Karlsruhe Institute of Technology

Software Design and Quality

Modelling and Analysing Zero-Trust-Architectures Regarding Performance and Security

Master's Thesis of

Evgeni Cholakov

at the Department of Informatics
KASTEL – Institute of Information Security and Dependability

Reviewer:	Prof. Dr. Ralf Reussner
Second reviewer:	Prof. Dr.-Ing. Anne Koziolek
Advisor:	M.Sc. Larissa Schmid
Second advisor:	M.Sc. Nicolas Boltz
Third advisor:	M.Sc. Bahareh Taghavi

17. July 2023 – 17. January 2024

Abstract

Integrating a Zero Trust Architecture (ZTA) into a system is a step towards establishing a good defence against external and internal threats. However, there are different approaches to integrating a ZTA which vary in the used components, their assembly and allocation. The earlier in the development process those approaches are evaluated and the right one is selected the more costs and effort can be reduced.

In this thesis, we analyse the most prominent standards and specifications for integrating a ZTA and derive a general model by extracting core ZTA tasks and logical components. We model these using the Palladio Component Model to enable assessing ZTAs at design time. In our components, we encapsulate different variations of the functionality of these components to make them reusable and adaptable to the varying ZTA approaches. We make our components extensible to allow developers to adjust them to their design requirements. We combine performance and security annotations to create a single model which supports both performance and security analysis. By doing this we also assess the possibility of combining performance and security analyses.

We demonstrate the ability to analyze different ZTAs in Palladio and assess the applicability of our model by modelling different ZTAs from the literature using the created in this thesis elements. We evaluate whether we can detect performance impact induced by different ZTA configurations by comparing the performance simulation of a Palladio example system with and without ZTA. For the security analyses, we evaluate unauthorized and unauthenticated access to resources as well as violations of the least privilege principle by using Data Flow Analysis.

From the evaluation, we conclude that we can model ZTA using Palladio without needing to extend the Palladio Component Model with new features. Our components can be used off-the-shelf with little adjustments to legacy systems and even less when designing a system with security in mind from the beginning. During the performance and security analyses, we identified compatibility issues when combining performance and security annotations. However, the model still allowed the detection of performance impact induced by new components as well as the detection of security violations such as unauthorized and unauthenticated access and violations of the least privilege principle.

Contents

1 Introduction

Zero Trust Architectures (ZTA) provide a shift from traditional perimeter-based defences to a more robust, flexible and fine-grained security. In the following chapter, we present why such security is needed nowadays and motivate the need to be able to evaluate such architectures at design time.

1.1 Motivation

The advancement of the Internet of Things (IoT), 5G networks and cloud technologies have allowed systems as well as critical infrastructures to shift to more distributed and off-premise practices in their way of functioning. For example, such technologies allow for evolution in the energy management systems and the introduction of Virtual Power Plants [5, 44]. Typical for such power plants is the interaction of multiple distributed power-generating components, energy-storing units and management systems. Another example of critical infrastructures which make use of distribution and remote communications are medical systems namely remote surgery [35, 50]. The IoT market is growing rapidly and IoT connections amount to almost 18 billion in 2023 with predicted growth to 38 billion until 2028 according to market insights [23]. Almost 77% of the examined global companies in [23] intend to implement IoT technologies in the following years.

In the business sector, the home office has been widely accepted recently and many companies have integrated it into the work life of their employees [22]. This means that business resources are being accessed off-site and through different devices. This is why secure software and system design with fine-grained access control should be more deeply researched.

These trends, however, are followed by an increased attack surface for the systems and spikes in cyber attacks.[23] These attacks cannot be underestimated since they can lead to cases where a city power grid is compromised [8] or a country's nuclear program is damaged [3]. This is why a secure software and system design should be more deeply researched.

A more traditional approach to cope with security problems is the perimeter-based architecture. In this type of architecture, the system is separated into internal and external domains with a border between them. The border is created using different access interfaces, firewalls, intrusion detection systems and intrusion prevention systems. Even strategies such as deploying resources inside the internal domain of the system and forbidding access to them from the external domain are also utilized. This type of approach to border definition suffers from one main drawback. Once an adversary manages to infiltrate the internal domain of the system, they gain significant trust and can easily access critical system components and resources without being checked. According to reports from 2019, the US health industry has suffered over 3,8 million cases of patient record breaches due to inside attacks, which was a 26% rise from the previous year [52]. The infamous WannaCry ransomware is an example of an attack

which had a severe impact on the National Healthcare Service (NHS). The worm was able to spread due to outdated OS versions.

ZTAs provide a new approach to system design which can cope with the issue of preventing internal attacks. ZTAs follow the principle that no subject, no matter whether it is internal or external, should be trusted before correctly authenticating itself. In addition, trust in the subject making the request should be constantly evaluated by analysing multiple information sources such as the subject's location or behaviour. Each access request to a protected resource should be evaluated for required permissions and permissions are granted only following the Least Privilege Principle (LPP), where only the required permissions for solving the task are assigned and no more. There are already suggestions about integrating ZTA into critical infrastructures similar to the ones we mentioned in the beginning. ZTA can applied to enhance security in a Virtual Power Plant [1] or in smart healthcare, [9]. Manufacturers from Industry 4.0 can also benefit from integrating a ZTA [39]. In the case of the WannaCry attack from the previous paragraph, if infected devices have been micro-segmented and requests evaluated according to Zero Trust principles, the impact of the attack could have been greatly contained [52].

However, designing or migrating to a ZTA is not a straightforward task [49]. The architecture introduces new components, such as Trust Algorithm (TA), Policy Decision Point (PDP), etc. and additional authentication and authorization checks. This can produce additional performance and cost issues. Additionally, there are multiple ways to design a ZTA [48]. Designing and testing each design alternative, in order to choose the right one, costs time and resources. Hence it is inefficient. By modelling the ZTA we can perform early performance analysis and try to identify access control violations, which can hinder the system's confidentiality. Analysis at design time can also detect errors in a system's architecture. An error may be induced in a system long before its implementation and deployment. Design time errors prove to be harder to diagnose, require more time to be corrected and cost much more resources when handled at a later stage of the development process[6, 7]. Currently, there is no existing model of a ZTA suitable for performance and security analyses.

Modelling of such architecture can be achieved using Architectural Design Languages. Palladio offers the opportunity to create a model of a system by defining its components, with their behaviour and relations as well as deployment environments and usage models. Then analysis can be performed on the created model to test the quality requirements such as performance or security[43].

1.2 Objective

Currently, ZTAs have not been modelled using Palladio. Additionally, performance and security analyses have been performed separately in Palladio until now. If we can combine both of the performance and security annotations in a single model we would enable the simultaneous analysis of both quality requirements.

The objective of the following master's thesis is to model reusable ZTA components and interfaces using elements of the Palladio Component Model. We want to create components which can be used by developers in their own Palladio projects to include ZTA concepts with minimal adjustments. We want to develop an approach to detect security violations in a system with ZTA using Data Flow Analysis. We want to include the security annotations alongside

performance annotations to enable both of the analyses at design time with a single model. We then evaluate if the created components can be reused off-the-shelf to model varying suggestions of ZTA from the literature. We also want to test if we can detect the performance impact induced by the ZTA components as well as security violations in an example model.

1.3 Structure of the Thesis

Chapter 2 presents the needed concepts to understand the work done in this thesis. We describe in more detail what ZTA is and provide an integration guide and maturity model which we will later use in our evaluation. We also provide a description of Palladio.

In Chapter 3 we present approaches for modelling systems for performance and security analyses from the literature. We also discuss suggested models of ZTA from the field.

In Chapter 4 we create a model of the JPlag [24] system which we use as a running example throughout the thesis.

In Chapter 5 we analyse standards for ZTA and extract from them the main tasks as well as logical components responsible for these tasks and their relations. We convert this knowledge into a ZTA meta-model.

In Chapter 6 we describe the the whole process of modelling ZTA concepts using Palladio and the resulting data types, interfaces and components. We then showcase how to apply the created elements to the running example in order to integrate into it a ZTA.

In Chapter 7 we present the development of the security model of a ZTA for Data Flow Analysis. We describe the label definitions and nodes' behaviour. We define the possible violations our model can detect and then present a reusable Java implementation for analysing ZTA for these violations by specifying constraints. We then apply the security annotations to our running example.

In Chapter 8 we perform the evaluation. We describe the example Palladio model which we use in our evaluation and describe how we integrate a ZTA into it according to the guide from Chapter 2. To provide a closer to real-scenario context we evaluate the created model according to the ZTA maturity model presented in Chapter 2. We propose scenarios for evaluating the security model. We assess how thoroughly we were able to recreate ZTA concepts using Palladio and test the applicability of our elements by modelling systems from the literature. We then execute performance and security analyses on the example Palladio model with ZTA.

In Chapter 9 we conclude the thesis and discuss topics for future work.

2 Foundations

In the following chapter are presented key topics for understanding the rest of the thesis. We first present what is the meaning of a ZTA and what principles it follows. Then we present Palladio which will be used in this project to create the model and perform the analyses. Finally, we introduce a guide for integrating a ZTA in a system and present also a maturity model which we use in the projector to evaluate the maturity level of the ZTA model we use in the evaluation.

2.1 Zero Trust Architecture

To explain what a ZTA means we present two standards from the literature which describe what is the definition of Zero Trust and what practices and concepts are included in a ZTA. We observe the NIST standard document as well as Microsoft's white paper on the topic of ZTA.

2.1.1 NIST Zero Trust Architecture

Zero Trust [48] can be described as a cybersecurity paradigm which is focused on access control of non-public resources (data, services, devices, Internet of Things actuators) per request based on trust and enforces the LPP. The LPP states that a subject should not be given more than the required authorization for accessing a resource. Trust in the context of Zero Trust can be given through a complex evaluation of the subject's (user, device or combination of both) identity, the resource's required permissions and the context of the environment as well as the request. The two main goals of Zero Trust are to restrict access to unauthorized subjects and enforce access control as granular as possible. ZTAs are architectures which apply the concepts of Zero Trust in their structure and functionality. The main concepts of Zero Trust can be summarized as follows [48]:

- Under the term resources we should consider all data sources, provided services and devices, which access classified resources.
- Communication is always secured.
- Access is granted per request.
- Access is determined through dynamic policies which observe identity state, resource state, and behavioural and environmental contexts.
- Security and integrity of resources, as well as access request and infrastructure state, are constantly monitored.

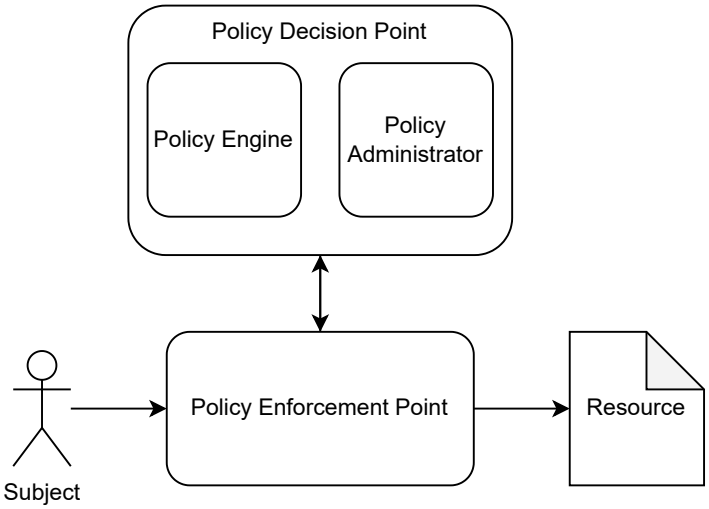


Figure 2.1: ZTA core components according to NIST standard [48]

- Access control is strictly enforced.

The two core components of a ZTA, which stand between the subject and the resource, as shown in Figure 2.1, are Policy Enforcement Point (PEP) and Policy Decision Point [48]. The PEP is responsible for intercepting the access request and then forwarding it to the PDP. It also monitors the access of the resources, if granted, and also terminates the connection to the resource. The PDP is broken down into two sub-components - the Policy Engine (PE) and Policy Administrator (PA). The PE is responsible for assessing the access request by evaluating static policies as well as information from additional sources such as threat intelligence, identity management systems, activity logs, etc. The PE takes the final decision of whether to grant access or deny it. This ultimate decision is passed on to the PA. The PA is responsible for configuring the communication between the subject and the resource based on the decision of the PE. If access is granted the PA may create needed access tokens or credentials for the subject and pass them on to the PEP to start the communication. If access is denied PA is responsible for signalling the PEP to terminate the connection.

Apart from the core components, a ZTA may include additional components to its structure which enhance the decision-making process. Such components may be [48]:

- Identity management system
- Continuous diagnostics and mitigation system
- Threat intelligence feed
- Network Monitor
- Industry compliance system
- Public Key Infrastructure

2.1.2 Microsoft Zero Trust Architecture

In their white paper on the topic ZTA [18], Microsoft outlines similar principles of the Zero Trust. These are:

- Always verify a request explicitly.
- Integrate all available data about identity, device health, location, anomalies, etc. into the decision-making process.
- Allow access only according to the LPP.
- Consider the system always as breached.
- Integrate micro-segmentation, encryption, monitoring as well as automated threat response to reduce the impact of breaches.

Furthermore, there are some logical components of a ZTA outlined. On the side of the accessing entities, there are Identities and Endpoints. Identities may represent a user or a device and require strong authentication. Least privileges should be provided to an Identity when accessing and behaviour should be monitored for anomalies. Endpoints represent the device which is used to perform the access request. Endpoints should be evaluated for device policy compliance as well and their health should be monitored. Resources, which are being accessed, are divided into *Data, Applications and Infrastructure*. Data is what is being ultimately protected and should be identified, labelled, classified, encrypted and accessed only through required attributes. Applications as well as Infrastructure are used to consume the data and should be monitored for abnormal use and faulty configurations. Since all data is accessed through the network, a *Network* component is outlined. The network should be segmented and protective measures such as end-to-end encryption as well as Intrusion Detection/Prevention Systems should be employed. A Threat Protection component feeds from the information provided by the monitoring of the previously mentioned components and calculates risk assessments. These assessments can be manually analyzed or sent to the Policy Enforcement component for decision-making. The *Policy Enforcement* intercepts requests and makes access decisions. It incorporates all of the provided information by the other components into its decision-making process. Lastly, the *Policy Optimization* component is mentioned. It covers the organization's specific policies which are adjusted to comply with business rules, industry requirements and end-user experience.

2.2 UK NCSC Zero Trust Architecture Design Principles

When designing a ZTA or looking to integrate Zero Trust principles in an existing system is a good practice to follow a structured guidance which outlines important aspects and suggests practices in covering the Zero Trust principles. For our evaluation later in the thesis we will be applying a ZTA to an existing Palladio model. We do not want to apply components from our repository in a random and unjustified way. We also want to put the integration process in a more real-life scenario to provide a better demonstration of a real-life application of our ZTA elements. Therefore we are going to use a guidance document which navigates us through

the steps of applying a ZTA and we will map the use of our components to these steps. For this case, we will be using the Zero Trust Design Principles Document [57] provided by the United Kingdom's National Cyber Security Center (UK NCSC). The guide is aimed at assisting systems in an enterprise environment to review their Zero Trust requirements and implement needed concepts to integrate a ZTA. The guide consists of 8 principles which we will briefly discuss in the following section.

1. Know your architecture including users, devices, services and data The first step in the integration process according to the document is to perform a full-scale examination of your system. This includes the discovery of all assets such as devices, services and accessing users. The data stored by the system should be examined as well and classified according to its sensitivity and locations of storage. Performing such a survey on your system helps to better understand the critical points and weaknesses of an architecture. It also allows us to identify which resources need protection and balance costs and efforts to achieve it. This examination would also allow for performing a better risk assessment and determining risk handling politics.

2. Know your User, Service and Device identities The second step of the guide is concerned with identities in the system. Identities are part of the authenticity security goal and without it, a system cannot determine whether a request is genuine or not. In a ZTA identities are fed to a *PE* for the decision-making process. Hence the guide suggests establishing unique user, device and service identities. Identities should be stored in a secure dedicated directory which is unique for a system. This would ensure that there is no ambiguous information stored about an identity in the system. Establishing identities also allows the assignment of roles to assets in the system and the enforcement of the LPP. However, implementing and maintaining an Identity Management System is a whole research topic. The authors of the document reference their dedicated guides for this purpose. This is why here the identities topic is covered on a more abstract level.

3. Assess user behaviour, service and device health The third step encompasses the processes of observing and assessing the behaviours of users as well as the health of devices and services. User accounts might be stolen and used in an unusual way and devices may be compromised after a successful enrollment in the system. Indications of such events should be constantly monitored in a ZTA and analysed. This is why in this step of the guide, a system should be provided with solutions to detect such events. For example, devices should be evaluated to predefined configuration requirements or their security state should be evaluated per request based on a device fingerprint. Users should be monitored from what location they are making a request as well as the frequency or type of requests they are performing. Monitoring the state of resources and services should also be performed to ensure that nothing has been corrupted. Data collected in those processes is then to be supplied to the *PE* of the system to impact the final decision.

4. Use policies to authorise requests The next topic of the guide is the policies and their evaluation. Policies are defined by the authors as the power of a ZTA and their evaluation happens in a *PE*. Enterprises should take the time to carefully integrate a *PE* and design it to make decisions based on multiple signals to improve the policy evaluation process. The policies and *PE* should be protected components of the architecture and should not be exposed to unauthorized access or access by user end devices in order to limit the possibility of corruption.

5. Authenticate and Authorise everywhere As a fifth step in the integrating process, we have the requirement for authentication and authorization everywhere. This means that requests for every critical resource should always be passed through a *PE* and evaluated there based on multiple inputs. Critical resources should not be accessible via an alternative path. The requirement for authentication ensures that the authenticity of a user or device performing the request can be checked and the authors discuss that a Multi-Factor authentication system should always be applied and not a simple authentication scheme.

6. Focus your monitoring on users, devices and services The next concept of the guide is the monitoring of users, devices and services. A system should constantly collect logs about who accesses what using which device as well as how services function upon requests. This data should be forwarded to a centralized monitor where analyzed and the result of the analysis should be used when evaluating requests. This would also help in identifying problems in the functionality of the ZTA itself as well as improve the quality of policies.

7. Do not trust any network, including your own The seventh guidance principle is aimed at the perception of the network a system is using. It directly quotes one of the core Zero Trust principles namely to not trust any network even your own. Trust should be built into devices, users and services and not in the communication between them. Although the network is always treated as hostile in a Zero Trust context, network hygiene should not be neglected and the network should constantly be inspected for unauthorized hosts, for example, or unpatched network components.

8. Choose services which have been designed for zero trust The last guidance principle discussed the types of services we are using in our system. The authors suggest that when choosing a new service, its compatibility with a ZTA should also be considered and carefully evaluated. This would reduce the complexity of architecture and reduce the costs of integrating the service in a secure environment. Legacy systems, meaning that they are not designed for zero trust, might require additional components and configurations which increases the time and effort required to integrate them as well as the complexity of the architecture. Zero Trust Solutions provided by trained professionals should always be preferred to self-made Zero Trust solutions because the error probability in developing Zero Trust solutions is too high to be neglected according to the authors.

2.3 CISA ZTA Maturity Model

Achieving Zero Trust is a task which extends to multiple aspects of an enterprise. Solutions should be applied to manage users and devices as well as policies and practices should be established about processes happening in an enterprise. Therefore enterprises apply multiple changes in their governance and introduce new components and services with the aim to cover the Zero Trust principles. In this process of implementing a ZTA, it is good to be able to evaluate the progress achieved in a structured way. This is where a maturity model plays a significant role. A maturity model defines levels of maturity of a certain concept and describes what requirements should be met to be able to achieve this level of maturity. The Cybersecurity and Infrastructure Security Agency (CISA) have developed such a maturity model for evaluating

the level of zero trust achieved by an agency. The CISA maturity model divides an agency into five pillars which are *Identity, Devices, Networks, Applications and Workloads and Data*. Each of these pillars might advance at its own pace, meaning that the level among pillars might vary. Additionally, the model outlines three cross-cutting capabilities which are contained in all of the pillars and for which each pillar provides a requirement. The maturity model outlines four levels of maturity. These are the *Traditional, Initial, Advanced and Optimal*. Each of the levels requires more effort and greater detail and complexity in the applied solutions than the previous one. Each of the pillars is further divided into functions and each function provides a requirement for each of the maturity levels. In the following section, we provide a summary of the pillars and the functions they are comprised of.

Identity Pillar An identity in the maturity model context is the set of attributes that uniquely describe a subject, no matter whether it is a person, a device or a service. The identity pillar encompasses aspects of authentication, identity management and the requirement to provide the right entities with the least privileged access to the right resources. Mentioned in this pillar are the needs for credential, identity and attributes manager combined with a strong authentication solution. The functions of this pillar are Authentication, Identity Stores, Risk Assessment, Access Management as well as the three cross-cutting capabilities. The Authentication function describes requirements for authentication solutions such as the use of a Multi-factor authentication system and its qualities such as phishing-resistant. Identity Stores function evaluates the stores where identity information is stored and their location. The Risk Assessment function evaluates the requirement for assessing risks concerned with the identity of an entity. The Access Management function observes the type of access provided to identities. Whether it is a permanent access or one that expires eventually and needs to be reevaluated.

Devices Pillar A device in the maturity model is considered every asset with its software, hardware and firmware that has the ability to connect to a network. The pillar encompasses requirements for monitoring and maintaining the health and security posture of company-owned devices as well as requirements for handling devices in a bring-your-own-device environment. The functions of the pillar are Policy Enforcement and Compliance Monitoring, Asset and Supply Chain Risk Management, Resource Access, Device Threat Protection and the three cross-cutting capabilities. Policy Enforcement and Compliance Monitoring encompasses practices and solutions to monitor the posture of devices, compare it to company policies and enforce changes and updates to a device's configuration and/or software. The Asset and Supply Management function evaluates the inventorying and storing of device information. Resource Access describes the requirements for considering device information when evaluating access requests. Lastly, the Device Threat Protection function describes solutions which need to be applied to devices in order to protect them from attacks and corruption.

Networks Pillar Networks in the maturity model are considered every open communication medium which is capable of transporting messages. In the pillar are mentioned requirements about segmenting the mediums into multiple separate parts and bringing security controls closer to the protected resource. The functions in this pillar are Network Segmentation, Network traffic, Traffic Encryption, Network Resilience and the three cross-cutting capabilities. The Network Segmentation function evaluates the level of segmentation of the system architecture.

Whether the system uses a single broad defence perimeter or is separated into multiple smaller ones. Network Traffic Management is concerned with establishing traffic rules and whether they are static for the system or different for applications based on application profiles. The Traffic Encryption function discusses security mechanisms and their use in the encryption of traffic when transporting along mediums. Network Resilience observes the configuring of the network to respond to applications' resource demands and the network's capability to dynamically adapt to these demands.

Applications and Workloads Pillar Applications and Workloads in the context of the model are all systems, computer programs or services executed on-premises, mobile device or cloud environments. The pillar encompasses requirements for the governance and management of applications throughout their lifecycle from development, through testing and deployment. It is evaluated also how access to applications is granted, how applications are protected and how are they exposed to the end user. The functions of the pillar are Application Access, Application Threat Protection, Accessible Application, Secure Application Development and Workflow and Application Security Testing and the three cross-cutting capabilities. In the Application Access function, it is observed whether access to applications is provided based on static attributes or contextual information is also evaluated. Similar to the threat protection in the devices pillar, here the Application Threat Protection is concentrated on applying threat protection solutions to applications and their workflows. Whether the applications are accessed only through dedicated networks or are available on public ones is evaluated in the Accessible Applications function. Secure Application Development and Deployment workflow observes the practices, tools and environments of the development and deployment of an application and how they are protected. Lastly, the Application Security Testing evaluates to what extent security tests are integrated into the applications' development process.

Data Pillar In this maturity model, data is defined as every structured or unstructured file or fragment and their metadata which is stored on on- or off-premise stores. The pillar determines the maturity level of how data is being handled in the agency. The functions which are contained in the pillar are Data Inventory Management, Data Categorization, Data Availability, Data Access, Data Encryption and the three cross-cutting capabilities. Data Inventory Management determines the level of inventorying the data and data categorization and labeling is observed by the Data Categorization function. From where data can be accessed is evaluated in the Data Availability function. The access controls to data and the LPP are observed in the Data Access function and lastly, whether stored data is encrypted and what mechanism are used for it is evaluated in the Data Encryption function.

2.4 Palladio

Palladio [43] is a component modelling approach which can be used to model systems and then perform analysis of quality requirements on those models. This allows for making predictions about a system's quality requirements at design time, which are not based only on educated guess but rather on the system's possible structure, behaviour, deployment and usage models.

The building blocks of a Palladio model are the first-order elements *Basic Component* and *Interface*. As per the Palladio book [43] the *Components* are defined as "contractually specified

building blocks which can be composed, deployed and adapted without understanding their internals" and *Interfaces* are the points of interaction between components. *Components* are described by specifying their required and provided *Interfaces*. They are specified in a *Repository* which holds all of the *Basic Components*, *Interfaces* and *Composite Components*. *Composite components* are components which are assembled by combining other *Basic Components* and again specifying required and provided interfaces.
The components and interfaces then can be used in order to specify a system. A system has again its own provided and required interfaces and contains inside components which are connected to each other via interfaces. This strongly resembles the structure of a composite component. However, the main difference is the possible allocation. Different parts of a system can be deployed on different resource containers. On the other hand, this is not possible for a composite component.

In order to specify the behaviour of a component a Service Effect Specification (SEFF) is used. SEFFs define the connection between a component's required and provided interface. Calls to the required interface are specified as *External Call Actions* and functions of the component, to which the SEFF belong, are modelled as *Internal Actions*. The internals of the actions, such as algorithms, are abstracted from the representation. The control flow is represented as a sequence of actions as well as loops, forks or branches. To further specify the quality model of a component, SEFFs can be extended to Resource Demanding SEFF (RDSFF). In RDSEFF resource demands, such as CPU, and HDD, are modelled using abstract work units, for example, 10 CPU work units. After a deployment context has been specified with exact resources, the actual processing time can be calculated. The RDSFFs allow for further specification of failure probabilities. These refer to probabilities of software errors and allow the analysis of the reliability of a component.

Palladio also allows the specification of the execution environment by modelling the resource container with their processing resources such as CPU and HDD. Resource containers can be connected by linking resources with specified communication resources such as network speed. After the execution environment has been defined, the allocation model can be created, which specifies on which containers the different parts of the system are deployed.
Finally, the usage model of a system is defined. This model represents how a system could be used in its operational phase. With the usage model, we can define different workloads which a system can experience.

2.4.1 Analysis

After putting all of the pieces from the previous sections together, we have a complete model of a system on which we can perform quality analysis. Palladio offers multiple analysis tools, which use different analysis techniques, in order to gather quality metrics about the system. One example is the SimuCom simulator which can provide response times and resource utilization for single components or the entire system, based on the provided Palladio model. Further metrics which can be calculated with Palladio include throughput and software failure potentials.

When doing a performance analysis, the simulator starts feeding requests to the system model according to the usage model. The system workload can be defined as a closed one and the number of requests which circulate in a system is specified. The workload might also be

open, where there is no limit on the requests which arrive at the system but we need to specify the interarrival time of the requests. We specify a limit to the simulation time and the number of measurements that should be made within this time limit.

Confidentiality issues arise when data arrives at a place in a system where it should not be. Since confidentiality issues compromise seriously a system's security and can be hard to resolve in later phases we need to be able to discover them at design time. Model-based confidentiality analyses prove to be suitable in this case and such analysis can be performed in the following way [46]:

1. Create an Analysis Definition
2. Model the system
3. Run the analysis

In the first step, which can be performed by a security expert or a system designer, the following elements should be specified:

- Properties of nodes
- Properties of data
- Behaviour of nodes, which specifies what are the output data properties based on the input data properties.
- Comparison function, which detects violations by comparing the properties of data and nodes.

In the second step, which is performed by the system designer, the system is modelled and the elements from the Analysis Definition are applied to the model. In the last step, the analysis is executed and violations are detected based on the comparison function. The analysis itself maps the model with the added Analysis Definition elements to a label propagation network. In a label propagation network the properties of nodes and data are treated as labels and the behaviour of nodes is the label propagation function. After the execution of the analysis, it is known what data at which node has arrived as well as the labels of the data at each node. According to the comparison function labels are evaluated to detect violations [46].

To specify labels and node behaviours in Palladio we use security annotations. Firstly, as per the confidentiality modelling documentation [17], labels and characteristic types are specified in a data dictionary. The dictionary is a file of type *.pddc* in the Palladio project and for specifying labels in it there is a dedicated syntax. Labels and their value range are specified as an *enum* and then characteristics of nodes are specified as *enumCharactersiticType* which uses an *enum*. After, we have specified the characteristics we can define how Palladio components handle them in the SEFF diagrams. For this purpose, we use confidentiality variable usages, where we set the variable and expressions for the characteristics. Expressions are evaluated to true or false and a true value means that a characteristic is available for this variable. Expressions have the form of *variable.characteristicType.value := Term* where *vairable* is the variable of the variable usage, *characteristicType* is the characteristic and *value* represents a value from the *enum* of the characteristic. *Terms* on the right side can take the constant values *true or*

false, can be evaluated using binary logic and can also reference other confidentiality variables, for example, parameters or return values of functions. Node characteristics are specified in a *.nodecharacteristics* file where we can set variable characterisations for component instances from the assembly model and resource containers from the resource environment model. In the usage mode of a Palladio model, we set the characteristic of input data to the system by using again variable usages and setting the input parameters of system entry calls.

When we have a Palladio model we need to extract a Data Flow Diagram from it. Then on the diagram, we need to evaluate the node behaviours in order to perform the label propagation. Then, in the end, we obtain a set of nodes with their node characteristics and the characteristics of the received data by the node. The required tools to extract data flow diagrams from a Palladio project and propagate labels are coded in a Java library in the *org.dataflowanalysis.analysis* package. We instantiate the *DataFlowConfidentialityAnalysis* class where we set the paths to the usage model, the allocation model and the node characteristics of the analysed system. Then we use the *findAllSequence* and *evluateDataFlows* functions of the class to obtain a list of *ActionSequence* objects. *ActionSequence* objects contain lists of *AbstractActionSequenceElements*, which we iterate to extract node labels. Constraints are defined as logical expressions which compare the extracted labels.

3 Related Work

In the following chapter, we present research in the field of modelling systems for performance and security analysis. In the second section, we discuss suggested models of ZTAs from the literature.

3.1 Modelling for Performance and Security Analysis

In this chapter, we explore how system architectures might be modelled in order to analyse quality requirements. We also look at different proposals of how a ZTA could be modelled for different systems and eventually implemented.

Fernandez et al. [19] perform an analysis on ZTAs regarding the principles of Saltzer and Schroeder for creating a secure system. Various ZTAs are mapped to the principles which they enforce. The security patterns which help to enforce those principles are discussed. The paper aims to outline a Security Reference Architecture (SRA) of ZTA. The proposed SRA extracts elements from different security patterns in order to form a concept model of a ZTA. The authors then try to evaluate ZTA by answering questions regarding the performance and security of such a system. However since no quantitative measures of ZTA systems are available as well as the proposed model is only a concept model and no analysis can be performed on it, the answers in the evaluation are based only on educated guess.

Cortellessa et al. [12] introduce a framework which allows modelling the performance and security aspects of a software architecture. The idea is to create a performance model which contains security annotations and can be compared to alternative performance models which may contain different security mechanisms or not at all. The authors have created a UML library which models the basic security mechanisms of encryption, decryption, signature generation and verification. Then with these models, they have modelled composite security mechanisms such as Data Confidentiality which contains in itself encryption and decryption mechanisms. When modelling a system, first an Application Model is created which consists of static and dynamic descriptions of the system in UML. Then, the model is annotated with security requirements. Then Security Enabling Model is obtained by embedding the modelled mechanisms from the described UML library into the UML model. Lastly, the Security Enabling Model is translated to a Generalized Stochastic Petri Nets model to analyse performance.

Sharma et al. [47] are using Discrete Time Markov Chains (DTMC) to make predictions about the performance, security, reliability and cache miss behaviour of a system. Their idea is to map a software architecture to an absorbing DTMC, which means that the DTMC has one state which has no outgoing transitions. This state represents the termination of the execution of a system. Each architectural component is then mapped to a state in DTMC and the transitions between states represent the flow of control between components in the architecture. The metric which is observed from this initial modelling of the architecture is the number of visits to

a state before reaching the absorbing state or in other words, the number of times a component is visited in an execution. Then the model is enhanced with additional information such as reliability or performance information so that different analyses can be performed.

3.2 Modelling Zero Trust Architectures

Google present their approach towards a ZTA, Beyondcorp, in [55]. The system aims to provide fine-grained access control for different company resources based on full authentication and authorization per each request. The ZTA has two database modules - *Device Inventory Database*, for storing company-registered devices and the *Users/Groups Database*, for storing users' roles and responsibilities. A Single Sign On (SSO) component is responsible for authenticating a user/device on requests and providing an authentication token to be presented to the Access Proxy (AP). The AP serves as a PEP and redirects requests to the Access Control Engine (ACE), where policies are evaluated. A *Pipeline* is responsible for feeding information to the *Access Control System* from the databases as well as the Trust Inference component. In the Trust Inference component, a trust score is calculated for the accessing user/device. The paper [37] continues the discussion of Beyondcorp's architecture. There a more abstract categorization of the ZTA components is presented. We can distinguish between:

- Data Sources - providing contextual information.
- Access Intelligence Components - which encompass the ACE, Access Policies and Trust Inference.
- Gateways - these are the various PEPs, such as network switches and web proxies.
- Resources - the assets, services and data which need to be protected.

In [39] Biplob and Muzaffar propose a ZTA model for a smart industry system, where manufacturing devices communicate with each other and need to be managed by an operator. The architecture consists of two core components. The first one is the Authentication Component and the second is the Authorization Component. An additional component for device discovery, validation and certificate assignment, named Enterprise Discovery System, is suggested. An Endpoint Compliance Management (ECM) system is suggested as a component responsible for regularly monitoring if the devices' state complies with security policies. When an operator is accessing a device the Authentication component performs a signature and integrity check in order to authenticate the operator. Then the Authorization component evaluates the operator's authority by using the information provided by the ECM. Internal device-to-device communication happens when manufacturing devices send data to a storage server. However, authors have only suggested encrypted communication in this case which can be argued whether this is enough to comply with the ZTA core concepts since no policy evaluation is enforced. The authors have modelled an additional scenario where the storage server resides on a cloud server. In this case, a Cloud Connector component is introduced which enforces policies when communicating with the cloud. The paper however does not provide additional performance or security analysis on the suggested model.

Ramezanpour et al. [42] propose a framework for an intelligent ZTA (i-ZTA) in next-generation networks 5G/6G. The system is categorized as intelligent since it incorporates machine-learning approaches in the decision-making process. Before presenting their framework the authors provide a general ZTA reference architecture based on ZTA focus areas, proposed by the U.S. Department of Defence. The proposed focus areas include users, devices, data, network, analytics, automation and workload. In the presented basic architecture we can observe the suggested core components and their relationships. A user device, which has its own security state, is accessing data, applications, assets, and services (DAAS). The request is intercepted by a PEP and forwarded to a Network Access Control (NAC), which can be mapped to a PDP. Inside the NAC is the core process of the architecture - the Trust Evaluation (TE). In order to evaluate the trust, data comes from multiple sources apart from the policies. The most important of these is an authentication component which provides identity information. The device communicates additionally its security state. There are also different analytic components such as user/device, network and anomaly detection. In the centre of the actual i-ZTA framework are again the PEP and PDP components. Additionally, the architecture contains three intelligent components. These are:

- Intelligent agent/portal (IGP) - responsible for analyzing the security state of the accessing device.
- Intelligent Network Security State Analysis (INSSA) - responsible for analysing the environment and providing dynamic rules using information from multiple components, mentioned under.
- Intelligent Policy Engine (IPE) - responsible for aggregating and analysing the information from the IGP, INSSA and static policies in order to make the final decision.

The static rules for the system come from the Data Access Policy, PKI, ID management and Industry Compliance. The INSSA is using a CDM System, SIEM system, Activity logs and Threat intelligence to perform its analysis. The authors then propose a design of a 5G network integrating their i-ZTA. However, no actual implementation is presented as well as performance and security evaluation of the model. This evaluation is important since integrating machine learning components in a system could result in additional performance issues as well as attack vectors.

In [10] Chen et al. are presenting a ZTA for 6G networks. The network consists of communities of user equipment (UE). The ZTA is described in a scenario where UE from one community is accessing another community. However, if we set the source and destination to be the same community we can assume that we can use the ZTA for internal communication. When a UE sends an access request to another community it is intercepted by an Access Control Proxy (ACP). The ACP forwards the request to the local controller. There an Identity Management system verifies the identity of the UE. A Trust Evaluation Engine calculates the trust score with the help of three external services - Vulnerability Database (VDB), Cybersecurity Event Ledger (CEL) and Anomalous Behaviour Detector (ABD). Then the trust score is provided to a Security Policy Engine (SPE) where the request is evaluated according to policies and the trust score. The SPE makes the decision whether to access or deny the request and forwards it to the ACP to implement it. The author evaluates the effectiveness of the architecture by

comparing it to two other security architectures. Worm spreading is simulated in the three architectures and the filtering rates of attack packets as well as the number of missed attack packets are measured. The authors also test the robustness of the system by setting a validity period on accepted requests, during which access by this device is not re-evaluated. The system maintains a good attack packet filtering rate while increasing performance. However, we can argue whether this characteristic should be evaluated since the practice of trust validity does not comply with a core ZTA principle - enforce access control on every request. Finally, the authors pay attention to the performance of the system as an open issue. Trust evaluation is categorized as a costly process, which can be optimized using different design alternatives, such as distributed external services across the communities or periodic active synchronisation of logs instead of on-demand requests. However, in order to evaluate these alternatives a more detailed model of the architecture is required on which to perform performance analysis.

Boo et al. [26] propose a ZTA for blocking malicious access to enterprise resources. The architecture consists of three main components - a PDP, PEP and an Authentication Server Function (AUSF), which consists of multiple sub-components. The PEP consists of Wi-Fi routers and a VPN server. In the PDP the authors outline the User Control Function (UCF), Device Control Function (DCF) and Connection Control Function (CCF). The AUSF provides a database for storing user identities and credentials which are provided and managed by the UCF. The DCF is responsible for keeping track of all devices which have access to the enterprise resources as well as checking their security state. The DCF also manages the routers in the PEP. Users first should be authenticated by the PDP and the PDP deploys policies to the PEP which describe the resources which can be accessed by the user. Users should connect to the VPN in order to be able to send access requests for the resources.

In [31], Lee et al. incorporate security situational awareness (SSA) into a Risk Adaptable Access Control (RAdAC) model in order to propose a ZTA. Typical for RAdAC is the attempt to balance the operational and security needs of an enterprise. This means that access to corporate resources might be relaxed in order to maximize the benefits in safer situations. If the enterprise is, however, in a tight security state then the risk is evaluated as higher and access to resources might become more fine-grained. As PEP the authors suggest elements of the network infrastructure, such as a WiFi base station, which intercepts requests and sends them to a Context Handler. The Context Handler is an unusual component, compared to the other suggested architectures, and is used as a centralized control unit which coordinates the access evaluation process. The architecture also has a Policy DB, a Risk Evaluation Function and an Access Decision Function. These three components form the PDP. Context information is provided by the Environment Evaluation component. It consists of multiple plug-in sub-components such as SSA or Location service. The Environment Evaluation Components translate context information into attributes which can be used by the PDP in the decision-making process. Information about subjects and objects is stored in a Subject/Object DB. Lastly, policies are enforced with the help of firewalls on the path between subject and object. The Firewall Provisioning component of the architecture is responsible for identifying the firewalls standing on the path of a request and supplying them with policies by translating PDP decisions into firewall-specific rules.

The Cloud Security Alliance (CSA) provides a specification for a Software Defined Perimeter (SDP) in [13]. SDP provides a way to segment the network by using software-defined perimeters and thus hide resources. This also allows to control access and prevent unauthorized access.

SDP enforces a policy-based, identity-centric access model. In SDP the two main components are an SDP Controller and an SDP Host. Hosts can be either an Accepting Host (AH) or an Initiating Host (IH). AHs can be seen as the PEP of the system. They might be deployed on the same device as the resources or on the path to it. AHs evaluate requests based on the policies provided prior to the request from the SDP Controller. By default, all access requests are denied if no instructions are present. The IH represents the entity making the request and thus can be either a user or a device. IH should authenticate themselves to the SDP Controller before requesting access to resources. After authenticating, the SDP controller determines what resources an IH can access and deploys instructions to the AH of those resources to accept requests from this IH. Since the SDP Controller evaluates an IH and makes the decision of which resources to allow the IH to access, the SDP Controller is the PDP of the architecture. To authorize access, the controller may use an on-premise policies provider or a third-party service such as an Active Directory which can be hosted off-premise. Additionally, the controls may obtain information about the context of the IH such as geolocation or a host validation service. This type of data as well as identity and device attributes of the IH are integrated into the decision-making process. For the authentication, the SDP controller may utilize an on-premise service such as an internal user table or a third-party Identity Management System which can be hosted again on-premise or in the cloud.

4 Running example: JPlag

As a running example, which we will use throughout the chapters to demonstrate how we apply the developed in this thesis concepts, we define a system. We create a model of the JPlag[24] system in this chapter, with the intent to later extend it with a ZTA and security annotations.

4.1 JPlag

JPlag [24] is a system which enables the automatic detection of plagiarism among multiple files containing source code, EMF meta-models or natural language. JPlag performs a rather complex text analysis than simply comparing text bytes. This is because JPlag considers the programming syntax and structure of programs. Therefore, JPlag can even detect plagiarism in files which are trying to circumvent traditional similarity detection approaches. The system accepts files or folders containing files to compare and analyse them. Based on the result of the analysis the system creates a report which can be viewed using the JPlag's dedicated report viewer. Currently, the system is available for local use via the CLI or programmatically via its Java API. The report viewer can similarly be used locally but is also available on the project's GitHub pages[25].

4.2 JPlag Scenario

For our example, we have the following scenario. We assume that JPlag is a publically deployed system on a university network. which accepts files for evaluation and generates reports. The JPlag system should offer the user a GUI through which the user can send their files for analysis. The JPlag algorithm resides on a backend server alongside the report generator. After the analysis is performed the system requests a report from the report generator and saves the result in a database. The user can access the database to obtain the result or take it directly from the JPlag response. The system should also offer a deployed version of the report viewer where the user can send the generated reports for displaying.

