimeter-bases systems, a blockchain-based zero-trust system as designed in Chapter 4, has a lot more components to implement and maintain. The system consists of two sub-networks which are already complex on their own, namely the zero-trust network with all the components for enforcing policies and validating trustworthiness of actors and requests, and the permissioned blockchain for saving the request history immutably.

RQ3: How scalable/performant is the system compared to today's conventional systems?

Another challenge when deciding to implement such a system is performance. The performance suffers from intense policy enforcement. The actual validation is done by dedicated components (policy engines) with a consensus algorithm in place which makes sure that the request validation is done correctly even if some policy engines are hijacked or manipulated. As the system validates each and every request and all policy engines are involved in this process, the overall execution time of request - i.e. the time from sending the request to actually executing it - is longer than requests executed in a perimeter-based security system. This can, however, be mitigated with a more efficient consensus algorithm in the validation process.

Adding more policy engines to the validation process is as easy as starting up a new policy engine instance and calling an API method for registering it in the system. Scaling the number of policy engines up will, however, slow down the system remarkably, as the number of nodes which need to validate an incoming request - and thus the wait time for reaching consensus - increases. Again, a more efficient consensus algorithm can help here. On the other hand, adding more blockchain peers for hosting more copies of the request history is not that easy, but it does not affect the validation time that much (see the evaluation results in Section 5.2.4).

6.2 Limitations

This thesis concentrated on *designing* and *implementing* a proof-of-concept system with zero-trust properties and an integrated blockchain component. Therefore, the evaluation

is done on non-functional properties which refer to the design and implementation of said system, i.e. implementation effort, complexity, scalability and performance. The most obvious non-functional property of such systems, namely the *security* property, is not evaluated as it is assumed that the security of a zero-trust system is by design more effective than the security in a perimeter-based system.

Additionally, the implemented PoC is not a solution ready to be used in production as it is. We concentrated on demonstrating the feasibility of such a system and intentionally left out the implementation of some non-functional properties which would be needed to make the system run in a real-world scenario, e.g. fault-tolerance, high-availability, etc.

6.3 Future work

The PoC implementation is not complete by any means. As already mentioned a couple of times throughout this thesis, the system lacks some crucial properties for making it production-ready. There is no concept for making the *whole* system fault-tolerant - some parts are fault-tolerant, like the validation process (see Section 4.2) and the blockchain network (immutability). High availability is also not dealt with during the design of the system. There is a lot of potential for increasing the mentioned and similar non-functional properties.

Additionally, performance can be increased with a more efficient consensus algorithm during the validation process, which is able to work with a multiple of policy engines and a high request load, e.g. the consensus algorithm could always select a random subset of policy engines for validation instead of all. Another possible improvement could be to add more policy administrators to the system which are preceded by a load balancer that makes sure that the system's workload is distributed to all policy engines and consequently to all policy engines. Caching is also not utilized in the whole system. Although it could improve the system's performance a lot, developers should be very careful to not cache anything which affects validation results. This could hinder or eliminate the flexibility of policy validation.

Last but not least, the policy enforcement process can be improved by adding a machine learning component. This component could be trained to recognize malicious behaviour patterns and to predict actors' behaviour to be able to reject attackers or compromised actors more effectively.

List of Figures

2.1 ZTA core components (within the Control Plane and Data Plane) and additional components used as data sources to enforce the policies (outside the Control/Data plan area) [RBMC20, p. 9] 13

4.1 Network diagram of the PoC 22
4.2 Validation process starting from the PA. PE2 is compromised and acts maliciously (it returns "not OK" for a valid request). The PA still returns OK to the caller, as three out of four PEs answered with OK. For readability, only the AS is shown from the validation components and the rest is represented by the "..." lifeline 30
4.3 Sequence diagram of a use case: An administrator updates the access rights of an actor. The marked area (red) is where the Trust algorithm is executed. *Note:* for readability only one PE and only two validation components are shown, but it should be easy to imagine all remaining PEs and validation components in this diagram. 34

5.1 The two additional systems implemented for the performance evaluation of the proof-of-concept system. The "no-blockchain" variant (b) has yet another variant with 12 policy engines. 44
5.2 Process for executing a test case against a system variant. This process is done for each pair of test cases and variants, i.e. 25 times in total 46
5.3 Execution times (in seconds) of the test cases against five different system setups: "Conventional" system (blue), ZTA without BC (red), ZTA without BC with 4x more PEs (lightbrown), ZTA with BC (grey), ZTA with BC with 4x more PEs (gold) 46

List of Tables

5.1 Software and hardware specifications of the test environment 43
5.2 Five different variants were used for testing. The variants are described relative to the originally implemented PoC system, i.e. the "excludes" column in this table means that the listed components are missing compared to the PoC system. 43

List of Algorithms

4.1 Trust algorithm pseudo-code 36
4.2 executeSecurityChecks 36
4.3 evaluateValidationFailures 37
4.4 Identify behavioural issues 38

Acronyms

AS Authentication Service. 24, 25, 30, 31, 37, 42, 49, 55

BC Blockchain. 31

CA Certificate Authority. 29, 31

HLF Hyperledger Fabric. 15, 26–28, 31, 32, 42

IoT Internet of Things. 1–4, 7, 8, 18–20, 23, 32, 33, 51, 52

MSP Membership Service Provider. 28

NIST National Institute of Standards and Technology. 2, 3, 8–12, 17–19, 21, 22, 51, 52

