View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Online Publikationen der Universität Stuttgart

Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Securing Cloud Service Archives for Function and Data Shipping in Industrial Environments

Muhammad Ali Haider

Course of Study: M.Sc. Computer Science

Examiner: Prof. Dr. Dr. h. c. Frank Leymann

Supervisor: Michael Zimmermann, M.Sc.

Commenced: February 1, 2017

Completed: August 1, 2017

CR-Classification: C.2.4, D.2.11, D.4.6, E.0

Abstract

Cloud Computing paradigm needs a standard for portability, and automated deployment and management of cloud services, to eliminate vendor lock-in and minimization of management efforts respectively. Topology and Orchestration Specification for Cloud Applications (TOSCA) language provides such standard by employing semantics for representation of components and business processes of a cloud application. Advancements in the fields of Cloud Computing and Internet of Things (IoT) has opened new research areas to support 4th industrial revolution (Industry 4.0), which in turn has resulted in the emergence of smart services. One application of smart services is predictive maintenance, which enables the anticipation of future devices' states by implementing functions, for example, analytics algorithms, and collecting huge amounts of data from sensors. Considering performance demands and runtime constraints, either the data can be shipped to the function site, called data shipping or the functionality is provisioned closely to the data site, called function shipping. However, since this data can contain confidential information, it has to be assured that access to the data is strictly controlled. Although TOSCA already enables defining policies in general, a concrete data security policy approach is missing. Moreover, constituents of TOSCA are packaged in a self-contained and portable archive, called Cloud Service Archive (CSAR), which is also required to be secured and restricted to authorized personals only.

Taking the above facts into account, the goal of this thesis is to refine and enhance the TOSCA standard to the field of smart services in production environments through the usage of policies, for example, being effectively able to define the security aspects. In this thesis, various available policy languages with frameworks supporting them are researched, and their applicability for the field of Industry 4.0 is analyzed. An approach is formulated with one language selected, to define policies for TOSCA compliant cloud applications. Furthermore, a prototype is developed to secure the content of CSAR using the proposed approach.

Contents

1 Introduction 15

2 Fundamentals 19
2.1 Cloud Computing 19
2.2 Topology and Orchestration Specification for Cloud Applications (TOSCA) 20
2.3 Smart Services 25
2.4 Function Shipping versus Data Shipping 26

3 Policy Language Characteristics and Frameworks 27
3.1 Policy Frameworks 27
3.2 Characteristics of Policy Languages 31

4 Shortlisted Policy Languages 37
4.1 TPL/ DTPL 37
4.2 EPAL 40
4.3 XACML 42
4.4 Ponder 45
4.5 WS-Policy 48

5 Policy Based Approach to Secure TOSCA-based Cloud Services 53
5.1 Evaluation of Discussed Policy Languages 53
5.2 Approach to Address Security Requirements of Smart Services 59

6 Prototype to Secure Cloud Service Archives 67
6.1 Technologies 67
6.2 Secure CSAR Use cases and Algorithms 68

7 Conclusion and Future Work 81

Bibliography 83

List of Figures

2.1 Constructs of TOSCA 21
2.2 Sample CSAR Directory Structure 25

3.1 Trust negotiation through bilateral exchange of digital certificates 28
3.2 Policy Core Information Model 29
3.3 Overview of Sticky Policy Approach 31

4.1 EPAL Policy Enforcement Model 41
4.2 XACML Context 43
4.3 Policy Implementation in Ponder 46
4.4 WS-Policy assertions intersection example 50

5.1 Correspondence between the decision requests of XACML and EPAL . . . 56
5.2 Correspondence between the elements of XACML and WS-Policy framework 58
5.3 An approach to secure cloud service archives 62

6.1 Encrypt CSAR Service - Algorithm Flowchart 70
6.2 Sign CSAR Service - Algorithm Flowchart 72
6.3 Verify CSAR Service - Algorithm Flowchart 75
6.4 Decrypt CSAR Service - Algorithm Flowchart 77

List of Tables

4.1 EPAL Rule Example - Protecting a chemical formula 41

5.1 Comparison of Policy Languages with respect to the Defined Characteristics 54
5.2 Possible solutions of smart services security requirements 60

List of Listings

2.1 TOSCA syntax to define an application . . . 23
2.2 Structure of first block of TOSCA meta file . . . 24
2.3 Structure of non-first blocks of TOSCA meta file . . . 24

4.1 TPL simple constraints example - Webservice accessibility to member devices of a trusted manufacturing industry with name "Smart Industry" 38
4.2 TPL external function call example - Challenge-response to prove that the mobile device possesses the corresponding private key . . . 39
4.3 EPAL Syntax Example - Protecting a chemical formula . . . 42
4.4 XACML example - Login to servers in Germany is only allowed between 9AM and 5PM . . . 44
4.5 Ponder example - authorization policy definition [DDLS01] . . . 46
4.6 Ponder example - policy type and instantiation [Slo] . . . 47
4.7 Ponder example - obligation policy definition [Slo] . . . 47
4.8 WS-Policy example - Using compact form expression . . . 49
4.9 WS-Policy example - Using normal form expression . . . 49
4.10 WS-Policy example - Reffering policies using WS-Policy Attachment . . . 49
4.11 WS-Policy schema - Including one policy in another . . . 50
4.12 WS-Policy example - Attaching a policy to a webservice using WS-Policy Attachment . . . 51

5.1 Secure smart service example - X509 Authentication using WS-Policy . . 63
5.2 Secure smart service example - Policy Type definition database encryption for storage (Adopted from [OAS13b]) . . . 63
5.3 Secure smart service example - Policy Template definition database encryption for storage (Adopted from [OAS13b]) . . . 64

6.1 CSAR TOSCA.meta file - Encrypted entry . . . 69
6.2 CSAR TOSCA.meta file - Signed entry . . . 73
6.3 CSAR TOSCA.meta file - Signing a CSAR multiple times with different digest algorithms . . . 74
6.4 Signature file (with .SF extension) of a signed CSAR . . . 74
6.5 Secure CSAR Policy Template for Signing and Encryption . . . 78
6.6 Secure CSAR Policy Schema for Signing and Encryption . . . 79

6.7 Secure CSAR Policy Type for Signing and Encryption 80

List of Abbreviations

API Application Program Interface. 19

BPMN Business Process Model and Notation. 20

CA Certificate Authority. 30

CSAR Cloud Service Archive. 3

DMTF Distributed Management Task Force. 28

DTPL Definite Trust Policy Language. 37

EAR Enterprise Application aRchive. 21

EPAL Enterprise Privacy Authorization Language. 39

GUI Graphical User Interface. 65

IaaS Infrastructure as as Service. 19

IETF Internet Engineering Task Force. 28

IoT Internet of Things. 3

JSON JavaScript Object Notation. 67

LDAP Lightweight Directory Access Protocol. 45

MVC Model View Controller. 67

OASIS Organization for the Advancement of Structured Information Standards. 15

PaaS Platform as a Serivce. 19

PAP Policy Authorization Point. 28

PCIM Policy Core Information Model. 28

PDP Policy Decision Point. 28

PEP Policy Enforcement Point. 28

PII Personally Identifiable Information. 29

PIP Policy Information Point. 28

PR Policy Repository. 28

QoS Quality-of-Service. 61

REST REpresentational State Transfer. 60

SaaS Software as a Service. 19

SAML Security Assertion Markup Language. 50

SLA Service Level Agreement. 26

SOAP Simple Object Access Protocol. 50

TA Trusted Authority. 29

TOSCA Topology and Orchestration Specification for Cloud Applications. 3

TPL Trust Policy Language. 37

W3C World Wide Web Consortium. 39

WAR Web application ARchive. 67

WSBPEL Web Services Business Process Execution Language. 20

WSDL Web Services Description Language. 51

XACML eXtensible Access Control Markup Language. 42

1 Introduction

Cloud Computing is a paradigm that enables to access computing resources (e.g., networks, servers, storage, applications, and services) over a network ubiquitously [MG11]. Cloud Computing has revolutionized Information Technology (IT), assisting the idea of IT services being used in the same way we use utility services like electricity, natural gas, water, etc., with supporting different pricing models like pay-per-use [BBKL14]. Cloud-based services can be easily scale-up and scale-down by adjusting services' deployment instances with changing bandwidth requirements. In addition to that, a lot of other benefits of Cloud Computing as well have convinced many businesses to move to the cloud. When enterprises now prefer to use cloud services to create and deploy their application, they find it hard to move from one cloud provider to another cloud provider. Generally, pieces of an application need to be adjusted when migrating to a new cloud provider [SAL15]. This problem is known as vendor lock-in. TOSCA is a standard for Cloud Computing which solves this problem.

TOSCA is a Cloud Computing standard from Organization for the Advancement of Structured Information Standards (OASIS) which enables to create a cloud application, seamlessly manageable across any TOSCA compliant cloud provider. Besides supporting portability of application between different cloud providers, TOSCA also enables automatic deployment and management of application, and interoperability through the usage of TOSCA service templates. TOSCA achieves this by providing a metamodel to describe the structure and management lifecycle of cloud applications as topologies. The topologies define the components, and relationships between the components of a cloud application. The management plans can also be optionally defined to describe each step of how to achieve what is being defined in the topologies. The plans orchestrate management operations exposed by the components to define management tasks. The files of a TOSCA compliant application are packaged in a self-contained and portable archive, called Cloud Service Archive (CSAR) [OAS13b].

Internet of things (IoT) is a concept that has changed the way we interact with the physical world. IoT has no universally agreed definition but it is safe to say that it is a collaboration of devices (things) embedded with sensors, actuators, and software, over a network (or Internet) with active data exchange [AKN17; WIK17]. The availability of cheap sensors, growing access to high-speed internet, and advancements in the field of

Cloud Computing have promoted IoT to get the attention of many enterprises and have triggered 4th Industrial Revolution (also called Industry 4.0).