4.3 JPlag System without ZTA

Following the scenario from the previous section we model the following components inside a JPlag repository. Since we are designing the system from scratch we are going to do it with security in mind. This means that we will enable components to propagate requests throughout the system that contain security features such as authentication or authorization for example. First, we create a *JPlagGUI* which requires and provides the *IJPlag* interface. The *IJPlag*

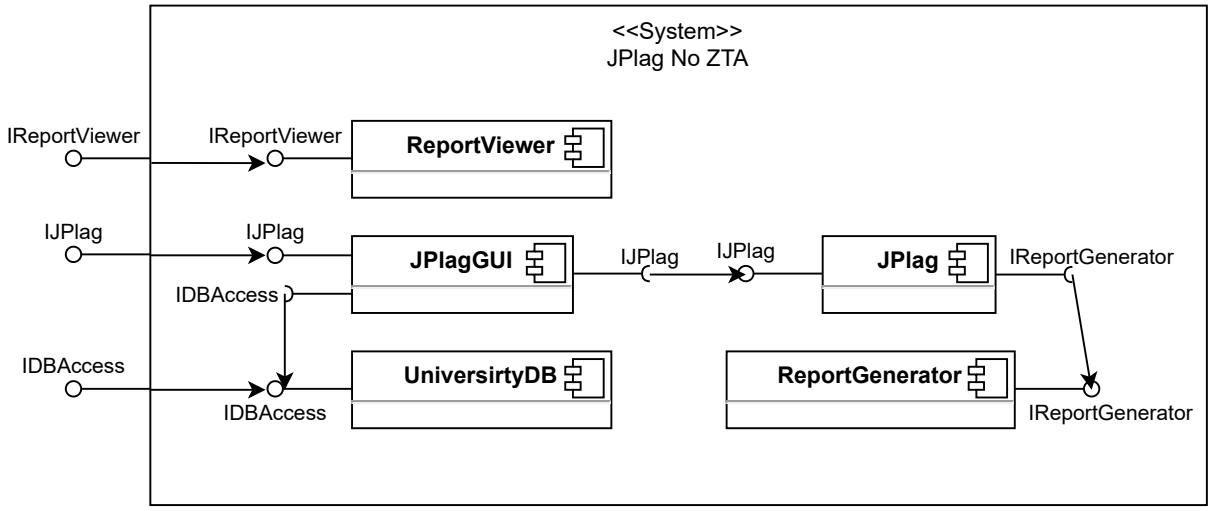


Figure 4.1: JPlag Assembly Diagram

interface has a single signature *run()*. When the component's provided function is called it forwards the request to its required interface to obtain a JPlag report and finally calls the *IDBAccess* interface to store the report. The *IDBAccess* interface has two signatures *put()* and *get()*. It is used to describe database access.
Next, we have the *JPlag* component which represents the JPlag's analysing functionality. The component provides the *IJPlag* interface and requires *IReportGenerator*. When the provided functionality is called, the component performs the inner action of analysing the input and then makes a request to its required interface to obtain a report for its result.

The *ReportGenerator* component provides the functionality of generating a report through the *IReportGenerator* interface. When called, the component simply performs the inner action of generating a report.

Lastly, we define the components *UniversityDB* and *ReportViewer*. The *UniversityDB* provided the *IDBAccess* interfaces and is our database where reports are stored. The *ReportViewer* component provides the *IReportViewer* interface and we use it to represent the entity where users send reports of JPlag to be visualized.

Using the components from our JPlag repository we create the example system shown in Figure 4.1. The system provides the interface *IReportViewer, IJPlag and IDBAccess* to allow the user to request analysis, view reports and access the university's database.

5 ZTA Meta-Model

In the following chapter, we summarize and present the general request evaluating process in a ZTA. Next, we extract the logical components which are responsible for the different processes in the general workflow. We identify the connections between those components. After performing this analysis, we end up with a meta-model of a ZTA which we instantiate in the next chapter.

5.1 ZTA Request Evaluation Process

When analysing the Zero Trust solutions presented in the NIST document[48], the SDP specifications [13] and Beyondcorp paper [55], we can extract three evaluation processes of a request in a ZTA.

From the NIST document [48] we extract the process of evaluating an access request shown in Figure 5.1. After the subject has sent a request it is intercepted and forwarded to a PDP. For the evaluation, the PDP obtains policies and context data before proceeding with the evaluation. According to this evaluation, it forms a decision and creates a configuration based on this decision. Then it forwards the configuration to the interceptor. The interceptor of the request acts based on this configuration and either forwards the request to the requested resource or denies it.

The situation in an SDP-integrating architecture is a bit different and is presented in Figure 5.2. There the subject must first interact with the controller entity before sending a request to a resource. The subject first contacts the controller and an authentication takes place. Then the controller evaluates the request and creates configurations for the Accepting Hosts. The configurations are then distributed to the Accepting Hosts and the subject. After that, the subject sends the actual access request for a resource to an Accepting Host. Based on the distributed configurations the Accepting Hosts allow or deny the request. It is important to notice that we are paying more attention to the interaction of the subject with the SDP, rather than the interaction with the Accepting Host. The request evaluation process in the Accepting Host resembles strongly the functionality of a simple firewall. Technically the Accepting Host enforces policies but the enforcement manner depends on a previous decision made by the SDP. Therefore, the SDP is indirectly enforcing the policies. Since the SDP also evaluates the context of a subject, we want to focus on its policy evaluation process.

In the case of Beyondcorp, as shown in Figure 5.3, the subject sends an access request to the Accessing Proxy which checks the authentication and forwards the request to an authenticating entity if needed. Then the request is forwarded to an Access Control Engine where policies are evaluated. The engine acquires context data from different sources to evaluate context. Then a decision is made and is sent to the Accessing Proxy. It then either forwards the request to a resource or drops it.

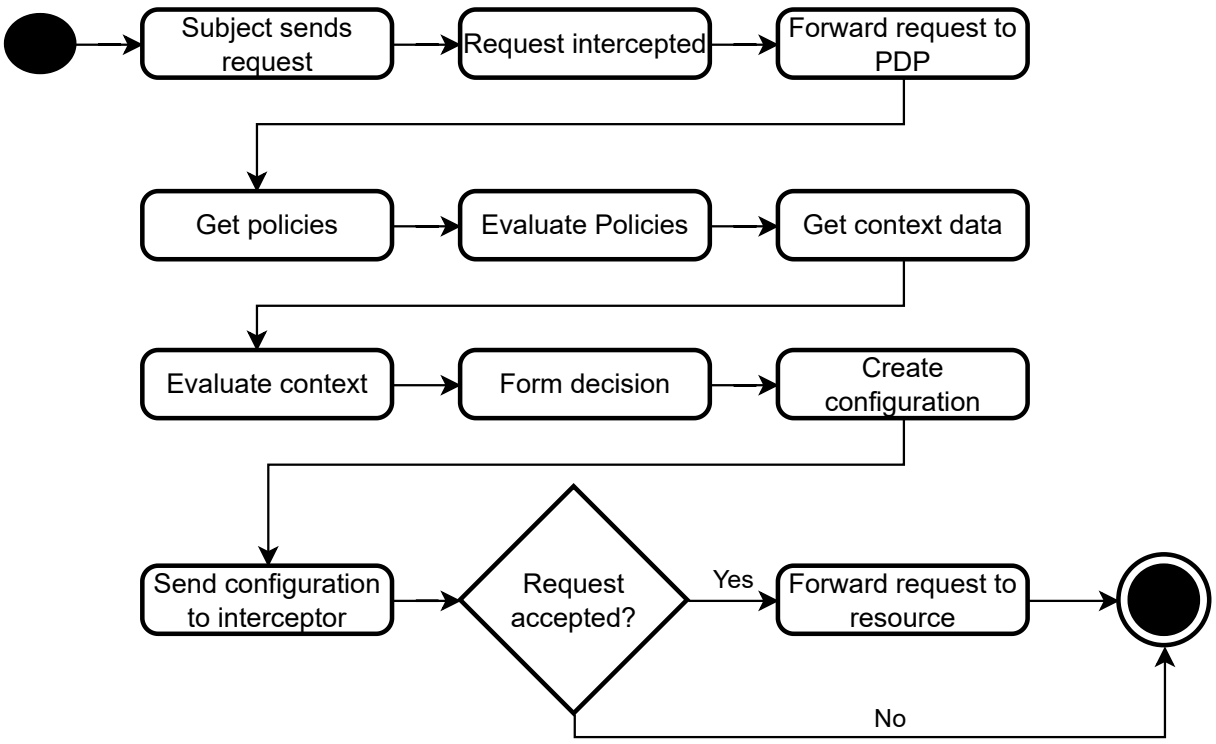


Figure 5.1: NIST ZTA Access workflow

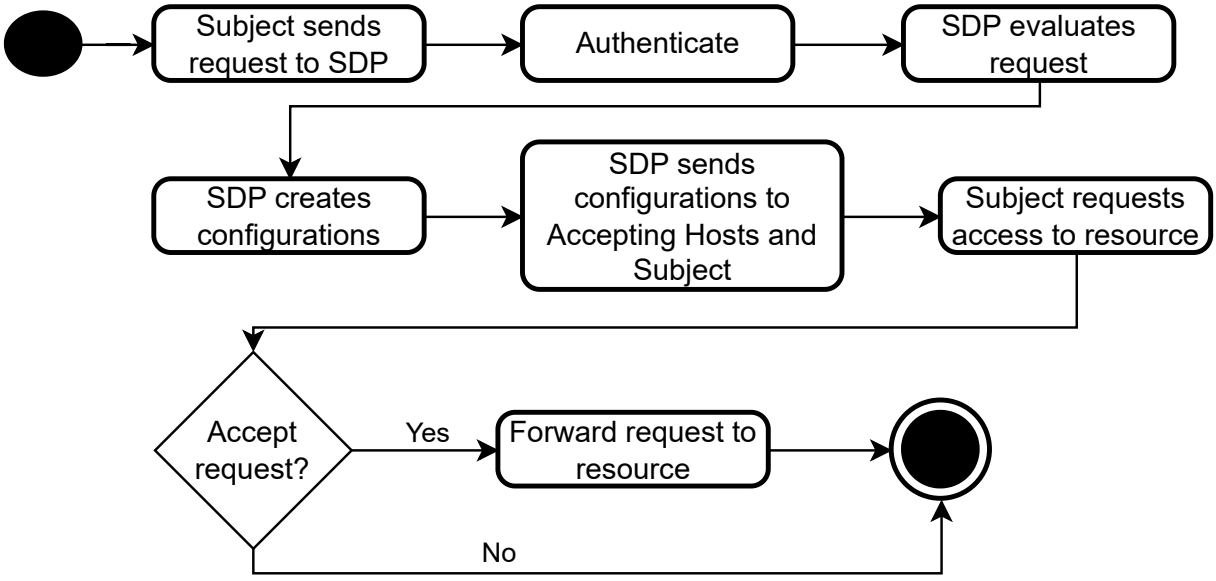


Figure 5.2: SDP Access workflow

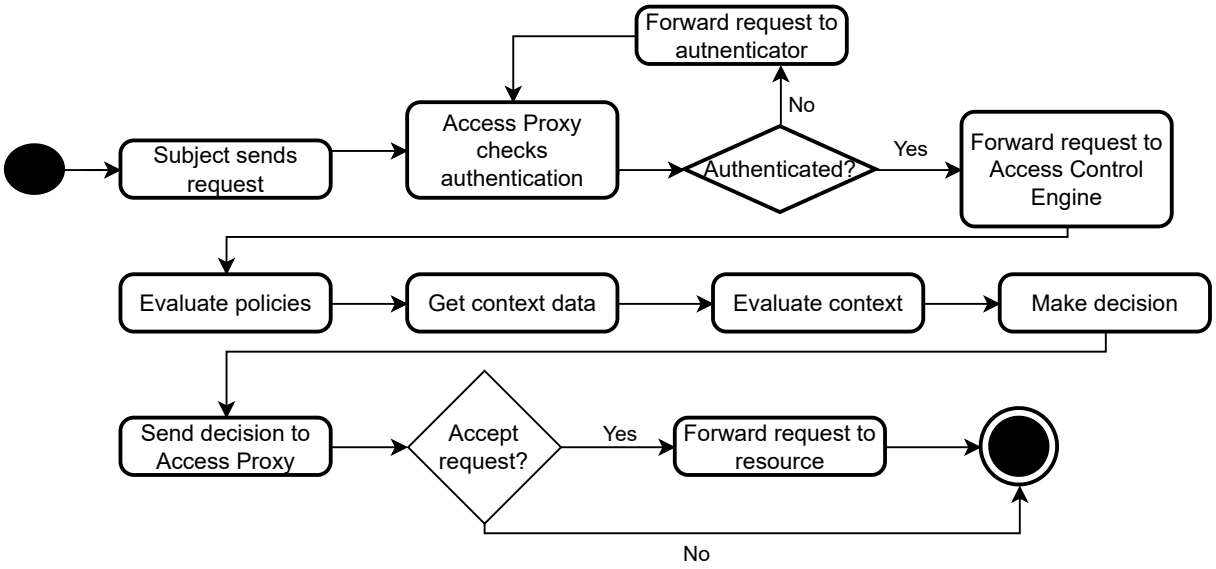


Figure 5.3: Beyondcorp Access workflow

5.2 ZTA Tasks

When looking at a system we need to identify what tasks happen during its workflow. If we can identify these, we can then determine different roles and create components which are responsible for performing those roles. In the previous section, we observed three different workflows of architectures which follow Zero Trust principles. Although they may vary in some aspects we can identify the main tasks which need to be performed in a ZTA. We end up with the following tasks:

1. **Request interception** - request for accessing a resource should be collected before getting to the resource.
2. **Forwarding requests to a Decision Point** - each unauthorized request should be forwarded to a Decision Point.
3. **Evaluating policies** - a request should be evaluated according to the system's policies.
4. **Supplying context data** - information about the context of a request should be proved. Context data may be information about authentication or geolocation.
5. **Evaluating context** - the provided context data should be used in the decision-making process.
6. **Making a decision about the request.**
7. **Creating configuration** - based on the decision about a request a configuration is created to establish or terminate a communication path between subject and resource.
8. **Supplying configuration to interceptors.**

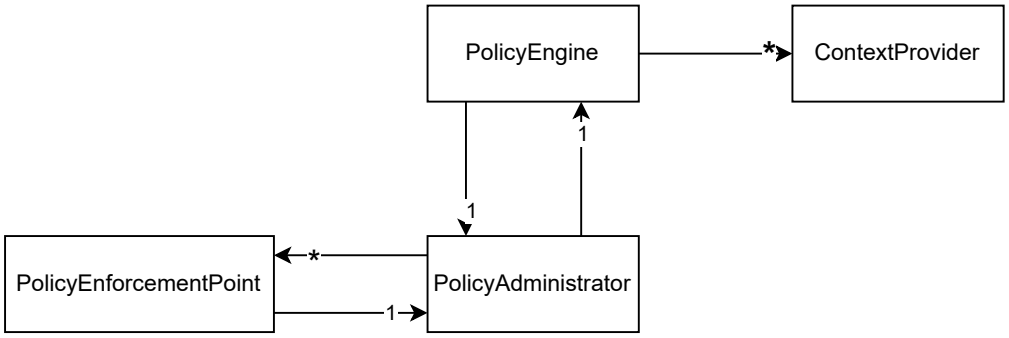


Figure 5.4: ZTA Meta-model

9. **Enforcing a decision** - a decision made by the decision point should be enforced according to the created configuration. This happens by either forwarding the request to the resource or dropping it.

5.3 Logical Components

Following the request evaluation process analysis and the identified tasks, we specify logical components. The components have a specific role in the architecture, described by their tasks, and need to be included in an architecture to be able to recreate the Zero Trust request evaluation process. The components are:

1. **Policy Enforcement Point** which is responsible for performing tasks of intercepting a request, forwarding it to a decision point and lastly enforcing the decision.
2. **Policy Engine** which is responsible for evaluating policies as well as context and making the decision.
3. **Policy Administrator** which also forwards requests to the decision point, prepares configurations for the interceptors and distributes them.
4. **Context Provider** which takes the role of supplying context data to the decision point.

5.4 ZTA Meta-model

When we combine the components from the previous section we end up with the following meta-model of a ZTA as shown in Figure 5.4. The *PolicEnforcementPoint* communicates with a *PolicyAdministrator*. The *PolicyAdministrator* may manage multiple *PolicyEnforcementPoints*. The *PolicyAdministrator* communicates with a *PolicyEngine* thus forming a PDP. We divide the PDP into two separate components for a better separation of privileges. The *PolicyEngine* has multiple *ContextProviders* which provide context data.
Then we have the following general access workflow of a ZTA. First, the *PolicyEnforcementPoint* receives the request by the user and is an entry point to the ZTA. The *PolicyEnforcementPoint* forwards the request to the *PolicyAdministrator* which forwards it to the *PolicyEngine* for

evaluation. After evaluation by the *PolicyEngine*, the *PolicyAdministrator* component also prepares a configuration and manages the *PolicyEnforcementPoint* to accept or deny a request. When receiving the request the *PolicyEngine* evaluates whether the request complies with the policies. If yes, it proceeds to evaluate the context of the request. The three main aspects which form the base of a request context in a ZTA are the user authentication, the device authentication and the output of the trust algorithm. The processes behind user and device authentication and trust calculation may be performed by their own components. However since they all provide information about the context of a request, we can summarize them as instances of a *ContextProvider* component. After context evaluation has finished the *PolicyEngine* makes the decision and propagates back the decision to the *PolicyEnforcementPoint* through the same path it has reached the *PolicyEngine*. The *PolicyEnforcementPoint* then based on the decision by the *PolicyEngine* and configuration by the *PolicyAdministrator*, forwards the request to a protected resource or terminates the process if the request has been denied.

6 Modelling with Palladio

In the following chapter, we model the core concepts of a ZTA using Palladio. We first define *Composite Data Types* to model the data flowing in a ZTA. Then we model interfaces and basic ZTA components. The idea is to model the basic ZTA components so they can be reused later by developers when modelling systems that integrate the ZTA. The basic components encapsulate the main functionalities of a ZTA identified in Section 5.2, such as intercepting and forwarding a request, evaluating policies and context, and managing interceptors. Since we want to be able to reuse these components we try to model them as generally as possible while still preserving the specific functionality of each component. However, different standards and architectural examples introduce variations of the basic components which makes it hard to completely summarize the functionality of a basic component in a single model. Therefore, extension points are introduced into the model that capture the points of variation (POV). Moreover, these extension points allow developers to customize the basic components and adjust them better to their architectural needs.

As per the NIST ZTA standard document [48], apart from the basic components such as a *PE*, *PA* and *PEP*, there can be multiple other components which may exist in a ZTA. These components have the role of additional data sources which provide data to the *PE* for the decision-making process. They may also provide or aid in fulfilling processes in a ZTA such as authentication or logging and monitoring. The NIST document mentions an SIEM system, a PKI, Threat Intelligence, ID Management, a CDM system and Activity Logs as additional data sources in a ZTA. These components may be internal to the system or external, provided by a third party. As we see further from the SDP specification [13] and the BeyondCorp paper [55], not every one of these components is mandatory to be present in a system in order to achieve Zero Trust. However, some are still crucial for the principles of Zero Trust, such as an authenticator component. From the CISA [11] and Microsoft [34] Zero Trust Maturity Models, we observe that the presence or absence of components and technologies may change the maturity level of a company's Zero Trust integration. This also shows that if a certain component is not integrated it does not necessarily mean that a ZTA is not present. Therefore, we separate those components from the basic ones and dedicate a separate section for them. Another point for the separation is the fact that for each of those systems, different solutions exist on the market which vary in functionality. It is really hard to generalise those components and model them as universal plug-ins for the ZTA. Therefore, we are going to model a basic functionality for some of those systems which can be used to simulate the presence of a such system in a ZTA model. Developers may further extend that general functionality to match their needs or remodel such systems from the bottom to customize them according to their requirements completely.

After creating our repository, we are going to model components for the ZTA solutions BeyondCorp and SDP. With these models, we demonstrate basic combinations of the modelled

ZTA components to represent more complex ZTA solutions. Moreover, we create templates which can be later reused in systems where SDP or BeyondCorp may be applied.

Lastly, we apply our created components and templates to the JPlag running example. We create three different ZTA configurations to demonstrate the application of the model in a system and show the model's flexibility.

6.1 Composite Data Types

We define two composite data types in the repository - *Request* and *ContextData*. They are used to specify parameters passed between the components as well as return types of signatures. In a ZTA the main data which is passed between different components is a request. A request is sent by the user to the system and is then propagated throughout the system for evaluation. Therefore, we model it as a composite data type. In the *Request* data type we define the following boolean fields:

1. *Authorized* - this field describes whether the request has been authorized or not. The field is modified by the *PE* in case of evaluation. We can also use it to simulate authorized requests sent to the *PEP* by the user in the case of a firewall-like *PEP* (see Subsection 6.3.1).

2. *EvaluationEligible* - this field represents whether the request requires pre-processing before being sent to the *PDP*. In some cases, other functionalities may be triggered (e.g. authentication) before forwarding to the *PDP* (see Subsection 6.3.1).

3. *PolicyAuthorized* - this field represents whether the request complies with the policies. The field plays a significant role in the path a request may take in the evaluation process as well as the execution path which comes after the evaluation process. If a request is policy-authorized it will pass the first check in the *PE*, see Subsection 6.3.2. If a request is not policy-authorized it will be rejected by the system. Hence the execution path for accessing a resource which comes after the request evaluation will not be started. Therefore by using this field, a developer can control the frequency of requests being accepted at the first check in the *PE*.

4. *ContextAuthorized* - this field represents whether a request has been authorized according to its context. This is the second check which a request passes in the *PE*. When the value of the field is set to false, this would trigger a context evaluation process by the *PE*. According to the standard of Zero Trust by NIST context evaluation is mandatory. However, by providing the modeller with this field we provide them with more flexibility through the option to switch off the context evaluation process, in case of a system which may require only policies.

5. *UserAuthenticated* - this field represents whether the user of the request has passed the authentication process.

6. *DeviceAuthenticated* - this field represents whether the device which sends the request has passed the device authentication process.

7. *Trusted* - this field represents whether the request is trusted by the trust algorithm.

Based on the values of these fields, components modelled in this section change their functionality. For example, setting the *Trusted* field to true could result in a system not starting a trust calculation process and hence will not make calls to the components responsible for that. Therefore, the fields modelled in this composite type are meant to be used to control the execution paths which a request takes during simulation.

The next composite data type - *ContextData* - is used to represent data returned by context providers. It has only one field *ContextAuthorized* which represents whether the context provider has authorized the request according to its evaluation process.

6.2 Interfaces

We create interfaces in our ZTA repository. We use them later in component modelling to describe the required and provided roles of the components.

IRequest

We model the interface *IRequest* which has one signature *request(Request request)*. The signature takes one parameter of the composite data type *Request*. It returns also data of type *Request*. We create this interface since passing around a request is one of the main communications which happen in a ZTA. Since the *Request* data type has fields which are altered during execution, the idea is to pass around a request with this interface and then act according to the altered fields from the returned data. For example, a *PEP* uses the interface to forward a request to a *PE* and then acts according to the *Authorized* attribute of the returned data.

IPolicies

The next interface we model is the *IPolicies*. It has a single signature *providePolicies()* which takes no parameter and has a void return type. In ZTA, policy evaluation is one of the core tasks, as we discussed in Section 5.2. To do this, a component is required to obtain policies and with this interface, we can describe this required role of a component. Furthermore, components which are responsible for loading and supplying policies should provide this interface.

IContext

We model the interface *IContext*. It has a single signature *provideContext(Request request)* which takes one parameter of type *Request*. The signature has a return type of *ContextData*. Providing context data and evaluating context are other core tasks of a ZTA. Therefore, there are components which perform these actions and to describe their roles of a *ContextProvider* we use this interface. We pass the *Request* as a parameter so context providers can act based on the fields for which they supply data or provide context evaluation. With the return type *ContextData* we describe what the outcome of the context evaluation is.

IManage

The next interface we introduce is *IManage*. The interface has a single signature named *manage()*. It takes no parameters and has a void return type. In a ZTA, rules and configurations

of *PEP* are being constantly updated. To enable this type of communication we propose this interface. Components which need to be configured should require this interface. Respectively, a component which is responsible for configuring rules and deploying them provides the *IManage* interface.

IObligation

We propose the interface *IObligation* with the single signature named *performObligations()*. The signature has no parameters and a void return type. Romain Laborder et al. [30, 29] specify the behaviour of enforcing obligations. They propose this behaviour in the context of the *PEP*. Apart from the decision, the *PEP* receives obligations that it needs to perform. Examples of obligations could be logging or storing configuration. To satisfy the obligations, the *PEP* can be plugged with additional components such as a logging component. When receiving the obligation of logging, for example, the *PEP* calls the service of the logging component. We want to use this concept of obligations to provide extension points in our components. We discuss the need for such extension points in more detail later in this chapter. The idea is for components to require this interface when we want to make them extendable and customizable in their behaviour.

ICredentials

The *ICredentials* interface has a single signature *provideCredentials* which has a void return type and has no parameters. In the process of authentication, an authenticating component needs to obtain the subject's credentials to compare them with the stored ones. This can be done using the *ICredentials* interface. Respectively, components with which we model the process of inputting or generating credentials should provide this interface.

IStoreAccess

We propose also the interface *IStoreAccess*. The interface has two signatures - *store()* and *retrieve()*. Both of the functions do not require parameters and have a void return type. In a ZTA we may need to store some kind of data such as logs, for example, in a storage. To communicate with a store, we use the *IStoreAccess*. The names of the signatures are also a description of their purpose - *store* is for storing data and *retrieve* is for retrieving data from a store.

ILog

We model the interface *ILog*. The interface has a single signature *log()* which takes no parameters and has a void return type. Logging is an important part of the Zero Trust request evaluation process. Components responsible for logging data in a ZTA should provide this role. If components need to perform some logging then a call to this interface's signature should be included in the execution flow.

IResource

Lastly, we propose the *IResource* interface. It has one signature named *accessResource(Request*

request) which takes a *Request* type parameter. Components which serve as a gateway to a protected resource should require this interface to communicate with resources. Respectively, components which represent the assets protected by the ZTA should provide this interface.

6.3 Basic Components

The following section describes the components which form the core of a ZTA. These components should be used to model the foundations of a system that integrates Zero Trust principles. We discuss possible variations in the context of *PEP* and model two variations of the component. The checking of policies and context is captured in a *PolicyEngine* component. Turkmen and Crispo [51] present some implementations of policy evaluators and we can identify two main phases of the process. They are the policy loading and policy evaluation. Following the separation of privileges principle, we model the process similarly. We delegate the tasks of loading policies to a *PoliciesProvider* components and the evaluation, as mentioned in previous sentences, to the *PolicyEngine*. We also model a *PolicyAdministrator* component. It is mentioned in [48] that the *PolicyAdministrator* can be directly combined with the *PolicyEngine* in a single PDP component. Further suggestions of ZTA support this claim. For example, in the SDP architecture [13], the controller makes the decision and sends configurations to the *PEP*s. However, separating the *PolicyAdministrator* from the *PolicyEngine* enforces better separation of privileges. Furthermore, it allows the developer to specify systems where the *PolicyEngine* and the *PolicyAdministrator* reside on different hardware.

6.3.1 Policy Enforcement Point

The *PEP* is the component which intercepts the request from the client and forwards it for further evaluation to the *PA* as well as monitors, establishes and terminates connections to a resource, according to the NIST standard [48]. In [14], Carlos Da Silva, Welkson Medeiros, and Silvio Sampaio implement the *PEP* as a service request interceptor. It forwards then the request to the PDP. After receiving the decision from the *PDP*, the PEP either forwards the request to the service it protects or denies the connection. Romain Laborde et al [30, 29] define the *PEP* in their works with similar functionality. Although this general pattern of functionality is similar across different *PEP* models there exist POVs which we consider when creating the final model of a *PEP*.

Discussion on Points of Variation

Request Translation - In [14], the *PEP* collects information such as requesting user or URL of access and sends it to the PDP. In [30, 29], the PEP is again mentioned as a component which translates application-specific requests to requests compatible with the language of the PDP. This is a critical point for legacy systems which are looking to integrate the Zero Trust principles into their workflow. An application-specific request may contain in its data the whole information required by a PDP and in this case, a simple data extraction may be enough. However, it is possible that the *PEP* needs to make additional calls to other components if the required data for translation is not present.

Communication Behaviour - Furthermore, after analysing the NIST standard[48], Beyondcorp paper[55] and the SDP specification[13], we extract another varying functionality of a PEP in the communication behaviour. The NIST [48] and Beyondcorp's [55] *PEPs* forward requests to the *PE* and accept rule updates. In the case of the SDP [13], the *PEP* can be summarized as a simple firewall, which only receives rule updates from the controller, without forwarding requests to it, and drops every unrecognized request. When we observe the rule updates the communication is similar in all of the cases. However, when it comes to request processing, the communication differs. In one case the communication channel is duplex where *PEP* communicates with the *PDP* to forward a request and receive updates. In the other case, the *PEP* only receives messages from the *PEP* and therefore the communication channel can be categorized as simplex.

Additional Functionalities - The next POV which can be identified from the descriptions are the additional functionalities of a *PEP*. In the Beyondcorp architecture [55], apart from the standard functionalities the PEP also communicates with an authenticating entity to initiate authentication in the event of an unrecognized request. The NIST standard [48] defines the *PEP* also as a communication monitor. Such tasks could be triggered at different steps of the request handling process of the *PEP*. For example, on a dropped request, authentication is triggered or on an accepted request the decision and received configuration are logged. Therefore, we need to enable the developer to customize the additional tasks performed by a *PEP*.

Modelling Decisions for discussed Points of Variation

Request Translation - Since translating application-specific requests to *PE*-compatible requests may vary between different systems, we are not including it in the general model of a *PEP*. We suggest that developers define their own translation component which requires and provides the *IRequest* interface and specify in that component the overhead required for request translation. As for the general model of a *PEP*, there might still be little overhead for request processing which we can model as an internal action.

Communication Behaviour - First, we separate the communications of *PEP* into a communication channel for request forwarding and a communication channel for receiving rule updates. We do this because the request processing communication differs and we want to capture both of the variations there. The rule updates channel is similar in all *PEP* and we will use a single modelling approach for it. We model two basic components for the *PEP*. One of the components is a *SimplexPEP* and represents one-way communication in the context of request processing. The second component we model is a *DuplexPEP*.

Additional Functionalities - To enable our model to represent as many as possible additional functionalities we will include extension points in the SEFFs. For this purpose, we are going the use the concept of obligations, as described in the *IObligation* interface definition from Section 6.2. Therefore, we include calls to the *IObligation* interface where we want to model varying functionality.

SimplexPEP

The *Simplex PEP* provides the *IRequest* interface so it can intercept access requests and the *IManage* interface so it can be configured by a *PA*. The component also has two required roles for the *IObligation* interface to provide extension points on the events of accepting or denying

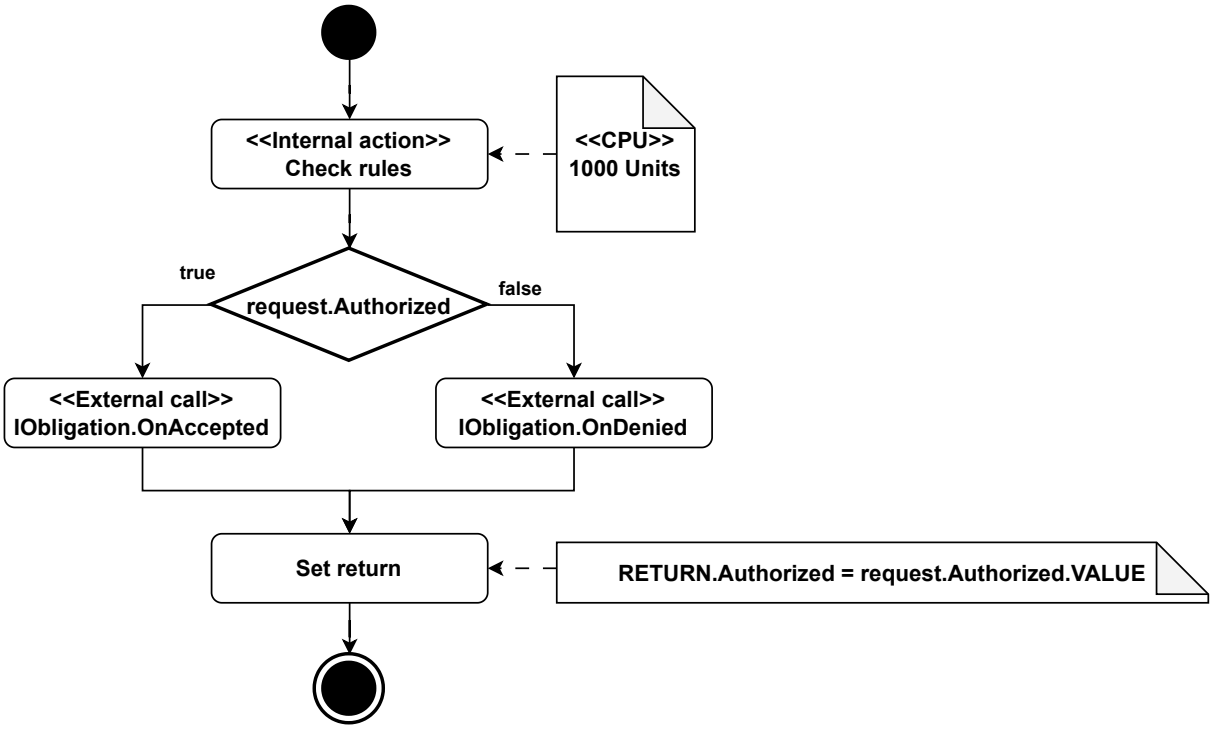


Figure 6.1: *SimplexPEP* SEFF of *request(Request request)*

a request. The component provides a simple firewall-like functionality. We provide the SEFF of the *request* function of the *IRequest* interface in Figure 6.1. When the function is called it simulates the task of checking its rules with an internal action. Then, based on the *Authorized* field of the *Request* parameter it triggers either the *OnAccepted* or *OnDenied* obligation through an external call to the respective*IObligation* required role. In the end, the component returns the value of the *Authorized* field of the input parameter without altering it.

The *manage* function of the *IManage* provided role simply executes an internal action to simulate the process of updating internal rules. Since there is not any complex logic present in its SEFF we do not provide an activity diagram for it.

DuplexPEP

The *DuplexPEP* provides the *IRequest* interface and the *IManage* interface. It again requires two *IObligation* interfaces for events of accepting and denying a request. The component requires additionally the *IRequest* interface to be able to forward requests to a PDP another instance of the *IObligatoin* interface for including an extension point when a request requires pre-processing before being forwarded to a *PDP*. In Figure 6.2, we show the SEFF of the *request* signature of the *IRequest* provided role. When it receives a request, it first simulates the overhead of translating a request through an internal action. Then it decides to accept the request for evaluation or drop it based on the *EvaluationEligible* field of the *Request* parameter. This is how we model the *Additional Functionalities* POV. If a request is not suitable for evaluation it triggers the *OnRequestIneligible* obligation, where a developer can plug in a component providing additional functionality. On accepting a request for evaluation, it forwards the request to a *PDP* by calling

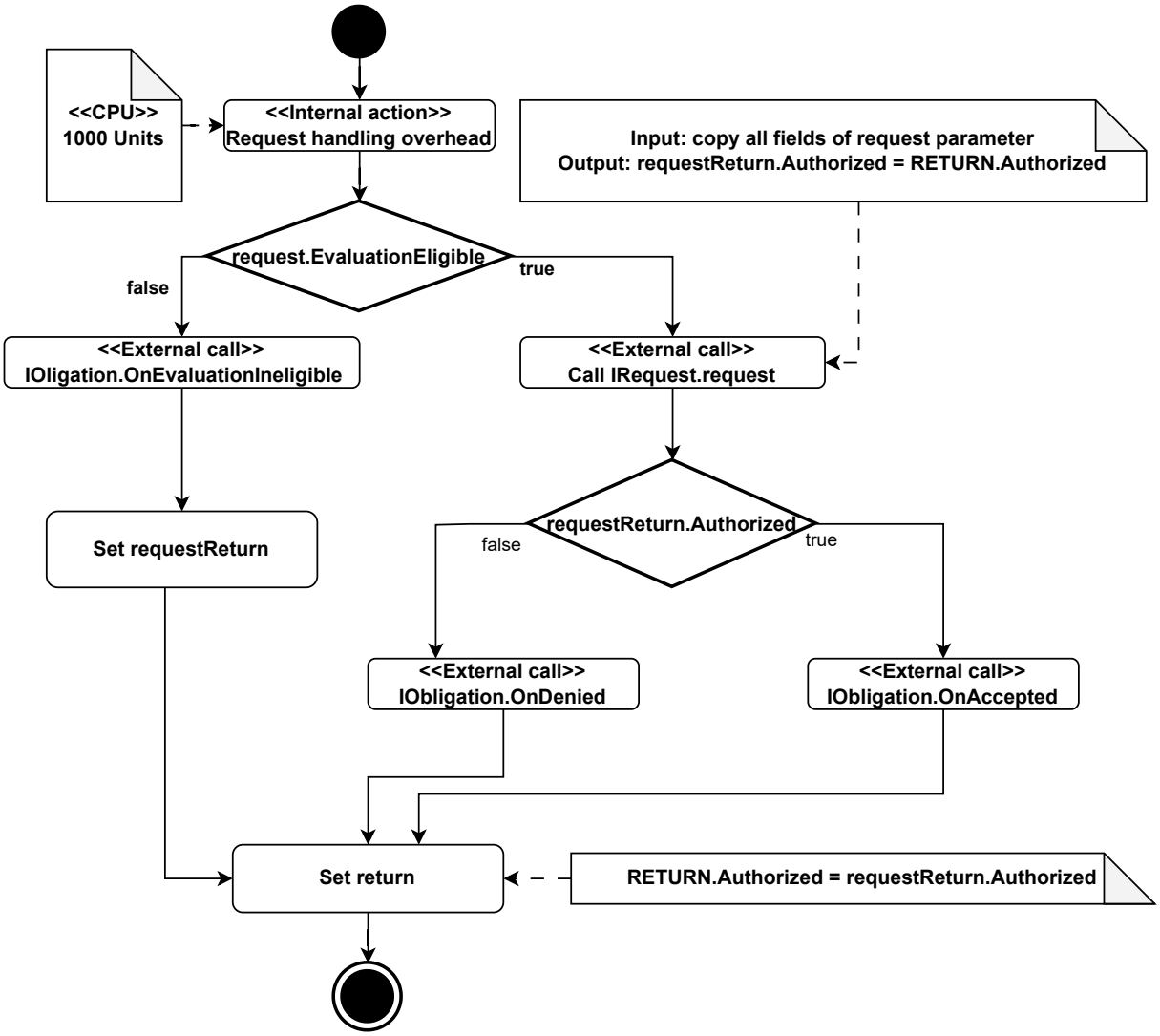


Figure 6.2: Duplex PEP SEFF *request(Request request)*

the *IRequest* interface. This external call returns a *Request* response, stored in the *requestReturn* variable, which contains the decision about the request in the *Authorized* field. Based on the value of this field, the *PEP* triggers respectively the *OnAccepted* or *OnDenied* obligation. Before finishing execution, the *PEP* sets the *Authorized* field of its *RETURN* variable to the value of the *Authorized* field from the *requestReturn*.

Customizing a PEP with multiple obligations

We want to allow the model of the *PEP* to support multiple obligations. However, it is not possible to plug in multiple components to the same required interface and call different implementations of the same interface in a loop. We propose the following way to combine multiple obligations:

1. The component developer specifies a component *ObligationAggregator* which provides *IObligation* and requires it multiple times.

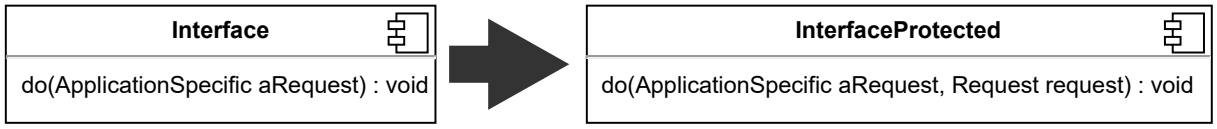


Figure 6.3: Copying an interface

2. Multiple *IObligation* components can be plugged into the requiring ends.
3. In the SEFF of the *ObligationAggregator*, the developer specifies how the services are being called (sequential, parallel, etc.).
4. The *ObligatoinAggregator* is plugged to the PEP via the *IObligation* interface.

The same pattern can be followed for every other extension point of the other components.

Gateways

In Palladio, we cannot directly copy an input parameter in a SEFF and forward it to another external call. We need to specify explicitly each field of an input parameter to an external call. Different resources require different parameters and it is impossible to know in advance every possible field which needs to be forwarded. This limits our ability to create a generic *PEP* which, apart from the request parameter, accepts an application-specific request parameter and forwards it to the resource it is protecting using a generic interface such as the *IResource*. To tackle this limitation we can use gateways we propose the following template for modelling and general functionality. A gateway should provide an interface which should mimic the signatures of the protected by the *PEP* interface and should extend them with an additional parameter of the *Request* type. We demonstrate this copying of interface in Figure 6.3. The gateway component provides this protected version of the interface and requires the unprotected interface as well as the *IRequest* interface, as shown in Figure 6.4. As a result, the gateway receives two parameters when called - one is the application-specific request and the other is a *Request*, as defined in Section 6.1. In Figure 6.5 we propose a generic SEFF for such gateways. The gateway first forwards the *Request* parameter to the *PEP* through the *IRequest* interface to start a request evaluation. As a response, it receives a *Request* data containing the decision. Then, according to the *Authorized* field, if the value is true, the gateway calls the interface it is protecting and forwards the application-specific parameters.

In our ZTA repository, we model a *GenericGateway*, as shown Figure 6.9, which provides and requires the *IResource* interface. Additionally, the gateway also requires the *IRequest* interface to forward a request to a *PEP*. The component has the general functionality described in this paragraph. The component serves as an example in the repository of how component developers should define a gateway with its SEFF and variable usages. There is still the possibility to use the generic component directly in projects which integrate a ZTA from the beginning and use the *IResource* to provide protected resources.

6.3.2 Policy Engine

The *PE* is the component that makes the ultimate decision to deny or accept a request based on policies and context information about the request [48]. Therefore, we introduce the basic

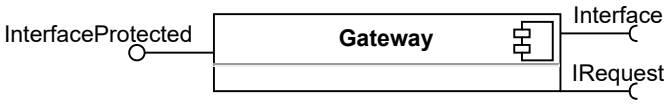


Figure 6.4: Gateway component example

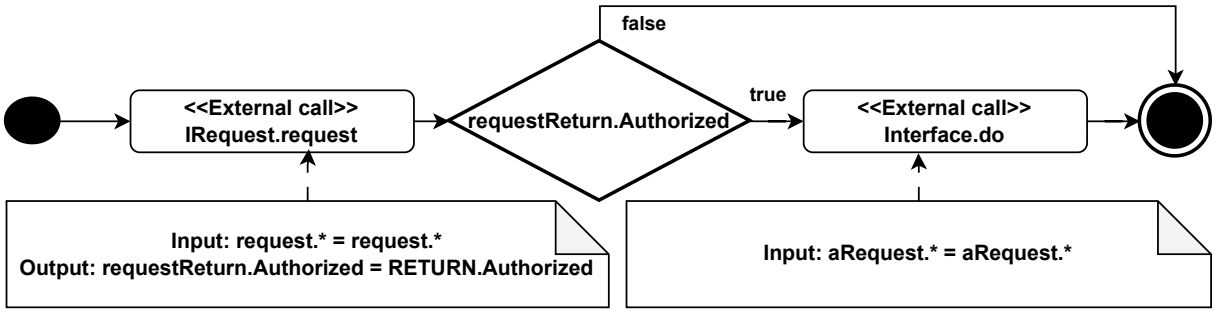


Figure 6.5: Gateway general functionality

component *PolicyEngine*, shown in Figure 6.9, for evaluating policies. The *PolicyEngine* provides the *IRequest* interface so it can receive a request for evaluation and decision-making. To make the decision the component requires the *IPolicies* interface, to obtain policies for evaluation and the *IContext* interface to acquire information about context evaluation.