OSV Operating System Vulnerabilities. 24, 30, 37, 42, 49

PA Policy Administrator. 11, 12, 23–25, 29, 34, 35, 49

PBFT Practical Byzantine Fault Tolerance. 29

PC Parameter Checker. 24, 49

PE Policy Engine. 11–13, 23–25, 29, 34, 35, 47, 49, 55

PEP Policy Enforcement Point. 11, 12, 22, 23, 25, 29–35, 47–49

PM Persistence Manager. 23–25, 30, 31, 35, 49

PoC Proof-of-Concept. 2–5, 21–33, 35, 36, 38, 39, 41–43, 48–51, 54, 55, 57

TA Trust Algorithm. 23–25, 36–38

TLS Transport Layer Security. 28

ZT Zero-Trust. 2–4, 9, 10, 12, 13, 25, 26, 28

ZTA Zero Trust Architecture. 2–4, 7–13, 18, 21, 25, 29, 31, 32, 41, 42, 51, 55

Bibliography

[ABB+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, page 13–16, New York, NY, USA, 2012. Association for Computing Machinery.

[CL+99] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[CQZ+20] Baozhan Chen, Siyuan Qiao, Jie Zhao, Dongqing Liu, Xiaobing Shi, Minzhao Lyu, Haotian Chen, Huimin Lu, and Yunkai Zhai. A security awareness and protection system for 5g smart healthcare based on zero-trust architecture. *IEEE Internet of Things Journal*, 8(13):10248–10263, 2020.

[DKJG17] Ali Dorri, Salil S Kanhere, Raja Jurdak, and Praveen Gauravaram. Blockchain for iot security and privacy: The case study of a smart home. In *2017 IEEE international conference on pervasive computing and communications workshops (PerCom workshops)*, pages 618–623. IEEE, 2017.

[DLSP16] Casimer DeCusatis, Piradon Liengtiraphan, Anthony Sager, and Mark Pinelli. Implementing zero trust cloud networks with transport access control and first packet authentication. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 5–10. IEEE, 2016.

[GTCMF+22] P García-Teodoro, J Camacho, G Maciá-Fernández, JA Gómez-Hernández, and VJ López-Marín. A novel zero-trust network access control scheme based on the security profile of devices and users. *Computer Networks*, 212:109068, 2022.

[Hyp22a] Hyperledger. Frequently asked questions - application-side programming model. https://hyperledger-fabric.readthedocs.io/en/release-2.4/Fabric-FAQ.html#application-side-programming-model, Last checked: 13.03.2023, 2020-2022.

[Hyp22b] Hyperledger. Hyperledger fabric documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.4/, Last checked: 14.02.2023, 2020-2022.

[Hyp22c] Hyperledger. Write your first chaincode. https://hyperledger-fabric.readthedocs.io/en/release-2.4/chaincode4ade.html, Last checked: 14.02.2023, 2020-2022.

[KB+10] John Kindervag, S Balaouras, et al. No more chewy centers: Introducing the zero trust model of information security. *Forrester Research*, 3, 2010.

[KN12] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19(1), 2012.

[KÖ19] Sinan Küfeoğlu and Mahmut Özkuran. Bitcoin mining: A global review of energy and power demand. *Energy Research & Social Science*, 58:101273, 2019.

[LLT+21] Dapeng Lan, Yu Liu, Amir Taherkordi, Frank Eliassen, Stéphane Delbruel, and Liu Lei. A federated fog-cloud framework for data processing and orchestration: A case study in smart cities. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, page 729–736, New York, NY, USA, 2021. Association for Computing Machinery.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, pages 382–401, July 1982.

[Lue20] Knud Lasse Lueth. State of the iot 2020: 12 billion iot connections, surpassing non-iot for the first time, Nov 2020.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, page 21260, 2008.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[RBMC20] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero trust architecture. Technical report, National Institute of Standards and Technology, 2020.

[SD18] Mayra Samaniego and Ralph Deters. Zero-trust hierarchical management in iot. In *2018 IEEE international congress on Internet of Things (ICIOT)*, pages 88–95. IEEE, 2018.

[SHL+20] Maliha Sultana, Afrida Hossain, Fabiha Laila, Kazi Abu Taher, and Muhammad Nazrul Islam. Towards developing a secure medical image sharing system based on zero trust principles and blockchain technology. *BMC Medical Informatics and Decision Making*, 20(1):1–10, 2020.

[SMG19] Sarwar Sayeed and Hector Marco-Gisbert. Assessing blockchain consensus and security mechanisms against the 51% attack. *Applied sciences*, 9(9):1788, 2019.

[Sza97] Nick Szabo. Formalizing and securing relationships on public networks. *First monday*, 1997.

[XLJ+21] Zhang Xiaojian, Chen Liandong, Fan Jie, Wang Xiangqun, and Wang Qi. Power iot security protection architecture based on zero trust framework. In *2021 IEEE 5th International Conference on Cryptography, Security and Privacy (CSP)*, pages 166–170. IEEE, 2021.

[YXC+15] ChuanTao Yin, Zhang Xiong, Hui Chen, JingYuan Wang, Daven Cooper, and Bertrand David. A literature survey on smart cities. *Sci. China Inf. Sci.*, 58(10):1–18, 2015.