Industry 4.0 is an attempt to augment conventional pieces of machinery to become smart cyber-physical systems [WIKa]. An important use case of the Industry 4.0 is the predictive maintenance of the machinery [FBC+16]. The goal of this idea is to improve production flow of industrial set-up by detecting any potential risk of failure of machines (devices) and adjusting configurations of any machine in the industrial set-up which is not functioning to its fullest. This can be made possible by using smart services which implement functions (e.g, analytics algorithms) and process the data from sensors in the industrial environment. The data could be static as part of smart service, or a reference to live data could be given to the service. The huge amount of metered data has to be analyzed and processed in a timely manner by these functions, to optimize system functioning at runtime. Moreover, this data contains confidential and proprietary information of the industry, employing the smart services. Considering security and runtime issues, for instance, latencies and limited bandwidth, it does not suffice to transfer this data to a central processing system and give a response to trigger actuation. Therefore, two approaches, *function shipping* and *data shipping* are used in this context. In function shipping, functionality is shipped to the data source; and in data shipping, data is transferred to the central execution environment, hosted in-house or in a public cloud environment [FBC+16].

The whole business process goes as follows. Service developer designs and implements a smart service based on TOSCA specifications, and optionally publishes it to public repository or marketplace. Depending on permissions and licenses, another developer can customize the service algorithm according to its specific industry needs. Customers can search and purchase the service from the marketplace or directly from service developer, and use it by deploying it to a TOSCA compliant cloud provider. This raises many points of concerns, particularly regarding integrity and security [FBC+16].

The implementation of the smart services is packaged in CSAR. It is very critical to secure the contents of CSAR from unauthorized access as the logic of algorithm and static data in CSAR can be analyzed and re-engineered to exploit trade secrets of an industry. Moreover, the algorithm is an intellectual property of the developer, and unless granted rights, cannot be reused, manipulated or adopted; hence integrity and security of the contents of CSAR must be maintained. Further, metered data from a production process contains trade secrets and confidential information of an industry, hence transport and persistency of the data must be done in an encrypted way. And there must be a way to define data policies, for example, "the metering data must not leave the IT infrastructure of the customer that will run the smart service" [FBC+16]. Although TOSCA already enables defining policies in general, a concrete data security policy approach is missing [FBC+16].

The agenda of this master's thesis is to research on available policy languages and their applicability to Industry 4.0, and formulate an approach to fulfill above-mentioned security requirements of smart services. To accomplish this, non-functional requirements of any system based on concepts of Cloud Computing and Industry 4.0 are gathered. A conclusion is drawn that which policy language best suits to define the gathered requirements. If the selected policy language does not satisfy any security requirement, then the possibility to extend the selected policy language or development of new policy language out of it is researched. Concrete use-cases specific to common security requirements are formulated. An approach is outlined to define the security requirements of smart service ecosystem. This thesis also contains a prototype, built to encrypt, sign, verify, and decrypt the CSAR with finalized approach.

This thesis is structured in the following way:

Chapter 2 – Fundamentals explains the concepts this thesis is based upon. This includes a brief outline of concepts of Cloud Computing, detailed explanation of TOSCA standard and its constructs, and overview of smart services. Finally, the approaches of function shipping and data shipping in industrial environments are discussed.

Chapter 3 – Policy Language Characteristics and Frameworks discusses the characteristics, defining criteria for policy language selection. Further, some of the popular policy frameworks are explained in this chapter. The research on policy language characteristics and available policy framework are beneficial to shortlist a set of policy languages to base the research of this master' thesis.

Chapter 4 – Shortlisted Policy Languages analyses various policy languages, their pros, and shortcomings. The policy languages are shortlisted based on criteria inferred from research work already done and relevance of the policy language to the security domain.

Chapter 5 – Policy Based Approach to Secure TOSCA-based Cloud Services discusses the comparisons of researched policy languages. One language is selected which best suits the characteristics defined in Chapter 3 and which is able to cater the security requirements of Industry 4.0 in the best way. In the end, an approach is defined to secure Cloud Service Archives for Function and Data Shipping in Industrial Environment.

Chapter 6 – Prototype explains the details of the prototype, built to secure cloud service archives for function and data shipping. This chapter starts off with the details on technologies used in the prototype. Then it discusses the use cases of the prototype and the algorithms used to implement the use cases. Finally, a brief guideline of using this prototype is also a part of this chapter.

Chapter 7 – Conclusion and Future Work summarizes the work done in this thesis and points to the need for future research relevant to the topic of this thesis.

2 Fundamentals

This chapter explains the fundamentals and concepts, which this thesis is based upon. This includes a brief outline of Cloud Computing and detailed explanation of TOSCA constructs. A basic idea of Internet of Things (IoT), Industry 4.0, and smart services is also included. Finally, this chapter discusses the approaches of function shipping and data shipping in industrial environments.

2.1 Cloud Computing

Cloud Computing is a paradigm shift in Information Technology. For the sake of completeness, the definition of Cloud Computing from National Institute of Standards and Technology (NIST) is stated as it is, as follows:

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [MG11]

NIST categorizes Cloud Computing in three service models, including Software as a Service (SaaS), Platform as a Serivce (PaaS), and Infrastructure as as Service (IaaS). Some of the important actors involved in the use of cloud applications are cloud provider (cloud vendors), service provider (cloud consumer), and service consumer. In one line, the service providers create cloud services/ applications, which are hosted on the infrastructure provided by the cloud providers, and consumed by the service consumers. The cloud providers provide different services to service providers (developers) based on the service model of cloud providers. For example, a PaaS cloud provider may provide different database Application Program Interfaces (APIs) to developers to build their applications, facilitating developers to set-up their database online, without having to set up the required infrastructure in-house, thus saving cost and time. A SaaS cloud provider may provide end users with services to manage their documents online so that service consumers do not have to install any software on their personal computers. Similarly,

IaaS cloud providers provide virtual infrastructure like virtual storage, operating systems, etc. which can minimize costly IT maintenance of setting up own infrastructure.

Cloud services are usually comprised of a comprehensive software stack and complex deployment procedures. Businesses keep evolving, and so their needs change. There may be times when current cloud provider cannot fulfill the needs of a business (service provider). In such cases, the service provider may have to migrate to another cloud provider. This requires the service provider to patch, configure, and change parts of their service, costing them time and money. This problem is called as vendor lock-in. TOSCA is a Cloud Computing standard which has solved this problem [Lip13].

2.2 Topology and Orchestration Specification for Cloud Applications (TOSCA)

When the problem of vendor lock-in was realized, a consortium of cloud vendors created the Cloud Computing standard TOSCA and proposed a new technical committee in OASIS [Lip13]. OASIS is "[...] a global nonprofit consortium that works on the development, convergence, and adoption of standards for security, Internet of Things, [...]" [WIKb]. TOSCA enables service providers to create cloud provider agnostic applications by defining *application topology* and *management plans* (Orchestration Specification for Cloud Applications).

2.2.1 Constructs of TOSCA Language

The application topology defines the structure of an application using TOSCA language, and the management plans use business process modeling languages like Business Process Model and Notation (BPMN) [MN] and Web Services Business Process Execution Language (WSBPEL) [TC] to define management of the application lifecycle. Figure 2.1 is the bird-view of constructs of TOSCA language.

Application topology defines the structure of a cloud application in terms of *Node Type*, *Node Template*, *Relationship Type*, *Relationship Template*, *Deployment Artifact*, *Implementation Artifact*, *Policy Type*, and *Policy Template*, with the usage of service template.

A Node Type defines the structure of a node in terms of properties a node can have, for example, username, password, etc, and management operations, for instance, install, start, stop, etc. The corresponding Node Templates set the value of these properties, hence initializing the topology template. There can be many Node Templates based on a Node Type. The concept of types and templates enable reusability as the same Node

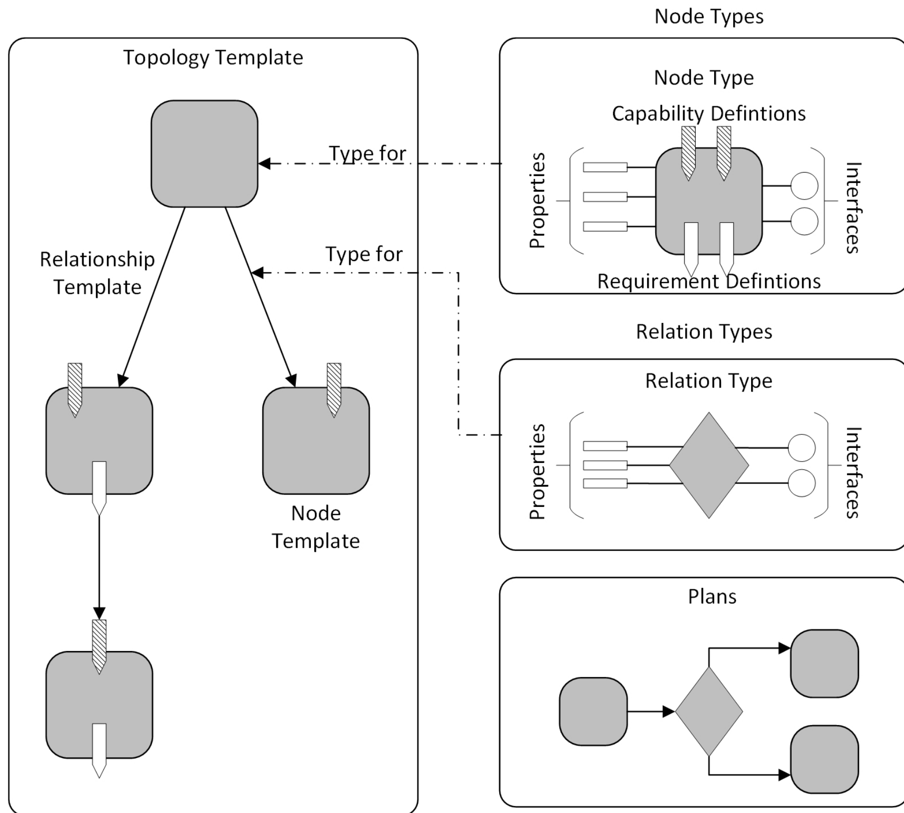

Figure 2.1: Constructs of TOSCA [OAS13b]

Type can be used for many templates, even across different cloud applications. The same concept is used for *Relationship Types* and *Relationship Templates*. A Relationship Type with corresponding Relationship Template defines the relationship between two nodes. Relationship Templates are directional, meaning they have a source Node Template and a target Node Template, and they set properties defined by corresponding Relationship Type.