The SEFF of the *request(Request request)* functions of the provided *IRequest* roles is shown in Figure 6.6. When it receives a request the *PolicyEngine* evaluates first whether the request complies with policies. This process is modelled with an external call to the *IPolicies* interface to trigger the policies loading process. Then with an external action, we simulate the process of checking policies. The actual decision of whether a request complies with policies is present in the *PolicyAuthorized* field of the *request* parameter. Based on this field of the request, the *PolicyEngine* either terminates the process or continues to evaluate the context of a request. The modeller is also provided with the option to switch off the context evaluation by setting the *ContextAuthorized* field of a request to *true* in the usage model. This would cause the execution to jump directly to the set return variable action and modify the *Authorized* field of the return variable based only on the *PolicyAutohrized* field which in this case should be *true*. If the request's context should be evaluated then the *PolicyEngine* will start the process with an external call to its required *IContext* interface. The call returns a result of the context evaluation in the *requestReturn* variable. At the end of the execution, it will set the *Authorized* field of the return variable based on the *PolicyAuthorized* of the *reuqest* parameter and *ContextAuthorized* field of the *requestReturn* variable. At the end of the *PolicyExecution* execution flow we provide an extension point by making a call to the *IObligations* interface.

Similar to the obligations in paragraph 6.3.1, multiple context providers might be used in a context evaluation process. However, since it is implementation dependent, we do not know in advance the number of all used context providers and we cannot specify in the general SEFF of the *PolicyEngine* all of the possible combinations. Hence, we propose the following way of plugging multiple context providers to the *PolicyEngine*:

1. The component developer creates a component named *ContextAggregator*, which provides *IContext* and requires it multiple times equal to the used context providers.

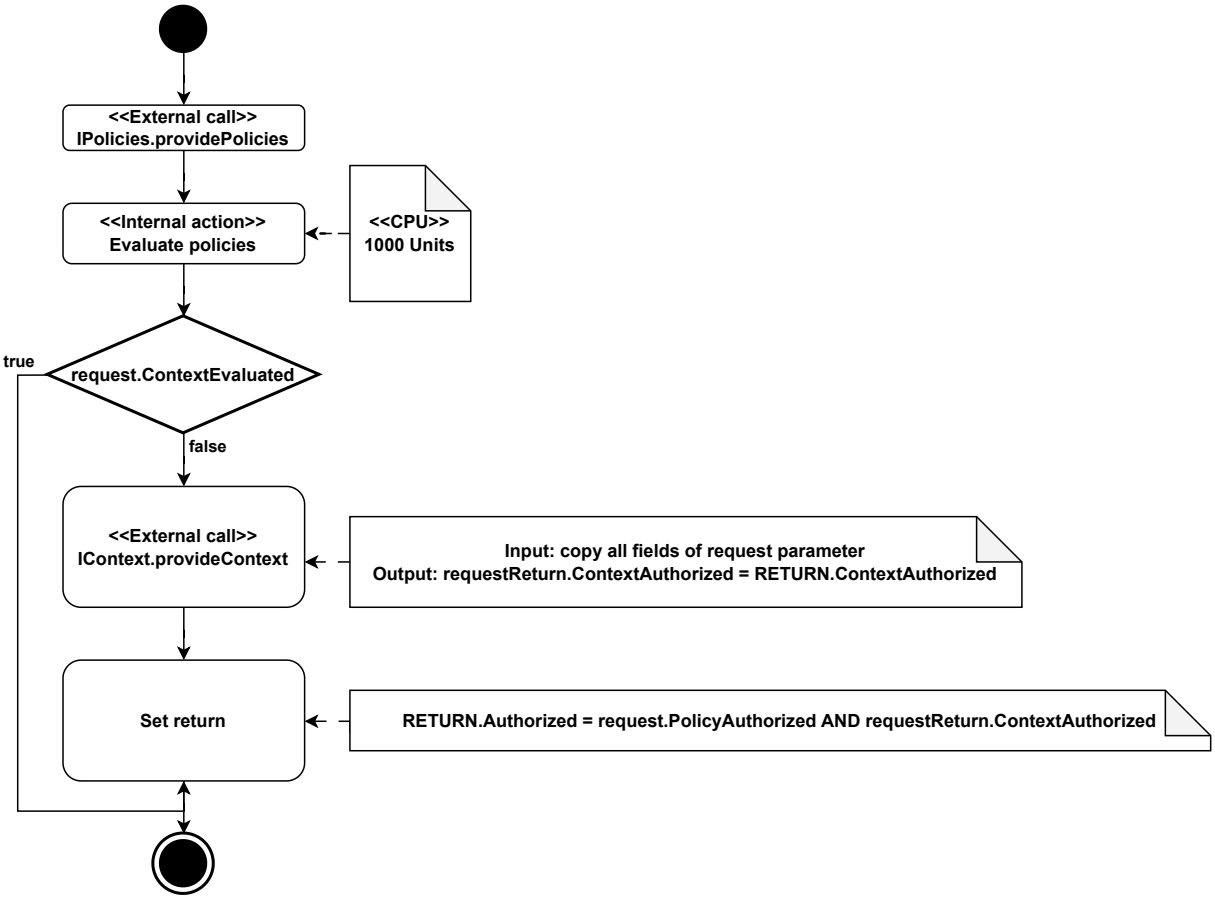


Figure 6.6: Policy Engine SEFF *request(Request request)*

2. In the SEFF of the *ContextAggregator*, the developer specifies how the different providers are called (sequential, parallel, etc.).

3. The *ContextAggregator* is plugged to the *PolicyEngine*.

PoliciesProvider

A *PE* needs policies in order to operate. Therefore we specify the basic component *PoliciesProvider*, as shown in Figure 6.9, which is responsible for loading the policies. By doing this we also allow the specification of different policy providers, since policies might originate from external providers [51] (e.g. industry compliance policies [48]). Furthermore, the main workload generator in the policy evaluation process is the loading of policies. Algorithms for more efficient loading are being researched [51, 16]. By externalizing the policy loading module, we can allow developers to experiment with the policy provider model using different algorithm models and further refine the policy loading process. The *PoliciesProvider* component provides the *IPolicies* interface. The component aims to simulate a simple policy loading process. The resource requirements and performance of such components are strongly dependent on the number of policies and the number of rules contained in a single policy [16]. Therefore, we model the two variables *numberOfPolicies and rulesPerPolicy* as component parameters. The

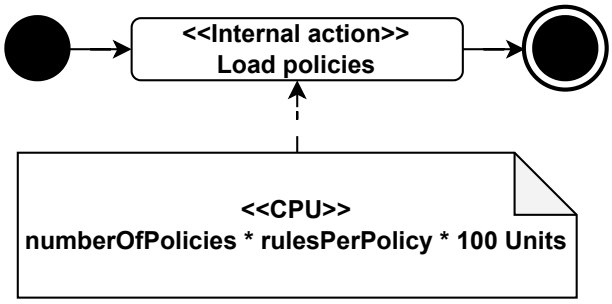


Figure 6.7: Policies Provider SEFF *providePolicies()*

variables influence directly the processing time of the internal action of evaluating policies, as shown in the SEFF diagram of the component Figure 6.7. These variables should be set when instantiating the component in an assembly model of a system.

6.3.3 Policy Administrator

The *PA* is responsible for configuring the connection between the subject and the resource based on the decision provided by the *PE* [48]. It does so by creating configurations, generating session-specific tokens and instructing the PEP which actions to perform and how to respond to the user's request. The *PolicyAdministrator* requires the *IRequest* interface to forward the request to the *PE* for decision-making. The component also provides the interface *IRequest*, as shown in Figure 6.9. When the execution of the *request* function of the provided role is called, the component first makes a call to its required *IRequest* interface. The response of the external call is saved in the *requestReturn* variable. It then generates a configuration based on the decision of the *PolicyEngine*. This process, however, is implementation-specific since generating a configuration might include generating session tokens or creating firewall rules [53]. We abstract it as an internal action, as shown in Figure 6.8. Then the component performs a call to the *IManage* interface to simulate the process of managing a *PEP*. We model an extension point at this part of the SEFF by making a call to the *IObligation* component. The *PolicyAdminitrator* finishes its execution by setting the *Authorized* field of its return variable to the *Authorized* field of the *requestReturn* variable.

Managing Multiple PEPs

There are cases where a *PolicyAdministrator* should manage more than a single PEP. For example, in an SDP[13] architecture, the Controller configures multiple Accepting Hosts. The configuring process of a *PEP* is expressed in the SEFF of the *PolicyAdministrator* with the external call to the *manage* signature of the *IManage* interface, as shown in Figure 6.8. However, we can only plug a single *IManage* provided role to the required role of the *PolicyAdministrator* component. To cope with this problem we suggest using a similar modelling pattern to the one of modelling multiple context providers, Subsection 6.3.2. Therefore we define the component *PEPsManager*. The component provides *IManage* interface and has as many *IManage* required roles as PEPs it has to manage. We model a starting case of managing two PEPs, so we have two required *IManage* interfaces. The component can be extended to support more PEPs by

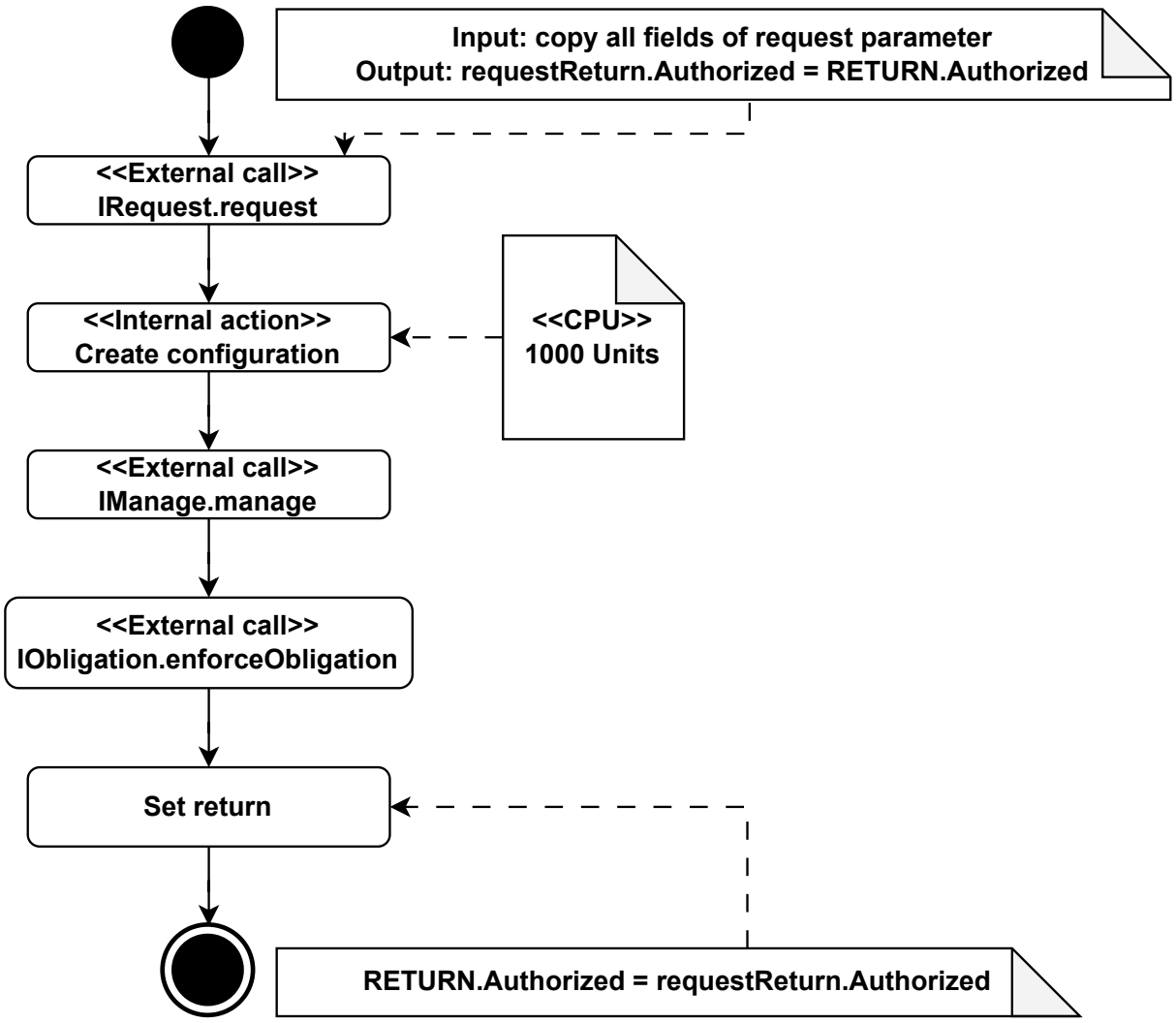


Figure 6.8: Policy Administrator SEFF *request(Request request)*

adding more required roles to it. In its SEFF, the component simply calls all of its required roles in a sequential order and performs no internal actions.

6.4 Context Providers

In the following section, we discuss important Context Providers. As we mentioned previously in the chapter, Context Providers are systems or components for which different solutions exist on the market. These systems are not a strict part of a Zero Trust environment and are also used outside of the context of a ZTA. However, when integrated into a ZTA they enhance its functionalities and help in covering the Zero Trust principles. Although these systems originate from their own research fields and may require a broader analysis and even separate projects in order to model all of their specialities, we generalize as much as possible about their functionality in the following section. We model templates for these systems which can be combined with the proposed in the previous section basic components to achieve a ZTA. Developers, however, can still opt to model these systems on their own when combining them with our basic components. In this section, we provide modelling solutions for a multi-factor user authentication system, a device authentication system, an events logging system, data storage and a trust algorithm.

6.4.1 Authentication

According to a review of authentication systems [4], there are multiple ways in which an entity can be authenticated. Authentication can be based on possession (smart cards, hardware tokens, etc.), knowledge (password, pin, etc.) or biometrics (fingerprint, voice, etc.). These methods could be used separately or combined in various ways in multi-factor authentication systems. Furthermore, authentication might happen at different places in a ZTA. For example, in the SDP model [13], the authentication happens in a communication between the client and the SDP controller, which has the role of a *PDP*. In the Google approach [55], the user has to authenticate themselves with the help of a separate authentication component before communicating with *PEPs*. As we see, a common authentication approach does not exist as standard which tells us how exactly we should apply authentication in a ZTA. However, if we observe the authentication process on an abstract level, we can identify its core activities. They are the supply of credentials by the entity being authenticated, the acquisition of credentials stored for this entity and the comparison of both credentials. In a suggested authentication system for ZTA [15], we observe that the user sends a token, which is compared to a token stored in an Identity Access Management system. Similarly, in the SDP specification [13], the system issues credentials, which are stored in a database and later compared to the credentials provided by a requesting entity. In an authentication method based on Windows Active Directory [27], we identify in its core again the process of collecting user credentials and comparing them to stored ones in the Active Directory. Following this core process, we model the authentication in the following way.

We introduce a basic component *Authenticator*, shown in Figure 6.12, which is responsible for comparing the provided by the user and by the system credentials. The component provides the *IContext* interface. The *Authenticator* requires the interfaces *IStoreAccess* to obtain credentials

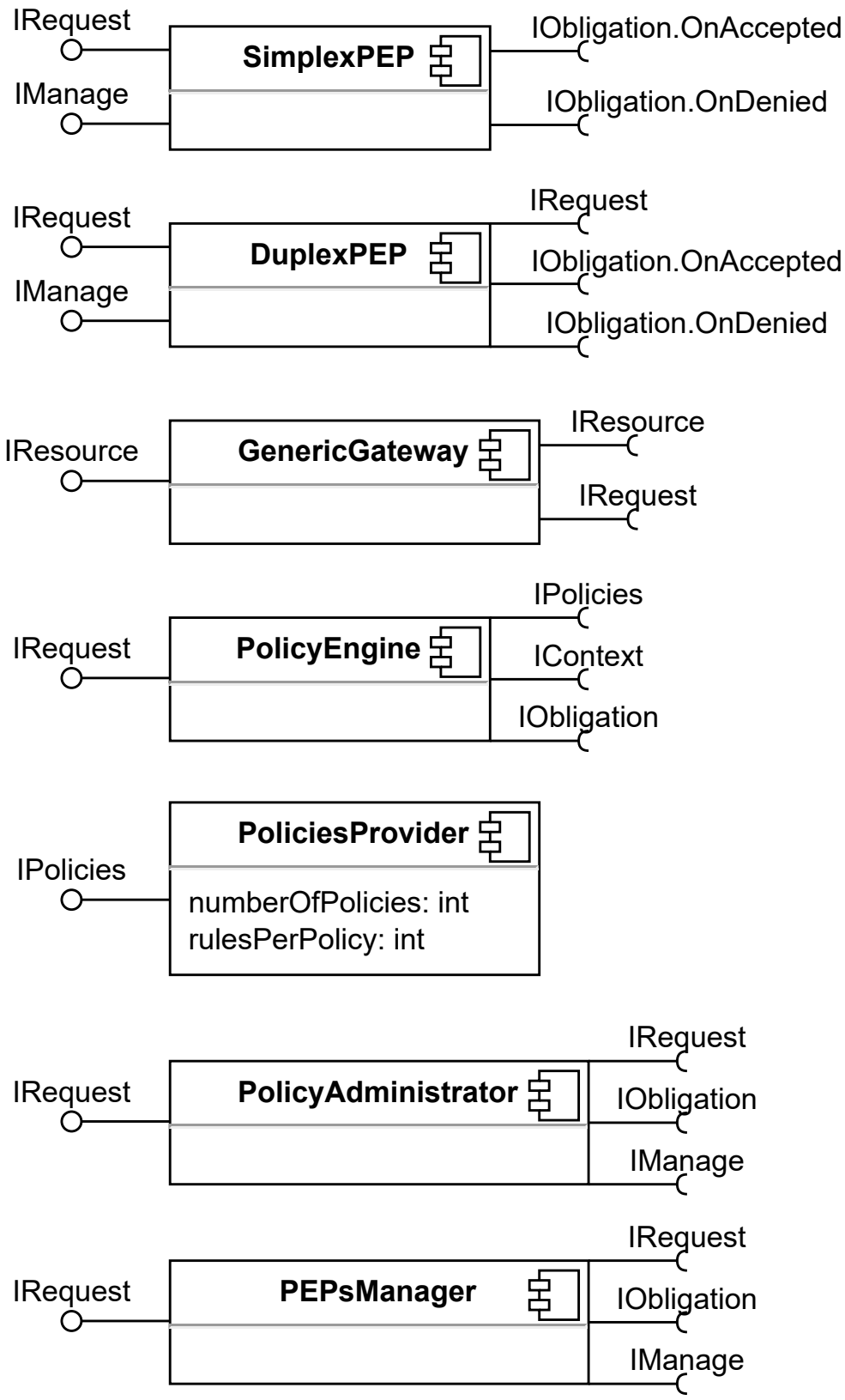


Figure 6.9: Basic components

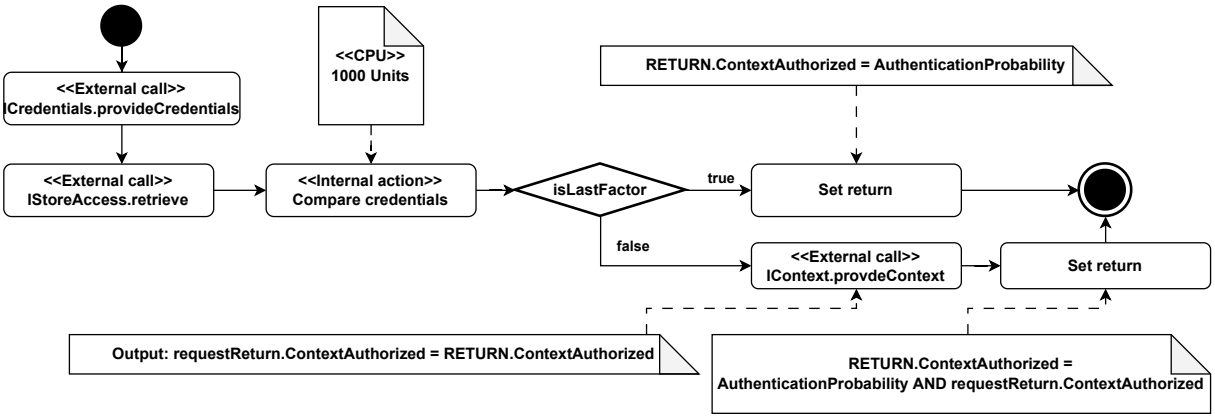


Figure 6.10: Authenticator SEFF *provideContext(Request request)*

from an Identity Store and requires the *ICredentials* interface to obtain credentials from the user. A *CredentialsProvider* component should provide the *ICredentials* interface and this allows the developer to specify multiple different credentials (passwords, fingerprints, tokens) as well as model the time and resources required to supply them in their SEFF. Furthermore, this type of structure would allow the authentication process to be triggered from anywhere in the system since it is not required to propagate credentials alongside the request. The system may call the provided by the *Authenticator provideContext* service and the *Authenticator* will require the user to supply credentials through a *CredentialsProvider*.

To allow for modelling Multi-Factor-Authentication (MFA), the *Authenticator* component has the *isLastFactor* component parameter and requires the *IContext* interface. When the parameter is set to false, the component calls the authentication service of the next plugged-in *Authenticator*. This allows the developer to chain multiple *Authenticators* and form an MFA authentication system. In the end, the component returns whether the user is authenticated or not based on another component parameter *AuthenticationProbability* and the result of the next in-chain authentication factor. The *AuthenitcationProbability* parameter is specified in an assembly model when instantiating the *Authenticator* component. To ensure that the result of such an MFA system is only true when all of the factors have returned true we use the logical AND operation to compare the output of the current component and the result produced by the next-in-chain *Authenticator*. We suggest that the *AuthenticationProbability* variable is set as a boolean Probability Mass Function (PMF). Then, tuning the probability of true and false we can simulate different scenarios with more or less successful authentications.

6.4.2 Device Authentication

Device authentication happens by comparing attributes of a device such as the version of the OS, enabled security features, installed software, etc. with policies which describe what should be the state of a device. In a ZTA we have two options to obtain the information about the device. The first one is to have a database that stores device fingerprints which are constantly updated there. Upon authentication, we retrieve the device information from there. The second option is to request the device to provide a fingerprint each time an authentication is happening. It is debatable whether the second option is suitable from a security point of view since devices

might lie about their fingerprint. Since it provides more flexibility for modelling systems which have not yet integrated a device managing system in their architecture, we model both of the options for authenticating a device.

We propose two types of the *DeviceAuthenitcator* component, as shown in Figure 6.12. First, we have the *DeviceAuthenticatorStore* which provides the *IContext* interface and requires the *IStoreAccess* interface. The second component *DeviceAuthenticatorLogs* is different in the aspect that it requires the *ILog* interface. Both of the components perform an external call to their required services after execution starts and then perform an internal action of authenticating the device. In the end, the authentication result is set according to the component parameter *AuthenitcationProbability* value. This component parameter is set when the component is instantiated in an assembly model of a system. The developer is encouraged to use a boolean PMF for this variable so they can tune the rate of successful device authentications.

6.4.3 Store

In a ZTA there is data such as a subject's identity or logs of events and interactions. This is important information which needs to be stored. However, the data is not stored in a single format. As an example, we can store logs as files on a hard drive and identities might be stored in a relational database. To cope with this, the *Store* component is introduced. The *Store* component abstracts the possible ways of storing data in a ZTA. It could represent a database or a hard drive where files are stored. The component provides the *IStoreAccess* interface.

6.4.4 Logging of Events and Data

Logging interactions and events is another concept of a ZTA [48]. A *PE* might log the decisions it has made, and a *PEP* might log the enforcement of these decisions and the obligations which were applied parallel with a decision. Further examples of logging could be keeping track of user interactions with the system to monitor user behaviour as well as logging security events of devices to monitor their security posture. Keeping track of things happening in the system helps in identifying anomalous behaviour, and detecting early security threats. It additionally supports the non-repudiation security goal.

If we observe the core functionality of the logging process, we see that it is nothing more than storing data about a component's action or state in some kind of store (file, database, etc.). On a high abstract level, a basic logging component can be observed as a store. It stores logs which can be retrieved on demand. However, logs in their raw format may be difficult to analyze and correlate. Moreover, manually expecting logs and deriving important information is a time and resource-consuming task. The process of logging and processing of system events and interactions can be modelled in a whole separate system which can then be included in the ZTA. These types of systems are called Security Information and Event Management (SIEM) systems. Currently, on the market exist multiple SIEM solutions [21]. Although these solutions vary in the functionalities they offer or in the extent to which certain functionalities are implemented, we can still extract some common components which are present in the systems and summarize a generic SIEM system. In general, the main roles of a SIEM are to collect, store and analyse logs as well as provide reports and alerts based on its analysis [21, 54, 2]. In the architecture models suggested in [54, 2] there are components for collecting and

storing the logs, components for normalizing and analyzing and components for presenting the data for further use. In [54], the authors specify additional components about enriching logs as well as for different types of result representation. However, in order to keep the overall ZTA model simple we are not going to go into full detail when specifying the SIEM system. We specify only three components that summarise a generic SIEM's functionality and then combine them into a composite component.

Firstly, we define a *LogCollector* component, as shown in Figure 6.12, which requires the interfaces *IStoreAcces* and *ILog* and provides the *IObligation* and *ILog* interfaces. The role of the component is to collect logs through both of its providing interfaces. The *IObligation* interface can be called, for example, by the *PEP* after receiving a decision from the *PDP*, see paragraph 6.3.1. The *ILog* interface can be called by another component which directly sends logs. In both of the SEFFs of the *IObligation* and *ILog*, the *LogCollector* simply performs a call to the *IStoreAccess* interface to store the logs. Next, we define a *LogNormalizer*, as shown in Figure 6.12, since logs may originate from different sources and we simulate converting those logs to a single format, ready for analysis. Lastly, we define the *LogAnalyzer*, as shown in Figure 6.12, which analyses logs and presents the results to the ZTA as context information through the *IContext* interface. We then combine these three components in a composite component to form a *SIEM* component which can be plugged into a ZTA. Keep in mind that this component represents a generic SIEM system and a developer is free to further specify additional components and functionality in order to model a more detailed SIEM system.

6.4.5 Trust Algorithm

The trust algorithm is described as the "thought process" of a ZTA according to the NIST standard document [48]. It has a significant role in improving the evaluation process of a request since it aggregates and processes multiple sources of information about the request and outputs a trust score based on which more fine-grained access control can be performed. Therefore we want to include the trust algorithm in our set of ZTA components. However, generalizing a trust algorithm is not an easy task. In the NIST document[48] trust algorithms are classified as score or criteria-based and singular or contextual. Score-based algorithms assign weights to inputs and output a quantity metric about the trust whereas criteria-based algorithms evaluate a request according to rules. The trust inference in BeyondCorp[55], for example, evaluates data about the user and device according to heuristics and rules in order to reduce or increase trust. The difference between a singular and contextual algorithm is the scope of requests the algorithm is observing when evaluating the trust for the current request. In the case of a singular algorithm, it only evaluates the current request whereas contextual algorithms also consider previous requests by the same user. Trying to represent all of these variations of a trust algorithm in our model may cross the boundaries of implementation-specific details. What we are interested in here is the most basic concept of the trust algorithm. Observed abstractly, the trust algorithm simply performs calls to multiple input sources before performing the trust evaluation and producing an output. Therefore, this is what we are going to represent in the model of a general trust algorithm. We cannot predict how many input sources a system might use for its trust algorithm. We will model the general component with just two sources and describe briefly how it can be extended to support more. As shown in Figure 6.12, we create the component *TrustAlgorithm* which requires two *IContext* interfaces for

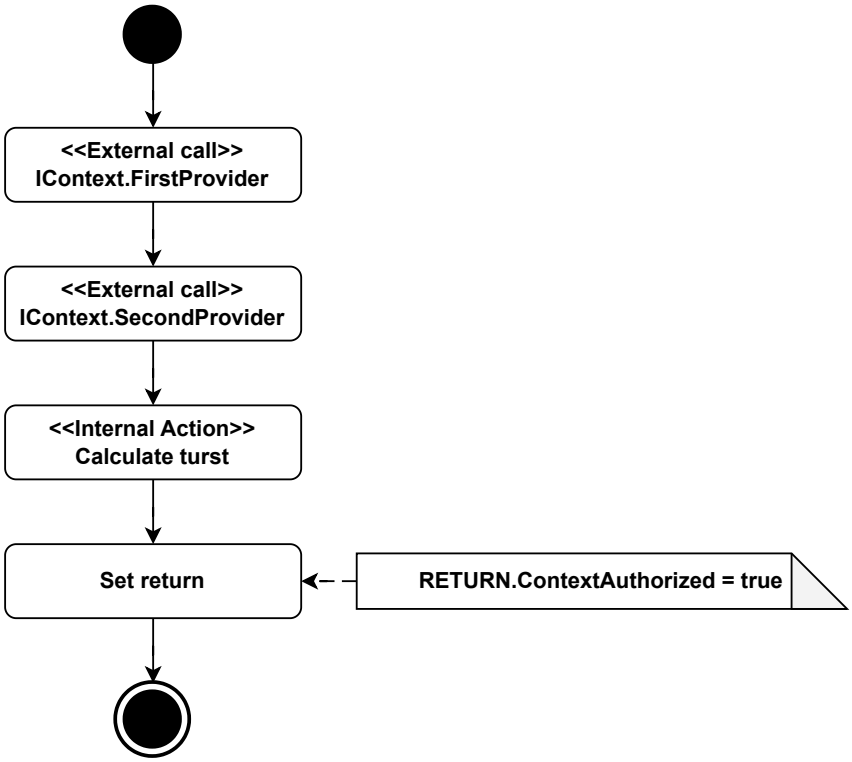


Figure 6.11: TrustAlgorithm provideContext(Request request) SEFF

the two input sources. We can classify the trust algorithm also as a context provider according to our meta-model, see Section 5.3 since it supplies data about the context of a request to the *PE* for the evaluation process. Hence, the component provides the *IContext* interface. In the SEFF diagram of the component, Figure 6.11, we model the actions as described previously in this section. First, we call sequentially both information providers before performing the internal action of calculating trust. Lastly, the component sets the *ContextAuthorized* field of its return variable of type *ContextData* to true. As we said we cannot cover all possible input sources of a trust algorithm and therefore we offer the option to component developers to extend the current model in case they need more sources. The developer should simply add additional required roles for the *IContext* interface and then include them in the SEFF as external calls between the call to the second provider and the trust calculation internal action.

6.4.6 Context Evaluator

We have already specified models of user and device authenticating systems as well as a simple trust algorithm. We can use these components to form the context of a request and include it in the request evaluation process presented in the *PolicyEngine*, Subsection 6.3.2. To combine these models in the process of context evaluation we propose a *ContextEvaluator* component. The component provides the *IContext* interface and requires three times the same interface - one for the user authenticator, one for the device authenticator and one for the trust algorithm. In the end, it evaluates the context based on the returned decision by each component. The modeller here has again the option, as shown in the SEFF of the component Figure 6.13, to

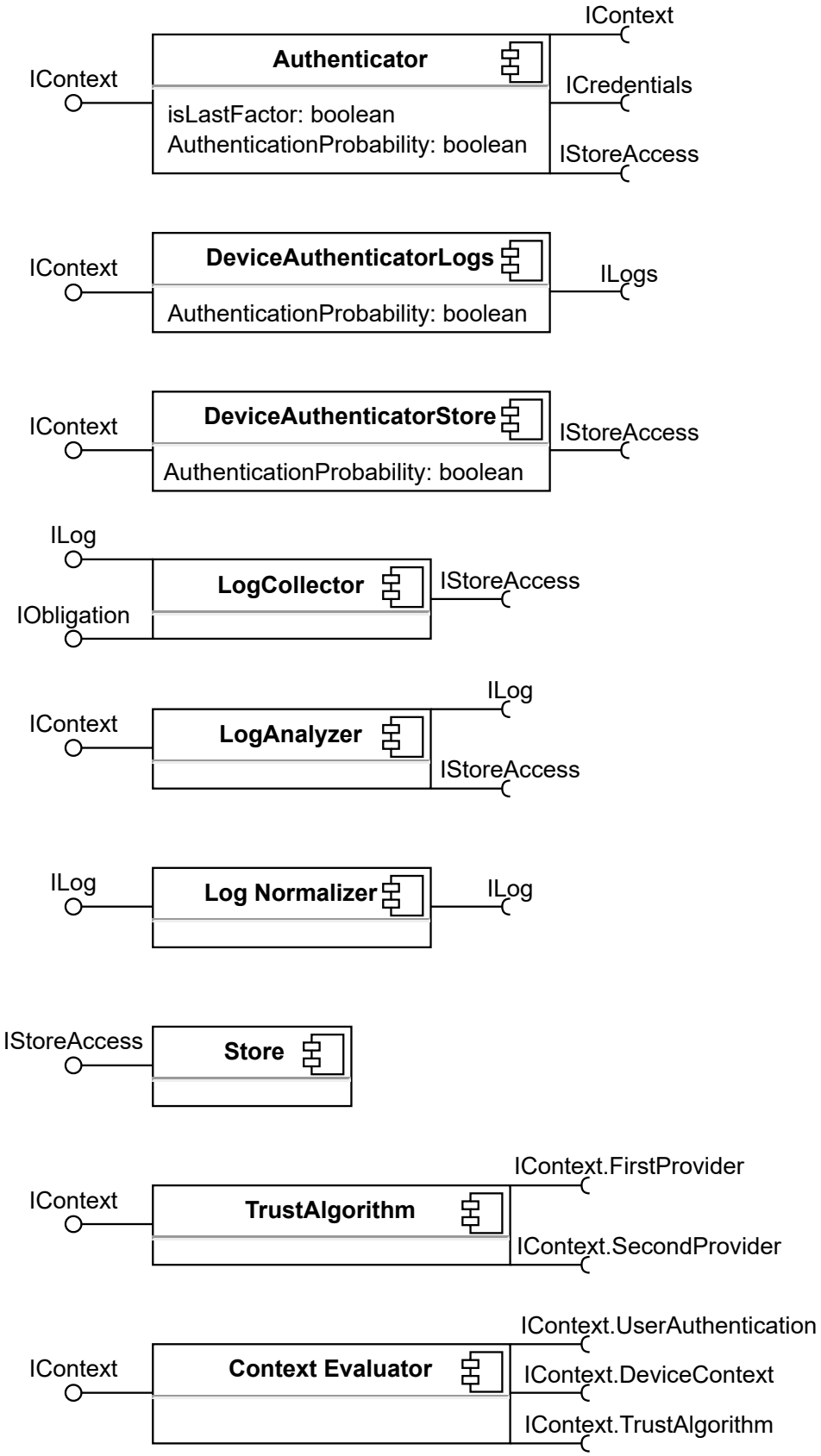


Figure 6.12: Context Providers

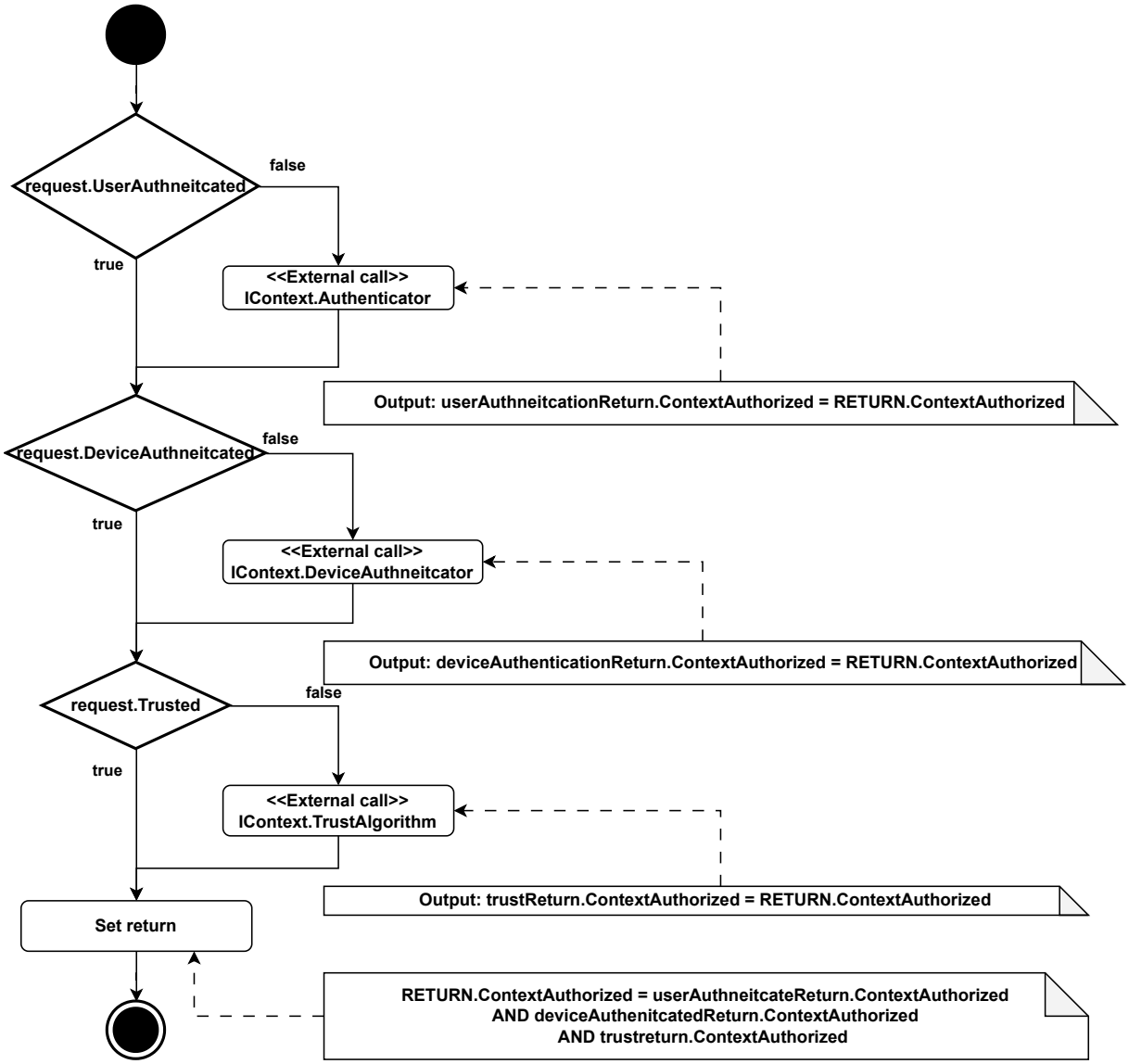


Figure 6.13: ContextEvaluator SEFF *provideContext(Request request)*

switch off different components by using the *UserAuthenticated, DeviceAuthenticated, Trusted* fields in the request. In the end, the component sets its returned variable based on the output of the three external calls to context providers using the logical *AND* operator.

6.5 Modelling Templates

In the following section, we are going to take the components and interfaces we have modelled in our base ZTA repository and use them to model ZTA approaches from the literature. We are going to model the SDP according to its specification [13] and the Beyondcorp architecture [37] which we described in Chapter 3. This is the first demonstration of how we can use off-the-shelf the modelled in this section components and reuse them among different ZTA approaches. As

a result, we can further reuse the created models of SDP and Beyondcorp to model systems which are integrating specifically the SDP or the Beyondcorp architecture.

6.5.1 SDP Model

We create a repository where we model the base components of an SDP - the SDP Controller and the Accepting Host - as composite components.

SDP Controller

According to the specification of the SDP, firstly the client should authenticate themselves through the SDP controller, then an access request is started in the controller to determine the resources to which the client has access. Lastly, the Accepting Hosts are configured to accept requests from the client. To be able to guide the process in this order we specify *SDP-ContextHandler* component. Therefore, we model the component, as shown in Figure 6.14, to provide the *IRequest* interface from the ZTA base repository as well as to require an *IContext*, where an authentication provider should be plugged, an *IRequest* interface, where we plug a *PDP* and the *IObligation* interface as to allow further customization of the process. Upon authentication request, the SDP context handler first initiates authentication and then sends a request for evaluation through a call to the *IRequest* required role. Finally, it executes the additional obligations, if plugged any. We use the *SDPContextHandler* inside the *SDPController-InternalAuthentication* composite component to guide the process. We integrate a single-factor authentication inside the SDP controller to create a basic case of an SDP controller. Developers should specify an external Identity Store provider and plug it in the SDP controller through the *IStoreAccess* interface. Inside the SDP controller composite component, we instantiate and connect a *PolicyAdministrator* and a *PolicyEngnine*. Since it is a basic SDP controller we include the *PoliciesProvider* inside, though it can be also externalized by simply removing the component from there and making the SDP controller require the *IPolicies* interface. Further data sources can be plugged into the SDP controller with the help of the *IContext* interface.

SDP Accepting Hosts

From the SDP specification, we determine that the *AcceptingHost* composite component, as shown in Figure 6.15, is no more than a simple firewall which has its rule managed to accept to deny requests by clients. Therefore, to model it we use a *SimplexPEP* and close its obligation points to represent a simple firewall functionality. If customization is required then the composite component can be directly swapped with a *SimplexPEP* and obligations can be plugged into the extension points.

6.5.2 Beyondcorp Model

Now we are going to do the same as in the previous section but this time we will model the BeyondCorp approach using our components. First, we will introduce some additional basic components which are specific to BeyondCorp. Then we will model the more complex parts of the architecture - the *AccessingProxy*, the *AccessControlEngine* and the *SingleSignOn* system - as composite components.