The example Service Template (Listing 2.1), depicting the structure and management lifecycle of a hypothetical payroll application, can be taken to understand the constituents of the application topology. The application can be packaged as an Enterprise Application aRchive (EAR), and deployed to an application server, for example, `IBM WebSphere Application Server` (a well-known application server). The application server must be installed on an operating system, for example, `Ubuntu` (a flavor of Linux operating system). In this scenario, the required Node Types may be `EAR Application`, `Application Server`, and `Operating System`, where `Payroll Application`, `IBM WebSphere Application Server`, and `Ubuntu Operating System` be the corresponding Node Templates [OAS13b]. The example shows `deployed on` Relationship Template, defining that payroll application is deployed on `IBM WebSphere`

`Application Server`, where the Relationship Template can set any property required for deployment. The corresponding Relationship Type (not shown in the example) declare those properties. This is how TOSCA language defines the structure of an application. The business logic, for example, code, scripts, etc. are defined by *Deployment Artifacts*. On other hand *Implementation Artifacts* support the business plan (service orchestration for deployment and management) by implementing management operations exposed by the Node Types.

A Deployment Artifact is associated with a Node Template, and defines the concrete instance of a node. For example, in the above example, `Payroll.ear` file could be Deployment Artifact associated with the `Payroll Application` Node Template. Implementation artifacts support management of a node. For example, in the above example, the `EAR Application` Node Type have an operation `start` that is implemented by an Implementation Artifact `payrolladm.jar` [OAS13b]. Further, Node Templates can have capabilities and requirements which can be defined using TOSCA *Capability Types* and *Requirement Types* constructs. They are used to find a suitable Node Template fulfilling the requirements of another Node Template, for example, memory, bandwidth, version, etc. In the discussed example, `IBM WebSphere Application Server` Node Template can define requirements in its definition that the node can only be installed on the `Linux operating system`.

TOSCA language also enables defining policies for a Node Type using *Policy Type* and *Policy Template*. Policy Type defines the place holders for policy properties using XML schema. Policy Templates define the actual values of the properties. For example, in the above example, the `Application Server` Node type may have "high availability" policy with heartbeat frequency" count defined by a Policy Type, whose actual value is set by the corresponding Policy Template, keeping the `Application Server` up and running with high availability [OAS13b].

Service Template file can refer to constructs of TOSCA from other TOSCA definition files as well. For example in the Listing 2.1, the corresponding Node Types and Relationship Types are being referred from another TOSCA definition document, `PayrollTypes.tosca`. Artifacts can be referred even from other CSARs, for example, Deployment artifact `Payroll.ear` is being referenced from another CSAR in Listing 2.1.

2.2.2 Cloud Service Archive (CSAR)

All the files related to application topology, for example, schemas, templates, types, definitions, binary files, and business process related files, are packaged in a ZIP file (with extension ".csar") known as Cloud Service Archive (CSAR). The CSAR follow a standard format which enables it to get deployed in any TOSCA compliant cloud provider, hence

Listing 2.1 TOSCA syntax to define an application (adopted from [OAS13b, p.77-78])

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Definitions xmlns:pay="http://www.example.com/tosca/Types" id="PayrollDefinitions"
       targetNamespace="http://www.example.com/tosca">
3  <Import namespace="http://www.example.com/tosca/Types"
       location="http://www.example.com/tosca/Types/PayrollTypes.tosca" importType="
       http://docs.oasis-open.org/tosca/ns/2011/12" />
4  <Types>...</Types>
5  <ServiceTemplate id="Payroll" name="Payroll Service Template">
6  <TopologyTemplate ID="PayrollTemplate">
7  <NodeTemplate id="Payroll Application" type="pay:EAR Application">
8  ...
9  <DeploymentArtifacts>
10 <DeploymentArtifact name="PayrollEAR" type="http://www.example.com/ ns/tosca/2011/12/
       DeploymentArtifactTypes/CSARref">EARs/Payroll.ear</DeploymentArtifact>
11 </DeploymentArtifacts>
12 ...
13 <!-- ImplementationArtifacts -->
14 ...
15 </NodeTemplate>
16 <NodeTemplate id="IBM WebSphere Application Server" type="pay:Application Server">
17 <DeploymentArtifacts>
18 <DeploymentArtifact name="IBMWebSphereAS" type="http://www.example.com/ ns/tosca/2011/12/
       DeploymentArtifactTypes/WebSphereASref">ibm-websphere-edf2cf99</DeploymentArtifact>
19 </DeploymentArtifacts>
20 </NodeTemplate>
21 ...
22 <!-- Linux Operating System node template -->
23 ...
24 <RelationshipTemplate id="deployed_on" type="pay:deployed_on">
25 <SourceElement ref="Payroll Application" />
26 <TargetElement ref="IBM WebSphere Application Server" />
27 </RelationshipTemplate>
28 </TopologyTemplate>
29 </ServiceTemplate>
30 </Definitions>
```

enabling portability. The CSAR contains at least two directories, the *TOSCA-Metadata* directory and the *Definitions* directory. Apart from these two directories, the creator of CSAR can have any structure in CSAR; what suits best for the cloud application modularity.

The TOSCA-Metadata directory essentially has a TOSCA meta file, *TOSCA.meta* which is consisted of name/value pairs. A name/value pair is separated by a colon and followed by a space. The name must not have any colons in it. Each name/value pair is on a new line. Related name/value pairs are separated by an empty line. The consecutive

Listing 2.2 Structure of first block of TOSCA meta file [OAS13b, p.75]

```
1 TOSCA-Meta-File-Version: digit.digit
2 CSAR-Version: digit.digit
3 Created-By: string
4 Entry-Definitions: string ?
```

Listing 2.3 Structure of non-first blocks of TOSCA meta file [OAS13b, p.76]

```
1  Name: <path-name_1>
2  Content-Type: type_1/subtype_1
3  <name_11>: <value_11>
4  <name_12>: <value_12>
5  ...
6  <name_1n>: <value_1n>
7
8  ...
9
10 Name: <path-name_k>
11 Content-Type: type_k/subtype_k
12 <name_k1>: <value_k1>
13 <name_k2>: <value_k2>
14 ...
15 <name_km>: <value_km>
```

name/value pairs are called as *block*. Each block describes an artifact in CSAR. The first block represents the CSAR itself [OAS13b]. The content of the first block in the TOSCA meta file is shown in Listing 2.2.

Each block except the first block has the first name/value pair, with key name "Name", referring to the path of the artifact within CSAR (relative to CSAR). Each non-first block of TOSCA meta file also have a key *Content-Type* which tells about the MIME type of the referred artifact. The structure of non-first blocks of TOSCA meta file is shown in Listing 2.3.

Figure 2.2 illustrates the content of CSAR for the example payroll application that we discussed in the previous section.

Having discussed all the constructs of TOSCA standard, it is important to realize that TOSCA only presents a grammar for describing cloud applications by means of topology templates and business plans [OAS13b]. TOSCA needs a runtime to deploy and manage instances of cloud application as defined in the Service Template of CSAR. "Without such a container, TOSCA could be used as pure exchange format and manually operated according to the definitions in the Service Template" [BBKL14].

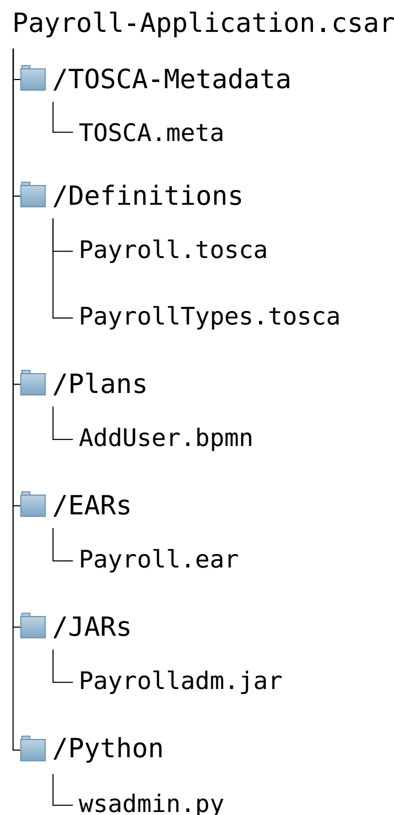

Figure 2.2: Sample CSAR Directory Structure [OAS13b, p.78]

2.3 Smart Services

Devices (Things) embedded with electronics, for example, sensors and actuators, etc. are interconnected with the usage of a shared platform, such as cloud services, to perform functions adaptively, on their own, or with the input from the physical world [AKN17]. Industry 4.0 is inspired from the concepts of IoT. Four design principals of Industry 4.0 as mentioned on [WIKa] are Interoperability, Information transparency, Technical assistance, and Decentralized decisions. Interoperability means the ability of devices, sensors, machines, actuators, and humans to connect with each other via Internet of Things (IoT). Information transparency is the ability of the system to create a virtual copy of the physical model by aggregating data from all the devices and sensors in the industrial environment. Technical assistance design principals mean that the system should update the users about the status of the industrial environment so that the users can take decisions effectively. The last design principal of Industry 4.0, Decentralized decisions enables the system to take decisions on its own. Only if there is a conflict in decisions, request for decision is delegated to humans.[WIKa]

In general, a smart service is just a smart (web) service with provided awareness of context, makes it "*reactive* or even proactive" [AL05]. In the context of Industry 4.0, one application of the smart services is predictive maintenance of the devices in an industrial environment. The idea is that all the mounted sensors send data to the smart service. The smart service based on the metered data, categorize a device as good or bad. Good means that the device is functioning correctly and bad means that the device is not functioning correctly and needs to be replaced or re-configured.

Considering the advantages of Cloud Computing and TOSCA standard, as discussed in the previous sections, the smart services are deployed in a TOSCA compliant cloud provider. Hence they are packaged as CSAR. The business process is that a data scientist (service developer) develops an analytics algorithm and put it in CSAR. Apart from the algorithm, CSAR may have static data, for example, if the analytics algorithm needs some data that does not change over time, and reference to the data source of sensors. Further, policies to manage the smart service, for example, a security policy stating that metering data must not leave the industrial environment, is also part of the CSAR. The two approaches to incorporate a smart service in an industrial environment are *function shipping* and *data shipping* [FBC+16].

2.4 Function Shipping versus Data Shipping

Function Shipping and Data Shipping are two approaches that are used for incorporating Smart Services in industrial environments. In general, function shipping means that the function, for example, analytics algorithm is provisioned close to a data source. On other hand, in data shipping, data from the data source is provisioned to the function. In the function shipping approach, data ownership is more in control as the data never leaves the environment where it is created. Further, function shipping enables to perform parallel processing in a timely manner. This is because, in real-time systems, the existence of high latency hinders transport of data (request), and command to actuators (response) in a parallel manner [FBC+16].

But sometimes, processing of metered data from sensors require a powerful processing environment. Thus, insufficient IT infrastructure close to the data source requires shipping and aggregation of data from sensors to a centralized location. This is also the case when the purpose of use-case is only to analyze the data and not real time processing. But this approach is only suitable if the industry does not have security constraints on their data, and the industry can afford to have their data leave their premises. In such cases, the smart service is deployed in self-hosted or public TOSCA compliant cloud infrastructure. It totally depends on the use-case requirements, and Service Level Agreements (SLAs), that which approach is used [FBC+16].

3 Policy Language Characteristics and Frameworks

This chapter discusses some of the existing frameworks for policy negotiation between different parties. It also reviews the characteristics of policy languages, some of which may count for the goodness of a policy language. The research in this chapter is beneficial to shortlist a set of policy languages to base the research of this master thesis.

3.1 Policy Frameworks

Although the purpose of this thesis is to research on available policy languages, the reason of discussing policy frameworks is that the policy languages need a policy framework to work with.

3.1.1 Trust Negotiation Model

The first framework worth considering is proposed by Seamons et al. [SWY+]; even though the work was done in 2002, but their research is still considered valuable and forms the basis of a lot of researches in policy languages. Their work was mainly about the requirements for the policy languages, but they also introduced a model based on which a trust negotiation between two parties can be made. Figure 3.1 illustrates main components of Trust Negotiation Model. It is a very basic model to exchange policies, with the consideration that policies themselves are confidential as well. The model is peer-to-peer, which means that both parties hold sensitive resources, policies, and credentials.

Trust negotiation is initiated when a client wants to access a secret resource on a server. The server demands credentials from the client to access that resource, by using policies. The client cannot give the credentials straight away to the server. Thus, before establishing trust, the client may also require the server to provide, for example, a digital

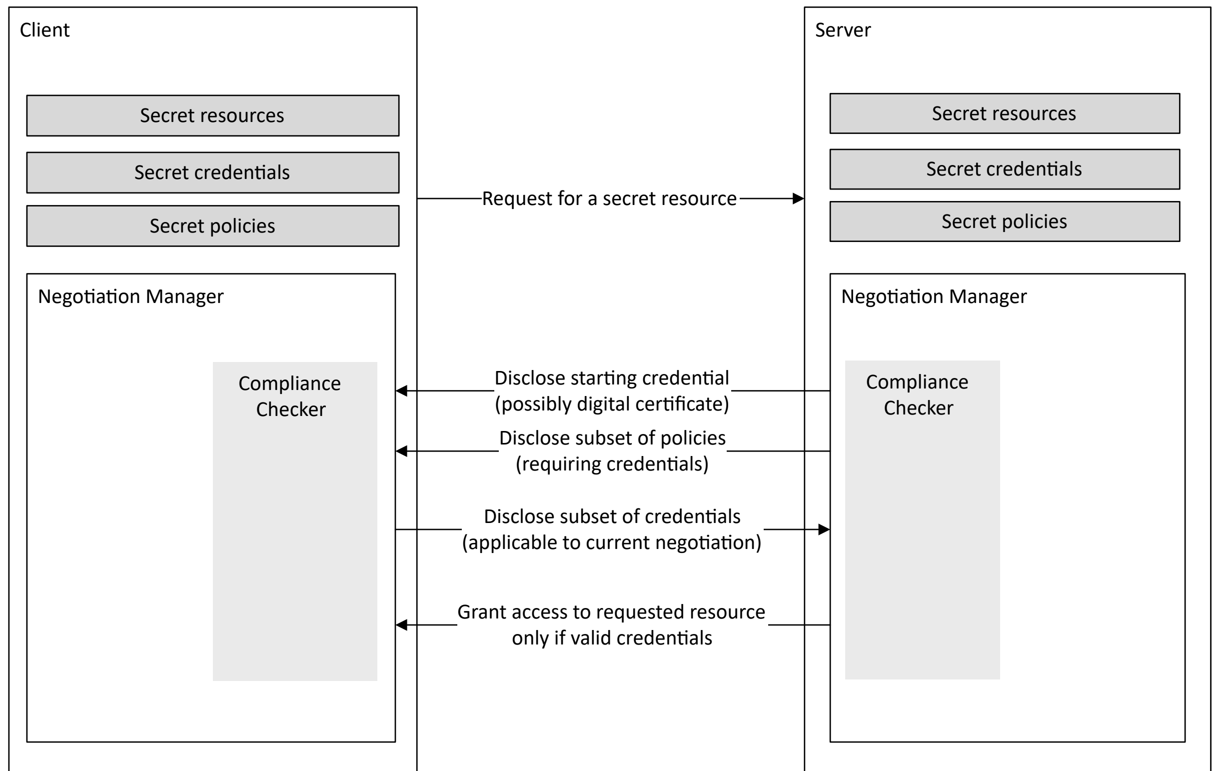

Figure 3.1: Trust negotiation through bilateral exchange of digital certificates [SWY+]

certificate, certifying the server to be a part of a trusted organization. So besides the initial policy, the server must also give a starting credential to the client [SWY+].

The policies themselves are confidential as well because by viewing requirements specified in a policy, sensitive information about a party could be inferred. Considering this, the exchange of such policies happen in round-trips comprising a set of negotiations. Each negotiation only exposes a subset of policies. If both parties satisfy each other policy requirements, access to the secret resource is granted to the client [SWY+].

Negotiations are managed by a *negotiation manager*. The manager is responsible for regulating a strategy which determines "which credentials and policies to disclose, and when to disclosed them" [SWY+]. The manager is assisted by a *compliane checker* which analyze whether the credentials against the policies under the current negotiation are satisfied or not. Based on the decision from compliance checker about an assertion of access control policies, negotiation manager could grant or reject access to the secret resource, or proceed with further negotiations [SWY+].

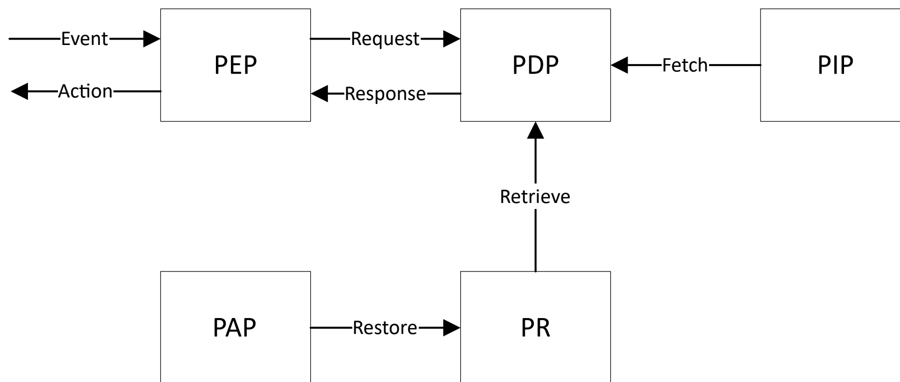

Figure 3.2: Policy Core Information Model (adopted from [HL12])

3.1.2 Policy Core Information Model

Policy Core Information Model (PCIM) was developed jointly by Internet Engineering Task Force (IETF) and Distributed Management Task Force (DMTF). It consists of three pivotal components: Policy Repository (PR), Policy Decision Point (PDP), and Policy Enforcement Point (PEP) [HL12]. There are two other components which have been repeatedly proposed and incorporated by policy language standards: Policy Authorization Point (PAP) and Policy Information Point (PIP). This should be noted that these two components are not part of actual IETF standard. But, their use makes a lot of purpose in the PCIM. Figure 3.2 illustrates how the components of PCIM fit in a policy-driven management system.

The PAP provides an interface to define, update, or delete policies. The policies are stored in the PR. The PEP keeps polling for any request or event to the system, and forward all the requests to the PDP. The PDP looks into PR for all the applicable policies and makes decisions like *permit*, *deny*, and *NotApplicable*, and give a response back to PEP. While making the decision, PDP may interact with PIP (systems like LDAP, database, web service, etc.), to fetch request detailed attributes. Finally, PEP takes the concrete action based on the decision made by the PDP [HL12].

3.1.3 Sticky Policies

Sticky policies is an approach as part of EnCoRe project [HP] to manage the privacy of data across multiple parties. By party, it means the organizations or service providers which need confidential data from the users for processing. The data is made available to parties with the consent of the users. The consents are defined in terms of machine-readable policies. The data is encrypted and attached to the policies, as the data travels across multiple parties [PM11]. The data could comprise anything but the

approach sticky policies was designed with motivation to protect Personally Identifiable Information (PII), for example, name, date of birth, address, phone number, credit card number, password, etc. The attached policies define how the data needs to be treated, including constraints, conditions, or compliance with standards. The policies also contain a list of Trusted Authorities (TAs) which provide keys to decrypt the data on the satisfaction of the policies [PM11].

Figure 3.3 illustrates the basic idea behind the management of sticky policies. Organizations which incorporate sticky policy framework publishes lists of supported policies and TAs. The data subject (data owner) encrypts the subset of the data which needs to be protected. Policies are created by the data subject and TAs are added to the policies. Finally, the created policies are defined on that data subset. The policy could include preferences like data should be deleted after 2 years of use, or data should not be made accessible to particular parties, etc. The data subject then includes a list of TAs which can be requested for the key to decrypt the data. The data subject may also encrypt different subsets of data with different keys. The system then sends the encrypted data with the sticky policies to the service provider. After receiving the encrypted data and sticky policies, the service provider sends the sticky policies to one of the listed TAs, with the assertion that it complies with the sticky policies. The TA can confirm this assurance from the service provider with methods, for example, verifying digital certificates from trusted Certificate Authorities (CAs) that the service provider may hold, or by verifying the absence of the service provider from a known blacklist, or verifying from an external reputation management system. The verification actions are logged by the TA, which creates an audit trail made available to the data owner and the TA in cases, for example, policy violation. Once the TA verifies all the policy requirements, the key to decrypt the data is released to the service provider [PM11].

Sticky policy approach does not have any fixed underlying encryption mechanism to protect data, though [PM11] suggests a public-key encryption technique to secure propagation of data along the service provision chain. This technique assumes that all stakeholders (data subject, service provider, and TA) possess certified public or private key pairs from trusted CAs. In this technique, data owner generates the policy, together with a symmetric key K. Data owner sends two items to service provider: (1) Data encrypted with K, (2) Sticky policy, having K appended to the policy's hash, with K and hash encrypted with TA's public key and signed with the data owner's private key. The signing makes it possible to verify the policy's authenticity and integrity. When the service provider receives the sticky policy with the encrypted data, it sends the sticky policy which contains encrypted K and policy hash, to one of the TAs. The TA decrypts K and policy hash, verifies the policy's signature with data owner's public key, and challenges the service provider for fulfillment of the policy. If all the verification steps get passed, TA sends K to the service provider by encrypting it with the service

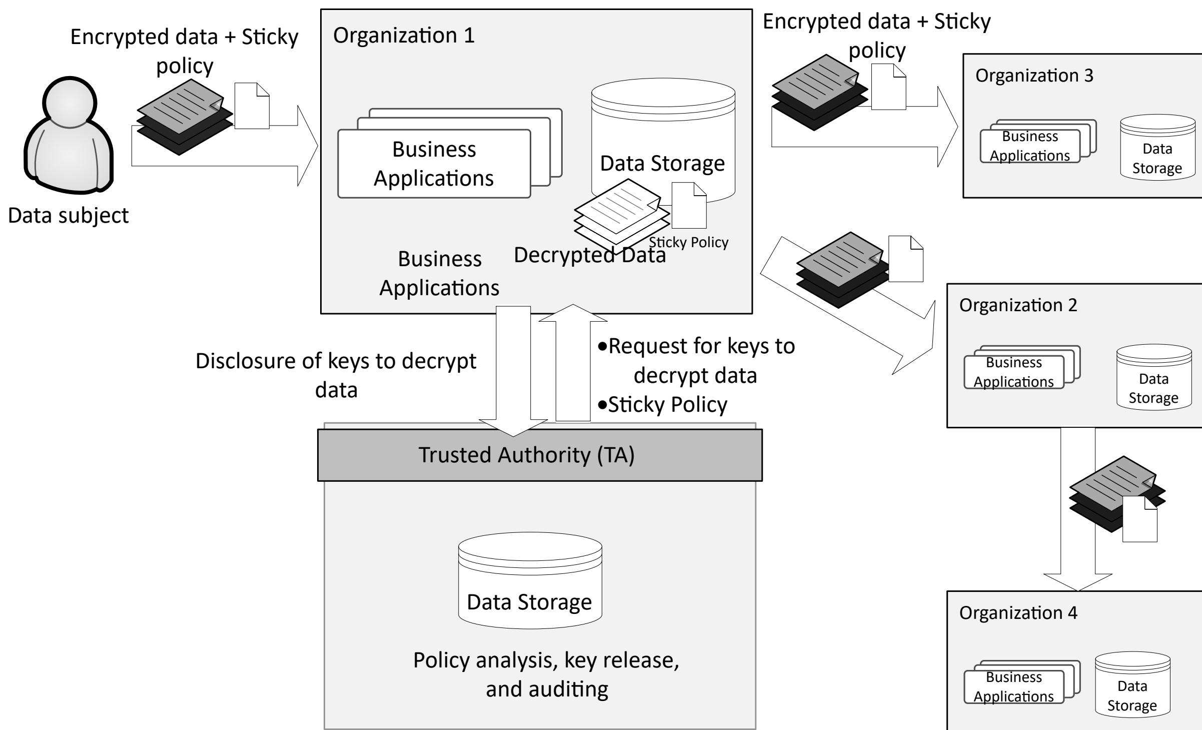

Figure 3.3: Overview of Sticky Policy Approach [PM11]

provider's public key. The service provider can then decrypt K with its private key and in turn decrypt the data using K [PM11].

3.2 Characteristics of Policy Languages

Seamons et al., 2002 [SWY+] presented requirements for a policy language in their work on the trust negotiation model. To date, a lot of advancements have been done in the field of policy languages, and dynamics of information technology have changed. But the work of Seamons et al. is still considered valuable. Duma et al., 2007 [DHS07] and Coi et al., 2008 [CO08] referred to [SWY+] for their work on the scenario based comparison of policy languages, and review of security policy languages respectively. Policy properties included in this section have been taken from [SWY+], [CO08], [DHS07], and [HL12]. The properties should only be considered as general characteristics of policy languages, and are not necessarily the criteria for a policy language to qualify for goodness. First fifteen characteristics are general language features and last four are related to credential support.

Well-defined semantics. A policy language semantics are well defined if its constructs are not implementation specific, for example, relational algebra is not implementation specific but SQL is. A language having this quality allows to confidently infer something from it, irrespective of how and when the language is used [CO08; SWY+].

Monotonicity. A policy language should be monotonic, which means the disclosure of more policies or more credentials should only give more privileges or access rights to the requester. For example, an insurance company ABC issues *ABC credentials* to its clients certifying them to be part of the insurance company. If a health care club web-service should not be accessible to the clients of ABC, imagine a policy language that grants access to the health care club web-service as long as the *ABC credentials* are not disclosed. From this case, it does not mean that negation of credentials should not be allowed, it only means that a policy language should allow policy writers to define the absence of credentials. Further, the monotonicity requirement does not prevent the use of inequality constraints or temporal predicates, for example, access to the health care club web-service should only be made between 06:00 and 21:00 [CO08; SWY+].

Condition expressiveness means expressiveness of a policy language in defining conditions under which a request should be fulfilled. Policy languages differ in expressiveness: Many policy languages allow to add constraints on attributes of requesters, values of credentials, or properties of the requested action. Some policy languages enable to define pre-conditions for a request to be considered. Further, some policy languages even allow to add constraints on environmental factors (for example, time), and a few also support the notion of *purpose* in the requests [CO08; SWY+].

Action execution is the ability of a policy language to execute an action during evaluation of a policy. For example, being able to get system current time and send it with the response of the evaluation. Some policies also enable executing an action depending on the outcome of an evaluation, for example, sending an email to a customer if his discount gets approved [CO08].

External functions. Some policy languages come with a library of functions, for example, date comparisons, time formatting, currency conversions, etc. The well-defined semantics of these functions enable the involved parties to agree on the same outcome of an evaluation of the policy [SWY+].

Negotiation support. Negotiation takes place when a request is initiated with the consideration of multiple policies individually defined by the parties involved in the negotiation, and evaluation of the request is done in a distributed way. The important point here is that not only the party offering a service has a policy, but also the party requesting for a service also have a policy. Policy evaluation is successful if both of the parties satisfy each other's policy requirements. For example, in the case of the sticky

policy approach that we discussed in the previous section, policy language ought to have this characteristic [CO08].

Type of evaluation. Those policy languages which support negotiation between multiple parties, perform the evaluation of policies in a distributed way. Each involved party evaluates a policy step by step, and advance the negotiation. This type is named as *distributed evaluation*. Some languages enable their runtime to break a policy on several nodes. Each node can further split the policy or return the evaluation result to the peer who sent the policy for evaluation. Finally, the root node could accumulate the results from each node, evaluate all the responses, and return the final result. This type of evaluation is called as *distributed policies*. The policy languages which do not offer negotiations, perform all their evaluations locally, hence the name *local evaluation* [CO08].

Policy engine decision. Policy language semantics should enable the policy engine to notify requester about the policy evaluation. This could be made possible by, for example, having a detailed policy response structure. The result sent back to the requester may consists of simple boolean value *true/ false*, or values like *allow*, *deny*, and *don't care*, or detailed reasons, for instance, why the evaluation of the policy got failed [CO08].

ECA. This refers to *Event*, *Condition*, *Action* paradigm. It is recommended that a policy language is based on ECA paradigm because this paradigm enables a policy language to handle more complex events, for example, asynchronous event calls, or composite policies. If a policy language is not based on ECA paradigm, then it follows *Condition-Action* way of evaluating policies [HL12].

Indexing. This characteristic allows a policy language to create indexes on entities of policy languages, for example, subject, resources, etc. Normally, there are a lot of policies in a system, and at runtime, there is the need for a policy engine to quickly access the required policy for evaluation. Indexes enable a policy engine to retrieve the required policy more efficiently [HL12].

RBAC stands for Role-Based Access Control. The policy languages which have this characteristic, perform policy evaluation by mapping subjects (for example, users, machines, etc.) to roles. Then, the roles are given permissions, and access control is calculated. It is good for a policy language to have this characteristic as it is easier to manage access control policies based on roles, rather than individual subjects. Also, the access permissions of a subject automatically change with role changes [HL12].

Obligation means that apart from performing an action based on a condition, perform another task. This task can be, for example, log the request, email the requester, start a scheduler, etc. The policy languages having this characteristic are able to produce a side effect based on a policy evaluation [HL12].

Extensibility is the ability of a policy language to facilitate extensions. A policy language should be extensible with new features, and customizable for a specific system needs [CO08].

Conflict Resolution is the ability of a policy language to resolve the conflicts if two policies are satisfied but both are contradictory, for example, one policy grants access to a web service call and the other denies the access. The policy engine cannot take both of the actions. In such cases, there must be some way in the language itself to resolve the conflicts and choose one action.

Protecting sensitive policies. A policy itself may also be confidential. For instance, if the policy protecting a person's medical record requires that the requester of the medical record must submit a certificate issued by a mental health organization, it could be inferred that the person may have some mental health problem. So during evaluation of policies, it should be able to protect such sensitive information either at policy syntax level or using the policy engine. For example, the policy engine could disclose sensitive policies only when required, or the policy content is encrypted, which can be decrypted with external function calls [SWY+].

Evidences. It should be able to specify signed statements (credentials) in a policy language and should be able to specify authentication requirements by means of external function calls or as part of policy language itself. For example, a student discount system must be able to authenticate that a customer indeed is the student mentioned in the student digital ID credential. There should be a mechanism to confirm a credential ownership by, for example, being able to refer to the private key associated with a public key referred to in a credential [CO08; SWY+].

Credential chains. A policy language should allow to construct and constrain a chain of credentials. This means that a policy language should be expressive enough to verify that the issuer name in one of the credential is the subject name of the next credential in a credential chain [SWY+].

Credential combinations. Many times, in order to establish trust, it is required to prove authenticity using multiple credentials from different trusted authorities. For example, a health care service may require student certificate and insurance certificate to offer a particular discount. This also ensures greater credibility because an attacker would have to fabricate several private/public key pairs before being able to get illegal access. So, policy language having this characteristic enable the policy writers to specify combinations of credentials using a conjunction, disjunction or other semantics [SWY+].

Inter-credential constraints. If two certificates are issued to the same subject, it is very likely that subject name is same in both of the certificates. For example, if a health care service may require student certificate and insurance certificate from a customer, then

the policy language should allow adding a constraint that subject name on both of the certificates is same or at least close enough [SWY+].

Based on the discussion included in this chapter, different policy languages are short-listed and will be discussed in the next chapter. In the later chapters, the discussed policy languages and their compliant frameworks will be compared, based on which an approach will be defined to secure applications of Industry 4.0, particularly smart services deployed in TOSCA runtime.

4 Shortlisted Policy Languages

The languages considered for this thesis are TPL/ DTPL, XACML, EPAL, X-Sec, Rei, PSPL, KeyNote, P3P/ APPEL, ASL, VALID, PDL, PFDL, WS-Policy, and Ponder. The languages are considered based on their popularity. Only the prominent features and the domain in which the considered policy languages deal in, are briefly researched. Out of these 14 policy languages, 5 policy languages are shortlisted for the detailed research due to their closeness with the security domain and applicability for the field of Industry 4.0. Further, the general characteristics defined in the previous section are also considered while selecting the policy languages.

4.1 TPL/ DTPL

Trust Policy Language (TPL) [IBM] created by IBM Research centre is an XML based policy language which maps requesters to *groups/ roles* based on the certificates they possess. The terms roles and groups are used in the language interchangeably. Definite Trust Policy Language (DTPL) is the subset of TPL which does not contain negative rules and hence is monotonic. The constructs of TPL are as follow.

The root of TPL is POLICY tag. The POLICY tag may contain one or more GROUP tags. Membership of the group is determined by multiple RULE tags. The RULE tag expresses constraints on a certificate using a sequence of INCLUSION and EXCLUSION tags and an optional FUNCTION tag. The INCLUSION tag adds basic constraints on existence of a certificate, for example, *type* and *issuer*. Similarly, the EXCLUSION tag adds basic constraints on the nonexistence of a certificate. Both INCLUSION and EXCLUSION tags have an attribute ID, which is used to refer the certificate associated with the RULE tag. The scope of ID attribute is only limited to the RULE tag. The FUNCTION tag is used to express complex constraints on a certificate by using TPL logical constructs like AND, OR, NOT, EQ, GE, etc. TPL expresses the reference to a field in a certificate using FIELD tag. The FIELD tag consists of two attributes: (1) ID attribute, which refers to INCLUSION or EXCLUSION ID (a unique id of the certificate where the field appears), (2) NAME attribute which refers to the name of a field in the certificate. TPL enables external function calls using EXTERN tag. The FIELD and EXTERN tags could come inside TPL

Listing 4.1 TPL simple constraints example - Webservice accessibility to member devices of a trusted manufacturing industry with name "Smart Industry"