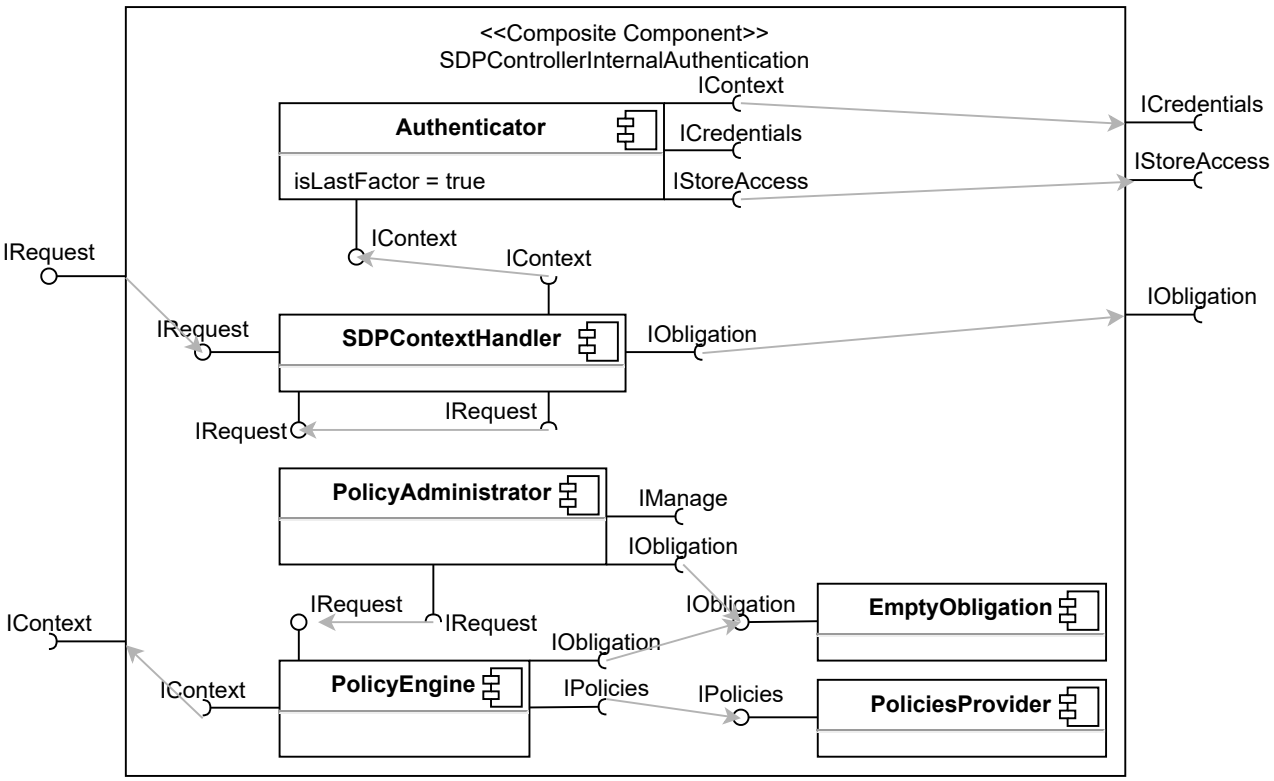


Figure 6.14: *SDPControllerInternalAuthentication* Composite Component

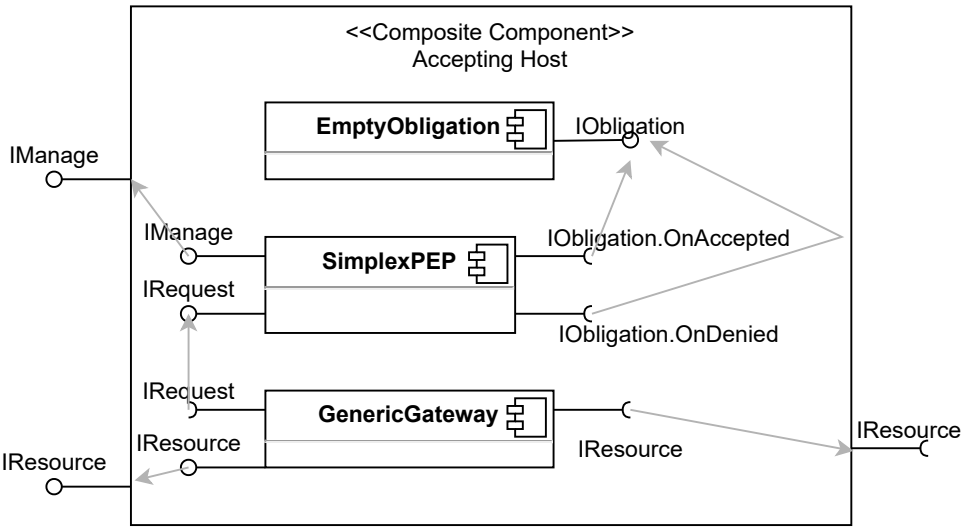


Figure 6.15: SDP's *AcceptingHost* Composite Component

BeyondCorp Basic Components

Firstly, we model the *Pipeline* as a basic component, mentioned in BeyondCorp's specification. The component is responsible for feeding data to the *PE* for the request evaluation. Therefore we can classify the component as a context provider. Since it is a context provider the *Pipeline* provides the *IContext* interface. The component should feed data from a user's database, a device's database as well as a trust algorithm. That is why we add to the component two required roles for the *IStoreAccess* interface and a required role for the *IContext* interface. When called, the *Pipeline* simply calls its required interfaces in a sequential order to simulate the process of collecting data to feed from input sources. Next, we create a *TrustInference* component. The trust inference in BeyondCorp is the trust algorithm of the system. However, in the provided architecture diagram in the document, the algorithm is not connected to any input sources. In the description of the component it is mentioned that it might process data about a device's OS or a user's location but it is not further specified how it does so. Therefore, we will not be using our more complex *TrustAlgorithm* component from the ZTA repository but we model a new one here. The *TrustInference* component simply provides the *IContext* interface and when called it performs a single internal action of calculating the trust. As a last basic component of this repository, we create an obligation component. According to the BeyondCorp specification when a request reaches the *AccessingProxy* it is first checked for a successful authentication and in the absence of such the request is forwarded to an authenticating system. We create the *OnAuthenitcationMissingObligation* component which provides the *IObligation* interface and requires the *IContext and IRequest* interfaces. When called the component forwards a request to its *IContext* required role, where an authenticating system should be plugged and after that, it returns the request to the *AccessingProxy* through the *IRequest* interface.

Accessing Proxy

The *PEP* in BeyondCorp is called Accessing Proxy. It should be able to intercept requests and forward them to an authentication system if needed as well as to the *PE*. We model it as a composite component, as shown in Figure 6.16. In contrast to the SDP approach, here the *PEP* has more functionalities than a simple firewall. That is why we are going to use a *DuplexPEP* component so we can forward requests to an authenticator and a *PE*. As a gateway, we instantiate the *GenericGateway* component from the ZTA repository. To forward unauthenticated requests to an authenticating system we use the modelled in the basic components section, see paragraph 6.5.2, *OnMissingAuthenticationObligation*.

Single Sign On

The authentication system in BeyondCorp is called Single Sign On. It utilizes a 2-factor Authentication where users should provide first credentials and then a passcode from a token. Since we need to combine two *Authenticator* components to represent this authentication approach we are going to model the Single Sign On as a composite component. We create the composite component *SingleSignOn*, as shown in Figure 6.17. We instantiate two *Authenitcator* components - one for the first factor and one for the second factor. The authentication probabilities we set as boolean PMF and give priority to successful authentication. These values can

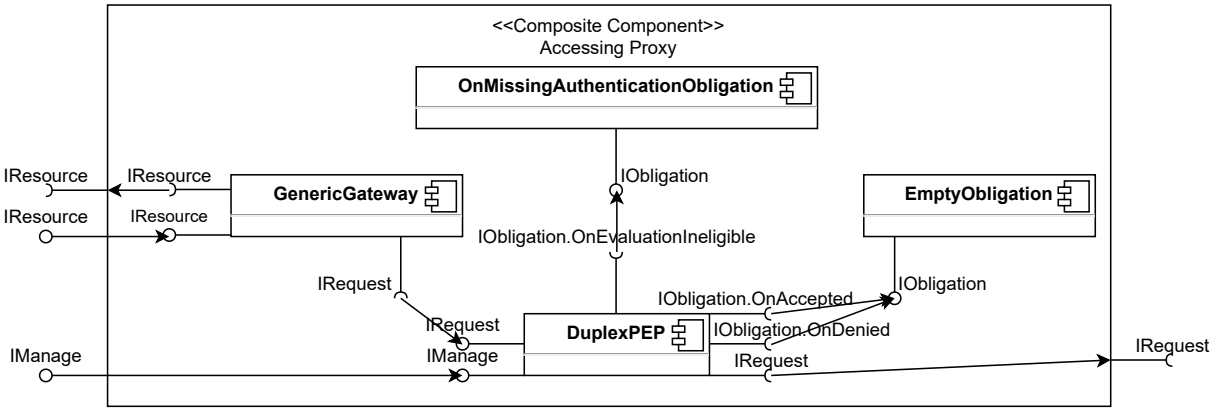


Figure 6.16: Accessing Proxy composite component

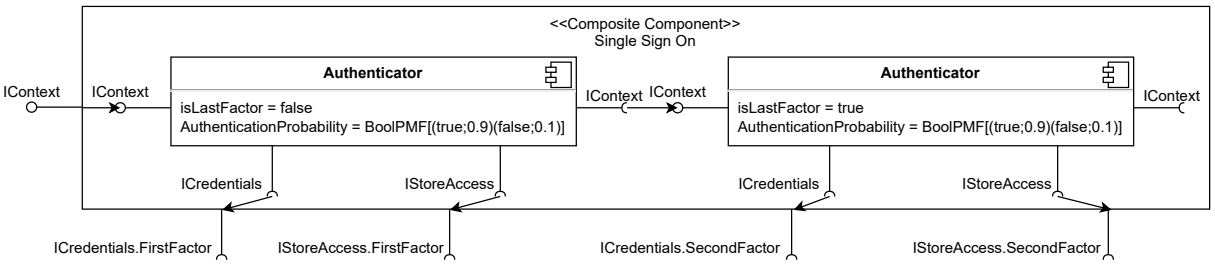


Figure 6.17: Single Sign On composite component

however be tuned by component developers. The component provides the *IContext* interface and requires *IStoreAccess and ICredentials* for both of the authentication factors.

Access Control Engine

The last composite component which we specify in the BeyondCorp repository is the *PE*, which in the context of BeyondCorp is called Access Control Engine. As shown in Figure 6.18, we model the component *AccessControlEngine* by instantiating in it a *PolicyAdministrator* and a *PolicyEngine* components. The policies are also integrated directly into the control engine and therefore we instantiate a *PoliciesProvdier* component. As example values for the *numberOfPolicies* and *rulesPerPolicy* we set respectively 100 and 10 since we do not have access to BeyondCorp's policies database and cannot approximate more accurately the number of policies.

6.6 Applying ZTAs on the Running Example

In this section, we use the created elements from the ZTA repository as well as the templates from the SDP and Beyondcorp repositories to integrate different ZTAs into the JPlag running example.

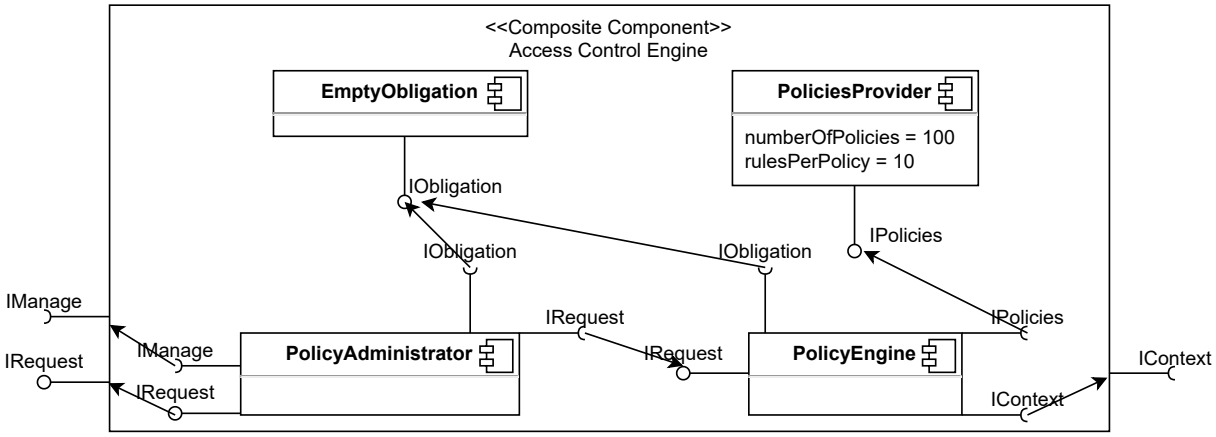


Figure 6.18: Access Control Engine composite component

6.6.1 JPlag with ZTA

We are going to integrate a ZTA into the JPlag system from the previous section, following a basic design from the NIST standard document [48]. The architecture of the base JPlag system is the same as in Figure 4.1. In this scenario, we want to protect the JPlag functionality. This means that we want to intercept requests to access the JPlag GUI and evaluate them. To do this we need to instantiate a *PEP* from our ZTA repository in front of the JPlag GUI and make the users send requests to the PEP. We select the *DuplexPEP* component from the repository since it allows us to accept requests and forward them to a *PolicyEngine*. As a gateway, we are going to use the *GenericGateway* from the ZTA repository. We also need to add an additional component to our JPlag repository in order to bridge the interfaces of the *GenericGateway* and *JPlag*. That is why we introduce the *JPlagResourceAdapter*. The component provides the *IResource* interface and requires *IJPlag* interface. Its sole purpose is to translate a request from the *IResource* to *IJPlag*. There is another way to model this without using the *GenericGateway* provided in the ZTA repository. A component developer might directly define a JPlag gateway following the modelling of the generic gateway and only change the *IResource* interface with the *IJPlag* interface. However, a more complex SEFF should be declared compared to the modelling of a simple adapter like the one above. After we have done this, we are ready to fully integrate the *PEP* into our system. We instantiate the *DuplexPEP*, *GenericGateway* and the *JPlagResourceAdapter* and connect them using the corresponding interfaces. We change the *JPlag* provided interface of the system to *IResource* and now our gateway is the entry point to the JPlag functionality. Next, we need to instantiate the *PolicyEngine* component and the *PolicyAdministrator*. The *PolicyEngine* requires policies to be able to operate. Therefore, we instantiate the *PoliciesProvider* component from the ZTA repository. This component requires us to specify values for the number of policies and the rules per policy. For this scenario, we are going to use example values and define that the number of policies is 100 and the rules contained per policy is also 100. Next, the *PolicyEngine* needs to evaluate context and for this scenario as a context, we are going to observe only user authentication. For this purpose, we instantiate an *Authenitcator* component from the ZTA repository. We are going to use a single-factor authentication and therefore in the component's *isLastFactor* variable, we define *true*. Additionally, we need to set the probability of correct user authentication. We define that

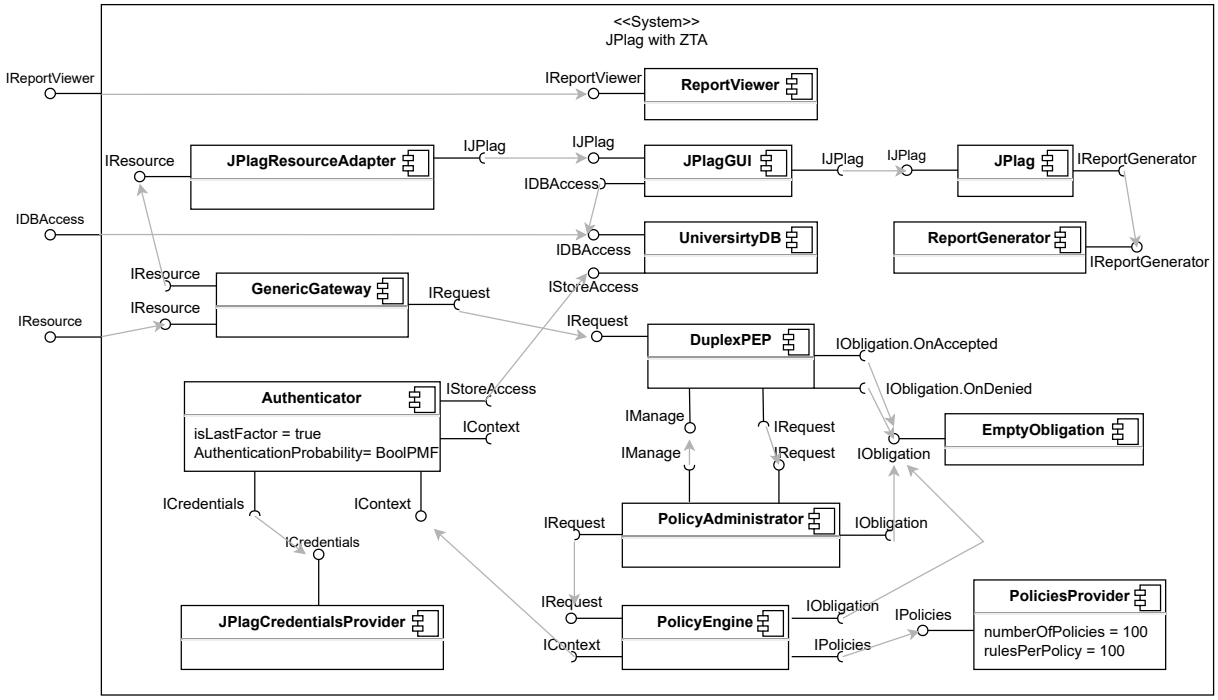


Figure 6.19: JPlag with ZTA

in 80% of the cases the users authenticate themselves correctly and in 20% of the cases they do not. The *Authenitcator* requires access to a store to be able to obtain user data. We already have a *UniversityDB* component in the system which can provide such data. We only need to edit the component to provide the *IStoreAccess* so it can be compatible with the *Authneitcator's* required interface. Lastly, we specify in the JPlag repository a *JPlagCredentialProvider* component which we use to simulate the input of credentials by the user. To keep the example simple we will not specify obligations for our components. Therefore, we can use the dummy obligation component provided in the ZTA repository - the *EmptyObligation* - and plug it into each required *IObligation* interface. This is how we can integrate a basic ZTA architecture in our already present JPlag system and model it using components from Chapter 6 to obtain the system displayed in Figure 6.19.

6.6.2 JPlag with SDP

Now we are going to model another version of JPlag integrating a ZTA but this time we are following the SDP approach. This means that now the protected resources are hidden behind a PEP which only filters requests based on its configuration and does not forward them to a *PolicyEngine*. In this scenario, the user needs to be able to contact the *PolicyEngine* for authentication and authorisation and then the *PEPs* are distributed configurations to filter the user's requests. We start again by identifying the resource which we are going to protect. To demonstrate a different scenario from the one in the previous section we will not protect the JPlag algorithm now. We will be shielding the *ReportViewer* and the *UniversityDB* with two separate *PEPs*. Recall that in the SDP approach, *PEPs* are called Accepting Hosts and in the SDP repository from Subsection 6.5.1, we already have modelled an *AcceptingHost* component

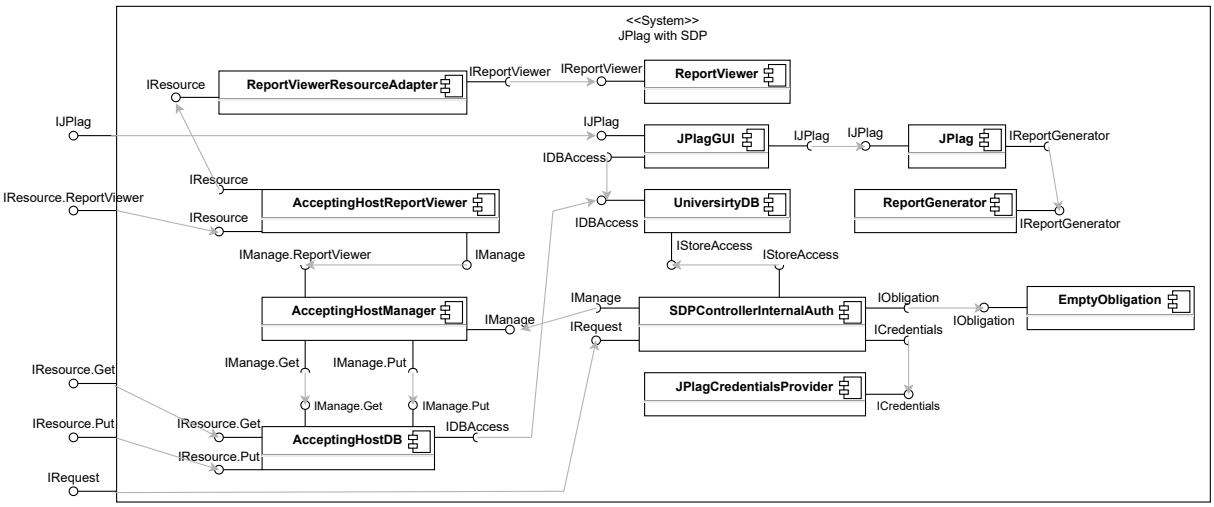


Figure 6.20: JPlag with SDP Assembly

using a *SimplexPEP* with a *GenericGateway*. We can use this composite component in our JPlag system. Since the *AcceptingHost* is using the *IResource* interface as an access point to a protected resource, the *AcceptingHost* in its initial form can be used to protect a single resource. In the case of the *ReportViewer*, we are protecting only the *viewReport()* functionality. Therefore, we create an *ReportViewerResourceAdapter* similar to the *JPlagReosurceAdapter* from the previous section to bridge the interfaces *IResource* and *IReportViewer*. Then we can go ahead and integrate the *AcceptingHost* alongside the created adapter to establish a perimeter around the *ReportViewer* as shown in Figure 6.20. For the *UniversityDB*, we need to make some adjustments to the *AcceptingHost*. The *UniversityDB* component provides the *IDBAccess* interface and therefore there are two functions to be protected - the *get()* and *put()* functionalities. That is why we create a composite component which combines two *AcceptingHosts* as shown in Figure 6.21. To bridge the *IDBAccess* and *IResource* interfaces of the accepting hosts we create two adapters - *DBResourceAdapterGet* and *DBResourceAdapterPut*. Next, we need an SDP controller. We have modelled one in our SDP repository and we are going to use it here directly. The whole system now needs to provide the *IRequest* interface of the *SDPControllerWithInternalAuthentication* component to allow the user to make authentication and authorization requests. We can again use the *JPlagCredentialsProvider* which we modelled in the previous section and again to keep the model simple we will not attach additional obligations to the SDP controller. Lastly, we need to enable the controller to manage multiple Accepting Hosts. To do this, we use *PEPsManager* component from our ZTA base repository, which accepts *manage()* calls from the SDP controller and forwards them to all of its required *IManage* roles. This is how we integrate an SDP architecture into our JPlag system with two Accepting Hosts and a single SDP controller.

6.6.3 JPlag with BeyondCorp

In this subsection, we are showing one last example of integrating ZTA into the JPlag system. This time we are going to use the BeyondCorp approach. We have already modelled the core BeyondCore component as per in the BeyondCorp repository from Subsection 6.5.2.

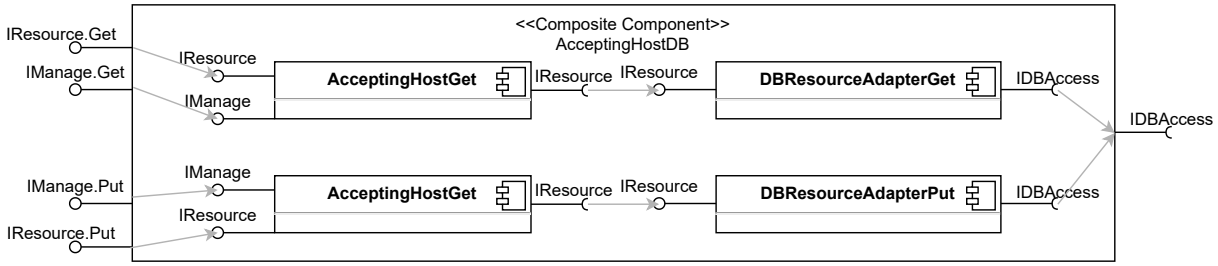


Figure 6.21: Accepting Host for UniversityDB Composite Component

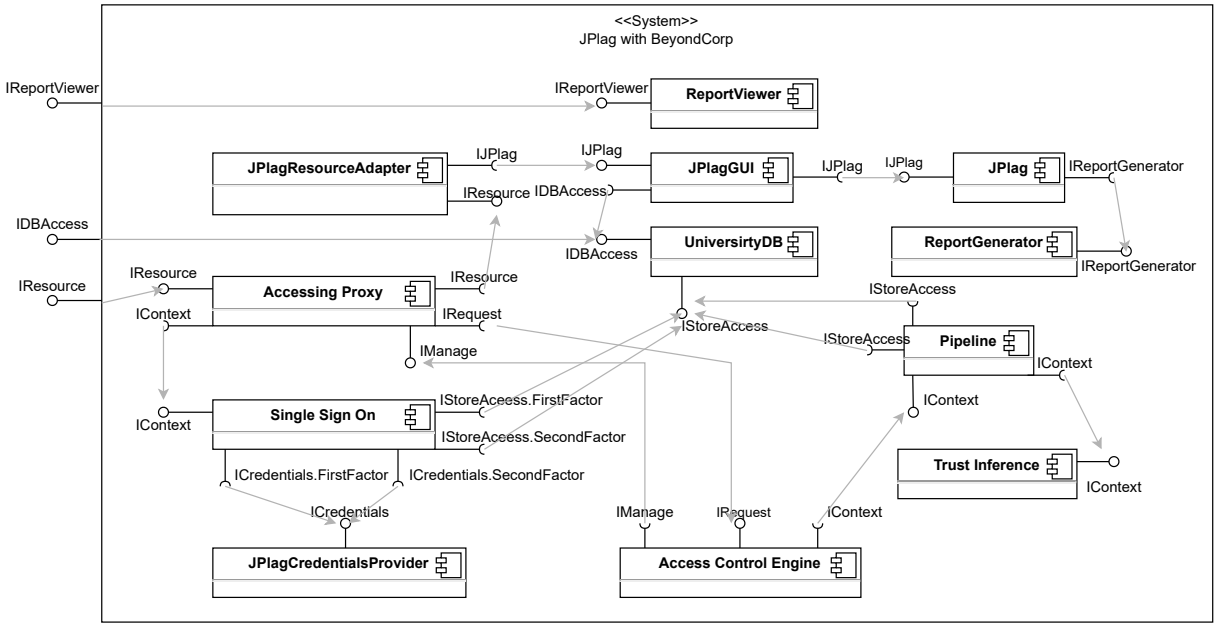


Figure 6.22: JPlag with BeyondCorp

Therefore, we can directly use the components from the repository and instantiate them in the system to obtain the modified system shown in Figure 6.22. Again the only adjustments which we need to make in the underlying JPlag system are the modelling of the *JPlagResourceResourceAdapter* and extending the *UniversityDB* to provide the *IDBAccess* interface. We instantiate the *AccessProxy* and the adapter components to form a perimeter around the JPlag functionality. Since in BeyondCorp requests might be stopped by the Access Proxy due to missing authentication we instantiate the *SingleSignOn* component to authenticate users. We plug it into the *UniversityDB* to get information about user data for the first- and second-factor authentication. As a credentials provider, we use the same *JPlagCredentialsProvider* for both factors. It is possible here to plug different credentials providers that have different delays to simulate the input of a password and the input of a secret token for example. Then, we instantiate the *AccessControlEngine* component which is the PDP of the system. It requires context data that is being provided by the instantiated *Pipeline* component. The *Pipeline* uses the *UniversityDB* to obtain users' and devices' data. For the trust mechanism, the pipeline is using again the modelled in the BeyondCorp repository *TrustInference* component.

6.6.4 Discussion on applying ZTA Elements

As we see from the previous examples, we can use the modelled ZTA components to create different versions of ZTA. We created an architecture following the NIST standard, the SDP specification and the Beyondcorp, described in Section 2.1 and Section 3.2, with little changes made to the underlying system. There is even an easier approach to integrating a ZTA into a system. This is the case where from the beginning of the design we know what variation of a ZTA we want to apply to a system and design the system's components to be compatible with the ZTA components. For example, in the ZTA system, we might not design the *IDBAccess* interface and directly use the *IStoreAccess* since it provides similar functionality. Then we will not need to extend the *UniveristyDB* component since it will directly provide the *IStoreAccess* interface. If we go even further we can directly replace the *UniversityDB* with the *Store* component from the ZTA repository. Another possibility to enable compatibility with ZTA components in the system's base design is to design the *JPlagGUI* component to provide a gateway functionality. We can edit its SEFF to forward requests to a *PEP* and to access the protected resource, in this case, the JPlag algorithm, only after a response from a *PEP* component. This would omit the need to specify additional adapter or gateway components.

7 Data Flow Analysis Model

In Palladio, security analyses can be performed using the Data Flow Analysis, presented in Section 2.4. However, to enable it, we first need to define a security model. The security model includes defining a dictionary with characteristics as well as specifying the behaviour of components when handling characteristics. In the following chapter, we propose a concept for a security model in a ZTA. Then following the concept we derive and define data dictionary and node behaviours for the proposed components from Chapter 6. We then propose six security violations which can be detected with our security model. Next, we want to make our Data Flow Analysis of ZTA models reusable. Therefore, we propose a Java implementation of our analysis which can be used off-the-shelf and can be extended to cover different result reporting formats. In our implementation, we include constraints for label evaluation, data classes to hold identified violations as well as basic reporting capability. Lastly, we demonstrate how our security model can be integrated by including it in the JPlag running example, presented in Chapter 4.

7.1 Concept

When we are assessing the security of a ZTA we need first to identify what we consider as secure. In the case of the ZTA, we need to make sure that only *authorized* requests reach protected resources. Additionally, only requests performed by an *authenticated user*, from an *authenticated device* and *trusted* by the trust algorithm reach the protected resource. Furthermore, we examine also the level of authorization which is granted to a request. Since the LPP is one of the core concepts of Zero Trust, we also observe if the level granted to a request does not supersede the required by the resource level. To be able to detect such violations we need the model the following things.

7.1.1 Authorization

We need to model labels which represent authorized and unauthorized data. These labels should also represent the level of authorization. That is why we can create labels "Authorized Level N", where N represents the level of authorization. Those levels are ordered which means that the label "Authorized Level 2" would supersede "Authorized Level 1". We set the label "Authorized level 0" to express unauthorized data. With these labels, we can annotate the highest level a user can be authorized to in a system as well as the current authorized level of data. In the end, we try to match the authorization levels of the data and the required authorization level of the resource node the data has reached. If the level of data is lower then we detect that unauthorized access was allowed. If we detect a data level higher than the resource node level, then it means that the LPP was not enforced.

Data should be set with default "Authorization Level 0" and only one component in a ZTA should be able to modify the authorization levels of the data. Other nodes should only perform forwarding actions. If data with level 0 reaches a protected resource node then it means that data has not been propagated through an authorizing entity.

7.1.2 Authentication

To be able to detect whether a request was made by an authenticated user from an authenticated device and has passed the trust algorithm we model authentication labels. We outline the "Device Authenticated" and "Trust Authenticated" labels. For user authentication, we also need to consider for which of the authentication factors a user was authenticated. Hence we need to model the user authentication label in such a manner that it captures different factors. Therefore we create the label "Authenticated N Factor" where N represent the factor number. Then similarly to the authorization, these labels should only be modified by dedicated components. Any other node should perform only forwarding action for these labels. At the end of the analysis, we check whether the data has all of these labels. A missing label would mean that data was not passed through the needed component for authentication and therefore a violation can be identified.

7.2 Data Dictionary

Now that we have specified a concept for our security model and proposed concepts for labels we try to model them in Palladio. We need to specify a data dictionary which holds all of the characteristic types and their possible values. The values which a characteristic can take are defined as an *enum*. Following the concept for authorization labels from Subsection 7.1.1, the value with which data can be labelled is the level of authorization - *Level 1, Level 2*, etc. Therefore we define an *enum* to capture the possible authorization levels, Listing 7.1 lines 1-6. Only 3 levels plus the unauthorized level 0 are hard-coded as labels. These however can be expanded with more levels if required.

Characteristics of data are defined as *enumCharacteristicTypes* which use a previously defined *enum*. We use these characteristics later to label nodes and data that propagates in the system. When we label data we set one or more of the values of the *enum* used by this characteristic to *true*. We want to be able to characterize requests and nodes with their authorization level. Additionally, we want to characterize data with the highest authorization level which a user can achieve and the required authorization level of the resource. As shown in Listing 7.1 lines 8-10, we use *enumCharactersiticTypes* to represent those aspects of the security concept. The enum characteristic type *Authorized* represents the authorization level that is assigned to the request. The *MaxUserLevel* represent the highest level of the user who makes the request and is the highest level which can be assigned to a request. The *ResourceRequireLevel* represents what is the level needed to access the resource.

```
1    enum AuthorizationLevels {
2        Level 0
3        Level 1
4        Level 2
5        Level 3
```

```
6   }
7
8   enumCharacteristicType Authorized using AuthorizationLevels
9   enumCharacteristicType MaxUserLevel using AuthorizationLevels
10  enumCharacteristicType ResourceRequriedLevel using AuthorizationLevels
```

Listing 7.1: Authorization characteristics definition

Next, we do a similar label definition but this time we capture the concepts for user authentication. As discussed, we want to represent different factors of authentication. There, we model the *AuthenticationLevels enum*, Listing 7.2 lines 1-4. We define only two values for authentication labels. The definition can be extended with more authentication factors if required. Then to represent the user authentication characteristic of data we define the *enumCharactersiticType Authenticated*, Listing 7.2 line 6.

```
1   enum AuthenticationLevel {
2       FirstFactor
3       SecondFactor
4   }
5
6   enumCharacteristicType Authenticated using AuthenticationLevel
```

Listing 7.2: User Authentication characteristics definition

For device authentication and trust we want to have a simple value which signals if data has passed a device authentication unit and respectively a trust algorithm. This is a binary state. Data is either authenticated/trusted or it is not. Therefore we can use a single-valued label. The presence of the label signals a positive state of authentication/trust and its lack signals a negative state. This is why we model the *Authenticated*, Listing 7.3 lines 1-3, with the single value *authenticated*. Then to represent whether data has been sent by an authenticated device or the subject is trusted we model the characteristics *DeviceAuthenticated and Trusted*, Listing 7.3 lines 5 and 6.

```
1   enum Authenticated {
2       authenticated
3   }
4
5   enumCharacteristicType DeviceAuthenticated using Authenticated
6   enumCharacteristicType Trusted using Authenticated
```

Listing 7.3: Device Authentication and Trust characteristics definition

We want to be able to label nodes which represent the protected by the ZTA resources. We want to distinguish them so we can evaluate constraints at these nodes in the Data Flow Analysis. For this purpose, we define the *enum Protected* with a single value *protected*, Listing 7.4 lines 1-3. We define also the *enumCharactersitictype* with the similar name *Protected*, Listing 7.4 line 4, to characterize nodes as protected.

```
1   enum Protected {
2       protected
3   }
4   enumCharacteristicType Protected using Protected
```

Listing 7.4: Protected characteristic definition

Lastly, we need to define a characterisation which describes that data has passed through a *PolicyEngine* component. Therefore, we define the *enum Protected* which is used by the *enumCharacteristicType Evaluated*, as shown in Listing 7.5. Although data may be propagated through a *PolicyEngine* and authorization levels correctly set, data with a lower authorization level will still arrive at the protected node. However, if the lower level data is also labelled with the *Evaluated* characteristic, it represents that data has passed policy evaluation and in a real-life scenario will be stopped at that point. A combination of a lower authorization level label than required and the *Evlauated* label invalidates the raising of an unauthorized access issue.

```
1  enum Evaluated {
2      evaluated
3  }
4  enumCharacteristicType Evaluated using Evaluated
```

Listing 7.5: Evaluated characteristic definition

7.3 Nodes Behaviour

We need to specify the behaviour of processing data labels of our proposed components from Chapter 6. This behaviour defines whether a component alters data labels. If this is the case we specify how it alters the data labels. Node behaviours are part of the security model. As we mentioned in the concept Section 7.1, most of the nodes have a forwarding behaviour which means that they simply take the labels from the input data and apply them unchanged on the output data. Therefore, we set in their return variables *RETURN.*.* = request.*.** where *request* is the received data. This expression means that all labels of the *RETURN* variable are the same as all of the labels of the *reuqest* parameter. Still, there are certain components which have special behaviour and can alter labels. These components are:

1. *PolicyEngine* - can alter the *Authorized* and *Evaluated* enum characteristic types.
2. *Authenticator* - can alter the *UserAuthenticated*.
3. *DeviceAuthenticator* - can alter the *DeviceAuthenticated*.
4. *TrustAlgorithm* - can alter the *Trusted*.

Although we are using the names of the components as proposed in Chapter 6, the behaviours specified in this section are not tightly connected to these specific models. If a developer swaps one of these components with their own modelled version they can still reuse the proposed here annotations and behaviour definitions without changing them. This can be done by simply copying the presented here annotations and pasting them into their model of a component.

7.3.1 PolicyEngine Behaviour

Since the *PolicyEngine* is the component which is responsible for making decisions about the authorization of a request, it is logical that this component should be able to alter the *Authorized* characteristic in the security model. When doing so, the component should consider the,

required by the resource, authorization level as well as the highest possible authorization level of the subject. Since we have modelled characteristics for these requirements - *MaxUserLevel and ResourceRequiredLevel* - the component should consider them in its behaviour definition. The *PolicyEngine* should not authorize a request if the *MaxUserLevel* is lower than the *ResourceRequiredLevel*. Additionally, a *PolicyEngine* should follow the LPP. Hence, it should not set the *Authorized* label to a higher level than the *ResourceRequiredLevle*, even if the *MaxUserLevel* allows it. Following these rules, we define the behaviour of a *PolicyEngine* as shown in Listing 7.6. In lines 3,8 and 14 we observe that the first condition for assigning an authorized level of a certain degree to the return variable is the same level to be required by the resource. For example, this eliminates the option of assigning a *Level3* label to a *Level1* resource, since in the *ResourceRequiredLevel* characteristic of a request, the *Level3* label is missing which would cause the expression from lines 14-16 to be evaluated to false. Next, we make sure that only a subject does not get authorized to a higher level than their highest possible. We only assign a level if the *MaxUserLevel* characteristic of the request supersedes it. For example, in lines 3-5, we authorize with *Level1* when the request has any of the currently defined authorization levels in its *MaxUserLevel* characteristic. However, from lines 10 and 11 we see that a request cannot be authorized to *Level2* if its *MaxUserLevel* characteristic has *Level1* authorization.

Lastly, the *PolicyEngine* sets the *Evaluated.evaluated* characteristic of the data as well as removes the *Level0* label from the data, Listing 7.6 lines 17 and 18. This indicates later in the analysis that data has passed through a policy evaluation component.

Apart from the *Authorized and Evaluated* characteristics, the *PolicyEngine* does not alter any other characteristics. It simply forwards them.

```
1   RETURN.Authorized.Level1 = request.ResourceRequiredLevel.Level1
2       AND (
3               request.MaxUserLevel.Level1
4               OR request.MaxUserLevel.Level2
5               OR request.MaxUserLevel.Level3
6           )
7
8   RETURN.Authorized.Level2 = request.ResourceRequiredLevel.Level2
9       AND (
10              request.MaxUserLevel.Level2
11              OR request.MaxUserLevel.Level3
12          )
13
14  RETURN.Authorized.Level3 = request.ResourceRequiredLevel.Level3
15      AND request.MaxUserLevel.Level3
16
17  RETURN.Authorized.Level0 = false
18  RETURN.Evaluated.evaluated = true
```

Listing 7.6: Node behaviour of the *PolicyEngine* component

7.3.2 Authenticator Behaviour

Next, we are modelling the behaviour of the *Authenticator* component. It is a model of an authentication system and therefore can alter the *UserAuthenitcated* characteristic. Since the modelled component is responsible for authenticating only one factor and passing the

authentication request to the next factor it should also only alter its respective factor label. To achieve this, we propose that the component should have a node characteristic *UserAuthenitcated* defined when it is instantiated in an assembly model. In this characteristic, we set only the label of the factor for which the instance of the *Authenticator* component is responsible. For example, if an instance of the component is responsible for authenticating the first factor in an authentication process, it should have only the *UserAuthenitcated.FirstFactor* label set to true, as shown in Listing 7.7.

When altering the *UserAuthenitcated* characteristic, the component needs to consider the output of the next in chain *Authenticators*, if present. If the *Authenticator* is just a single factor or it is the last factor in an authentication chain then the component directly copies all of the *UserAuthenitcated* labels of the *ComponentConfiguration* variable, which represents the node characteristic, to the output data labels, as shown in Listing 7.8 line 1. However, if there is an additional *Authenticator* component in the authentication chain, then at some point of its execution before returning, the component calls it and receives a *requestReturn* variable containing the authentication labels of all next-in-chain *Authenticators*. Then, in its return variable, the *Authenticator* needs to copy it *ComponentConfiguration* without changing the received from other *Authenitcaotrs* labels. To do this we perform a logical *OR* operation on the *UserAuthenitcated* characteristics of the *requestReturn and ComponentConfiguration* variables. The *OR* operation ensures also that no labels get overwritten.

To provide a better understanding of the behaviour we demonstrate it in an example. We take a two-factor authentication system which consists of two *Authenticator* components - *FirstFactor* and *SecondFactor*.

1. When data arrives at *FirstFactor* it is propagated to *SecondFactor*.
2. Since *SecondFactor* is the last factor in the process it does not propagate it further and directly jumps to setting the labels for its return data. Since it has not received any labels by subsequent component it copies its *ComponentConfigurtion*, as shown in Listing 7.8 line 1, and returns the data to *FirstFactor*.
3. *FirstFactor* receives this data and stores it in the *requestReturn* variable.
4. *FirstFactor* jumps to the setting the return variable and performs an *OR* operation, as shown in Listing 7.8 lines 3 and 4.

```
1  UserAuthenticated.FirstFactor = true
2  UserAuthenitcated.SecondFactor = false
```

Listing 7.7: Node characteristic of *Authenitcator* component responsible for the first factor

```
1  RETURN.UserAuthenticated.* = ComponentConfiguration.UserAuthenticated.*
2
3  RETURN.UserAuthenticated.* =
4  ComponentConfiguration.UserAuthenticated OR requestReturn.UserAuthenticated.*
```

Listing 7.8: Node behaviour of the *Authenticator* component

7.3.3 DeviceAuthenticator and TrustAlgorithm

We have modelled *DeviceAuthenitcator* components which are responsible for authenticating a device and a *TrustAlgorithm* component responsible for calculating trust. Therefore, these components should also be able to alter respectively the *DeviceAuthenitcated* and *Trusted* characteristics. When receiving a request these components simply alter the labels for which they are responsible by setting their value to true, as shown in Listing 7.9. For the other labels, the components perform a forwarding action.

```
1    RETURN.DeviceAuthenticated.authenticated = true
2
3    RETURN.Trusted.authenticated = true
```

Listing 7.9: Node behaviours of *DeviceAuthenticator* and *TrustAlgorithm* components

7.4 Violations

Now that we have defined our labels and have defined how our components handle these labels we can outline the possible violations. Since we have created a general security model, the violations which we can detect with it are also general for models which integrate the components from our repository.

7.4.1 Multiple Authorization Labels

Following the logic of the node behaviour of a *PE*, we know that multiple authorization labels cannot be assigned to the same request. Moreover, we cannot identify a case where this should even be logical. For example, a request cannot at the same time acquire *Authorization Level 1* and *Authorization Level 2*. If the request should be authorized at *Level 2* then it does not make sense to also have the *Level 1* label since those permissions are already contained in *Level 2*. We have defined our labels as an ordered set with each next level extending the permissions of the previous one. Next, if a request is authorized to Level 2 when it should only be *Level 1* then indicates a breach in the LPP. This violation is also an important prerequisite for the evaluation of the next defined violations. If there are multiple authorization labels to be evaluated then this would induce ambiguity in a request's authorisation and we will fail to detect which label was assigned by the *PE* and which was added after the request has left the *PE*. Then we will not be able to differentiate whether a violation breaches the LPP or if it is an unauthorized request. Additionally, this type of violation could help us identify the presence of components other than the *PE* on the request path which assigns authorization labels and therefore are capable of authorizing requests. This is a violation since in a ZTA, only the *PE* should be responsible for that. Furthermore, if a developer models their own *PE* and uses our security annotations in it they could check if their *PE* handles labels the right way.

7.4.2 Unauthorized Access

The next violation that it is possible to detect is the unauthorized access. This happens when a request arrives at a resource and the assigned authorization level is lower than the assigned

authorization level of the resource. This violation could be an indication of an alternative path to protected resources which avoids the PEP.