```
1  <?xml version="1.0"?>
2  <POLICY>
3        <GROUP NAME="self"></GROUP>
4        <GROUP NAME="industries">
5              <RULE>
6                    <INCLUSION ID="indCrt" TYPE="memberIndustry" FROM="self"></INCLUSION>
7              </RULE>
8        </GROUP>
9        <GROUP NAME="mobileDevices">
10       <!-- device identification from an industry -->
11             <RULE>
12                   <INCLUSION ID="deviceCrt" TYPE="DeviceID" FROM="industries"></INCLUSION>
13                   <FUNCTION>
14                         <EQ>
15                               <FIELD ID="deviceCrt" NAME="Industry Name"></FIELD>
16                               <CONST>"Smart Industry"</CONST>
17                         </EQ>
18                   </FUNCTION>
19             </RULE>
20       </GROUP>
21 </POLICY>
```

logical construct tags. Listing 4.1 is the TPL snippet, defining policy requirements for accessibility of a web service only for mobile devices possessing a certificate issued by a particular trusted manufacturing industry with name "Smart Industry" [SWY+].

As can be seen in Listing 4.1, TPL has well-defined semantics. A special group, 'self' group refers to the self public key. The FROM attribute in INCLUSION and EXCLUSION tags refers to the name of a group. Only certificates signed by a member in that group should be eligible to evaluate the function in INCLUSION or EXCLUSION; this enables TPL to verify *credential chains*. The CONST tag allows TPL to express a comparison for the value of a field in a certificate with a constant value. The ability of TPL to express external function calls enable it to specify complex constraints, evaluate *Inter-credential constraints*, and allows requester to specify *evidences*. The ability of TPL to execute external functions can be used to develop a library of standard functions. One function could be *checkStrongAuthentication*, which at the time of role mapping evaluation decrypt some data. The function should be provided with certificate public key in the input, and the data to decrypt could be provided from within function call itself. For example, Listing 4.1 can be extended to decrypt some data as shown in Listing 4.2. Invoking this function would trigger a challenge-response to prove that the mobile device possesses the corresponding private key [SWY+].

Listing 4.2 TPL external function call example - Challenge-response to prove that the mobile device possesses the corresponding private key

```
1  <?xml version="1.0"?>
2  <POLICY>
3        <GROUP NAME="self"></GROUP>
4        <GROUP NAME="industries">
5              <RULE>
6                    <INCLUSION ID="indCrt" TYPE="memberIndustry" FROM="self"></INCLUSION>
7              </RULE>
8        </GROUP>
9        <GROUP NAME="mobileDevices">
10             <RULE>
11                   <INCLUSION ID="deviceCrt" TYPE="DeviceID" FROM="industries"></INCLUSION>
12                   <FUNCTION>
13                         <AND>
14                               <EQ>
15                                     <FIELD ID="deviceCrt" NAME="Industry Name"></FIELD>
16                                     <CONST>"Smart Industry"</CONST>
17                               </EQ>
18                               <EXTERN CLASS="checkStrongAuthentication">
19                                     <PARAM NAME="publicKey">
20                                           <FIELD ID="deviceCrt" NAME="publicKey"/>
21                                     </PARAM>
22                               </EXTERN>
23                         </AND>
24                   </FUNCTION>
25             </RULE>
26       </GROUP>
27 </POLICY>
```

In Listing 4.2, to make the external call possible, a class must be created that implements the *checkStrongAuthentication* interface. The name of the class should match the value of the CLASS attribute in the EXTERN tag.

TPL does not support sensitive policies but Seamons et al. [SY01] have proposed a way to protect sensitive policies by using the policy graphs. Formally, the policy graph is a directed acyclic graph with exactly one node and sink. Every node except the sink represents a policy. The sink represents the resource to be protected. A policy node n should only be disclosed if the outcome of all the policy nodes in the path to n (except policy node n itself) have been satisfied by the credential [SWY+].

4.2 EPAL

Enterprise Privacy Authorization Language (EPAL) is an XML-based language proposed by IBM, and was submitted to World Wide Web Consortium (W3C) in 2003 for consideration as a standard [HKPS03], but has not been approved yet. EPAL addresses security issues of enterprise applications in a structured and interoperable manner. It is based on Policy Core Information Model that we discussed in the previous chapter. An EPAL policy expresses applications privacy by defining the following constructs: *data-categories*, *user-categories*, *conditions*, *purposes*, sets of (privacy) *actions*, and *obligations*.

Data-categories define the data to be protected, for example, medical record, chemical formula, etc. User-categories (users/ groups) are the entities which collect the data, for example, health researcher, chemical composition analyst, etc. Conditions consist of boolean expressions to expresses constraints on the Data-categories. The purposes define the intended use of the data, for instance, marketing, auditing, processing, analysis, etc. *Actions* tell how the data is used, for example, disclose, store, read, etc. Obligations model actions that should be taken by EPAL engine when interacting with the data, for instance, delete the data 1 week after using, or ask for consent from the data holder, etc.

The elements defined above are used to formulate *rules* to establish access control. The outcome of policy could be one of *allow*, *deny*, *Not-applicable*. Not-applicable means that the formulated policy does not care if the user-category should be allowed or disallowed to perform the specified action on the data-category for certain purpose while enforcing particular conditions. Figure 4.1 illustrates EPAL policy enforcement model.

EPAL has a strict but very generic XML schema. Hence, EPAL has well-defined semantics. The root element of EPAL is *epal-vocabulary* tag which has *user-category*, *date-category*, *action*, *purpose*, *container*, and *obligation*. It is possible that the *rules* conflict with each other. EPAL vocabulary allows to resolve such conflicts by for example, assigning precedence to a particular rule over others, etc. Table 4.1 shows an example of how a policy can be mapped to the elements of EPAL Rule.

The corrosponding EPAL syntax for EPAL Rule shown in Table 4.1 is presented in Listing 4.3. It must be noted that the individual elements, for example data-category, data-user, etc. need to be defined in epal-vocabulary. Separating rule definition from epal-vocabulary provides modularity. There can be a sequence of one or more `<user-category>`, `<data-category>`, `<purpose>`, and `<action>` elements in `<epal-query>`.

Constructs and vocabulary of EPAL are very abstract. It does not enforce how different elements need to be implemented, for example, how an *action* is triggered when the policy rule is evaluated, or how to call an application which is implementing a

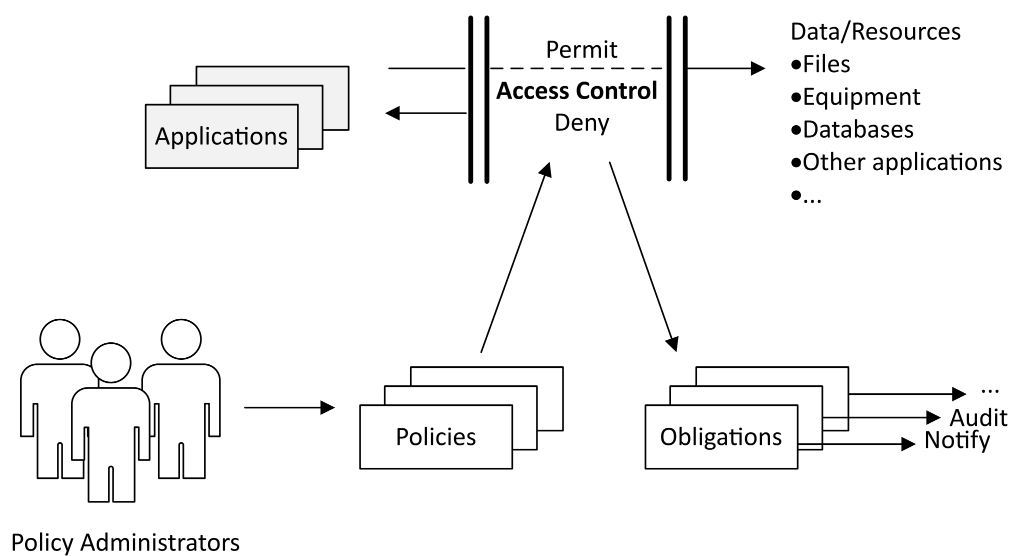

Figure 4.1: EPAL Policy Enforcement Model [And04]

Policy Rule	Allow a chemical researcher or chemical engineer to collect a product formula's for analysis if the formula inventor has not patented the formula, and the formula inventor has been notified of the privacy policy. Delete the data 5 years from now.
ruling	allow
user category	chemical department
action	store
data category	product-formula
purpose	analysis
condition	the product formula is not patented
obligation	delete the data 5 years from now

Table 4.1: EPAL Rule Example - Protecting a chemical formula

Listing 4.3 EPAL Syntax Example - Protecting a chemical formula