7.4.3 Least Privilege Principle

Due to the ordered set of labels, we can enable the identification of LPP breaches. This could happen when a request arrives at a resource and the assigned authorization level of the request is higher than the level required by the resource. This violation could indicate that either a *PE* is not authorizing a request following the principle or there is a component on the path to the resource which elevates the permissions of a request.

7.4.4 Unauthenitcated Access

Another violation which we can detect with our annotations is the unauthenticated access. This violation can happen when a request arrives at a resource and is missing authentication labels. We say labels since our authentication model includes labels for a two-factor authentication. The violation could indicate the lack of an authentication entity on the path if the labels are missing. It is also possible that only one of the authentication labels is missing. This also results in an unauthenticated request and could represent that either one of the factors is not functioning properly or it was not traversed at all.

7.4.5 Device Unauthenticated Access, Untrusted Access

The last two violations are similar in their logic and therefore we are going to summarize them in a single section. How one is detected and what it represents is similar to the other two. Therefore we will talk about device unauthenticated access. It happens when a request arrives at a resource with a missing device authentication label. This would mean that along the path no entity responsible for authenticating the device was visited. Similar to that is detected an untrusted access and it represents the same problem. These two violations, however, can be considered less serious than the other violations in the section. This is because some systems might not have reached the level of ZTA where they authenticate a device or include a trust algorithm in their system. Therefore, in such systems entities which alter those labels are not present and the violations will be present. This presence of the violation in such systems is not unnecessary since it turns the attention of a developer to the need to eventually include device authentication and a trust algorithm in their system.

7.5 Generalising ZTA security analysis

After we have outlined the possible violations, we want to have a common approach to detect them. In the Data Flow Analysis, this happens by evaluating constraints. Since the violations are general for each model using our components we can also define general constraints in the form of Java functions which can be reused. We go even further and try to generalize the whole process of analysing a ZTA model for the defined above violations by providing a class which encapsulates all of the constraints and the whole process of iterating nodes and checking

them. We also implement data classes which form a ZTA report. Data classes in Java are only responsible for holding data and do not provide complex logic. Using these data classes, we can specify a universal way for reporting results from our analysis as well as provide the option to extend further the reporting process by allowing the data classes to be converted to data of other formats such as a JSON or XML schema. Apart from the re-usability of the whole analysis, this implementation would allow the obtaining of report results which can be analysed or visualised by other services.

7.5.1 Defining a Java Enum

To represent the violations in our Java implementation of the security analysis we specify a Java enum *ViolationTypes*, as shown in Listing 7.10, to represent each of the possible violations.

```
    public enum ViolationTypes {
  MULTIPLE_AUTHORIZATION_LABELS,
  UNAUTHORIZED_ACCESS,
  LEAST_PRIVILEGE_VIOLATION,
  UNAUTHENTICATED_ACCESS,
  DEVICE_UNAUTHENTICATED_ACCESS,
  UNTRUSTED_ACCESS
}
```

Listing 7.10: ViolationTypes.java

7.5.2 Violation Data Class

First, we want to specify a data class which encapsulates all of the information concerning a violation. We want to store as attributes what is the type of violation, an identifier of the node where it was detected and lists of the characteristics present on the node and received by it. Therefore in the *Violation* Java class, we define a private field for each one of the mentioned attributes as shown in Listing 7.11.

```
public class Violation {

    private String type;
    private Violation node;
    private List<String> nodeLabels;
    private List<String> recievedLabels;
    ...
    }
```

Listing 7.11: Structure of the violation data class

The class has additional getters and setters for these fields, which we omit from the displayed code snippet for simplicity.

7.5.3 ZTAReport Data Class

Next, we define a data class, as shown in Listing 7.12, which represents the whole report. The report should contain all of the detected violations. Therefore, we are going to define a list

of *Violation* elements for each of the violation types (Listing 7.11, Lines 3-8) and define only report functions (Listing 7.11, Lines 10-20) which add a new violation to a list. Only one of the report functions is presented since all of them follow the same logic and we just change the list to which they are adding the violation and the violation type. We create the data class as shown below:

```
1  public class ZTAReport {
2
3      private List<Violation> multipleAuthorizationLabels;
4      private List<Violation> unauthorizedAccess;
5      private List<Violation> leastPrivilegeViolation;
6      private List<Violation> unauthneitcatedAccess;
7      private List<Violation> deviceUnauthenitcatedAccess;
8      private List<Violation> untrustedAccess;
9
10     public void reportMultipleAuthorization(String nodeName,
11                                             List<String> nodeLabels,
12                                             List<String> recievedLabels) {
13       this.multipleAuthorizationLabels.add(new Violation(
14                           ViolationTypes.MULTIPLE_AUTHORIZATION_LABELS.toString(),
15                       nodeName,
16                       nodeLabels,
17                       recievedLabels
18                   )
19               );
20         }
21     ...
22 }
```

Listing 7.12: Strucutre of ZTA report data class

7.5.4 ZTAReporter

Now we need to implement the logic of the reporter and we do this in the *ZTAReporter* class. The class has an instance of the *ZTAReport* class, which it uses to save violations. Then in the *ZTAReporter* we can define different functions which convert the *ZTAReport* object into different formats. Since it is out of the scope of this thesis we have not defined any conversion functions apart from the *printReport()*. This one simply prints out the ZTA report in the console in a readable format by using the *toString()* function of the report which we have overridden in the *ZTAReport* class. Now, the main logic which we need to implement is the constraints for our dataflow analysis. We are going to define the constraints as functions which receive an *AbstracSequenceElement* as a parameter, perform constraint evaluation on it and save the found violations in the *ZTAReport* instance. To reduce the complexity of the code snippets, only the constraints will be displayed which are defined in the form of *if*-statements. The variables *nodeLabels and dataLabels* are a list of strings which are respectively the labels on the node and the labels of the received data.

Multiple Authorization Labels Constraint The multiple authorization labels constraint is shown in listing Listing 7.13. Here the *nodeLabels and dataLabels* lists contain the labels for the *Authorized* characteristics from the data dictionary, see Section 7.2. To check the constraint we

look at the size of both *nodeLabels and dataLabels* lists and check if either of them is greater than one. If one of the checks is true then it indicates that either a node has been configured with multiple labels or data has arrived at the node with multiple authorization.

```
if(nodeAuthorization.size() > 1 || dataAuthorization.size() > 1) {
    report.reportMultipleAuthorization(node.toString(),
                                       nodeAuthorization,
                                       dataAuthorization);

    return true;
}
```

Listing 7.13: Multiple Authorization Labels constraint

Unauthorized Access and Least Privilege Violation Constraints Since we have created multiple authorization levels as an ordered set, we can use this definition to evaluate unauthorized access and violations of the LPP. To do this we compare labels but first, we need to extract a number from the label. The levels are defined as "Level N", see Section 7.2, and we can directly extract the number of the level from the label string as shown in listing Listing 7.14. We use for this the defined in the *ZTAReporter* private function *extratAuthorizationLevel(String label)*.

```
int nodeAuthorizationLevel = extractAuthenticationLevel(nodeAuthorization.get(0));
int dataAuthorizationLevel = extractAuthenticationLevel(dataAuthorization.get(0));
```

Listing 7.14: Extracting number of authorization level

After we have obtained the levels of authorization in the *nodeAuthorizationLevel and dataAuthorizationLevel* variables we can compare them to check for violations. First, we check if the node authorization level is higher than the data authorisation level, as shown in Listing 7.15. In the evaluation of the unauthorized access constraint, we have an additional labels list *evaluated*. This list holds the label *Evaluated.evaluated* if present. We check if the *evaluatedLabel* is missing from the data. If it is missing and the node label is larger than the data label, then this indicates unauthorized access. Then, as shown in Listing 7.16, we check if the data authorization is higher than the node authorization. If true, then this is an indication of a violation of the LPP.

```
if(nodeAuthorization > dataAuthorization) {
      report.reportLeastPrivilegeViolation(node.toString(), nodeAuthorizationLevel,
    dataAuthorizationLevel);
      return true;
}
```

Listing 7.15: Checking for unauthorized access

```
if(nodeAuthorization < dataAuthorization && evaluated.isEmpty()) {
      report.reportUnauthorizedAcess(node.toString(), nodeAuthorizationLevel,
    dataAuthorizationLevel);
      return true;
}
```

Listing 7.16: Checking for unauthorized access

Unauthenticated Access Constraint Next, we define the constraint for detecting unauthenticated access. In the *nodeAuthorization* list we obtain now the *Protected* characteristic of the node and in the *dataAuthorization* we obtain the *UserAuthenticated* characteristic. As shown below in listing Listing 7.17, we first check if the size of *nodeAuthorization* is greater than one. This indicates that the *Protected* characteristic is present and thus we should check for unauthenticated access. Then, we check if the *dataAuthorization* list has two labels - one for first-factor authentication and one for second-factor authentication. If one of the labels is missing then we report an unauthenticated access.

```
1 if (nodeAuthorizationLevel.isEmpty()) {
2       return false;
3 }
4
5 if(dataAuthorizationLevel.size() != 2) {
6       report.reportUnauthenitcatedAcess(node.toString(), nodeAuthorizationLevel,
      dataAuthorizationLevel);
7       return true;
8 }
```

Listing 7.17: Checking for unauthenticated access

Device Unauthenticated Access and Untrusted Access Constriants Both of the constraints for device unauthenticated access and untrusted access follow the same pattern only the checked labels differ. Therefore, we present here only the constraint for device unauthenticated access where we obtain in the *nodeAuthorization* list again the *Protected* characteristic of the node and in the *dataAuthorization* list we obtain the *DeviceAuthenticated* characteristic. In the case of an untrusted access check, the difference is that the *dataAuthorization* list stores the *Trusted* characteristic of the data. Similarly to the user authentication check we first check if the *Protected* characteristic is present on the node, as shown in listing Listing 7.18. Next, we check if the *DeviceAuthenticated* label is present. If it is not present, then this signals that the device used for the request has not been authenticated.

```
1 if (nodeAuthorizationLevel.isEmpty()) {
2       return false;
3 }
4
5 if(dataAuthorizationLevel.size() != 1) {
6       report.reportDeviceUnAuthneitcatedAccess(node.toString(), nodeAuthorizationLevel,
      dataAuthorizationLevel);
7       return true;
8 }
```

Listing 7.18: Checking for device unauthenticated access

7.6 Applying Security Annotations to JPlag Models

Now that we have defined security annotations and applied them to the ZTA components, we will discuss what else needs to be done to have a complete security model for JPlag. For the components which we take from the ZTA repository, we do not need to make any adjustments

since they already have the security annotations specified in their SEFFs in the previous sections. However, we still need to specify the behaviour of the components we created in the JPlag repository as well as a *.nodecharaacteristics* file for each of the assembly models.

JPlag with standard ZTA and with Beyondcorp We first observe the JPlag model with standard ZTA and Beyondcorp. The models are similar in the aspect that in both we are protecting the *JPlagGUI* component. For those models, we have only created one additional component in the JPlag repository - the *JPlagResourceAdapter*. In the SEFF of the component, there is only an external call to the *IJPlag.run()* function. We add forwarding behaviour to this external call, which means that we create an input variable usage for the *request* parameter which copies all of the characteristics of the data received by the *JPlagResourceAdapter*. Next, we create a *.nodecharacteristics* file for the model and inside we define an *Assembly Assignee* for the *IJPlagGUI* component. In this *Assembly Assignee* we set the *Protected.protected* label. Additionally, the authorized level label should be set according to the desired authorization level.

JPlag with SDP In this variation of JPlag with a ZTA we are protecting the *ReportViewer* and the *UniversityDB* components. Again, here we have modelled additionally three adapter components. Similar to the JPlag with standard ZTA scenario we need to define forwarding behaviour for these three components. Then we again need to define a *.nodecharactersitics* file where we set the *Protected.protected* label for the *ReportViewer and UniversityDB*.

8 Evaluation

In the following chapter, we present the evaluation of the developed in this thesis model. In order to have a structured evaluation we will be using the Goal-Question-Metric (GQM) [20] approach.

8.1 Evaluation Design

For the evaluation of the created ZTA model, we use the GQM approach. The approach requires defining clear goals. For each of the goals, we define what questions arise that need to be answered to prove the achievement of the goals. Where it is possible to answer the questions quantitatively, we define metrics. Following the approach, we define the goals:

- G1: Examine the completeness of the model regarding modelling power and identified elements from the literature.
- G2: Examine the applicability of our proposed ZTA components.
- G3: Examine the ability of the model to infer performance impact due to applied ZTA components.
- G4: Examine the ability of the model to detect security violations according to the proposed ZTA security violations concept.

8.1.1 Design for Evaluating Model Completeness

When evaluating the completeness of the proposed ZTA model, we consider first the modelling power of Palladio. Since there were no attempts to model ZTAs in Palladio until now, we want to observe if we were able to represent the researched concepts of ZTA using the currently implemented elements of the Palladio Component Model. Next, we want to concentrate on the identified ZTA elements during the research and analysis of ZTA standards. In the meta-model extraction, Chapter 5, we outlined 4 base logical components. In addition to that, we extracted also 9 main tasks which need to be performed by components in a ZTA. Here we raise the question if the modelling power of Palladio allows us to represent all of these components and tasks. Following these thoughts, we formulate the questions for the goal as follows:

- **Q1.1** Were all identified components and tasks represented using Palladio?
- **Q1.2** Did Palladio require adjustments during the modelling process?

Looking at the nature of the questions and the aspect they are evaluating, we claim that we are able to answer them quantitatively. Therefore we can define metrics for both of the questions. We do this as follows:

- **M1.1** Percentage of modelled elements from total identified elements.
- **M1.2** Number of adjustments made to Palladio.

8.1.2 Design for Evaluating Model Applicability

The next quality we want to evaluate in our proposed model is its applicability. Since as an objective of our work, we set the task of developing reusable components which generalize the concepts of ZTAs we want to focus on the generality of the components. This means that we want to test whether we can use the same components from our repository to model varying suggested ZTA models from the literature which were not used during the modelling process. As a next step in evaluating applicability, we want to assess what efforts are required to integrate the proposed ZTA elements into an existing system. Here we need to assess first what changes our components need to undergo to be used as well as what compatibility adjustments the model into which we integrate our components requires. In the context of this goal, we ask the following questions:

- **Q2.1** Are the proposed elements a general representation of ZTA components?
- **Q2.2** Can the proposed elements be used off-the-shelf in existing system models?
- **Q2.3** Does a system need to be modified to integrate the proposed ZTA components?

Question **Q2.1** does not allow a quantitative answer through the use of metrics. Therefore, we are going to approach the question by providing examples of mapping suggested ZTA models from the literature to the proposed in this thesis elements. We also provide a discussion on the mapping. For questions **Q2.2** and **Q2.3** we can specify metrics to help us answer the questions. We formulate them as follows:

- **M2.1** Number of adjustments to our components to make them compatible with an existing system model.
- **M2.2** Number of changes made to already present components of the model, where we integrate our elements.
- **M2.3** Number of new elements modelled to make a system compatible with our components.

8.1.3 Design for Evaluating Performance Inference

Since we want to enable performance analyses of systems integrating a ZTA we want to evaluate the ability of our components to induce performance overhead. We want to observe if the components and execution path which can be tuned using the modelled parameters from our repository, alter the simulated execution time of a system. This can be done by comparing

the simulations executed on a model without ZTA and simulations executed on the same model but with included ZTA components. By experimenting with multiple usage scenarios we can observe the variations in the execution time and resource utilization as well as paths taken during the execution. For this goal, we define the following questions:

- **Q3.1**Does the model allow the analysis of the performance impact induced by ZTA components and different execution flows?
- **Q3.2** Does the model allow the analysis of the performance impact of different ZTA execution flows?

To answer these questions we can observe the difference in the worst execution time achieved by an example model with and without ZTA components in varying usage scenarios.

8.1.4 Design for Evaluating Security Violations Detection

To evaluate the ability of the model to detect security violations we will be using Data Flow Analysis, as described in Section 2.4. We defined security violations in Section 7.4 which can happen in a ZTA. We evaluate whether our proposed model and security annotations are capable of detecting these violations. For this purpose, we introduce a total of five scenarios of the Media Store system with and without security issues. Issues may arise from wrong security annotations or missing authentication and/or authorization on the path to a protected resource. We define the evaluation scenarios as follows:

- **S0** - No security violations
- **S1** - Presence of unauthorized access
- **S2** - Presence of LPP violation
- **S3** - Missing user authentication factor
- **S4** - Lack of trust calculation

Following the evaluation approach, we ask the following question for this goal:

- **Q4.1** How accurately does the analysis detect introduced issues?

To be able to answer the questions we will be using the metrics of precision and recall. We define those metrics as follows:

- **M4.1** *Precision* - precision is the amount of true positive (*TP*) values to the sum of *TP* and false positives (*FP*) - $\frac{TP}{TP + FP}$. In our evaluation, *TP* are the correctly identified issues. *FP* are identified issues which were not introduced in the scenario. Therefore we calculate the ratio of the correctly identified issues in a scenario to the sum of correctly identified issues and identified issues which were not introduced in the scenario [40]. A high value for the precision metric ensures a higher credibility of the detection of issues.

- **M4.2** *Recall* - recall is calculated by dividing the *TP* by the sum of *TP* and false negatives *FN* - $\frac{TP}{TP + FN}$. In our case, *FN* are the issues which were present in a scenario but not identified. Hence, we calculate the ratio of correctly identified issues to the total amount of identified and unidentified issues [40]. The higher the value of the recall the better our analysis is at detecting introduced issues.

8.2 Evaluation Setup

To be able to answer the questions of goals G2, G3 and G4 presented in chapter 8, we need to apply the components of our ZTA repository to an already existing Palladio model. For this task, we have selected the case study of a Media Store. This model has been used to demonstrate the modelling power of Palladio and is also a running example used in the Palladio book and paper [43]. The model is suitable since it is a complete model of a system in the aspects that it has all of its components, interfaces, SEFFs and resource demands specified. This allows the model to produce response times and utilization of resources which simulate a real system behaviour. In the following section, an overview of the example model of the Media Store is presented. Then, we describe how we apply our ZTA according to the ZTA design principles by UK NCSC [57], described in Section 2.2. We then provide additional real-life context to the newly created Media Store with ZTA system by evaluating its maturity according to the CISA ZTA maturity model [11], described in Section 2.3.

8.2.1 Media Store

The Media Store example represents an audio storage system. The system allows users to access audio files stored in remote storage. The users are able to obtain a list of all available audio files on the server. The users can also upload files to the server as well as download files. The system also offers a packaging functionality. In the case of more than one file being downloaded, the system zips the requested files in a single package before sending it to the user.

Media Store Repository

In the initial repository of the Media Store, as shown in Figure 8.1, we observe the following components. A *FileStorage* component is modelled, to simulate the retrieving and storing of files on a server. It does so through its provided role of the *IFileStorage* interface. Next, we have the *MediaAccess* component which serves as a storage access component. It requires the *IFileStorage* interface and provides both the *IDownload* and *IMediaAccess* interfaces. Next, we have the *Packaging* component that simulates the zipping process of files. The component provides the *IPackaging*. Lastly, we have the *MediaManagement* component that serves as an access point to the system. To provide this functionality, the component provides the *IMediaManagement* interface. The interface consists of three signatures. They are *upload* which accepts a *FileContent* parameter, *download*, which accepts an *AudioCollectionRequest* parameter and returns *FileContent*, and *getFileList* which has no parameters.

Apart from the basic description of the components, we are not going to go into more detail and discuss thoroughly the SEFFs of each component and signature. A detailed modelling of

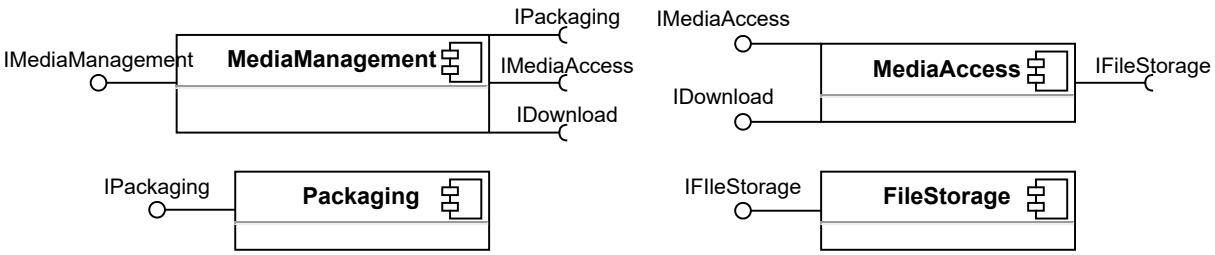


Figure 8.1: Components present in the initial Media Store repository

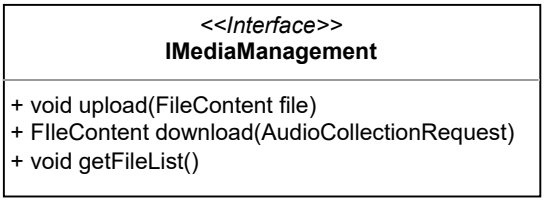


Figure 8.2: *IMediaManagement* interface from initial Media Store repository

the functionality of each media store component, as well as additional components which were not introduced here due to irrelevance, can be observed in the Palladio examples repository [38].

Media Store System

The initial cacheless Media Store System is presented in Figure 8.3. The *MediaManagement* component is instantiated and connected to an instance of the *Packaging* component with the *IPackage* interface. Next, the *MediaManagement* is connected to an instance of the *MediaAccess* component with both the *IDownload* and *IMediaAccess* interfaces. The *FileStorage* component is also instantiated and is connected to the *MediaAccess* through the *IFileStorage* interface. Lastly, the system offers an entry point through the *IMediaManagement* provided role.

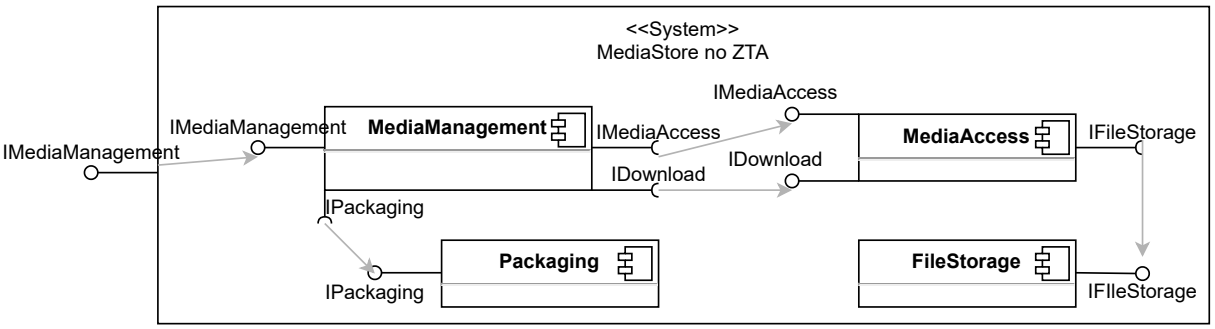


Figure 8.3: Initial Media Store assembly diagram

8.2.2 Integrating ZTA into the Media Store

Now we are going to integrate a ZTA into the Media Store system following the steps presented in the UK NSCS Zero Trust Guide [57], described in Section 2.2.

Knowing your architecture and assets

As a first step, we need to make sure that we are aware of the system's architecture namely the assets we are protecting and how they are accessed. When we look at the possible parts of the system we might want to protect we can identify:

- *File Access* - Audio files are digital assets. We might want to protect the *FileStorage* component.
- *Download Function* - Requests for multiple file downloads execute the zipping functionality of the system. Multiple requests to the *Packaging* component might result in a denial of service. Therefore, we might want to limit the downloading of multiple files only to permitted users.
- *Upload Function* - We might also want to restrict the uploading of files to the system only to allowed users.

Next, we take a look at the architecture of the system. In our case, we are protecting a legacy system from the Zero Trust point of view. This means that the model was developed without security in mind. Because of that it is harder to dissect the architecture in smaller pieces and establish multiple fine-grained defense perimeters without applying serious changes to the underlying model. We discuss this in more detail in the discussion part, Subsection 8.2.8. To be able to protect the discussed above assets we need to find a point to place a *PEP* to intercept the request to all of these assets. Such point is the *MediaManagement* component since it forwards requests for file access, downloading, uploading and file packaging. Thus, this is going to be the component where we will establish our defence parameter.

Since we have identified that we want to intercept requests to the *MediaManagement* components this is where we are placing our *PEP*. We want our *PEP* to forward each request to a *PE* for evaluation. Therefore, we select the *DuplexPEP* as a suitable *PEP* for this case. Then, due to the limitations discussed in paragraph 6.3.1, we need to include a gateway component. We cannot use the generic gateway modelled in the ZTA repository since it accesses the protected resource through the *IReosurce* interface and here the protected resources are accessed through the *IMediaManagement* interface. We create a new *MediaStoreGateway* component. The component resembles strongly the functionality of the generic gateway. This means that the SEFFs for each of its signatures follow the same execution:

1. Create a Zero Trust request from the received application-specific request and forward it to the *PDP* through the *IRequest* interface.
2. When a decision is received, if it is positive then forward the application-specific request to the protected functionality through the *IMediaManagement* interface. If the decision is negative - terminate the process.

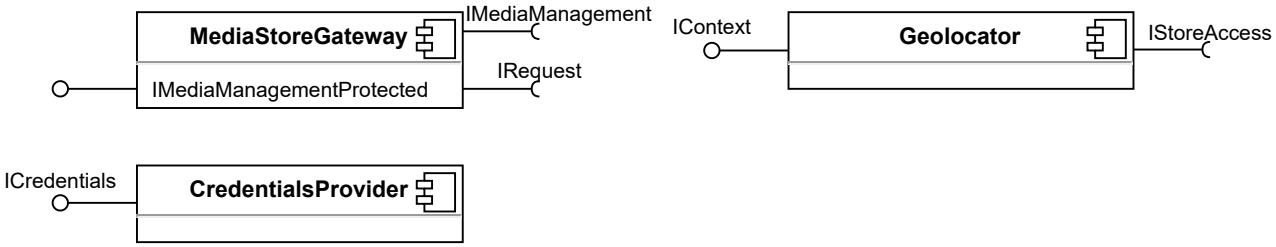


Figure 8.4: Basic components introduced during the ZTA integration

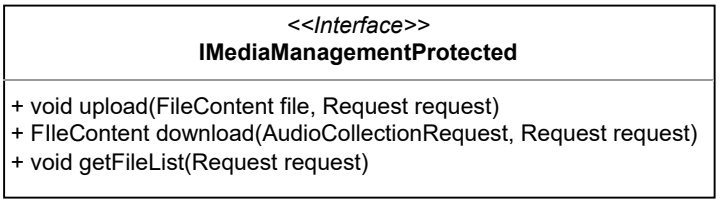


Figure 8.5: *IMediaManagementProtected* interface introduced during ZTA integration

Since the *IMediaManagment* interface only offers application-specific requests and its signatures do not accept Zero Trust requests as parameters, the gateway cannot simply provide it as an interface. If we do so, then the gateway will not have a source to obtain the required variables to create a request for the *PDD*. We solve this by introducing a new interface called *IMediaManagementProtected*, as shown in Figure 8.5. The interface has the same signatures as the *IMediaManagement*. However, its signatures are extended to also accept a *Request*(Section 6.1) parameter. This is the interface which our *MediaStoreGateway* will provide. In the end, we obtain the *MediaStoreGateway* as shown in Figure 8.4. Now that we have a gateway component we can create a composite component which will be our application-specific *PEP*. We create the composite component *MediaStorePEP*, as shown in Figure 8.6. Next, after we have modelled our *PEP* we can go ahead and integrate it into the Media Store system. We place the *PEP* in front of the *MediaManagement* and connect it to it through the *IMediaManagement* interface. The system now should provide as an entry point the interface *IMediaManagementProtected* instead of *IMediaManagement*. We delegate this provided role to the *MediaStorePEP*'s provided *IMediaManagementProtected* role. This is how we establish a perimeter to protect our identified assets: *File Access, Download Function and Upload Function*. The other required roles of the *PEP* will be discussed in the next steps of the ZTA integration process.

User and Device Identities

In the second step of the integration process, we need to make sure that we keep track of the identities of users and devices accessing the protected assets. To do this in our example system, we need to introduce databases where users' and devices' information is stored. These databases will later be used also for the authentication of users and devices and need to be compatible with our authentication components from the ZTA repository. Due to this fact, we are going to use the *Store* component from the ZTA repository. We instantiate two *Store* components, as shown in Figure 8.7, one for users and one for devices. We name them respectively - *UsersDB* and *DevicesDB*.

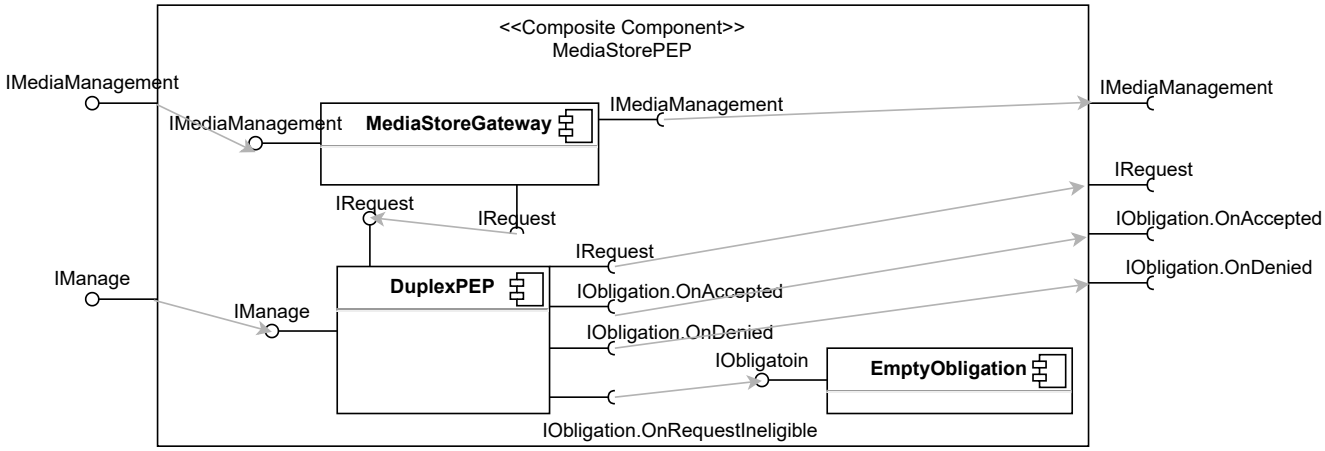


Figure 8.6: *MediaStorePEP* Composite component

Assess user behaviour and device health

As a next step of the process, we need to take into consideration the behaviour of users as well as the posture of devices when making decisions. To do this we are going to introduce a trust algorithm which forms its decision based on data about the user's behaviour and device state. In the ZTA repository, we have modelled a generic trust algorithm which combines data from two sources. We instantiate the component *TrustAlgorithm* in the Media Store assembly model. Now we need to define what we are going to observe when we assess user behaviour. Factors which can form a user behaviour can be logs from previous access requests by the user, the device's state and the user's current location. The state of a device can be assessed through constant logs it supplies to the system, as well as logs from previous requests made by the device. We already have integrated a logs source for the trust algorithm. Therefore, we can say that the call to the SIEM system also simulates the obtaining of information about the device's health. We can additionally include a second call to the SIEM system in the SEFF of the *TrustAlgorithm* component to increase the processing time. This however can be observed as inducing implementation details into the model. Therefore we will restrain from doing so. We will say that a single call to a logging system is enough for the trust algorithm to obtain its needed data about the user's behaviour as well as the device's health. These sources will be the input to our trust algorithm. Therefore we need two context providers which we will then connect to the required *IContext* roles of the *TrustAlgorithm* component.

First, we will discuss how we include the user's logged behaviour as well as device logs into the system. We need to be able to store logs somewhere. Similar to the users' and devices' information from the previous subsection we will need to include a database for logs. Therefore, we instantiate another *Store* component from our ZTA repository and name it *LogsDB*. Plain logs however may not be suitable for analysis since they might require pre-processing or correlation, see Subsection 6.4.4. We have already modelled such system which processes and analyzes logs namely the *SIEM* composite component from our ZTA repository. Thus we go ahead and instantiate this composite component into the Media Store system and connect it to the *LogsDB*. We also connect it to the *TrustAlgorithm* as a first context provider. We now have integrated assessing of user's logged behaviour in the system.

As a next part of the user's behaviour, we have selected to observe the location of the user at the time of request. Geolocation is not part of the core components of a ZTA. Furthermore, it is optional for a system and models of extracting a geolocation may vary from determining geolocation based on IP address to requesting data from a GPS tracker. This is why we have not modelled a general geolocation service in our core ZTA repository. If a component developer would like to have such a service integrated into their system they have to model it themselves in the repository of their system. Therefore, for our Media Store system, we will model a simple geolocation system. We base our model on the geolocation by IP approach namely the MaxMind GeoLite [33]. MaxMind GeoLite offers the user a database that contains geolocation data for IP addresses and is updated twice every week. The user simply downloads the database on their server and write their own service queries to the database to extract geolocation information. What we do in our system is to include another *Store* component into the Media Store assembly model and name it *GeolocationDB*. This is where the GeoLite database resides. Then in the Media Store repository, we create a *Geolocator* component which provides the *IContext* interface and requires the *IStoreAccess* interface. When the component is called, it simply makes a *retireve* call to its *IStoreAccess* required rolled and then in an internal action it simulates the mapping of IP to geolocation. We instantiate the *Geolocator* in the assembly model and connect it to the *GeolocationDB* as well as to the *TrustAlgorithm* as a second *IContext* provider.

Integrating Policies

As a next step in the process, we need to integrate policies and policy evaluation into the system. We instantiate and connect with each other a *PolicyEngine* component which will be responsible for policy evaluation and a *PolicyAdministrator* component which will create configurations based on the decisions and manage the *PEP*. Next, the *PolicyEngine* requires a policy source. From our ZTA repository, we take the *PoliciesProvider* component and instantiate it as a policies provider for the *PolicyEngine*. According to the documentation of the *PoliciesProvider*, see 6.3.2, we need to specify values for the *numberOfPolicies* and *rulesPerPolicy*. For this case, we will use example values and set both of the variables to 100.

Authorization based on context and authentication

The next part of the process is to take care of authentication as well as authorization based on request context. We need to make sure that we authenticate the users accessing the resources and the device they are using for this purpose. We also need to integrate additional factors such as user and device information and the authentication itself into the authorization process of the *PolicyEngine*.

We start by taking care of the user authentication. The UK NCSC guide [57] we are following requires that we integrate a Multi-Factor Authentication. Therefore, we need two instances of the modelled in the ZTA repository *Authenticator* component. The first instance will be responsible for the first factor. As per the documentation of the component, see Subsection 6.4.1, we set its *isLastFactor* variable to false. Then we also set its *AuthenticationProbability* variable as boolean PMF, where we define that in 90% of the cases, the authentication outcome will be true and in 10% it will be false. These values can be tuned later if we want to simulate different scenarios of the system - for example with more failed authentications. We also

connect the *Authenticator* to the *UsersDB* from paragraph 8.2.2. Now we need a credentials provider component. We specify this component in the Media Store repository. The component simply simulates the input of credentials by the user through an internal action in its SEFF. Then we instantiate the *CredentialsProivider* and connect it to our *Authenticator*. For the second factor, we use again an *Authenticator* component and set it up similarly to the component responsible for the first factor. The only difference here is that we set the *isLastFactor* variable to true. This is how we instantiate a 2-factor User Authentication in our Media Store assembly model.

For the device authentication, we are going to use a simple approach and not include multiple factors. In our ZTA repository, we have the *DeviceAuthenticatorWithStore* component which we are going to use here. We instantiate it and set its *AuthenitcationProbability* again to a boolean PMF with 90% of the authentication outcomes as true and 10% as false. Again these values can be tuned to simulate different scenarios. We connect the component to the *DevicesDB* from paragraph 8.2.2.

Since we now have user authentication, device authentication and a trust algorithm, see paragraph 8.2.2, which form the context of a user request, we can add context evaluation to our *PolicyEngine*. In our ZTA repository, we have modelled the *ContextEvaluator* component which is able to aggregate information from multiple context providers and simulate the process of context evaluation. We instantiate the component and connect it to the *PolicyEngine* component. Next, we connect the user authentication, the device authentication and the trust algorithm to its required *IContext* roles.

Monitoring

Lastly, we need to integrate monitoring into the system. We already have included an SIEM component that apart from providing context data can also collect and analyze logs from various components in the architecture. The component provides the *IObligation* interface, which when called simulates the process of logging data, see Subsection 6.4.4. Therefore, we are going to use the SIEM component as a monitor. From the documentation of the modelled ZTA components, see Chapter 6, we know that most of the components provide extension points through the *IObligation* interface. We can go ahead and connect each of the required *IObligation* roles of the *MediaStorePEP, PolicyAdministrator and PolicyEngine* to the *SIEM* component's *IObligation* provided role. We can now simulate the process of logging accepted or denied decisions, decisions made by the *PolicyEngine* and configurations made by the *PolicyAdministrator*.

8.2.3 Resource Environment

As a resource environment, we will be using the same resource environment of the initial Media Store model extended with only one additional container, as shown in figure 8.8. The environment consists of an *Application Server*, a *Database Server* and a linking resource - *Network*. We are extending the environment by adding a third container which is the *ClientDevice*. We define a processing resource for the new container with a *First Come First Serve* scheduling policy. Since the *ApplicationServer* provides a processing power of 1000 * 1000 * 1000 units and

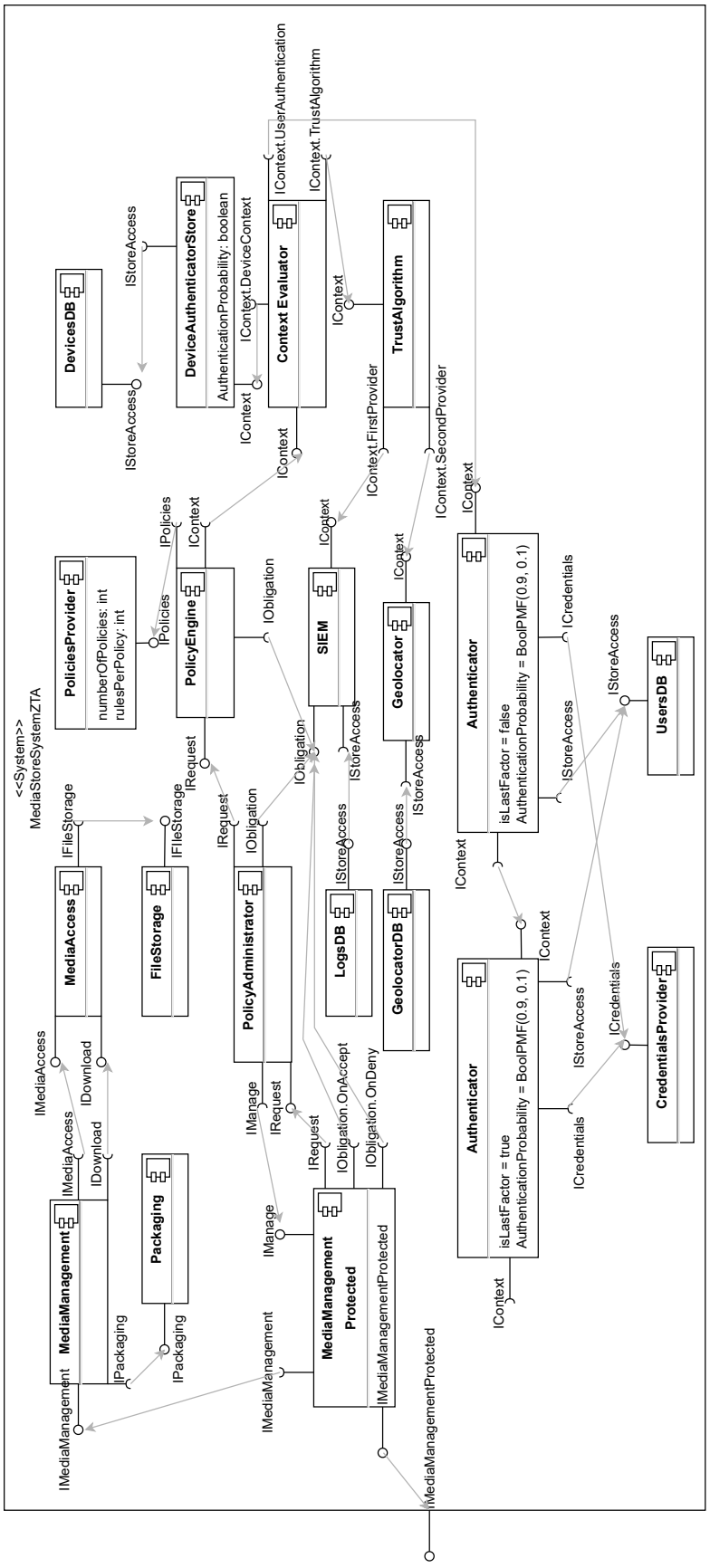


Figure 8.7: Media Store with ZTA Assembly

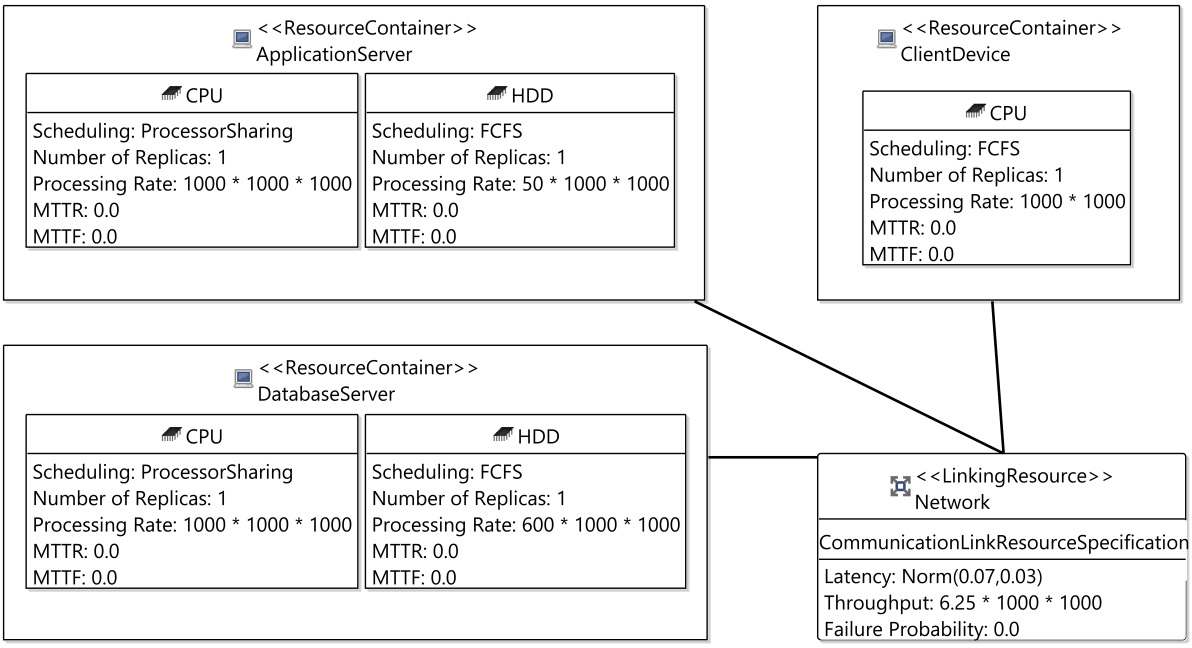


Figure 8.8: Media Store resource environment

a client device is normally less powerful than a server, we define the processing rate of the new container as only 1000 * 1000 units.