```
1  <epal-query>
2        <rule id="rule1" ruling="allow"/>
3        <data-user id="chemical department"/>
4        <data-category id="product-formula"/>
5        <purpose id="analysis"/>
6        <action id="store"/>
7        <condition id="not-patented"/>
8        <obligation id="retention">
9              <parameter id="years">5</parameter>
10       </obligation>
11 </epal-query>
```

particular *obligation* logic. Neither it has any concrete scheme to exchange key-pairs for authorization and security. Furthermore, it does not enforce how the data which needs to be protected is stored, and mapped to EPAL *data-category* vocabulary. It is the responsibility of the policy engine to enforce what has been defined in policies.

4.3 XACML

The eXtensible Access Control Markup Language (XACML) is an XML based policy language defined in 2001, the third version of it was approved as an OASIS standard in 2013 [Ris13]. XACML can be broken down into three parts: policy language, request/ response scheme, and underlying model. The policy language defines how to express access control constraints. The request/ response scheme defines the structure of request and response in XACML context. The request enables to form the query that whether an access to a resource is permitted or not. Similarly, the response enables to interpret a policy evaluation result. The policy engine must convert application context attributes, for example, JAVA, LDAP, etc. to create XACML context request, and convert XACML policy decision to an application context response. How to perform this conversion, does not come under the scope of XACML. The underlying model is PCIM (Figure 3.2) that we discussed in the previous chapter. Figure 4.2 illustrates how XACML fit in an application context. [OAS13a; Ris13]

When someone wants to access a resource (database, web service, filesystem, etc.), the request first goes to PEP. PEP creates XACML context request based on the requester's attribute, action, the resource to access, and any relevant information for access control. The request is forwarded to PDP. PDP fetches request's other details it may require from PIP, and looks into PR for all applicable policies. The PDP then evaluates the policies and takes a decision. The decision could be one of the following: Permit, Deny, Indeterminate (a decision cannot be made because some error occurred while evaluating policies, or

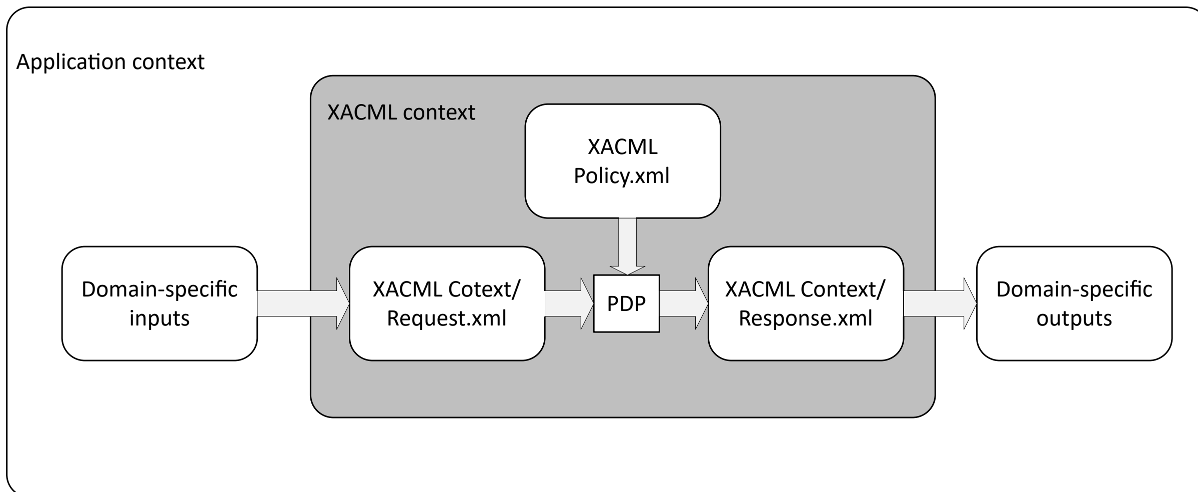

Figure 4.2: XACML Context [Ris13]

some required value was missing) or Not Applicable (the request is not applicable to any policies in PR). The decision is then returned to the PEP, based on which the requester is either allowed or denied the access to the resource.

The root of XACML policy document is either *Policy* element or *PolicySet* element. *PolicySet* can contain *Policy* elements, references to policies on remote locations, or further *PolicySet* elements. The *Policy* element defines the rule for access control. It is consisted of one *Target* element and multiple *Rule* elements. The *Target* specifies for which *Subject*, *Resources*, and *Actions* the policy holds. Beside enabling XACML to find applicability of a policy for a request, the *Target* also help XACML to create indexes on the policies to quickly shift from one policy to another. For instance, if a policy applies to resource country name, and thus *Target* must apply constraint on *Resource*, in this case an index can be made on Resource country name; when request arrives, PDP will know where to look for the policies.

Once the applicable policies have been found, they are evaluated using *Rule* elements. *Rule* is applicable to a certain *Target* and contains policy evaluation logic in *Condition* element. The *Condition* can calculate complex logics using XACML built-in functions or custom-made functions. If the condition results in true, then the effect of the Rule (Permit or Deny) is applied. The condition can also result in Intermediate (some error occurred) or NotApplicable (rule does not apply). As one *Policy* element can have many *Rule* elements, the result of rules need to be combined. XACML achieves that by providing *Combining Algorithms*. The XACML standard contains thirteen built-in Combining Algorithms but custom Combining Algorithms can also be defined.

XACML entire logic runs on XACML context attributes. So when PEP receives an event, PEP maps application context attributes to XACML context request attributes

Listing 4.4 XACML example - Login to servers in Germany is only allowed between 9AM and 5PM (adopted from [Ris13])

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Policy PolicyId="GermanyServer"
       RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
3  <Target>
4  <Subjects>
5  <AnySubject />
6  </Subjects>
7  <Resources>
8  <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
9  <AttributeValue
       DataType="http://www.w3.org/2001/XMLSchema#string">Germany</AttributeValue>
10 <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
       AttributeId="urn:oasis:names:tc:xacml:1.0:resource:country-name" />
11 </ResourceMatch>
12 </Resources>
13 <Actions>
14 <AnyAction />
15 </Actions>
16 </Target>
17 <Rule RuleId="LoginRule" Effect="Permit">
18 <!-- Rule Target specifying Action attibute "ServerAction" equal to "login"-->
19 ...
20 <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
21 <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-greater-than-or-equal">
22 <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-one-and-only">
23 <EnvironmentAttributeSelector DataType="http://www.w3.org/2001/XMLSchema#time"
       AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time" />
24 </Apply>
25 <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#time">09:00:00</AttributeValue>
26 </Apply>
27 <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-less-than-or-equal">
28 <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-one-and-only">
29 <EnvironmentAttributeSelector DataType="http://www.w3.org/2001/XMLSchema#time"
       AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time" />
30 </Apply>
31 <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#time">17:00:00</AttributeValue>
32 </Apply>
33 </Condition>
34 </Rule>
35 <Rule RuleId="FinalRule" Effect="Deny" />
36 </Policy>
```

which are characteristics of *Subject*, *Resource*, *Action*, and *Environment*. The request attributes can be referred and compared with policy attributes using two mechanisms: *AttributeDesignator* and *AttributeSelector*. The AttributeDesignator lets referring to attribute values using name and type. The AttributeSelector enables looking up attribute values by specifying XPath query.