8.2.4 Allocation

When allocating the components of the new system we will not be changing the initial allocation of the Media Store. This means that on the *DatabaseServer* will reside the *FileStorage*. On the *ApplicationServer* are allocated the *MediaAccess, MediaManagement and Packaging*. The newly introduced components are allocated in the following way. Each instance of the *Store* component is allocated on the database server since these components represent a database. So the *UsersDB, DevicesDB, LogsDB, GeolocationDB* are all allocated on the *DatabaseServer*. On the newly introduced container *ClientDevice* we allocate the *CredentialsProvider* component. Next, we observe that components which perform some kind of functionality, such as zipping files or forwarding audio requests, are allocated on the *ApplicationServer*. Furthermore, we do not want to add additional resource containers for our core ZTA components since this would influence our evaluation. Since we are out of options, on the *ApplicationServer*, we allocate:

- *TrustAlgorithm*
- *PolicesProvider*
- *PolicyEngine*
- *PolicyAdministrator*
- *SIEM*

	RUM	DUM
Probability calling *download*	0.8	1.0
Probability calling *upload*	0.2	0.0
audioRequest.Count	IntPMF	4
audioRequest.Size	IntPMF	40568000

Table 8.1: *RUM* compared to *DUM*

- *ContextEvaluator*
- *Geolocator*
- *MediaStorePEP*
- *AuthenticatorFirstFactor*
- *AuthenticatorSecondfactor*

8.2.5 Usage model

The initial usage model of the Media Store defines a realistic usage of the system. We will adjust it slightly so it can be compatible with the new system and use it in our evaluation. Instead of calling the *MediaManagement*, we now use the *IMediaManagementProtected* interface as an entry point to the system. We also need to define additional variable usages to the ones from the initial usage model. We will define multiple variations of the initialization of the newly introduced variables to trigger different execution flows and analyse the performance impact of the new components. However, before using the realistic usage model (RUM) we will also derive another usage from the realistic one where we eliminate the probabilities present in the initial model. We call it a derived usage model (DUM). We do this to obtain a constant behaviour of the system and perform a more accurate comparison of the basic functionality of the initial system model and the one with the ZTA elements. For example, in the initial usage model, there is a probabilistic behaviour when it comes to calling one of the system entry calls. In our (DUM), we alter the probabilities so only the download system call is used. The upload functionality has weaker resource demands than the download. In a scenario where more uploads are called, it could result in improved performance although ZTA elements are present in the model and this improvement is not because of the ZTA elements but because of probabilities in the initial usage model. Following the same logic we alter the specifications for the input parameters to the download function from the initial model - the *audioRequest.Count* and the *audioRequest.Size*. We set the *audiotRequest.Count* parameter to always request 4 files and the *audioRequest.Size* to 40568000 which is one of the possible sizes from the *RUM*. In Table 8.1 the differences between the *RUM* and the *DUM* are summarized.

Next, we define the variations for the values of the new parameters required by the Zero Trust components - the *request.PolicyAuthorized, request.EvaluationEligible, request.PolicyAuthorized, request.ContextAuthorized, request.UserAuthenticated, request.DeviceAuthenitcated and request.Trusted.* In Table 8.2, we summarize the different configurations of the initial variable values. *UM* stands

	UM1	UM2	UM3	UM4	UM5
Authorized	false	false	false	false	false
EvaluationEligible	true	true	true	true	true
PolicyAuthorized	true	BPMF 9 1	BPMF 9 1	true	true
ContextAuthorized	false	false	false	true	false
UserAuthenitcated	false	false	false	false	false
DeviceAuthenitcated	false	false	false	false	true
Trusted	false	false	false	false	true

Table 8.2: ZTA variables configurations

	UM1	UM2	UM3	UM4	UM5
User Authentication Probability	true	true	BPMF 9 1	true	true
Device Authentication Probability	true	true	BPMF 9 1	true	true

Table 8.3: Configurations for the authentication probability of the device and user authentication components

for Usage Model and *BPMF 9 1* stands for BooleanPMF with 0.9 probability for *true* and 0.1 probability for *false*.

Since the component configuration of the device and user authentication components also impact the execution flow we summarize in table 8.3 the different configurations of the authentication probability for the device authenticating component and the first factor for the user authentication.

In *UM1* we set all requests to be policy-authorized and not context-authorized. This would result in the execution of the complete path where for every request each part of the context is evaluated. Since authentication probabilities are set to true each request should pass the evaluation and should be forwarded to the initial Media Store components. In *UM2* we have BPMF for the policy authorized value and this should result in requests dropped at the *PE* without evaluating their context. In *UM3* we again have the possibility of not evaluating context and directly dropping a request but there is also the possibility of evaluating context and dropping requests due to failed authentication since the authentication probabilities are now a BPMF. In *UM4* we simulate the scenario where all requests are policy- and context-authorized. This means no context evaluation is triggered and all requests reach the initial Media Store components. Lastly, in *UM5* we simulate the execution path where the device authenticator and trust algorithm are not used in the system.

To obtain a *RUM1* we combine the usage model configuration for *RUM* from Table 8.1, the variable values for *UM1* from Table 8.2 and the authentication probabilities for *UM1* from Table 8.3. To obtain the other *RUM2-5* as well as *DUM1-5* we follow the same logic.

8.2.6 ZTA Maturity Evaluation

In the following section, we are going to evaluate the created ZTA model of the Media Store according to CISA's Zero Trust Maturity Model [11]. As described inSection 2.3, the Maturity Model evaluates the Zero Trust level of different pillars of an agency. The model observes the

level in the aspects of *Identities, Devices, Networks, Applications and Workload and Data*. We will assess our model by looking at the same pillars and following the observed functions for each of these pillars. However, since the model is designed to assess an agency as a whole including its system architecture, infrastructure, personnel and practices, there are aspects which cannot be covered by our architectural model and therefore evaluated according to the maturity model. For example, the governance capabilities are connected to the practices and behaviour of an agency as a whole. In the authentication function of the identity pillar, it is stated that an agency implements identity policies and the difference between the initial and advanced levels is the level of automation and the frequency of policy updates. Our authentication subsystem does not represent automation or frequency of updates and since it is abstract we can make any claims about these attributes. Therefore an assessment about this aspect of the authentication system becomes quite subjective and it is inappropriate.

Identities

In our model, we have integrated a database for user identities. We also have authentication components and observe authentication on each access request. Therefore, we can assess the Zero Trust level of the Identities pillar of the maturity model.

Authentication - First, we need to assess authentication itself. Since we have integrated a 2-factor authentication and we trigger both of the factors on each request, we cover the traditional level of the authentication function. The initial level requires that additional attributes apart from the credentials such as locale or activity are used in the authentication process. The current model of Authenticator in our ZTA repository does not support context data. The next step to the advanced level is the phishing resistance of the factors, however, this quality of a factor cannot be represented with our model. The last step requires to provide again phishing-resistant as well as continuous authentication. Currently, our model does not support continuous authentication.

Identity Stores - Next, we observe the identity stores. In our system, we have one identity store defined which is the *UsersDB* component. This is the only store for user identities in our system and is deployed according to our allocation model on the database server. We can assume the server is located on-premise. Further, identity stores located on remote servers are not present. Therefore, according to the maturity model our architecture covers the traditional level.

Risk Assessment - The risk assessment function is a little bit more complex to assess since it cannot be expressed thoroughly in a Palladio model. Risk assessment might include heuristics evaluation or manual actions performed by a user. However, looking at the definitions for advanced and optimal levels of zero trust for this function we claim that the model covers them. We have integrated a logging functionality in the system and therefore may claim that information about user activity and their identity is logged throughout the execution. Furthermore, since authorization and authentication in the current assembly and usage model are based on probabilities, see paragraph 8.2.2 and Subsection 8.2.5, this means that an accepted request might be rejected on the next attempt. This simulates the dynamic adjustment of policies. To sum up, we have an automatic collection of user behaviour and dynamic policies, hence we cover the optimal level of this function according to its definition.

Access Management - In our scenario, access is evaluated per each request. No previously authorized requests are fed to the system and the *DuplexPEP* which we have integrated always forwards requests for evaluation before forwarding to a resource. Therefore, no permanent access is granted. Then, following the description of the function in the maturity model document, our model covers an advanced level of zero trust. We may claim even that an optimal level is covered since we perform request evaluation for each of the exposed-to-user functions (*download, upload and getFileList*). However, we may also argue that the defence perimeter is not fine-grained, see Subsection 8.2.8, since it has only one access point beyond which access is not continuously checked. That is why we are going to assess the level here as advanced rather than optimal.

Visibility and Analytics Capabilities - In the visibility and analytics definition of the identities pillar, it is required to collect activity logs and perform some analysis on these logs. As we discussed in the *Risk Assessment* function in paragraph 8.2.6, we have integrated a logging functionality in the model. Moreover, the *SIEM* component also has a *LogAnalyzer* component and therefore we simulate also this functionality. Hence, we can say that we cover the initial level for the visibility and analytics function.

Devices

As a next step of the assessment, we observe the devices pillar. In our model, we have integrated, similarly to the identities, a store component for device information as well as a component for device authentication. We also have included the client's device as an additional resource container which provides us with more possibilities to simulate the user's device monitoring, as described in the next subsections. Therefore, we can evaluate the Zero Trust level which is covered from the devices' point of view.

Policy Enforcement and Compliance Monitoring - According to the description of this function, policy enforcement and compliance monitoring across devices represent the functionalities of evaluating a device's state according to an agency's device policies and enforcing commands such as software or configuration updates. Currently, our model has only a *DevicesDB* where we keep track of enrolled devices. Furthermore, we claim that, since device evaluation is also part of the request evaluation process, we do have some partial policy enforcement on a device's state. A request might fail due to an unauthenticated device and the next one might succeed. In a real-life scenario what happens between requests might be the update of software which causes the successful authentication on the next request. This is an indirect device policy enforcement and is currently simulated with our Palladio model although it is not explicitly represented. Hence, our current model covers the traditional level in the Zero Trust model with visibility over devices and partial policy enforcement. *Resource Access* - In the resource access aspect of the devices pillar, it is mentioned the requirement for device evaluation upon access request. In our model it is represented this type of evaluation when making a request and therefore our architecture can be categorized at least one level above the traditional. However, which certain higher level our model categorizes as can be subjective. The initial level requires device evaluation upon request which our model covers and to achieve an advanced level an agency considers verified devices. If we assume that our client container is verified then we cover this level simply by making an assumption. Currently, we cannot represent the verified quality of a component in our model. The highest level requires real-time risk analysis within

the device. Device risk analysis algorithms were not part of the research for this project since they are out of the scope of the researched ZTAs. Therefore, they are not modelled in our ZTA repository. On the other hand, due to the probability of authenticating a device, see paragraph 8.2.2, we can make the claim that device evaluation sometimes fails due to internal risk analysis and hence cover the requirement of simulating device internal risk analysis. This claim can be observed as subjective. Due to these reasons we will not be giving the model the highest maturity level and will simply stick with the initial level of this function.

Remaining functions - Our model cannot be assessed according to the remaining functions of the pillar since they observe aspects which are out of the scope of the architectural modelling of a system. Device threat protection is a topic which spans out of the scope of ZTA modelling. Protecting a device could range from integrating an anti-virus system to policies about how to handle a device. Also, modelling a threat protection system with its variations would require separate research [56]. As we see in the maturity model on all levels threat protection is mentioned as capabilities, processes or solutions. No single architectural solution can be outlined and therefore it is difficult to represent it with general components from our ZTA repository. This is why we cannot make assessments about this function of the devices pillar. The visibility and analytics functions encompass methods for labelling, analysing and using device information. The automation and orchestration part observes the practices in the processes of provisioning, configuring and registering devices. These are aspects which we cannot represent in our Palladio model.

Network

The network pillar observes aspects of a network infrastructure and network technologies implemented to make it more secure The functions observe concepts such as the application of network rules, encryption on traffic, network configurations for availability demands and resilience and communication protocols. Those concepts cannot be represented in our current model and therefore we cannot assess these functions. The only thing for which we can perform an assessment is the network segmentation function. The function observes how granularly the network is divided into separate defence perimeters and protected. In our case, we have a single *PEP* that defines our perimeter. If we assume that the critical workloads are the functions of the media store and the file storage they can only be accessed through this *PEP* and therefore are isolated. We can say that our model covers the initial level of network segmentation. We discuss a finer network segmentation and our limitation in achieving it in the Discussion section Subsection 8.2.8.

Applications and Workloads

The applications and workloads pillar is concentrated on the functions of the underlying system which is protected by Zero Trust principles, In our case this is the Media Store and its initial components. The pillar encompasses aspects of development, testing and deployment processes and practices of applications. It observes also whether an application is exposed on a public network or hidden behind a VPN. Monitoring of the application behaviour is also included here but this would require us to modify the components of the Media Store and its core functionality which is out of the scope of this research. The application threat protection aspect is similar to the threat protection discussed in the devices pillar. Hence, our model does

not provide the required concepts to assess the zero trust maturity of this pillar. The only thing which we can observe is the application access function. Our model represents an access evaluation based on complex context evaluation and assigning of authorization levels based on the LPP. Therefore, it covers the advanced maturity level of this function. The model also simulates logging throughout the whole evaluation process and therefore we may state that we log usage patterns. Also, due to the probabilistic nature of our decision-making process, a request might first be accepted and in the next interaction denied or vice versa. From a subjective point of view, we can claim that this is due to the fact of logged and analysed usage behaviour and therefore we cover the highest optimal level of the maturity model. However, we will remain with the assessment that our model goes as high as the advanced level.

Data

The data pillar concentrates on how an agency handles its data. It observes the practices of inventorying and categorizing data, encrypting data as well as managing the lifecycle of data and policies connected to it. Again we cannot represent those aspects in our architectural model and therefore cannot asses the maturity model. However, we can assess the maturity of the functions Data Availability and Data Access since they observe the deployment of data and management of access to it.

Data Availability - Data availability assesses where data is deployed in an agency infrastructure and what backups it has. In our architectural model, all of our database components are allocated on a single database server. Therefore, we are storing data in a database which we can assume is on-premise. Therefore we can cover only the first level of maturity in this function.

Data Access - Data access observes what type of control is performed when accessing data. In our model, we are accessing files from the file storage. The access to the file storage happens only through a request intercepted by a *PEP* and forwarded to the *PE* where it is authorized according to the LPP. For the access, context is evaluated and decisions are only valid for a single request. Hence, in this aspect, we cover the advanced level of maturity. When it comes to the data obtained from the database, currently only the components from the Zero Trust infrastructure have access to this data. The databases are not exposed to outside access.

ZTA Maturity Evaluation Summarized

Table 8.4 presents a summarized version of which level was covered for the different pillar functions. As we discussed at the beginning of the section since the maturity model is designed to evaluate an agent as a whole and not only the architecture of a service it provides, there are concepts which we cannot cover with our model. Therefore, the functions which cannot be covered are omitted from the table. As we see from the table, for every function which we could evaluate with our model, the model covers at least the first level of maturity. In four functions *Identity Store, Identities Visibility and Analytics, Resource Access and Network segmentation* we covered the initial level of maturity. In the functions *Access Management, Application Access and Data Access* we achieved an advanced level of maturity. Lastly, only for the *Risk Assessment*, we managed to cover the optimal level. For the rest, the *Authentication, Policy Enforcement and Compliance Monitoring and Data Availability*, we cover the traditional level.

Discussion on Possible Improvements

For the functions *Authentication, Identity Stores, Policy Enforcement and Compliance Monitoring and Data Availability* from the maturity model, we discussed the borderlines between higher levels. In those borderlines, we identify possible improvements which can be applied to our model to achieve the higher level. In the following paragraph, we propose how we can achieve those levels in our model using Palladio and our proposed ZTA components. The possible improvements were discovered during the maturity evaluation of the model. We choose not to apply those improvements in the final model since this could be observed as tuning the model to improve evaluation results and reduce the validity of the evaluation. Furthermore, some of the improvements require editing parts of the initial Media Store model. We avoid these types of actions during the whole ZTA integration process.

Authentiation - Lack of support for context data in our *Authenitcatior* is one of the limitations to achieving a higher maturity level. However, this can be easily fixed. The *Authenticator* component can be extended with a required *IContext* interface and in its SEFF a call to this required interface should be added. The next limitation our model can theoretically overcome is the support for continuous authentication. Continuous authentication can be represented in our model. We can extend different components of the system with the *IContext* interface where we plug the authentication subsystem. Then in the SEFF diagram, we add somewhere on the execution path a branching action. One of the branches simply continues the execution without changes. The other branch makes a call to the *IContext* interface and triggers an authentication. Then we define the branch execution to be guarded by probabilities. In our case, for example, we can extend the *Packaging* and *MediaAccess* components following this pattern and simulate continuous authentication.

Identity Stores - In this function, we are limited to the traditional level of the maturity model due to only a single identities database. A scenario in which we achieve a greater level would be a case where we externalize the authentication subsystem on a separate server defining its own additional *Store* component as an additional identity store. This would cover the initial level of the maturity model where the system combines an on-premise identity store with one stored in a remote location.

Policy Enforcement and Compliance Monitoring - For this function, we cannot achieve a higher maturity level due to the lack of dedicated device policy enforcement and compliance monitoring subsystem. However, as we know so far, some of the basic tasks of Zero Trust are indeed policy enforcement and compliance monitoring and we have the components for those tasks and it is possible to model a such subsystem inside the model with ZTA. For example, we can include in our system a second PDP created again by combining the *PolicyEngine* and *PolicyAdministrator* components. We include also an additional *PolicyProvider* which represents only device policies or we can assume that an organization has a single policy source for all its activities and use the already instantiated one. Next, we create a simple component which we can call *DeviceConfigurator* and which provides the *IManage* interface and in its SEFF has a single internal action with the resource demands for updating a device. We connect this component directly to the PDP's *IManage* required interface. Then, we expose the new PDP's *IRequest* interface as a system entry point. In the allocation model, we can allocate the PDP again on the application server and the new device updating component on the client device container. Lastly, we need to change the usage model to include, with some probability, calls to

	Traditional	Initial	Advanced	Optimal
Identities				
Authentication	✓			
Identity Stores		✓		
Risk Assessment				✓
Access Management			✓	
Visibility and Analytics		✓		
Devices				
Policy Enforcement and Compliance Monitoring	✓			
Resource Access		✓		
Network				
Network segmentation		✓		
Applications and Workload				
Application Access			✓	
Data				
Data Availability	✓			
Data Access			✓	

Table 8.4: Covered Maturity levels of pillar functions

this new system entry point and we are done. This is a quick scheme of how we can simulate this policy enforcement and monitoring process for a higher maturity level using our Palladio components.

Data Availability - The borderline between our achieved level and the higher one in this function is the lack of distribution in storing system data. If we want to improve the level we can introduce additional database components and deploy them on an additional database container. Then in the SEFFs where we are accessing some kind of data, we can define a branching action where each branch requests data from a different data storing component. The branches are defined to be guarded by probabilities, for example, 90% of the requests go to the initial database server and 10% of the cases are forwarded to the backup server. By doing this we can now simulate redundancy of data storing and increase the maturity level.

8.2.7 Setups for Data Flow Analysis

To carry out the evaluation of the security analysis we need to prepare the scenarios mentioned in Section 8.1. Additionally, due to limitations presented by the Data Flow Analysis implementation, discussed in Section 8.5, we need to introduce new components as well as make changes to the Media Store with ZTA assembly model.

First, we remove the *SIEM* composite component and replace it with its building components. Next, we introduce copy components of the *DuplexPEP*, *MediaStoreGateway*, *Authenticator* and *ContextEvaluator*. The copy components mimic closely the functionality of their root component without including complex branching actions. Additionally, we apply changes to the behaviour of the copy *Authenticator* component. Currently, component variables are not

propagated alongside other variables in a SEFF. In our initial definition of the behaviour of the *Authenticator*, we use such a variable. However, we now replace the behaviour definition with a simpler one by excluding the component variable. We replace the root components in the Media Store with the ZTA assembly model with their copies. Now that we have prepared the model for evaluation we set it for the different scenarios in the following way. Where not mentioned explicitly we use the same configuration for the *.nodecharacteristics* file. We set the following node labels for the instance of the *MediaManagement* component:

- *Authorized.Level1*
- *Protected.protected*

Again, if not mentioned explicitly, we use the same usage model where we specify the user behaviour as a call to the *download* functions of the *IMediaManagementProtected* system entry point. We define the following data labels for the input *request* parameter:

- *Authorized.Level0*
- *MaxUserLevel.Level2*
- *ResourceRequiredLevel.Level1*

For *S0* we will not be applying any additional changes to introduce issues. For this scenario, we do not expect any issues to be identified.

S1 Unauthorized Access

For *S1* we want to test the detection of unauthorized access. Therefore, we introduce an issue in the model in the following way. We integrate into the system a new component which allows employees to manage the Media Store. We add a *PlatformManager* component to our repository which requires and provides the *IMediaManagement* interface. Since we assume that the component will be accessed only by employees and on the premises of the Media Store it is directly connected to the *MediaManagement* component to allow employees faster access by skipping the request evaluation process. We expose a new entry point to the system with the *IMediaManagement* interface and delegate it to the *PlatformManager* instance. In the usage model, we add a branching behaviour, where half of the requests are made to the *IMediaManagementProtected* and the other half are made to the *IMediaManagement* entry point. We alter the data labels of the *request* parameter for the call to the *IMediaManagement* interface as follows:

- *Authorized.Level0* - a user who does not have the required permission to access the resource makes a request.
- *UserAuthenticated.*, DeviceAuthenitcated.* and Trusted.** are all set to true. We assume that in the usage scenario, if a call is made to *IMediaManagement*, it comes from the premises of the Media Store and therefore the user and device are authenticated and eventually trusted.

S2 Violation of LPP

In the second scenario, we introduce a violation of the LPP. We use the setup of the assembly model from *S1* with the *PlatformManager* instantiated. This time, however, we change the behaviour of the *PlatformManager*. To allow for easier access to all resources, the component elevates the privileges of every request to the system's highest possible level. In our case, this is *Level3*. We also use the usage model from *S1*.

S3 Missing user authentication factor

In this scenario, we test the detection of incomplete authentication. This could result when multi-factor authentication is not implemented in a system. For this purpose, we set one of the authentication factor components to always return false. This would cause data to be propagated with only one authentication label.

S4 Missing device authentication and trust calculation

In the last scenario, we introduce two issues in our model. These are the lack of device authentication and trust calculation. To do so, we are going to disconnect the *ContextEvaluator, DeviceAuthenticationStore and TrustAlgorithm* components in the assembly model. The only context provider which we leave connected to the *PolicyEngine* is the *Authenticator*. This would result in trust and device labels not being included in the data flow labels.

8.2.8 Discussion

For the following evaluation, we are going to observe only one approach to integrate a ZTA although we have demonstrated in chapter 4 that there are multiple ways to do it. We will not be applying the additional ZTA to the Media Store because of its initial design. In contrast to the system from Chapter 4, where we designed it from the bottom up and considered the security aspect, in the Media Store case study security is not included in the initial design. The model was constructed to represent only the required aspects of a system for a certain evaluation and no more. Therefore an integration of a ZTA was not foreseen. To be able to include more complex ZTA in the aspect of segmentation where the system is divided into smaller parts each protected by its own *PEP*, we need to make significant changes to the underlying model. To be able to integrate a *PEP* directly in front of the *FileStorage*, for example, we need to alter all interfaces that are on the path to it to accept an additional parameter in its signature. Without the *Request* parameter we cannot forward the required by the *PEP* and PE parameters to simulate the ZTA workflow. Additionally, we need to change each SEFF of each function along the way to forward those parameters. As an alternative, we can adjust only the last interface before the *FileStorage* which is the *IFileStorage* to accept a parameter and then in the SEFF of the *MediaAccess* component define the needed variable usages. This, however, would prevent us from defining different ZTA scenarios in the usage model since we cannot now set the *Request* fields in system entry point calls.

8.3 Evaluation Results

In the following section, we discuss the results obtained for the evaluation defined in Section 8.1.

8.3.1 Discussion on Model Completeness

We start with the first goal of evaluation in which we observe to what extent we have managed to model the ZTA components using Palladio. We want to see whether we were able to cover all tasks and logical components identified in Chapter 5. We also evaluate whether those modelled elements are representative enough to be applied to various ZTAs suggested in the literature.

Discussion on Q1.1

We have outlined *Policy Enforcement Point, Policy Engine, Policy Administrator and Context Provider*. In our repository from Chapter 6 we were able to model two types of *Policy Enforcement Points* to cover variations as described in Subsection 6.3.1, a *PolicyAdministrator* component and a *PolicyEngine*. When it comes to Context Providers, they cannot be represented with a single component since they are heterogeneous and may also play additional roles apart from providing context. What we did was to provide the needed interface *IContext* and patterns which demonstrate how a Context Provider can be modelled or adjusted to be compatible with the rest of the architecture. Examples of such Context Providers are the *Authenitcator, SIEM, Trust Algorithm, etc.*. Following these thoughts, we can say that we were able to cover 100% of the identified logical components using Palladio elements. When we look at the task which we identified in Section 5.2, we see that we were able to represent them all in our model. The tasks of intercepting a request, forwarding it to a decision point, and enforcing the decision are all modelled inside the created *PEP*s. The evaluation of policies and making a decision is simulated in the *PolicyEngine* component. The various Context Providers supply context data to a *PE*. Lastly, the tasks of creating configurations and distributing them amongst interceptors are represented in the *PolicyAdministrator*. This is why we claim that we have modelled 100% of the identified tasks in a ZTA using Palladio.

The next metric of this question is the number of adjustments needed by Palladio to model a ZTA. Although there were some obstacles, for example forwarding application-specific requests as described in Subsection 6.3.1 or calling multiple Context Providers as in Subsection 6.3.2, they were merely a limitation in the fine-tuning of the model to be as simple as possible without losing its representative power. We were able to find workarounds in the face of adding a component, as in paragraph 6.3.1 and providing patterns to overcome such obstacles, as the *ContextAggregator* in Subsection 6.3.2. Therefore, we were able to fully model all of the identified ZTA logical components and simulate the tasks in a ZTA without applying any changes to the underlying Palladio model.

8.3.2 Discussion on Model Applicability

In the second goal, we observe the applicability of the proposed model. We look at the generality of the proposed components. We also observe what changes our model as well as the model, where we integrate our components, require to be compatible with each other.

Discussion on Q2.1

We observe how we can map suggested ZTA models from the literature to components from our repository. We look at the systems presented in Section 3.2 and we exclude the models of BeyondCorp and SDP since they were used in the modelling process of the general components. We will present and describe diagrams where we have mapped components mentioned in the papers of the observed systems to components of our repository. The names used for the components are the same as the names mentioned in the papers to make the mapping more intuitive and easier to read. Where the name of the component is too application-specific and cannot be intuitively mapped to an element from the repository we mentioned in brackets under it what component from our repository we are using for it. We have removed some of the unplugged interfaces of the components from the diagrams to increase their simplicity. However, this does not mean that components have been changed in any way and if the presented systems are instantiated in the Palladio then everything as per the component's documentation from Chapter 6 will be present. If there are components which are not modelled in our repository but can be modelled using one of the patterns discussed in Chapter 6 it will be noted in the diagram description.

We begin by observing the system presented in the paper of Biplob and Muzafar [39]. The system consists of an Authorizer, Authenticator, Endpoint Compliance Management (ECM) system and Enterprise Discovery Service. Received requests are authenticated by the Authenticator and then evaluated by the Authorizer based on ECM information. The mapping of the system to ZTA components is displayed in Figure 8.9. Since requests are intercepted and authenticated first we can use a similar approach to the BeyondCorp modelling 6.5.2 and therefore use a *DuplexPEP* combined with *OnAuthenticationMissing* component which redirects the request to an *Authenticator* which we take directly from our repository. The *OnAuthenticationMissing* component can be defined additionally by a developer or taken from our BeyondCorp repository. The *PE* of the system is the *Authorizer* and for this component, we instantiate a *PolicyEngine* component from our repository. Since the ECM provides policies for authorization and also monitors devices in the system it can modelled with a composite component which has instances of a *PolicyProvider* and *SIEM* in it. According to the paper device monitoring is triggered periodically and therefore we expose the ECM's *IContext* interface which comes from the contained inside *SIEM* as an entry point to the system and in the usage model we define the probability of calling the device monitoring functionality. The model also exposes the *IRequest* interface to intercept requests. The mentioned in the paper Enterprise Discovery System is not mapped here since apart from mentioning it no more details about its integration in the whole process are described in the paper.

The next system we discuss is the intelligent ZTA, which integrates machine learning in its components, proposed by Ramezanpur et al. in [42]. In the architecture, the authors outline an Intelligent agent/portal (IGP), Intelligent Network Security State Analysis (INSSA) and Intelligent Policy Engine (IPE). Further, they mention multiple sources of policies such as Data Access Policy (DAP), Public Key Infrastructure (PKI), ID Management and Industry Compliance systems. There is also a SIEM system, activity logs and threat intelligence which supply data to the INSSA for analysis. We present the mapping of the system in Figure 8.10. The IPE is the component which plays the role of a *PE* and therefore we map it directly to the *PolicyEngine*. As per the paper, the DAP, PKI and Industry Compliance components provide policies and

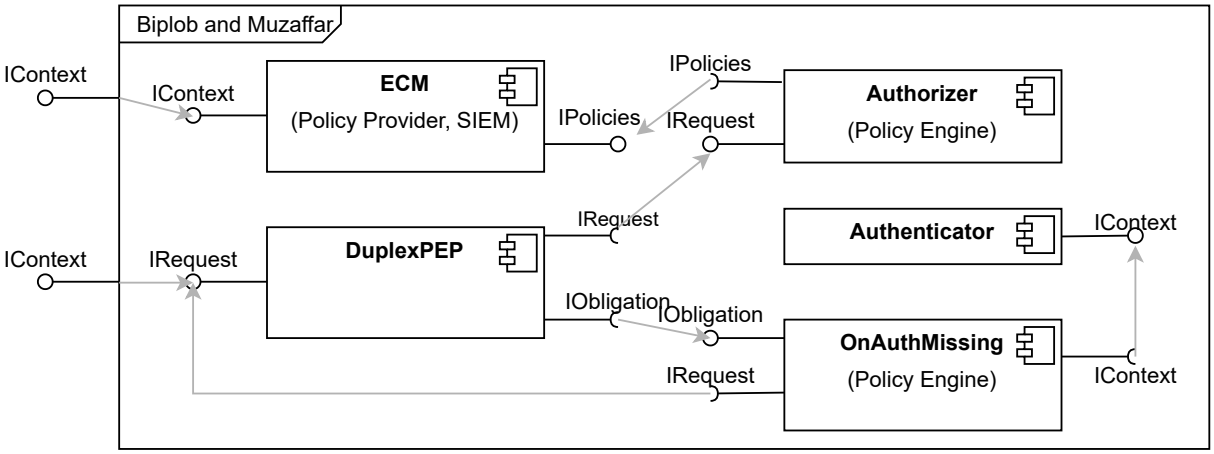


Figure 8.9: Biplob and Muzafar model [39] mapped to ZTA components

therefore we can instantiate them using the *PoliciesProvider* component. Since we cannot plug multiple policy providers into the *IPolicies* requiring role of the IPE we need an additional component *PoliciesAggregator*. This component can be modelled by following the pattern for creating a *ContextAggregator*, see Subsection 6.3.2. The INSSA component resembles the functionality of a Trust algorithm and therefore we can use the *TrustAlgorithm* component for it. We can map the SIEM directly to the *SIEM* component from the repository and for the Threat intelligence which can be mapped to the role of a Context Provider, a developer can specify a new component which provides the *IContext* interface. Next, since the IGP component observes devices and feeds data to the *PE* about them we can associate it with a Device Authenticator. In our repository, we have modelled two device authenticating components and for this case, we can use the *DeviceAuthenticatorWithLogs* component which obtains the device's data from the device directly and from a database. Lastly, a *ContextAggregator* is needed to be able to connect all of the Context Providers to the *PolicyEngine*. What makes this suggested model different is the fact that it integrates machine learning. This can also be represented in our components. The IPE (PolicyEngine), IGP (DeviceAuthenitcator) and INSSA (TrustAlgorithm) all perform an internal action inside their SEFF. A developer can edit this internal action by setting it to a resource demand estimated for their machine learning algorithm. By doing this, the performance impact of a machine learning algorithm can be introduced into the model.

Then we look at the system suggested by Chen et al. [10]. The components which the authors outline in the architecture are an Access Control Proxy (ACP), an Identity Management System and a Trust Evaluation System. The Trust Evaluation uses a Vulnerability Database (VDB), Cybersecurity Event Ledger (CEL) and Anomalous Behaviour Detector (ABD) for calculating the trust. Lastly, there is the Security Policy Engine (SPE), which makes the final decision of allowing or denying a request. The mapping of the system is presented in Figure 8.11. We first map the ACP to a *PEP* and according to the description of the ACP, we can use a *DuplexPEP* which forwards requests to a *PolicyEngine*. Then, we map the SPE to a *PolicyEngine* component. No policy provider has been discussed in this paper and therefore we leave this aspect of the architecture unmapped. Next, as per the description of the architecture, user requests are authenticated and trust evaluation is performed. Therefore we can go ahead and use the *ContextEvaluator* component from our repository to connect an *Authenticator* component and

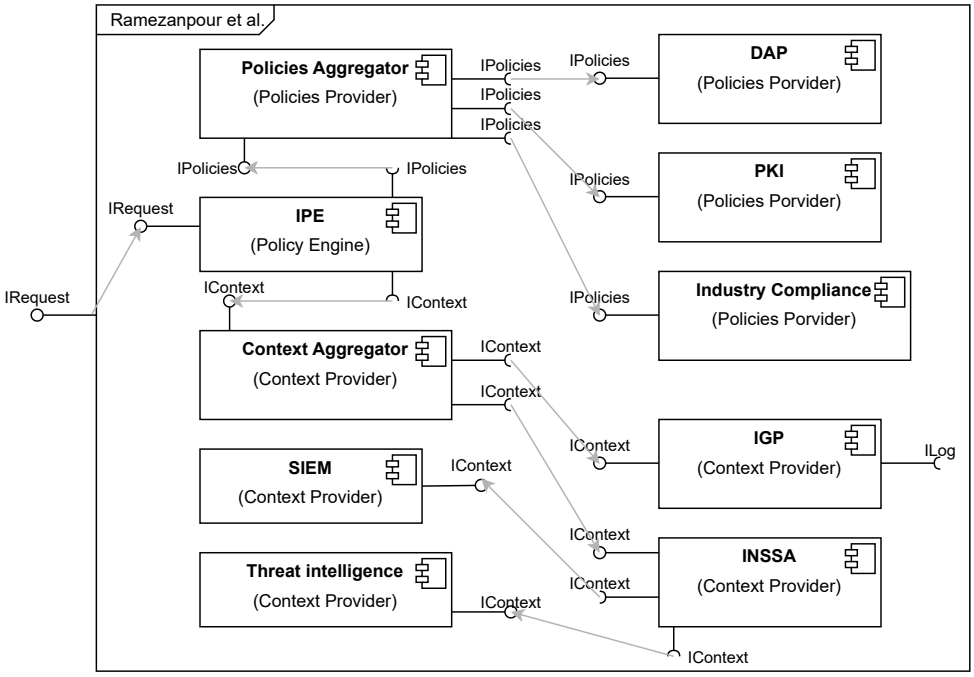


Figure 8.10: Ramenzapour et al. model [42] mapped to ZTA components

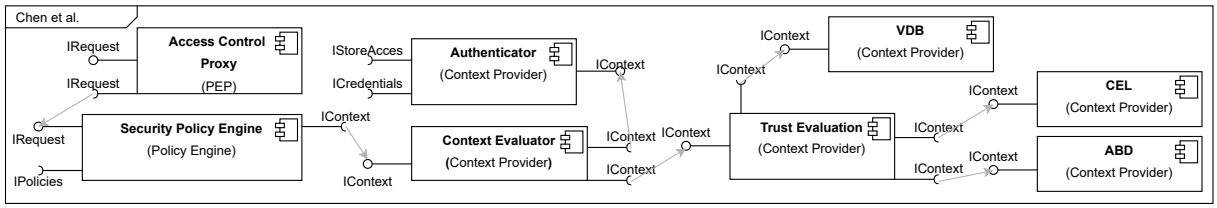


Figure 8.11: Chen et al. model [10] mapped to ZTA components

a *TrustAlgorithm* component to the SPE. At this point, the *TrustAlgorithm* from our repository is not fully suitable since we have modelled it to obtain input from only two Context Providers and in this paper are mentioned three. However, we have discussed this issue already in the modelling chapter and have described how our *TrustAlgorithm* component can be altered to accept more inputs. We can apply this method here to make the *TrustAlgorithm* compatible. Lastly, the sources for it are components which we have not specified in our repository. However, they can easily be mapped to Context Providers and therefore be modelled as basic components and made compatible with the use of the *IContext* interface.
Lastly, we observe the system of Lee et al. [31]. In the description of the architecture, we find the components WiFi base station, Context Handler, Policy Database, Subject/Object Database, Risk Evaluation Function, Access Decision Function, Environment Evaluation Function and Firewall Provisioning. The Environment Evaluation Function has as subcomponents mentioned a Security Situational Awareness (SSA) and Location. However, not enough data is provided to map the SSA component more precisely to one of our components and we can only say it is a Context Provider. The location service is also not fully specified and cannot be modelled. However, we have demonstrated in paragraph 8.2.2 how a location service can be represented

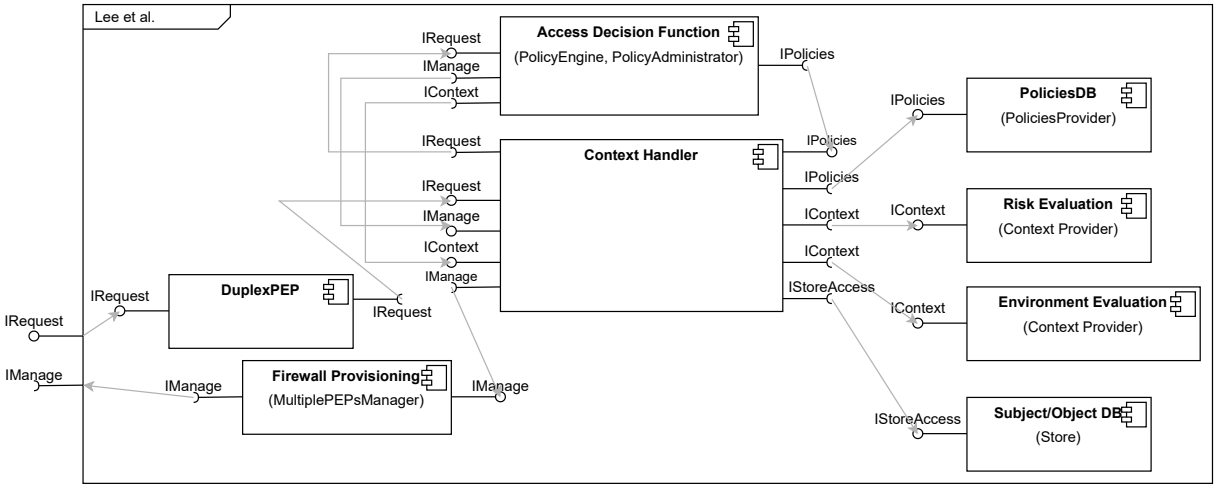


Figure 8.12: Lee et al. model [31] mapped to ZTA components

with our components. We will consider both components as part of the Environment Evaluation and display only this component in our diagram. The Context Handler is another component which we have to discuss before we start mapping. As per the description, the Context Handler navigates the process or in other words in what order other components are called. This type of component can be easily modelled using Palladio and the defined interfaces in our repository. We have modelled a Context Handler for the model of the SDP architecture, see 6.5.1. Similar to this one can be modelled for the system of Chen et al as shown in figure 8.12. We start the mapping with the WiFi base station which intercepts requests and hence we use for it a *DuplexPEP*. The Access Decision Function we can map to a composite component consisting of a *PolicyEngine* and a *PolicyAdministrator*. Policies are provided by the Policies Database component and therefore we map it to the *PoliciesProvider*. Since the Access Control Function uses the Risk Evaluation and Environment Evaluation components to make decisions, we map them to Context Provider components and model them as basic components which provide the *IContext* interface. Lastly, to represent the provisioning of firewalls we map the Firewall Provisioning component to a *Multiple*PEP*sManager* component. The full mapping of the system can be observed in Figure 8.12.

Discussion on Q2.2

We observe the quality of the components to be used off-the-shelf. This means that we can take an element from our repository and include it in its initial form in the existing model without altering its required or provided interfaces as well as its SEFF. We will be observing the assembly model of the Media Store, as shown in figure 8.7. We have instantiated a *MediaStorePEP* which is a composite component containing a *DuplexPEP* and a *MediaStoreGateway*. The *DuplexPEP* is instantiated directly without any changes. The *MediaStoreGateway* is a newly introduced component which we will discuss in the second question of **G2** and not here. Next, we have the *PolicyAdministrator and PolicyEngine* components. Both of them are also used in the assembly model without applying any changes to them. The *PolicyProvider* component, on the other hand, requires minimal adjustment. We have instantiated it as it is and then we have

specified values for the variables *numberOfPolicies and rulesPerPolicy*. Apart from these variable specifications, no other changes were made to the component. The *SIEM, TrustAlgorithm and ContextEvaluator* components are also instantiated without applying any changes. Then we have the *DeviceAuthenitcator* component and the two *Authenticator* instances - *FirstFactor and SecondFactor*. The three components were adjusted after being instantiated. For the *DeviceAuthenticator* we have specified a value for the *AuthenticationProbability* variable. For both *FirstFactor and SecondFactor* components, apart from the *AuthenticationProbability*, we have also defined the *isLastFactor* variable. One last change needs to be made to an *Authenticator* component and that is the definition of the security node characteristic *ComponentConfiguration.First and ComponentConfiguration.SecondFacotr* which describes which label in a request the component may alter. All of the *Store* instances which represent the databases in a ZTA are used in their initial version. These are all of the components in the Media Store assembly which are also part of the core ZTA repository. As we see most of them can be used directly in a system without changing required and provided roles and editing SEFF. For three of the components - the *DeviceAuthenticator, Authenticator and PolicyProvider* - minor adjustments need to be applied. Hence, we can claim that components from our ZTA repository may be used off-the-shelf.

Discussion on Q2.3

In the third question for *G2*, we look at the system into which we integrate a ZTA. We want to evaluate how much of a change needs to be applied in order to be made compatible with the elements from the ZTA repository. This is why we will observe first the number of changes to existing components and interfaces of the system and then the number of newly introduced components and interfaces in the system's repository. First, we start by examining the components which were in the initial version of the Media Store. These are the components *MediaManagement, Packaging, MediaAccess and FileStorage*. During the integration of the ZTA, none of them were altered in any way. This is because, in the current architecture of Media Store with ZTA, the process of evaluating a request happens prior to any processes performed by the initial Media Store model. Moreover, the chosen approach to create a greater defence perimeter and not multiple finer ones also contributes to the lack of modifications to initial components. However, as we have discussed in the Discussion section of the Evaluation Preparation chapter if we were to integrate *PEP*s in front of the *Packaging* component or the *FileStorage* then we would have had to modify each component on the way to these *PEP*s to be able to forward Zero Trust requests. This means that the interfaces should have been modified to accept an additional parameter and the SEFF diagrams should also be edited to forward these parameters. In the scenario of a Media Store with *PEP*s in front of the *Packaging* and *FileStorage* components this would require a change to the *IPakcaging, IMediaManagement, IMediaAccess, IDownload and IFileStorage* interfaces and changes in all SEFFs of the *MediaManagement and MediaAccess* components.

In the second metric of the question, we examine the new elements which were modelled inside the Media Store repository to enable integration of the components from the ZTA repository. We have modelled a total of three new elements which are crucial for integrating a ZTA. These are the *MediaStoreGateway, CredentialsProvider and IMediaManagementProtected*. The composite component *MediaStorePEP*, although also new for the repository can be represented

again from its building components directly in the assembly and therefore is not mandatory. The *Geolocator* component was also created for the ZTA integration, however, it is not part of the core elements of a ZTA. A ZTA may exist without a location service and hence this one is again not mandatory. In the theoretical scenario mentioned in the previous metric where we have *PEP*s in front of the *FileStorage* and *Packaging* components, we would again need to specify a protected version of the interface that is hidden behind the *PEP* and a gateway component per each protected interface. So in general if the system does not allow the use of the *GenericGateway* from the ZTA repository then always two new elements should be added per protected interface.

Final Discussion on G2

In conclusion, as we see from the mappings of the system we were able to reproduce the suggested models using only components, interface and modelling patterns described in our repository from 6. In cases where we did not have an already created component to which we could directly map we described how a new one can be introduced without needing to alter anything in our already present core elements. Therefore we can conclude that our components are general enough to represent various suggested ZTAs.

We saw that in the evaluation model where we have specified a broader defence perimeter we need to apply minimal changes to the underlying system to make it compatible with our ZTA components. The need for specifying a gateway is also due to the limitation by Palladio to forward whole variables. Hence in this case we can state that our ZTA model is easily applicable. On the other hand, when finer defence perimeters need to be established the underlying system requires more changes. But when a system was not designed with a certain concept in mind which later is decided to be added to a system it is normal to require changes in the core model. This is one of the main concerns which we mention in our introduction - flaws in the design lead to more required effort in later changes. In our case, the Media Store model was not intended to integrate a ZTA in its initial design and therefore makes further adjustments more costly.

8.3.3 Discussion on Performance Analysis

In the third goal of the evaluation, we want to observe whether the ZTA components introduce a performance impact in an existing model and as a consequence allow the evaluation of performance after adding a ZTA to a system. We also want to evaluate the flexibility of the performance analysis to represent performance differences in scenarios where different execution flows of the request evaluation process are triggered. We will be using the derived and initial usage models as described in 8.2.5. We will first execute simulations for the Media Store systems without the Zero Trust components in order to obtain the performance of a system without ZTA and later compare it to the simulation results of the same system but with ZTA integrated. We first start by analysing the results from executing the derived usage model before analysing the realistic one. We will be analysing the Cumulative Distribution Functions (CDF) of the usage model runs which describe the probability (y-axis) of response time to be equal to or less than a certain value (x-axis). We look at histograms which display

the probabilities of different execution times. We also examine XY plots which present on the x-axis a point in execution time and on the y-axis the response time at this point in execution time. Response time is measured in seconds.

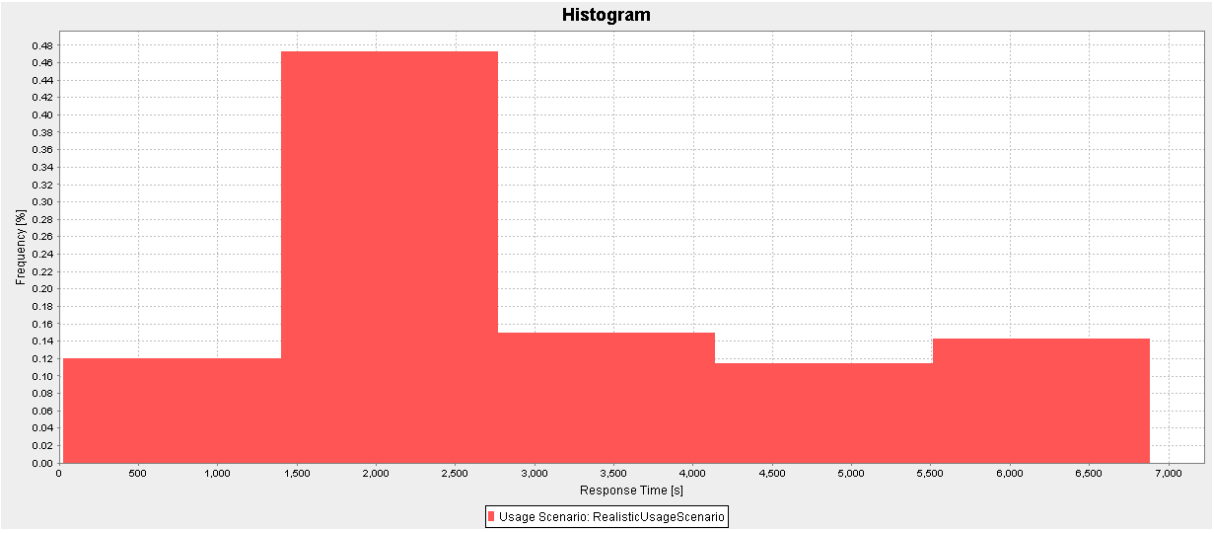
Derived Usage Model

We begin the analysis with the derived usage model executed on the Media Store version without ZTA and in Figure 8.13a it is displayed the CDF of the derived usage model of the Media Store without a ZTA. The chart shows us that the worst execution time achieved is around 7.000 seconds and the probability of execution time increases drastically at around 2.000 seconds. This means that half of the requests have achieved a time of less than or equal to 2.000 seconds response time. Next, we execute the different usage models presented in Subsection 8.2.5 on the Media Store model with ZTA and compare them.

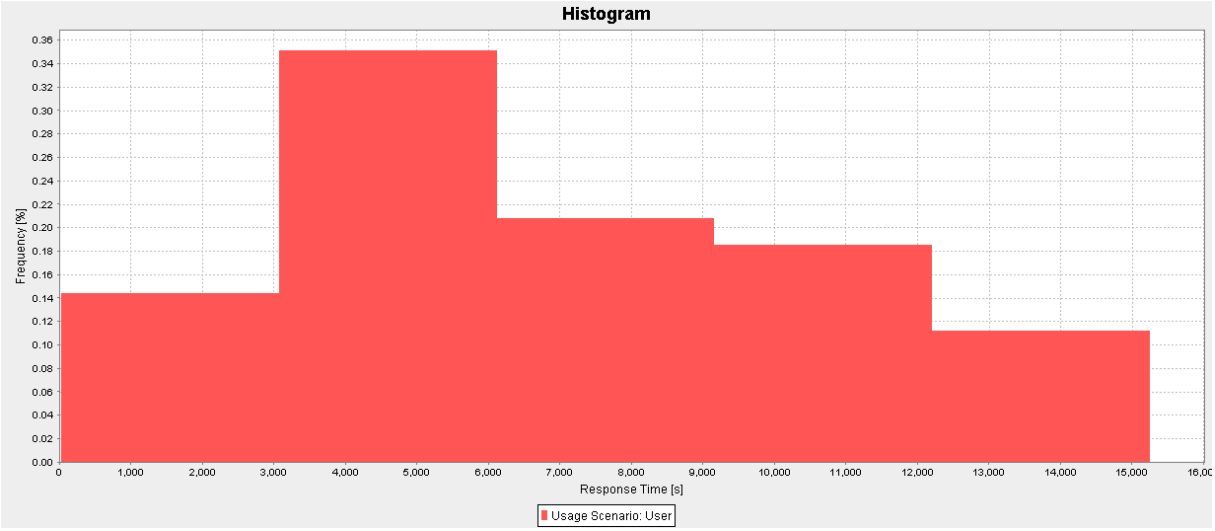
In **DUM1**, all of the requests are policy-authorized and require context evaluation. According to the configuration of the device and user authentication components for this usage model, the requests are always successfully authenticated. All requests should pass the evaluation successfully and be forwarded to the media store components. Therefore each request should traverse the full execution path and all of the internal actions should be triggered. This means, that we are expecting the worst response time to be increased since additionally to the executed actions from the initial store model, the execution triggers the newly introduced actions from the ZTA components. As we see from the CDF in Figure 8.17b, the worst execution time increases above 15.000 seconds. Additionally, we have a more gradual increase in the probability of achieving an execution time lower than a value. From the histograms of both executions, we see that the execution time with the highest probability is between 3.000 seconds and 6.000 seconds and in the DUM without ZTA is between 1.500 seconds and 2.500 seconds. The performance impact of the new components can be detected using the simulation.

In **DUM2**, we now introduce the possibility of having requests which are not policy-authorized and therefore the system drops them earlier without further processing. We still expect that the worst execution time should be increased compared to the default usage model since accepted requests still traverse the full execution path. From the CDF of the execution of the second usage model, shown in Figure 8.17c, we see that indeed the worst execution time increases but with less compared to usage model 1. In the XY plots in Figure 8.16 of usage models 1 and 2 we can observe the points in execution time where requests are dropped and the response times for these are significantly less. The overall better system performance might be due to the fact that dropped requests induce less resource utilization and resource containers have more resources to process authorized requests.

In the execution of **RUM3**, we observe an abnormal behaviour of the system. The third usage model introduces probabilities in the authentication of the user and device. This means that there are requests for either the user or device or both that are not successfully authenticated and the request is dropped before being sent to the initial Media Store components. However, there are still requests which reach the initial Media Store components and therefore we expect the worst response time to be similar to the time from the DUM2. However, the result presented in Figure 8.18a shows a completely different outcome where the response time significantly drops. We interpret that this might be due to the higher uncertainty in a request being dropped because compared to usage model 2, here apart from the probability of dropping



(a) Default DUM



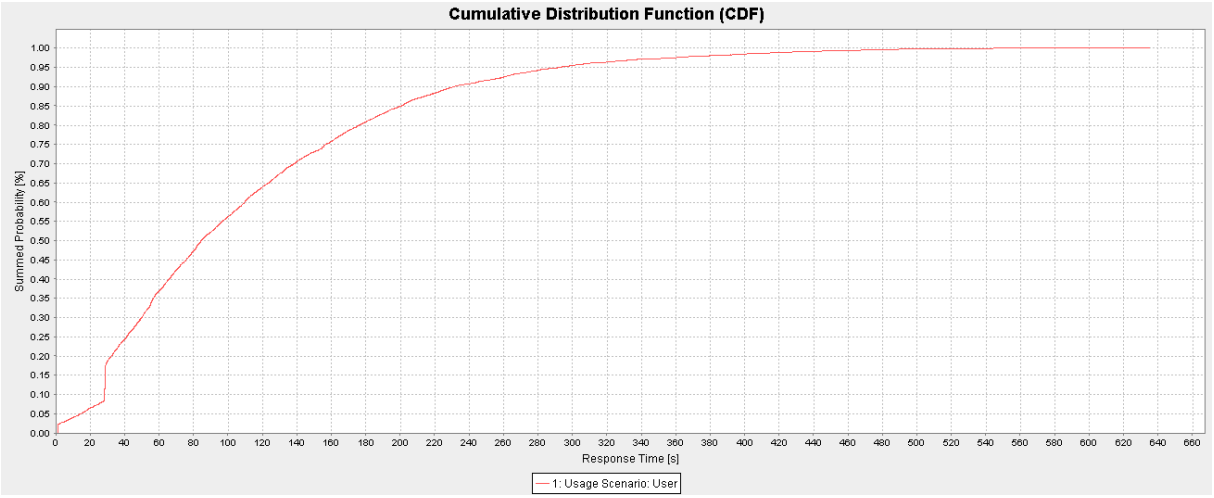
(b) DUM1

Figure 8.13: Probabilities of the execution time of default DUM and DUM1

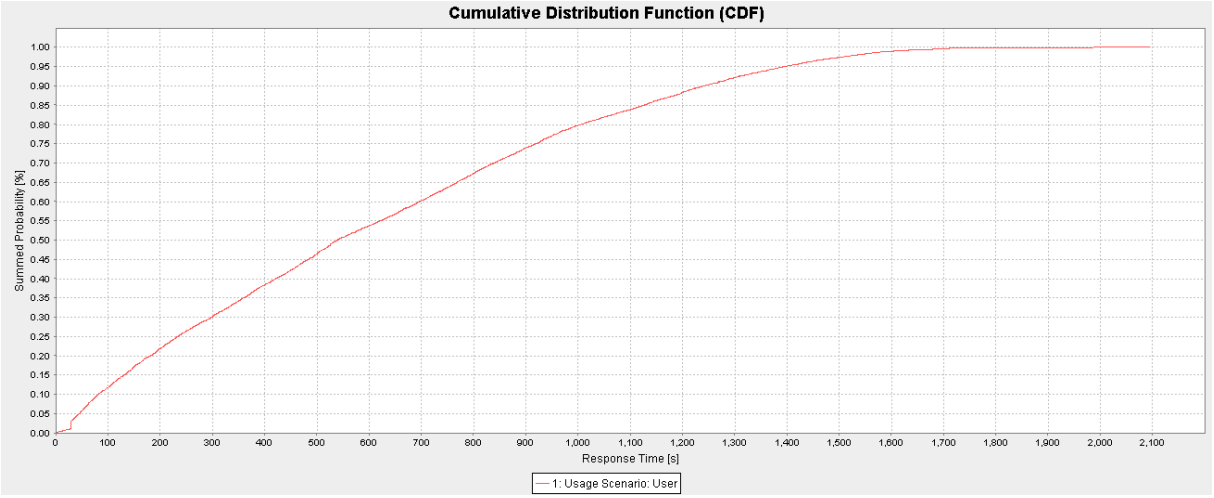
a request due to not being policy authorized, a request may be dropped also due to not being authenticated successfully. Therefore, we eliminate this uncertainty and run some additional simulations using usage models derived from usage model 3. As said we remove the BPMF value of the *PolicyAuthorized* variable and set it to always true and run a simulation. Then we run simulations also for the cases where we have a BPMF only for the *AuthenitcationProbability* of the *DeviceAuthenticator* component and one only for the *AuthenitcationProbability* of the *Authenticator* component. We do this to determine whether one of these components is causing the unusual behaviour. Lastly, we play with the probabilities of the BPMF functions, to see how they affect the outcome of the simulation. In Figure 8.14a are shown the results of the previously mentioned variations of usage model 3. As we see from the result, even if we eliminate the probability in the *PolicyAuthorized* variable, Figure 8.14a, the simulations still yield low response times. If we limit even the probabilities to one of the authentication components in order to check if one of them is causing this time decrease we again obtain really low response times as shown in Figure 8.14b and Figure 8.14c. Next, we tuned the probabilities of authentication to check how these affect the outcome. According to the results, when see set the probabilities to a successful authentication probability of 50% and thus limit the percentage of requests sent to the initial Media Store components, we see that the system starts behaving even faster. If we increase the probability to 99% of successful authentications then the system yields a closer response time to the one achieved in the other usage models. A concern emerged, when first observing the results of **DUM3**, that there is a possible error in the model which is triggered by setting authentication probabilities of the authentication components. Failed authentications cause the execution to not forward requests completely. This would reduce response time. Palladio offers the ability to trace what calls to what components were performed. When observing the calls made in the different variations of **DUM3**, we saw that requests were actually forwarded to the initial Media Store components and no calls were skipped. Theoretically, the response time should have been slower. Currently, we cannot provide a logical explanation for why this behaviour of the system is present. This indicates that maybe the model is not suitable for playing with authentication probabilities.

In **DUM4**, we evaluate the scenario where no context evaluation is performed. It is a valid usage model for systems which are at the beginning of their ZTA maturity and depend only on policy authorization. From the resulting CDF, shown in Figure 8.18b from the execution, we observe that the worst execution time is higher than the time from the default usage model and the second usage model. This is because although we do not perform the context evaluation process, all requests are still successfully authorized and none are dropped. The lack of evaluating context results in better performance time than the time achieved in **DUM1**.

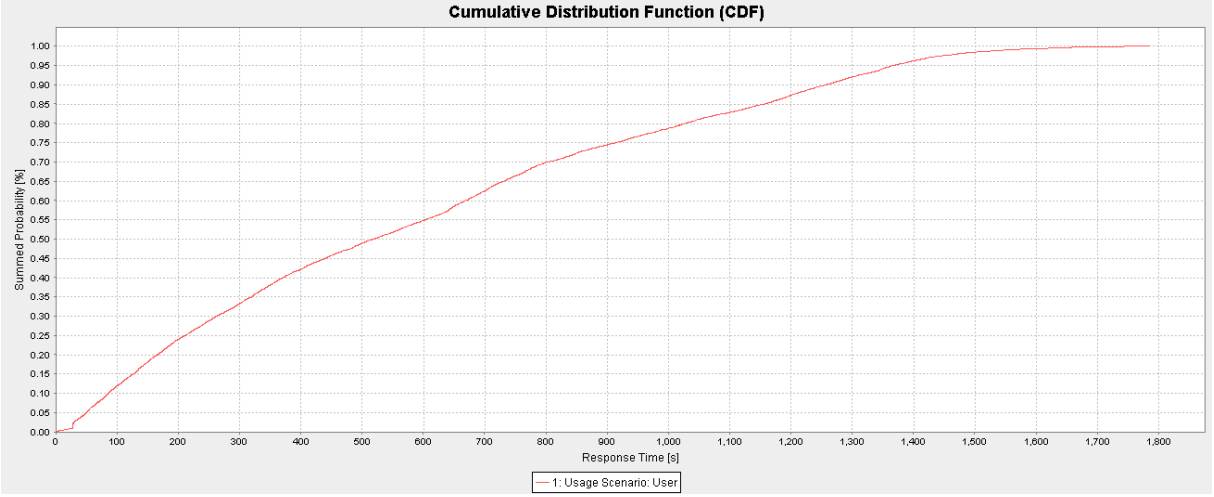
In the last usage model, **DUM5**, we want to evaluate scenarios where developers want to manage which paths of context are traversed and whether the resulting performance impact can be measured. For this purpose, in the usage model device authentication and trust calculation are set to true from the start which should result in not executing those paths. The scenario simulates an interaction similar to the one from **DUM1** where every request is accepted but device and trust are not checked. Therefore we should expect that the response time achieved is less than the one from **DUM1** but still above the **DUM** and above the usage model where requests are dropped, **DUM2**. From the CDF, shown in Figure 8.18c, we observe a worst execution time of around 14.000 seconds. This time is less than the time from **DUM1** where full path traversal is performed and above the time of **DUM2** since we do not drop requests.



(a) PolicyAuthorized - true

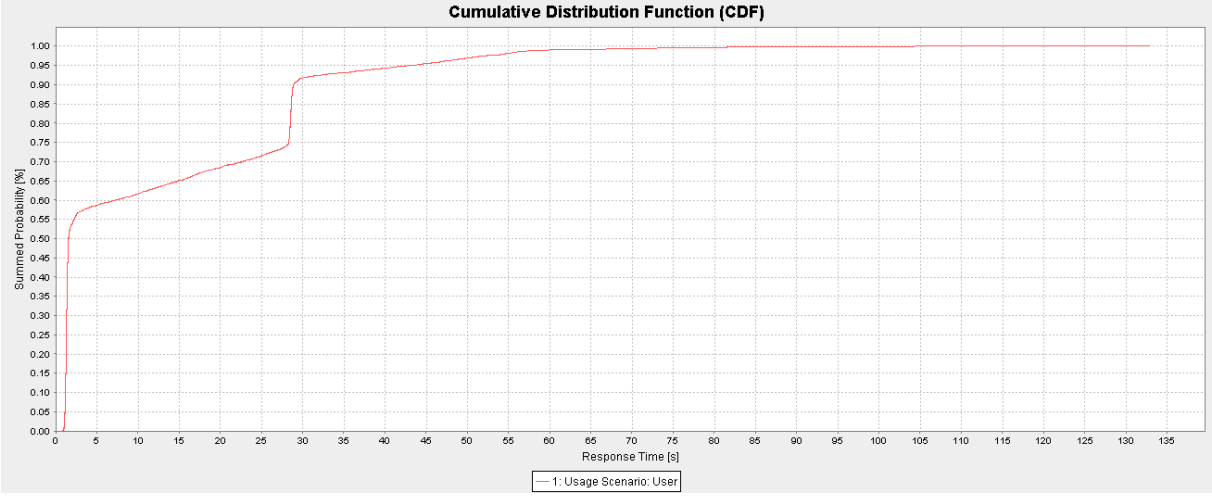


(b) PolicyAuthorized, Device AuthenitcationProbability - true

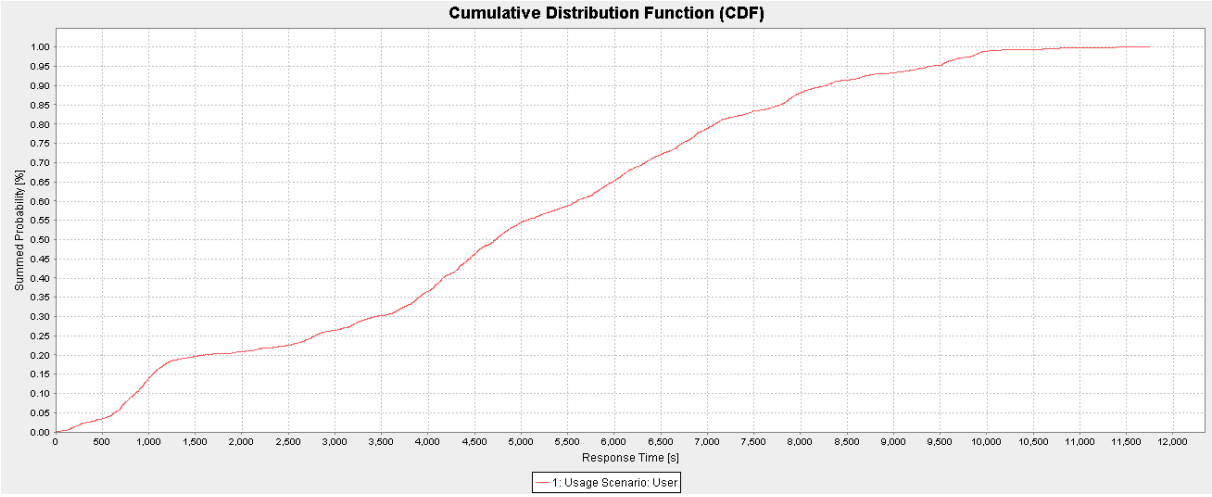


(c) PolicyAuthorized, User AuthenitcationProbability - true

Figure 8.14: DUM3 with different values for PolicyAuthorized and authentication probabilities

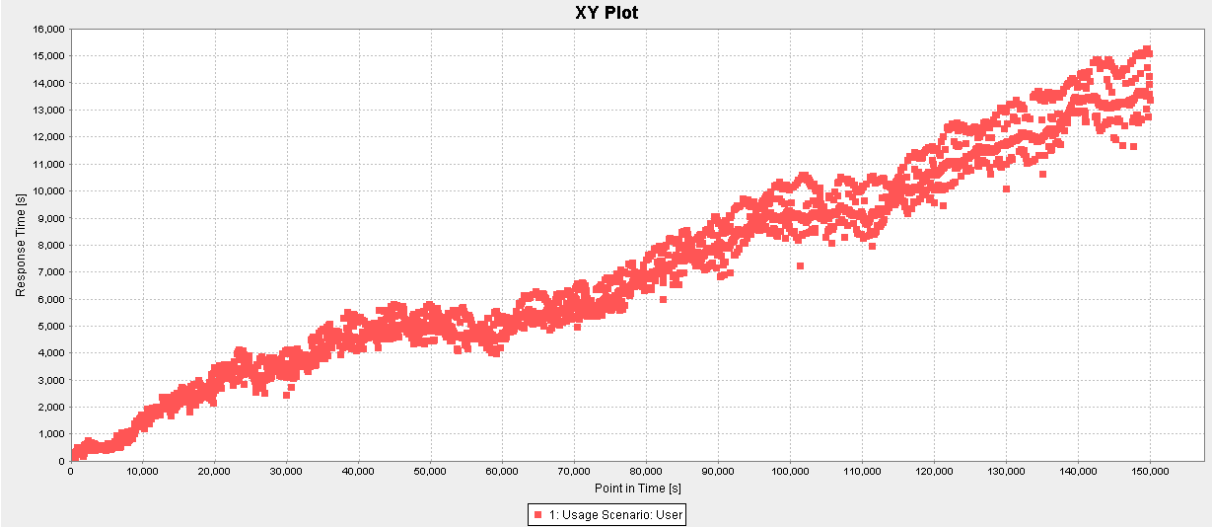


(a) PolicyAuthorized - true, AuthenitcationProbabilities - BoolPMF 50 50

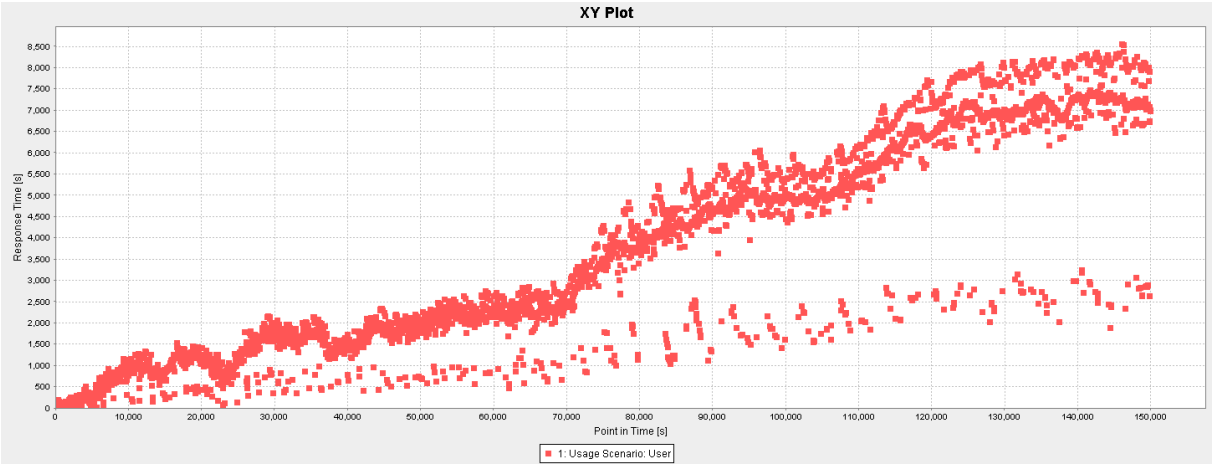


(b) PolicyAuthorized - true, AuthenitcationProbabilities - BoolPMF 99 1

Figure 8.15: DUM3 with different values for PolicyAuthorized and authentication probabilities

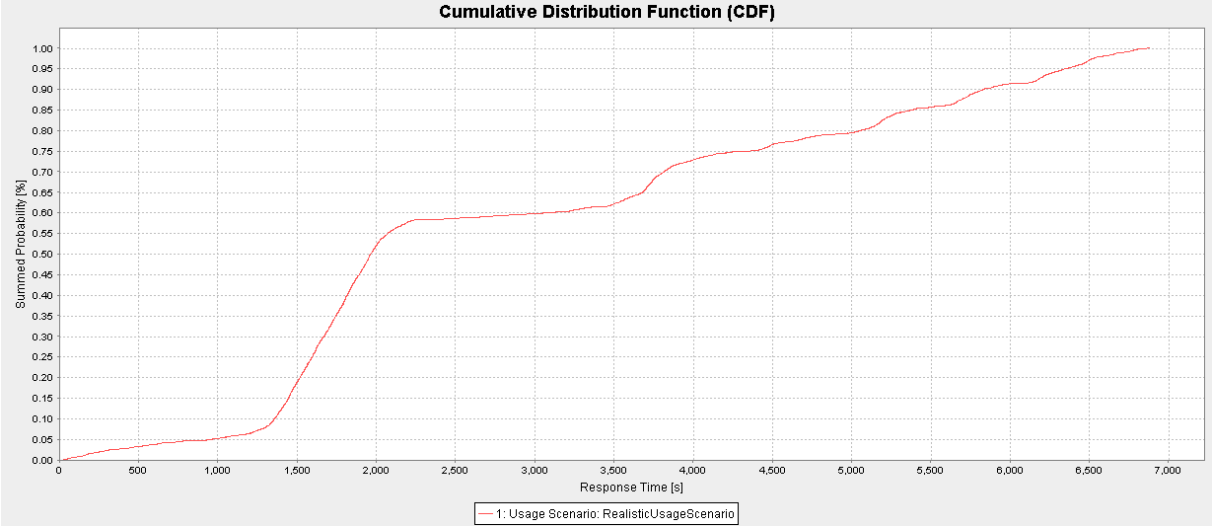


(a) UM1

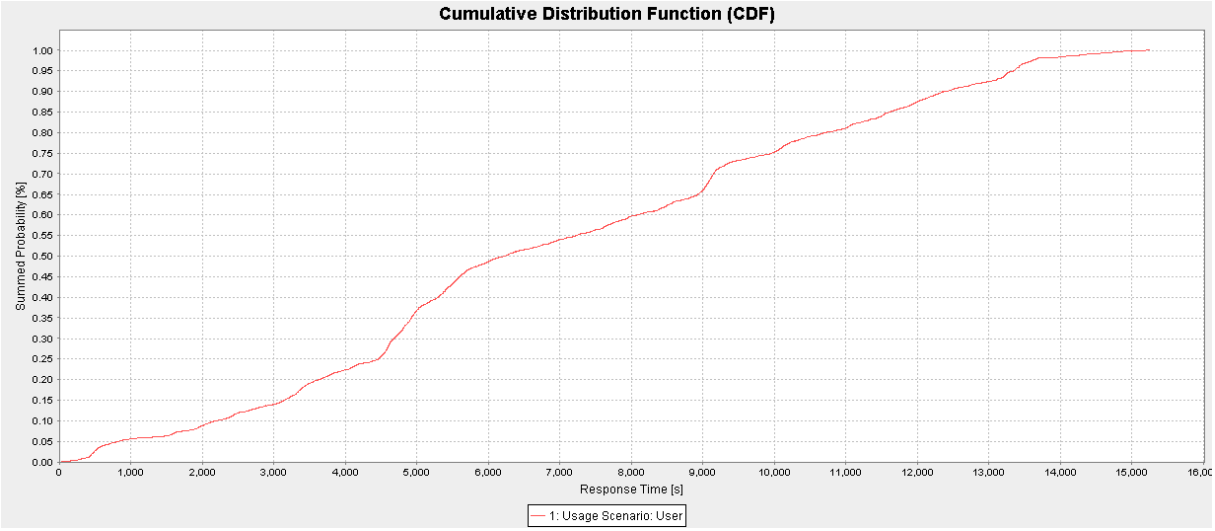


(b) UM2

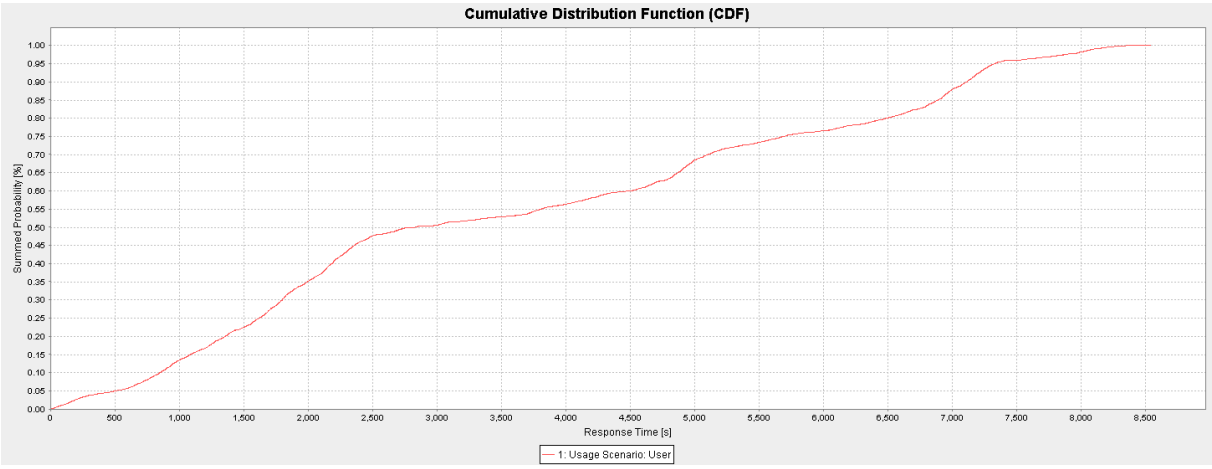
Figure 8.16: Response time at the different points in execution time for UM1 and UM2



(a) DUM no ZTA

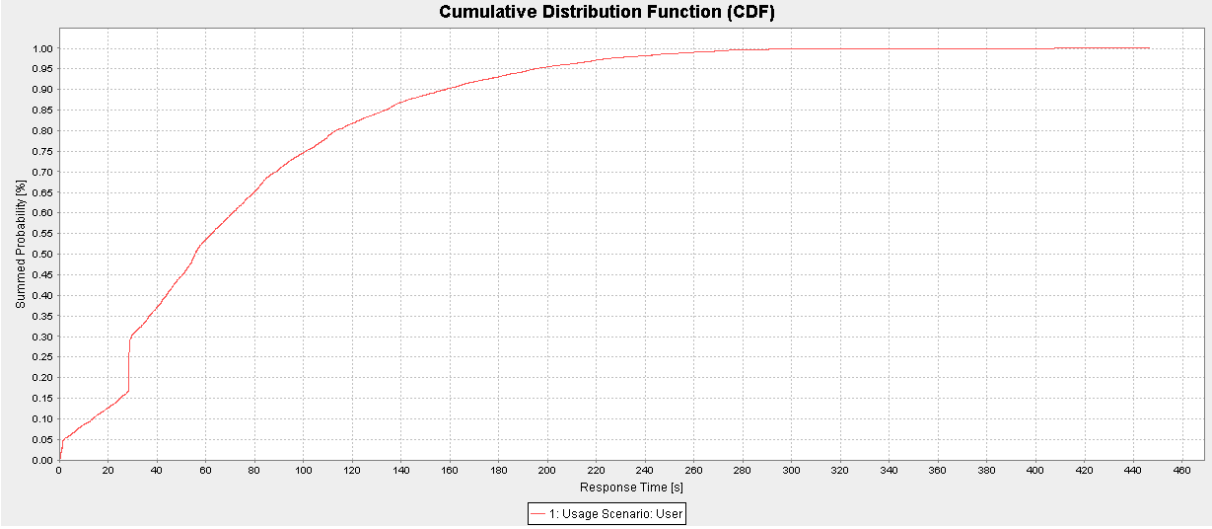


(b) DUM1 with ZTA

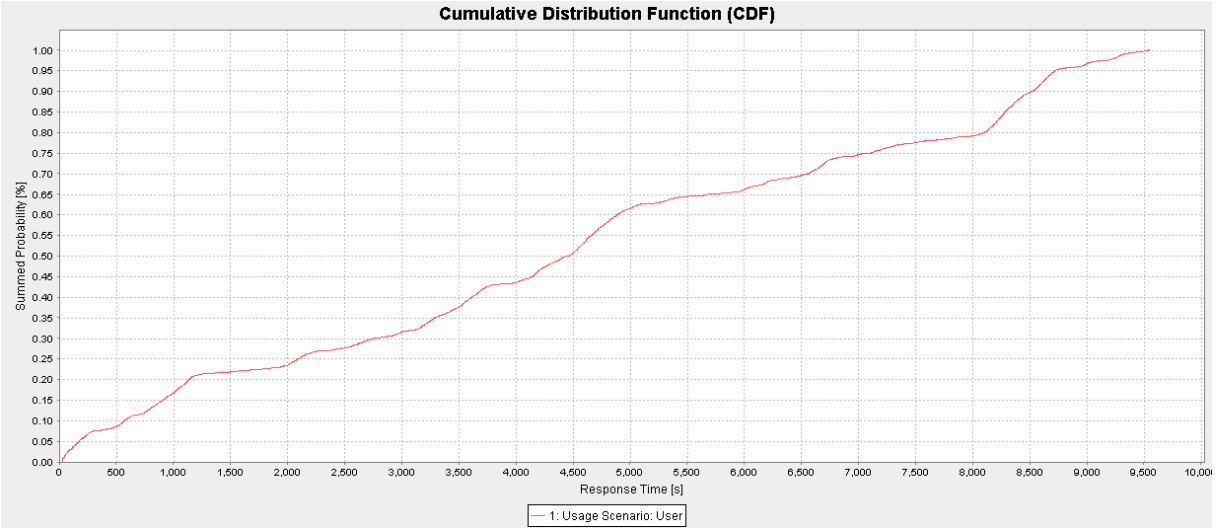


(c) DUM2 with ZTA

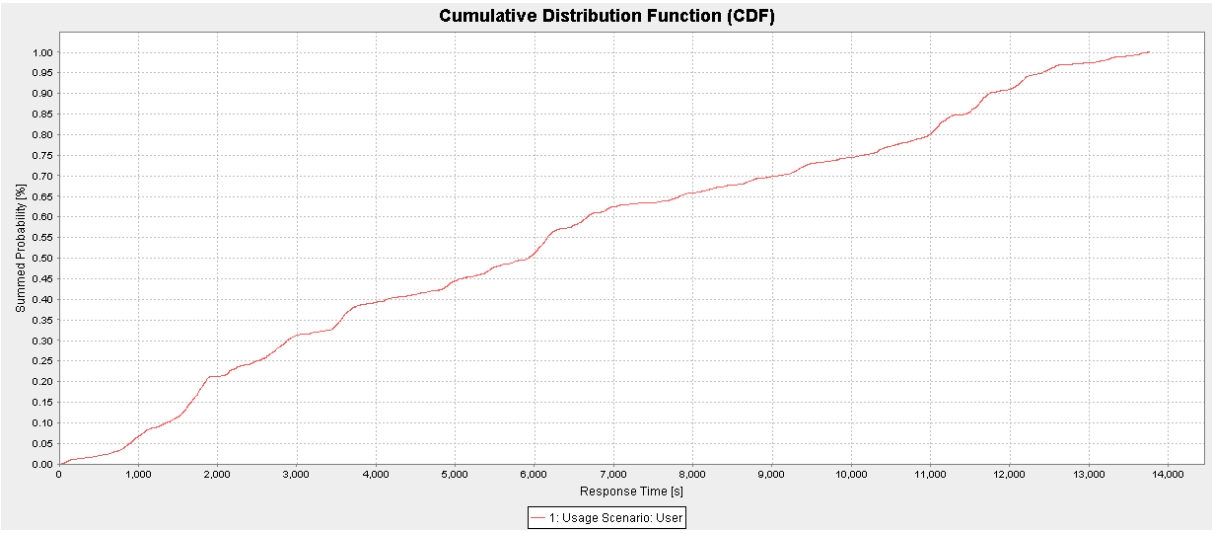
Figure 8.17: CDFs of DUM no ZTA and DUM1 and DUM2 with ZTA



(a) DUM3 with ZTA



(b) DUM4 with ZTA



(c) DUM5 with ZTA

Figure 8.18: CDFs of DUM3, DUM4 and DUM5 with ZTA

Realistic Usage Model

After we have analysed the performance impact on the derived usage models we will run simulations again but this time using variations of the realistic usage model. Like in the previous subsection, we first run a simulation on the Media Store model without ZTA, **RUM**, to obtain response time metrics which we can compare to our ZTA model. From the execution of the initial Media Store model with the realistic scenario we observe that the system achieves a worst execution time above 80.000 seconds, as shown in Figure 8.21a. We now run the simulation for our ZTA model and use the realistic usage model with the different configurations presented in Subsection 8.2.5.

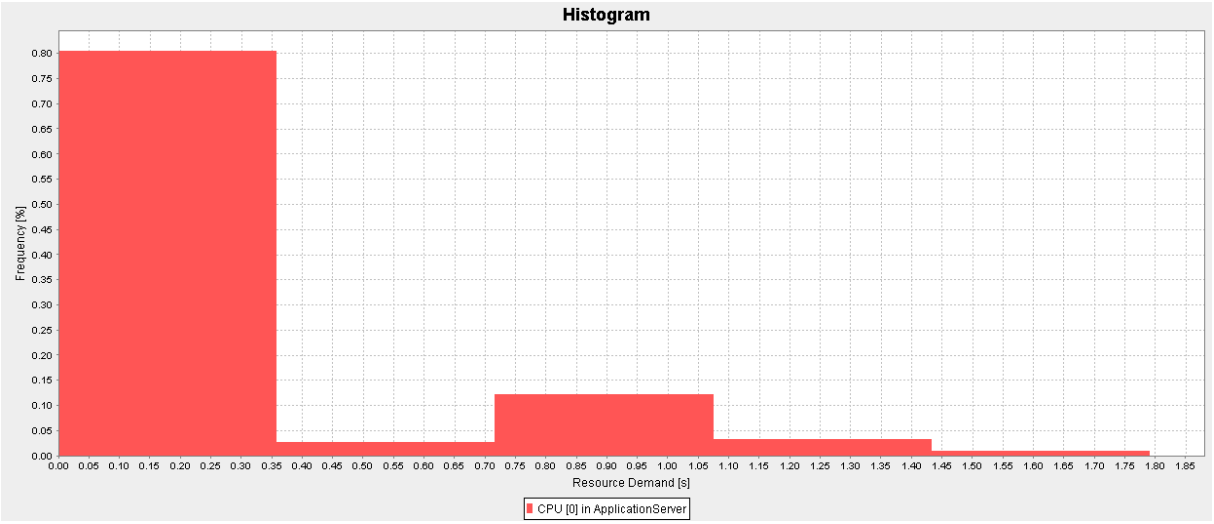
As discussed, requests in **RUM1** traverses the full execution path for each request and we expect a higher response time due to the impact of the ZTA components. The CDF of the usage model, shown in Figure 8.21b, shows that the system yields a worst execution time of 90.000 which exceeds the achieved time in **RUM**. This means that the performance impact of the ZTA components can be detected.

In **RUM2** where a request might be dropped, we observe a decrease in the wort response time achieved by the system. As shown in Figure 8.21c the time reaches at most 80.000 seconds which is lower than the time from **RUM**. We can interpret that this result is achieved because of the dropped requests which free up resources earlier and allow the system to handle accepted requests faster. From the histograms of the CPU utilization of the Application Server, shown in Figure 8.19, we observe that indeed the resource utilization is less for the model with ZTA.

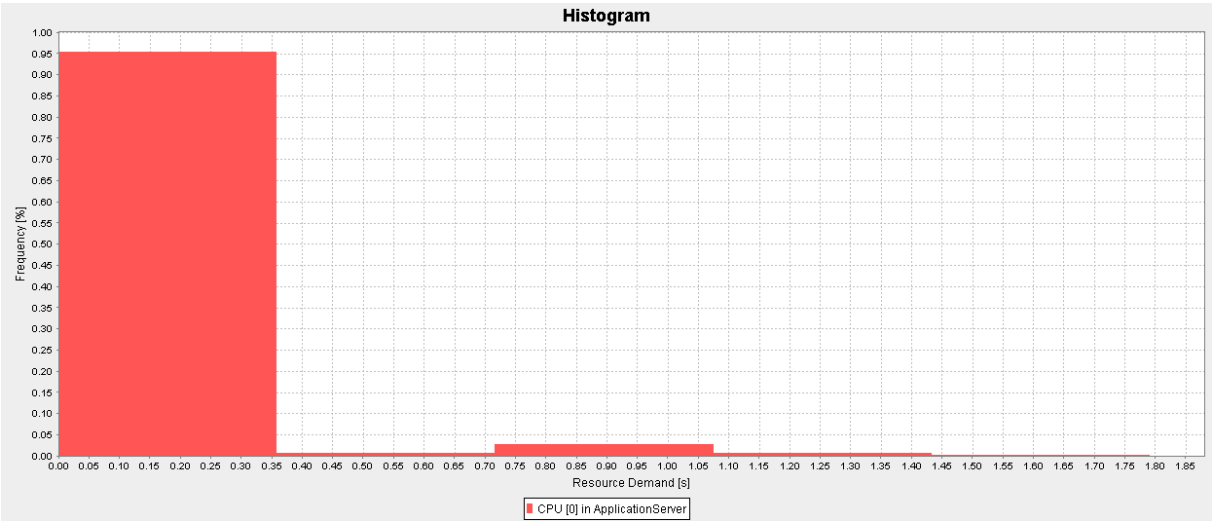
In **RUM3** again, the results of the simulation yield a lower response time, as shown in Figure 8.22a, similar to the discussed situation in paragraph 8.3.3 of **DUM3**. However, this time the difference is less significant than the difference achieved by the derived usage model. Since here the probability of denying a request is higher due to the authentication probabilities it is possible that there are more free resources than in the other usage scenarios. However, we already know from the derived usage model analysis that tuning authentication probabilities may not provide trustworthy results.

In **RUM4**, where all requests are approved but context evaluation is skipped we observe in the results shown in Figure 8.22b that the worst response time is better than the time of **RUM1** due to no context evaluation and still above the **RUM** due to the performance impact of the ZTA components. However, the worst response time is not significantly higher.

Observing again the ZTA model and the initial Media Store model we notice that there is a great difference in the units used to define resource demands. For the ZTA components, we have selected a default resource demand for all components which is 1000 units. On the other hand, resource demand units defined in the Media Store are of higher values such as 248240 units or 7943680 units and even higher. This could be the reason why the performance impact introduced by the ZTA components is not so significant. What we can do to test this is to increase the resource demands required by the ZTA components to a value taken from the possible resource demands of the Media Store components. In Figure 8.20 we display the result of the simulation of **RUM4** on a ZTA model with resource demands of every ZTA component increased to 7943680. We immediately see from the CDF that the worst response time has now increased to above 85.000 units which proves that if resource demands are of the same range as the Media Store demands then an even more accurate performance impact can be detected. This usage model evaluation also proved that when integrating ZTA components in



(a) RUM no ZTA



(b) RUM2 with ZTA

Figure 8.19: Histogram of CPU resource demand of Application server of Default Model and ZTA models with realistic UMs default and 2

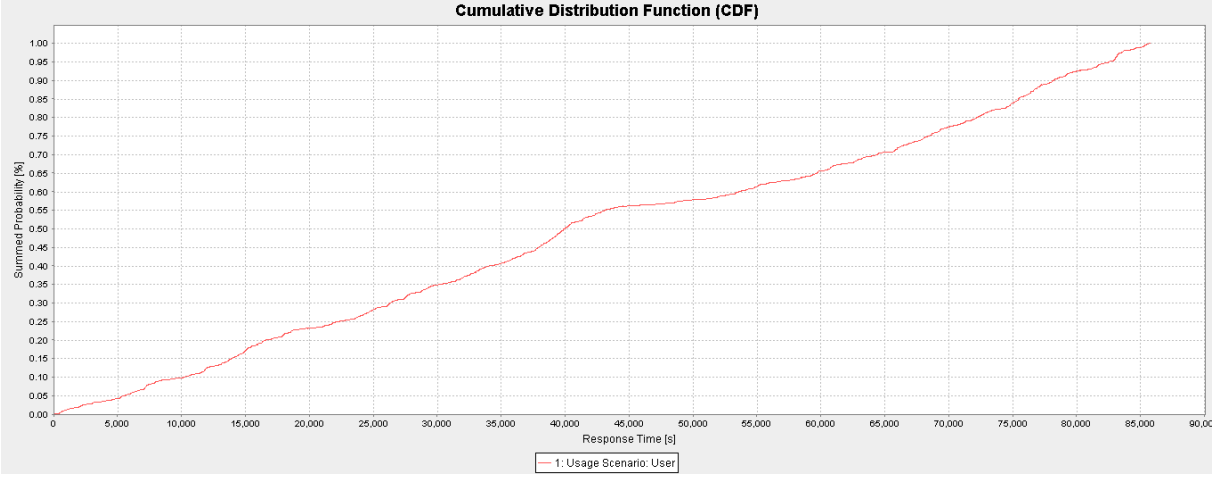


Figure 8.20: RUM4 with ZTA and increased resource demands

an existing system a unit normalization needs to be performed to make sure that the resource utilization and response time of ZTA components are not too great or too low in the context of the system where it is integrated. However, since we currently do not have an existing ZTA system for which we can measure execution time, we cannot determine whether a *PE*, for example, performs faster or slower than a Media Access component and hence annotate it with higher or lower resource demand units than those of the Media Access component.

Lastly, in the execution of **RUM5**, where we test the selection of execution paths traversed by eliminating device authentication and trust calculation, we see that the worst execution time almost reaches the one achieved in realistic usage model 1. However, it is still lower since we are skipping execution paths. When we traced the executed calls by the simulation, calls to the device authenticator and trust calculator were indeed missing from the execution.

Final Discussion on G3

To sum up, with the performed simulations on models with and without ZTA components we are able to answer the question **Q3.1** if the model allows analysing performance impact introduced by the ZTA. We saw in the derived usage scenario as well as in the realistic that when ZTA components are included in the model there is an increase in the worst execution time achieved by the system. The probability of a request requiring more time increases when every request traverses each of the execution paths in a ZTA model compared to the no ZTA model. Additionally, when executing usage models which activate different execution paths in a ZTA system we again observe an increase or a decrease in the worst response time. Therefore the model allows the detection of the performance impact of different ZTA architectures when compared to non-ZTA models. The next question we asked, **Q3.2**, is about whether we can detect performance impact in different ZTA configurations and usages. From the simulations of different usage model configurations, we observed different behaviours of the model with different response times due to dropped requests or skipped execution paths. We observed executed calls in the different scenarios and saw that the model allows for tuning different execution scenarios with the help of the ZTA variables we introduced in the model. However, along the way, we discovered also flaws in the model. The tuning of authentication probabilities

seems to be unstable for the moment and yields simulation results which are not trustworthy. Additionally, we determined that the resource demands of the ZTA components cannot be generalized. If we measure and define general resource demand units for the ZTA components, when we create a model of a system with ZTA from scratch we can adjust the system's component demands to match the general ones of the ZTA components. However, if we integrate the ZTA components as we did for our evaluation, then normalization of the units should be performed in order to avoid units of the ZTA being too low or too great when compared to the units of the underlying system.

8.3.4 Discussion on Security Analysis

In the last goal of the evaluation, we want to examine the ability of our model to detect the proposed security violations in Section 7.4. To do this we execute Data Flow Analysis on five scenarios with and without issues and calculate the precision and recall, defined in Section 8.1, of the analysis.

In **S0** we did not make changes to the model so issues would emerge. The execution of the analysis returned an empty list of violations since no violations were identified.

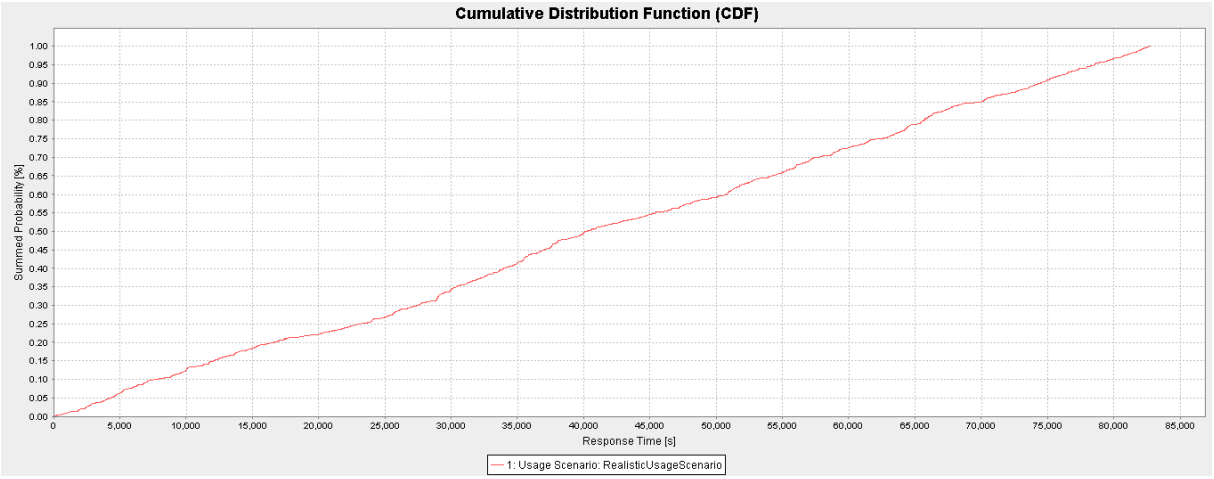
In the next scenario *S1*, where we tested the detection of unauthorized access, the analysis returned a result which contained a single identified issue of type *Unauthorized Access* due to the introduced data flow which skips the policy evaluation process. The analysis detected that a label of a lower level reached the resource and the label which signalled that the request was handled by a *PolicyEngine* was missing. Furthermore, as expected, no issue was identified for the data flow which goes through the *PolicyEngine* since there the *Evaluated.evaluated* label identified that this request was correctly handled by a *PolicyEngine*.

In **S2** where we evaluate the detection of LPP violations, the analysis identified one issue. Data with a label of *Authorized.Level3* has reached the resource which is labelled with *Authorized.Level1*. The analysis correctly identified that more permissions were given to a request than required. Additionally, no unauthorized access issue was raised due to mismatched *Autohrized* labels. The analysis correctly identifies that level 3 access is authorized in the context of accessing a level 1 resource.

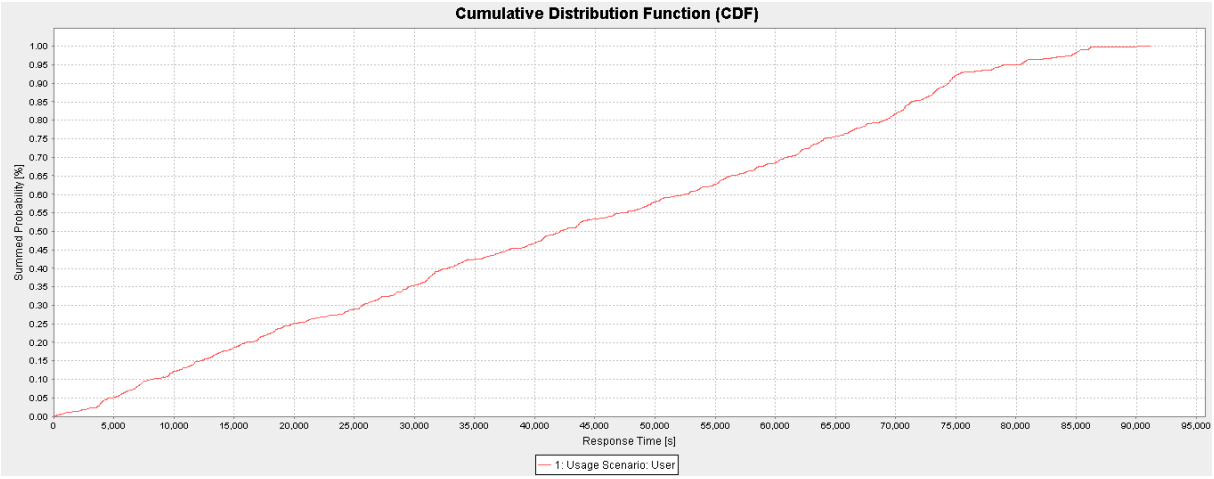
In **S3** we tested if incomplete authentication can be identified in our model. At the protected node arrived data with a single authentication label *UserAuthneitcated.SecondFactor*. Therefore, the analysis returned a list containing a single issue of type *Unauthenticated Access*.

Lastly, in scenario **S4** we introduced the issues of missing device authentication and trust calculation. The result of the analysis provided a list which contained two issues. One was of type *Device Unauthenticated Access* and the other issue was of type *Untrusted Access*. Hence, the analysis correctly identified both of the introduced issues.

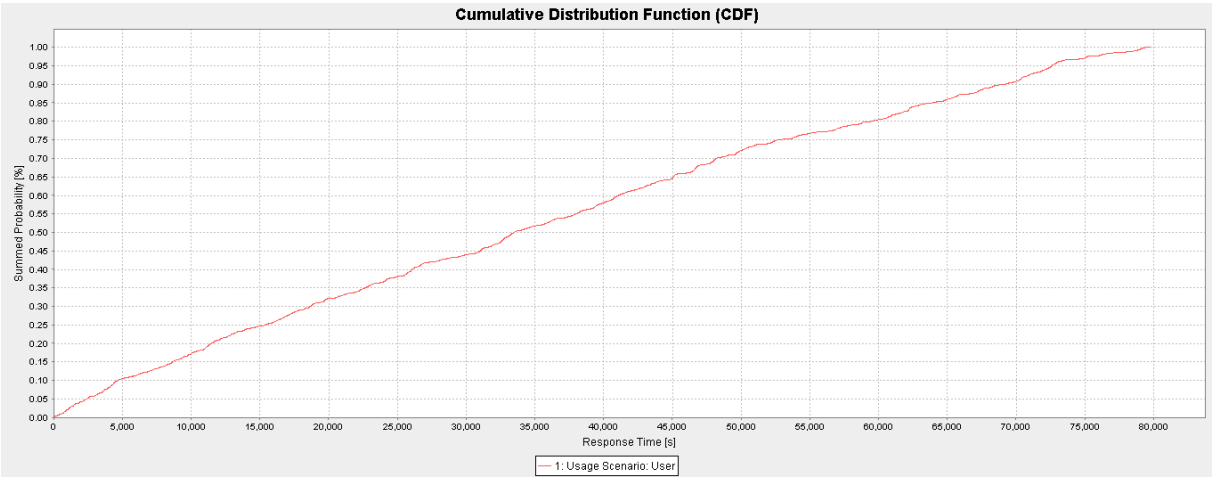
Now that we have performed the analysis and identified the issues we can calculate the results for these metrics. In table Table 8.5 we summarize the introduced versus identified security issues in scenarios 0 to 4. For each of the scenarios, the number of identified issues which were not introduced in the respective scenario is 0. Therefore the *FP* for each scenario is 0 and the sum of *FP* across all scenarios is again 0. The sum of all correctly identified issues across all scenarios is $0 + 1 + 1 + 1 + 2 = 5$. Therefore, the *TP* value is 5. Following the definition of precision presented in Section 8.1, we get the following equation: Therefore we get the following equation $\frac{TP}{TP+FP} = \frac{5}{5+0} = \frac{5}{5} = 1.0$. We achieve a precision of 1.0 for the security



(a) RUM no ZTA

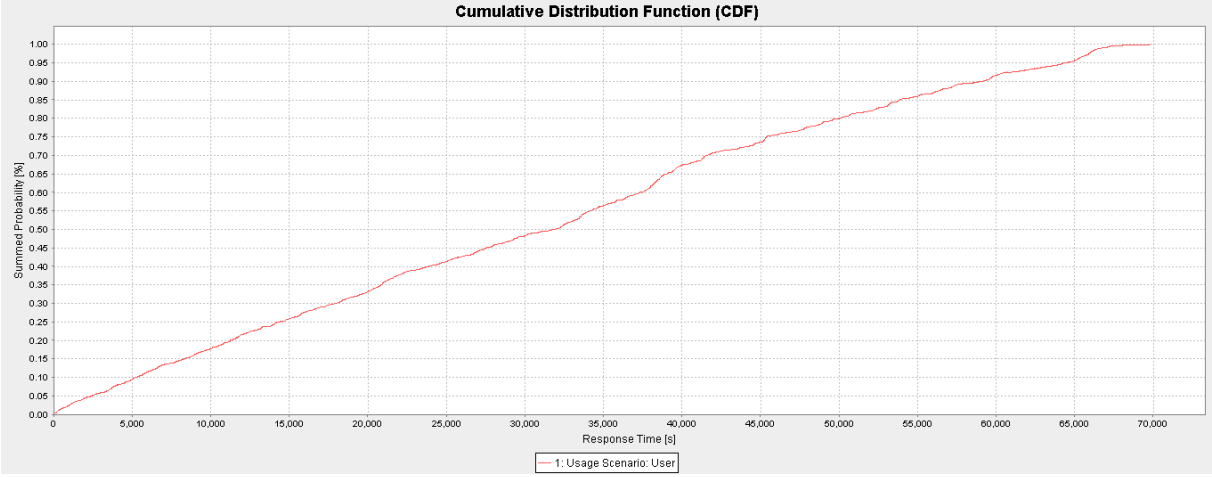


(b) RUM1 with ZTA

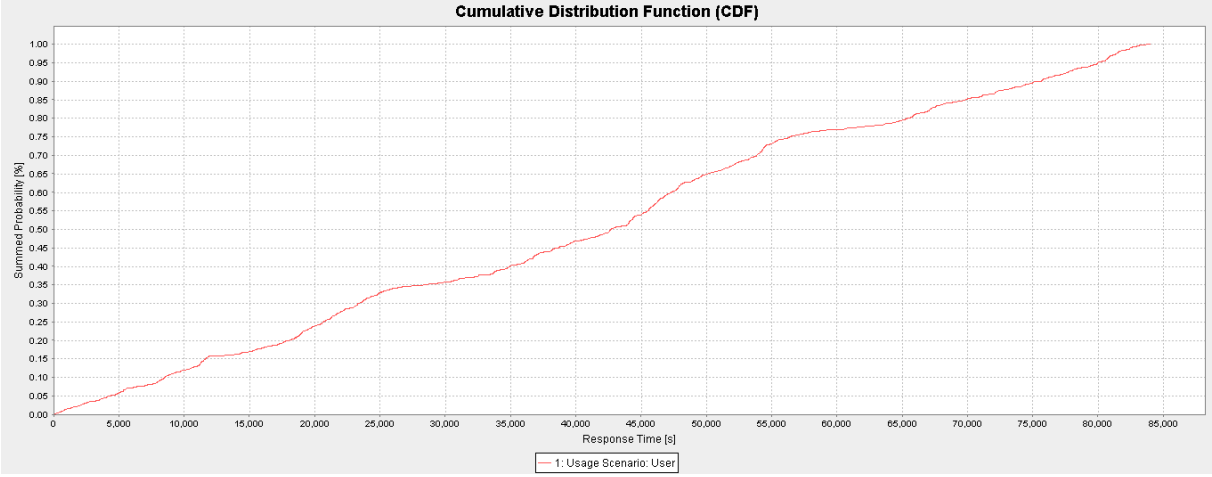


(c) RUM2 with ZTA

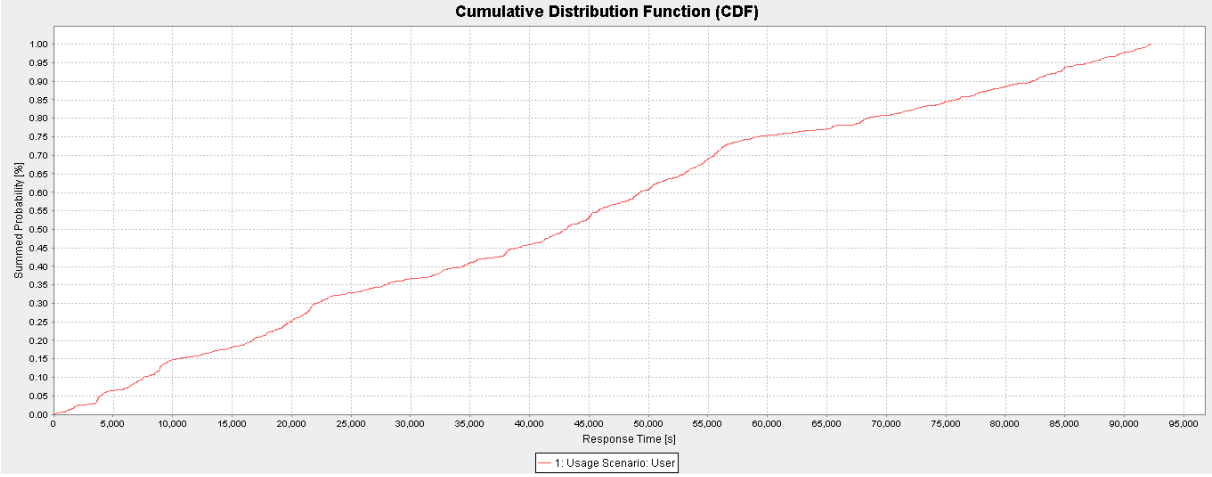
Figure 8.21: CDFs of Default Model and ZTA models with RUMs 1-2



(a) RUM3 with ZTA



(b) RUM4 with ZTA



(c) RUM5 with ZTA

Figure 8.22: CDFs of ZTA models with realistic UMs 3-5

	Introduced	Identified
S0	0	0
S1	1	1
S2	1	1
S3	1	1
S4	2	2

Table 8.5: Introduced and Identified issues in scenarios 0 to 4

analysis. Since our analysis has not failed to identify an introduced issue, our value for *FN* is 0. For the recall, we get the following equation $\frac{TP}{TP+FN} = \frac{5}{5+0} = \frac{5}{5} = 1.0$. Hence, we achieved a recall of value 1.0

Following the evaluation of the security analysis, we observe that our model allows us to detect the proposed security violations. Moreover, it does so by achieving a precision and a recall of 1.0. This means that the security analysis performed on our model is capable of detecting the proposed ZTA security violations with high credibility. Although we were unable to perform the planned security analysis on the initial Media Store with ZTA model, as discussed in Section 8.5, the partial analysis was designed to be as close as possible to the initial one. Hence, we believe that our results will hold if the same scenarios are evaluated using the initial model.

8.4 Threats to Validity

We address the validity of our evaluation in the following section. Since our evaluation includes parts where we use a case study to evaluate our contribution, we can classify the validity aspects as done by Runeson et al. [45]. According to the authors, validity can be classified to *Construct, Internal, External and Reliability.*

Construct Validity

In *Construct Validity* we discuss whether the goals we set for our evaluation are relevant to the contribution of the thesis as well as the questions we ask and the metrics we choose for them are suitable.

One of our objectives of the work is concerned with creating a ZTA model which captures all of the concepts of such architecture. Therefore we choose to evaluate the completeness of the model. Defining completeness is a problem of relativity. For example, our project might be complete in the context of ZTA but it might be incomplete in the context of Authentication Systems. To remove this bias we have limited the completeness of our model to identified elements from literature in the course of the project. In the applicability evaluation, we have chosen a discussion to answer *Q2.1.* However, this discussion is based on representations of ZTA models from the literature. Additionally, a question may arise whether the property of generality is suitable for evaluating applicability. One of our contributions is the development of a model which can be applied in different ZTA cases. Návrat and Filkorn [36] define generality as a quality of being able to capture multiple concepts and represent varying cases. Therefore, we believe that generality describes appropriately the aspect of applicability in the context of our contribution.

The performance analyses observe differences in execution time and resource utilization. These are the currently supported metrics by Palladio and were used to demonstrate the performance impacts of different Media Store configurations in [43].

For the security evaluation, we use precision and recall. The metrics have already been used to evaluate the accuracy of models and analysis using the Data Flow Analysis in [46, 7].

Internal Validity

In *Internal Validity* validity we discuss whether there are hidden factors that affect the evaluation of the impact of one factor on another factor.

We encounter such a case when evaluating the performance impact of our ZTA components. We evaluate there how the ZTA components impact the execution time and resource utilization of a system. However, the overall execution time may also be impacted by probabilities introduced in the initial case study. However, we engage this threat by providing an additional usage model that triggers a constant behaviour of the initial case study components.

When evaluating the detection of security violations an unwanted factor which impacts the evaluation of the questions is the compatibility of performance and security models in Palladio. Problems in the compatibility forced us to apply changes to the ZTA model of the case study in order to execute the evaluation. However, we mimicked the incompatible complex elements from the performance model as closely as possible using simpler components. We did not make any changes to the core concepts of the security or performance model.

External Validity

In the aspect of *External Validity*, we analyze the relevance of our evaluation and its results to cases outside of the work.

Generalizing findings of the evaluation outside of the scope of the project is difficult when using a case study for the evaluation. This is the case in this thesis. However, we mitigate this threat by using two case studies in the course of the project. One of the case studies is designed for the project whereas the second one we use is a well-known case study which has already been used in the evaluation of modelling projects [28, 32, 43]. We add real-world context to our case study by extending it according to the UK NCSC [57] guide and evaluate it using the CISA maturity model [11]. In addition, we perform an evaluation of the generality of the model. We use our work to model multiple suggested ZTA from the literature.

We recognize a threat to the external validity in the security evaluation. We performed an evaluation for each violation with just a single scenario. Additionally, the scenarios were derived from themselves.

Reliability

Reliability represents the ability of future researchers to be able to reproduce the evaluation and obtain the same results. In our evaluation setup, we have provided a step-by-step guide to preparing the model for the experiments. Additionally, we have described how our usage models and scenarios are constructed. Threats to reliability may arise from our discussion evaluations since future researchers might not reproduce the same interpretations. However, we cannot address these issues. We publish our model and implementation with all of the evaluation scenarios [41] to eliminate further threats regarding *Reliability*.

8.5 Assumptions and Limitations

In our evaluation, we first encountered limitations in the aspect of available systems to model. During the selection of a suitable system to model we were able to find two implementations of ZTAs. However, both of the implementations proved to be unsuitable. One of the systems was an open source project which has discontinued. Although we had access to the implemented components and were able to set up a *PEP* and a *PE*, we were not able to test the system. Its further use required the use of a dedicated mobile application for multi-factor application which is no longer present on the market. Attempts to replace the application with alternative apps from the application market we unsuccessful. The second system we found was still functioning and had a trial test version. However, its components were black-box and we had no access to the underlying architecture. The system was performing its task on remote company-owned servers. This type of component deployment does not allow us to measure resource utilization in order to derive resource demands for our model. Additionally, the lack of open source available ZTA system led us to make assumptions about the resource demands of the ZTA components.

The next limitations came from the implementation of the Data Flow Analysis. At the time of creating the security model and writing the thesis, the Data Flow Analysis has been simultaneously updated. Additionally, the fact that Data Flow Analysis has not been combined until now with complex performance models, was a prerequisite for possible incompatibilities. These expectations proved to be true. During the execution of the Data Flows Analysis on the model presented in Subsection 8.2.2 we detected errors connected to processing composite components, SEFF internal actions and variable specifications which had both performance and confidentiality variable characterisations. More serious issues emerged in compatibility with more complex branching actions and the unavailability of component variables defined in the assembly in the evaluation of the nodes' behaviour expressions. We were able to cope with composite components by replacing them with their building blocks. Processing of internal actions and performance variable characterisations were patched successfully by the team providing the Data Flow Analysis implementation. However, the last two identified issues with branching actions and component variables introduced a limitation to the security evaluation. This required us to apply changes to the model in order to assess the developed security model. The changes in the authenticating behaviour would prevent us from testing the generality of the initially specified behaviour definition. However, the detection of the security violation is not obstructed. Overall, we believe that the applied changes did not change the core functionality of the model and the results should hold once the issues have been resolved and evaluation is carried on on the initially proposed Media Store with ZTA.

8.6 Data Availability

We publish all of our data in the project's repository [41]. The repository includes all of the models described in this thesis. It includes also the implementation of the ZTA security analysis. We also provide there all of the usage models used for this evaluation.

9 Conclusion

In the following chapter, we conclude the thesis by summarizing the contributions of the work. In addition, we present what questions and activities remain open for future research projects.

9.1 Conclusion

In this master thesis, we analyzed ZTAs and mainly how they can be modelled so we can perform security and performance analyses at design time. We examined how we can do this using the Palladio Component Model. We determined if it was possible to represent ZTA concepts and processes using only the already implemented elements of the Palladio Component Model and if the component model requires any extensions. We also tried to create the ZTA model as general as possible so it can be used off-the-shelf in future projects. For the security analysis, we used Data Flow analysis which traces data labels through all of the possible system execution paths. Additionally, it was well known that in Palladio security and performance analyses can be performed separately. However, in this thesis, we tested the possibility of combining both security and performance annotations in a single model.

In the main part of the work we analyzed ZTA specifications such as the NIST standard document, Google's Beyondcorp and CCA's SDP specification. From these standards, we were able to extract core activities and processes which happen in a ZTA. We identified logical components which are responsible for those roles and should be present in an architecture to cover Zero Trust principles. We also outlined additional components and systems which could further improve the maturity of a system in the context of Zero Trust. As a result of this activity, we obtained a ZTA meta-model.

In the next part of the main body, we used Palladio to create a ZTA model. We created a repository where we defined interfaces to describe identified roles and included all of the core logical components of ZTA and their variations to cover all of the possible differences suggested in the literature. As mentioned before, we created all of these components in a way that they encapsulate and generalize their differences presented in the standard documents. This increases the applicability of our model in different projects and requires developers to make fewer changes to their underlying models in order to adapt our components to their models. However, there were still differences which were not possible to represent in a general manner. For this, we introduced extension points in our components and made them customizable. With our components, we modelled templates for the SDP and Beyondcorp ZTA approaches. However, since we did not have access to an implemented ZTA, we could not measure and estimate resource demands to include them in our performance model. We also applied our model to the JPlag case study defined in this thesis.

Next, we defined a model for Data Flow Analysis. First, we defined what were the possible security violations which can happen in a ZTA and how we can test for them. We developed

the required, by the Data Flow Analysis, labels and node behaviours and defined them in the ZTA Palladio model. We developed a reusable Java implementation of the Data Flow Analysis of a ZTA and defined an extensible reporting pattern for the analysis.

For the evaluation, we tested how many of the identified in the research ZTA concepts we were able to represent in our Palladio model and what changes would Palladio need to be able to represent ZTAs. We also tested the applicability of the model by modelling different ZTA models from the literature. We were not able to obtain a system which has ZTA components and try and model it in Palladio using the elements from the ZTA repository. This is why we used the Media Store case study which has been used in the past to demonstrate the modelling power of Palladio. We extended the Media Store model with a ZTA following a ZTA integration guide and evaluated the resulting model according to the CISA maturity model. We also designed multiple usage scenarios to evaluate different behaviours of the ZTA. We then compared the achieved response times and resource utilization of the no ZTA and the ZTA model of the Media Store and we were able to detect the performance impact introduced by the new components. For the security evaluation, we designed scenarios in which we introduced security violations. Then we tested and calculated the precision and recall of the Data Flow Analysis in those scenarios with our security model. However, due to limitations presented in the Data Flow Analysis, we were forced to apply changes to the initial Media Store model so it could be compatible with the Data Flow Analysis. As a result of this, we determined that the performance and security models of a system in Palladio are still not fully compatible. To sum up, we successfully created a performance and a security model of ZTA using only the present elements of Palladio and Data Flow Analysis. Zero Trust proves to be a growing topic in security and companies will be looking to integrate a ZTA into their systems. To evade the costly effects of false designs such as high implementation effort and financial losses, companies would prefer to first test out different ZTA approaches at design time. Using our created ZTA repositories, they can integrate the ZTA concepts in their design models and observe what performance changes would appear as well as detect security violations. Furthermore, with this project, we make also a step towards combining security and performance analyses in Palladio in a single model, as until now both of the analyses were performed on separate models.

9.2 Future work

Although we have been able to model ZTA as a Palladio model there still remain some open topics for future work. As we have mentioned multiple times in this thesis we did not have the option to evaluate a real system with ZTA components in order to estimate realistic resource demands. From this statement points of future work arise. The first one is the implementation of the modelled here ZTA components. For each of these components, it can be researched what technologies and algorithms can be used to implement them and then actually implement them to observe resource utilization and response times. For example, a *PE* component is present not only in ZTA but also in XACML architectures. As we discussed in the section where we modelled the *PE* there are different algorithms for loading and evaluating policies. These algorithms can be implemented and then used to estimate the resource demands of a *PE*. The second point of future work is the examination of existing solutions for the auxiliary components which we modelled in the project. For components such as authentication components or SIEM

systems, there are currently solutions on the market. However, these technologies require their own topic. For example, authentication approaches can vary in factors, and technologies enforced. We cannot examine all of the possible systems in order to derive general resource demands for Palladio. Therefore, different implementations of such systems can be researched and used to estimate resource demands which can then be applied to the developed in this thesis components.

Additionally, we recognize the possibilities of further refining the created model to represent even more concepts of ZTAs. For example, continuous authentication can be added to the model. This can be done by examining the option to include additional branches in the SEFF diagrams at different points in the execution path with probabilistic behaviour which triggers calls to an authentication component. Another improvement which would increase the variability of the model is the modelling of dynamic policy providers which communicate with context providers to update policies.

Lastly, we were not able to evaluate the initial Media Store model with ZTA which contained complex branching actions due to limitations from the Data Flow Analysis implementation. When the compatibility of both the security and performance models is increased, the evaluation should be reproduced this time with the unmodified Media Store ZTA assembly model.

Bibliography

[1] Annamalai Alagappan, Sampath Kumar Venkatachary, and Leo John Baptist Andrews. "Augmenting zero trust network architecture to enhance security in virtual power plants". In: *Energy Reports* 8 (2022), pp. 1309–1320.

[2] Igor Anastasov and Danco Davcev. "SIEM implementation for global and distributed environments". In: *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*. IEEE. 2014, pp. 1–6.

[3] Marie Baezner and Patrice Robin. *Stuxnet*. Tech. rep. ETH Zurich, 2017.

[4] Mohammadreza Hazhirpasand Barkadehi et al. "Authentication systems: A literature review and classification". In: *Telematics and Informatics* 35.5 (2018), pp. 1491–1511.

[5] Erphan A Bhuiyan et al. "Towards next generation virtual power plant: Technology review and frameworks". In: *Renewable and Sustainable Energy Reviews* 150 (2021), p. 111358.

[6] Barry W Boehm, Robert K Mcclean, and DB Urfrig. "Some experience with automated aids to the design of large-scale reliable software". In: *Proceedings of the international conference on Reliable software*. 1975, pp. 105–113.

[7] Nicolas Boltz et al. "Handling environmental uncertainty in design time access control analysis". In: *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2022, pp. 382–389.

[8] Defense Use Case. "Analysis of the cyber attack on the Ukrainian power grid". In: *Electricity Information Sharing and Analysis Center (E-ISAC)* 388.1-29 (2016), p. 3.

[9] Baozhan Chen et al. "A security awareness and protection system for 5G smart healthcare based on zero-trust architecture". In: *IEEE Internet of Things Journal* 8.13 (2020), pp. 10248–10263.

[10] Xu Chen et al. "Zero trust architecture for 6G security". In: *arXiv preprint arXiv:2203.07716* (2022).

[11] *CISA Zero Trust Maturity Model*. URL: https://www.cisa.gov/sites/default/files/2023-04/zero_trust_maturity_model_v2_508.pdf.

[12] Vittorio Cortellessa et al. "An architectural framework for analyzing tradeoffs between software security and performance". In: *Architecting Critical Systems: First International Symposium, ISARCS 2010, Prague, Czech Republic, June 23-25, 2010 Proceedings*. Springer. 2010, pp. 1–18.

[13] CSA. "Software-Defined Perimeter (SDP) Specification v2.0". In: (). URL: https://cloudsecurityalliance.org/artifacts/software-defined-perimeter-zero-trust-specification-v2/.

[14] Carlos Da Silva, Welkson Medeiros, and Silvio Sampaio. “PEP4Django-a policy enforcement point for python web applications”. In: (2019).

[15] Casimer DeCusatis et al. “Implementing zero trust cloud networks with transport access control and first packet authentication”. In: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. IEEE. 2016, pp. 5–10.

[16] Fan Deng et al. “An efficient policy evaluation engine for XACML policy management”. In: *Information Sciences* 547 (2021), pp. 1105–1121.

[17] *Documentation of Confidentiality Modeling and Analysis in Palladio*. URL: `https://fluidtrust.github.io/tutorial-ecsa2021-tooldoc/index.html`.

[18] “Evolving Zero Trust How real-world deployments and attacks are shaping the future of Zero Trust strategies”. In: ().

[19] Eduardo Buglioni Fernandez and Andrei Brazhuk. “A critical analysis of zero trust architecture (Zta)”. In: *Available at SSRN 4210104* (2022).

[20] *Goal-Question-Metric Evaluation Approach*. URL: `https://sdq.kastel.kit.edu/wiki/Goal_Question_Metric`.

[21] Gustavo González-Granadillo, Susana González-Zarzosa, and Rodrigo Diaz. “Security information and event management (SIEM): analysis, trends, and usage in critical infrastructures”. In: *Sensors* 21.14 (2021), p. 4759.

[22] “Homeoffice und mobiles Arbeiten”. In: (). URL: `https://de.statista.com/statistik/studie/id/86464/dokument/homeoffice-und-mobiles-arbeiten/`.

[23] “Internet of Things – Market data analysis and forecasts”. In: (). URL: `https://de.statista.com/statistik/studie/id/109209/dokument/internet-der-dinge-market-outlook-report/`.

[24] *JPlag - Detecting Software Plagiarism*. URL: `https://github.com/jplag/JPlag`.

[25] *JPlage Report Viewer*. URL: `https://jplag.github.io/JPlag/`.

[26] Boo Geum Jung et al. “ZTA-based Federated Policy Control Paradigm for Enterprise Wireless Network Infrastructure”. In: *2022 27th Asia Pacific Conference on Communications (APCC)*. IEEE. 2022, pp. 1–5.

[27] Jaroslav Kadlec, David Jaros, and Radek Kuchta. “Implementation of an Advanced Authentication Method within Microsoft Active Directory Network Services”. In: *2010 6th International Conference on Wireless and Mobile Communications*. IEEE. 2010, pp. 453–456.

[28] Heiko Klare et al. “Enabling consistency in view-based system development—the vitruvius approach”. In: *Journal of Systems and Software* 171 (2021), p. 110815.

[29] Romain Laborde et al. “An adaptive xacmlv3 policy enforcement point”. In: *2014 IEEE 38th International Computer Software and Applications Conference Workshops*. IEEE. 2014, pp. 620–625.

[30] Romain Laborde et al. “Pep= point to enhance particularly”. In: *2008 IEEE Workshop on Policies for Distributed Systems and Networks*. IEEE. 2008, pp. 93–96.

[31] Brian Lee et al. “Situational awareness based risk-adapatable access control in enterprise networks”. In: *arXiv preprint arXiv:1710.09696* (2017).

[32] Sebastian Lehrig and Thomas Zolynski. “Performance prototyping with protocom in a virtualised environment: A case study”. In: *Proceedings to Palladio Days* (2011), pp. 17–18.

[33] *MaxMind GeoIP Documentation*. URL: https://dev.maxmind.com/geoip.

[34] *Microsoft Zero Trust Maturity Model*. URL: https://download.microsoft.com/download/f/9/2/f92129bc-0d6e-4b8e-a47b-288432bae68e/Zero_Trust_Vision_Paper_Final%2010.28.pdf.

[35] George Moustris, Costas Tzafestas, and Konstantinos Konstantinidis. “A long distance telesurgical demonstration on robotic surgery phantoms over 5G”. In: *International Journal of Computer Assisted Radiology and Surgery* (2023), pp. 1–11.

[36] Pavol Návrat and Roman Filkorn. “A note on the role of abstraction and generality in software development”. In: *Journal of Computer Science* 1.1 (2005), pp. 98–102.

[37] Barclay Osborn et al. “Beyondcorp: Design to deployment at google”. In: (2016).

[38] *Palladio Example Models*. URL: https://github.com/PalladioSimulator/Palladio-Example-Models.

[39] Biplob Paul and Muzaffar Rao. “Zero-Trust Model for Smart Manufacturing Industry”. In: *Applied Sciences* 13.1 (2023), p. 221.

[40] David MW Powers. “Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation”. In: *arXiv preprint arXiv:2010.16061* (2020).

[41] *Project repository*. URL: https://gitlab.kit.edu/kit/kastel/sdq/stud/abschlussarbeiten/masterarbeiten/evgenicholakov.

[42] Keyvan Ramezanpour and Jithin Jagannath. “Intelligent zero trust architecture for 5G/6G networks: Principles, challenges, and the role of machine learning in the context of O-RAN”. In: *Computer Networks* (2022), p. 109358.

[43] Ralf H Reussner et al. *Modeling and simulating software architectures: The Palladio approach*. MIT Press, 2016.

[44] Hossein Mohammadi Rouzbahani, Hadis Karimipour, and Lei Lei. “A review on virtual power plant for energy management”. In: *Sustainable energy technologies and assessments* 47 (2021), p. 101370.

[45] Per Runeson et al. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[46] Stephan Seifermann et al. “Detecting violations of access control and information flow policies in data flow diagrams”. In: *Journal of Systems and Software* 184 (2022), p. 111138.

[47] Vibhu Saujanya Sharma and Kishor S Trivedi. “Quantifying software performance, reliability and security: An architecture-based approach”. In: *Journal of Systems and Software* 80.4 (2007), pp. 493–509.

[48] VA Stafford. “Zero trust architecture”. In: *NIST Special Publication* 800 (2020), p. 207.

[49] Songpon Teerakanok, Tetsutaro Uehara, and Atsuo Inomata. "Migrating to zero trust architecture: Reviews and challenges". In: *Security and Communication Networks* 2021 (2021), pp. 1–10.

[50] Wei Tian et al. "Telerobotic spinal surgery based on 5G network: the first 12 cases". In: *Neurospine* 17.1 (2020), p. 114.

[51] Fatih Turkmen and Bruno Crispo. "Performance evaluation of XACML PDP implementations". In: *Proceedings of the 2008 ACM workshop on Secure web services*. 2008, pp. 37–44.

[52] Dan Tyler and Thiago Viana. "Trust no one? a framework for assisting healthcare organisations in transitioning to a zero-trust network architecture". In: *Applied Sciences* 11.16 (2021), p. 7499.

[53] Romans Vanickis et al. "Access control policy enforcement for zero-trust-networking". In: *2018 29th Irish Signals and Systems Conference (ISSC)*. IEEE. 2018, pp. 1–6.

[54] Manfred Vielberth and Günther Pernul. "A security information and event management pattern". In: (2018).

[55] Rory Ward and Betsy Beyer. "Beyondcorp: A new approach to enterprise security". In: (2014).

[56] Eric Yuan, Naeem Esfahani, and Sam Malek. "A systematic survey of self-protecting software systems". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 8.4 (2014), pp. 1–41.

[57] *Zero trust architecture design principles*. URL: `https://www.ncsc.gov.uk/collection/zero-trust-architecture`.