Listing 4.4 is an example policy in XACML using the elements discussed above. Its Target says that the Policy is only applicable on servers in Germany. The Policy has a Rule (with RuleId "LoginRule") which has a Target, specifying the action to be "login" and a Condition that applies only if the Subject tries to log in to the server between 9 AM and 5 PM.

The *CombineAlgorithm* used in the Listing 4.4 is *permit-overrides* which means that *Permit* decision have priority over *Deny* decision. So if any Rule causes the Policy outcome to be *Permit*, other Rules' outcomes (Deny, NotApplicable, Indeterminate) are overridden. In the listing, the second Rule (with RuleId "FinalRule") is being used as the default rule, which always returns Deny; so if the first rule (LoginRule) does not apply, the default rule is considered. Rules are always evaluated in sequence. Other rules can be added in this listing to specify different actions and constraints.

Only basic elements of XACML were discussed in this section, to understand control flow of XACML. Other than that, XACML has a rich vocabulary, providing elements like StatusCodes, MissingAttributeDetails, ObligationExpressions, AdviceExpressions, etc. It provides a standard extension mechanism by defining XACML Attribute definitions for Subject, Resource, Action, and Environment. Similarly, custom CombiningAlgorithms, DataTypes, and Functions can be defined. It does not provide any mechanism for key-pairs exchange; but being based on PCIM, any request information can be defined and extracted from the PIP.

4.4 Ponder

Ponder is a declarative, object oriented, and role-based access control policy language created by Imperial College in 2001. The language consists of the following constructs: *subjects* (actors which interact with the resources to protect), *targets* (resources to protect), *actions* (what to do when a policy is satisfied), and *constraints* (limits the applicability of a policy). It introduces the concept of *Domains*, which means the grouping of subjects and targets. References to the domains are maintained by a domain service which could be implemented by protocols like Lightweight Directory Access Protocol (LDAP). This enables Ponder to assign policies to a group of objects, hence the policy changes go with the domain membership. Ponder has five types of policies for

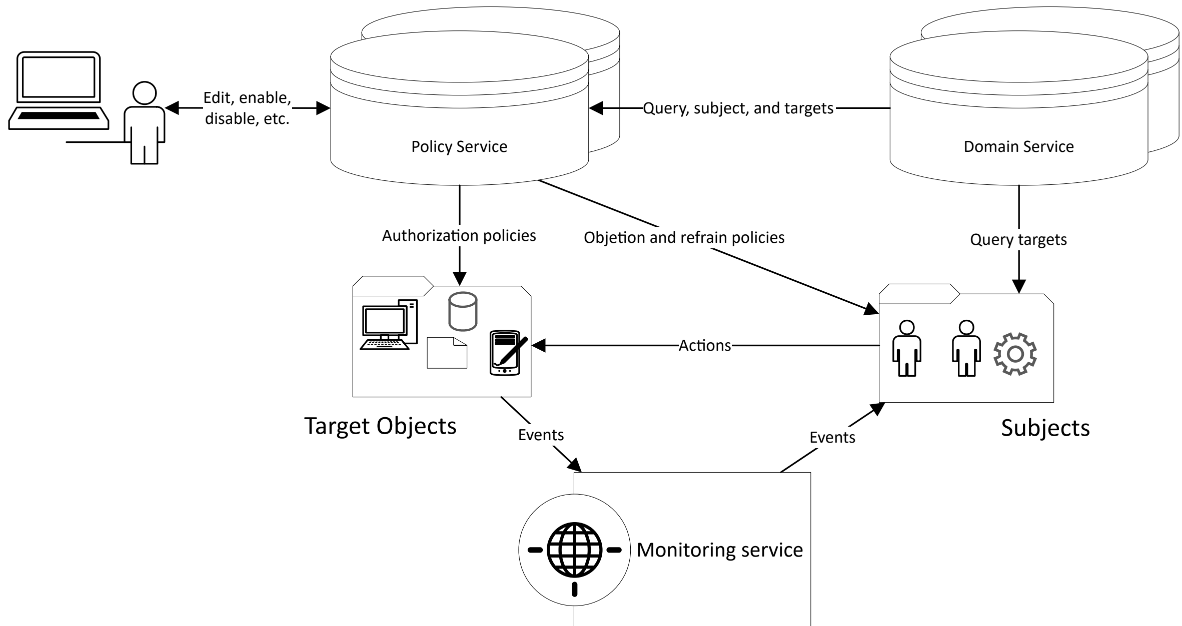

Figure 4.3: Policy Implementation in Ponder [Slo]

Listing 4.5 Ponder example - authorization policy definition [DDLS01]

```
1  inst auth+ switchPolicyOps {
2        subject /NetworkAdmin;
3        target <PolicyT> /Nregion/switches;
4        action load(), remove(), enable(), disable() ;
5  }
```

access control: Authorization policies, Refrain policies, Filter Policies, Obligation policies, and Delegation policies. Ponder is not based on any standard model, for example, PCIM or Trust Negotiation Model, etc. but it has its own model to enforce policies; Figure 4.3 illustrates how policies are implemented in Ponder. [DDLS01; Slo]

Authorization policies are of two types: Positive authorization policies which define what a member of the subject domain is permitted to do to a target, and Negative Authorization policies are used for revocation of access rights. Listing 4.5 is an example of how a positive authorization policy can be defined in ponder.

The example in the listing defines the policy for network switches in Nregion that a network administrator is authorized to load, remove, enable, and disable them. In the listing "+" sign means the definition of positive policy authorization. The same way, negative authorization can be defined using "-" sign. The keyword *inst* is used for direct declaration and instantiation. For the purpose of reusability, Ponder also provides a mechanism for defining the *Policy Type* with which instances of a policy could be created;

Listing 4.6 Ponder example - policy type and instantiation [Slo]

```
1 type auth+ doorcontrol (subject s, target t, string start, string end)
2 {
3       action enter ();
4       when time.between (start, end);
5 }
6 inst ChildSwimmer = doorControl (/hotelGuest/child, /leisure/pool, "1000","1700");
7 inst AdultSwimmer = doorControl (/hotelGuest/adult, /leisure/pool "1000", "2000");
```

Listing 4.7 Ponder example - obligation policy definition [Slo]

```
1 Inst oblig {
2       on          workexit(userid)
3       subject     securityAgent(userid)
4       do          enable.pda(userid, highSecurity)
5 }
```

this is shown in Listing 4.6. This listing is an example of defining access control on the doors of a swimming pool.

The Obligation policies define the actions that must be performed by the subjects (human or automated manager components) on objects in the target domain when a certain event occurs. Examples of an event could be temperature exceeding, component failing, disk space close to limit, web service call, etc. Actions could be, for example, auditing, email, shutting down a component, etc. A chain of actions could also be specified. Listing 4.7 is an example of this specification. In the listing, *workexit* is name of the event, *securityAgent* the subject, and *enable.pda* is name of the action.

Refrain policies specify the actions the subject should refrain to do. The difference between Negative Authorization Policies and Refrain Policies is that the Refrain Policies are enforced by Subjects rather than Target access controllers. They are used when it is not possible to define policies on target objects; for example, there may be the case when a target object is not exposing any interface for policy definition. This mechanism is a clear violation of Monotonicity that we discussed in the previous chapter. The Non-monotonic policy languages introduce complexities when it comes to resolving policy conflicts.

Filter policies are defined on top of Positive Authorization Policies to transform input or output parameters associated with their actions, based on attributes of subject or target. They are used when it is not possible for external authorization agent to provide different operations to reflect permitted parameters. The same way output parameter transformation is done to reflect the result in the format expected by an enforcement agent. Delegation Policies enables subjects to perform an action on behalf of some other subject. This delegation is temporary, and holds only till the expression in *valid* element

of Ponder is applicable. Roles group policies for subjects having same duties and rights. This enables Ponder to specify policies in terms of Roles rather than persons or any other individual system component.

A set of tools and services have been developed for the enforcement of policies defined in Ponder. These include implementations for the domain service, event enforcement agent, policy compiler, application communication agent, etc. Ponder2 is another language created in 2007, inspired by features and semantics of the Ponder. Ponder2 comes with a runtime that enables to interact with other software and hardware components, for example, tools to capture events from hardware, a JAVA based application to create Ponder objects (subjects, targets), etc [Pon13].

Ponder does not have well-defined semantics, this is because they are not based on Logic programming or Description logics [CO08]. Although, Ponder is not an XML based language, there are tools available to translate Ponder definitions into XML representations, for the exchange of policies across different domains. But it is hard to extend Ponder language. This is the reason Ponder2 is a redesign and reimplementation of the Ponder [Pon13]. It has no mechanism to protect sensitive policies and does not provide a concrete mechanism for trust building based on certificates.

4.5 WS-Policy

WS-Policy is an XML based framework for defining policies in web service based systems. It is a W3C recommendation as of September 2007 [BBC06]. WS-policy defines policies using a collection of policy alternatives, where each alternative consists of multiple policy assertions. Each policy assertion defines a policy requirement or capability. The requirements and capabilities defined by policy assertions could either be those which are directly manifested on the wire (for example, transport protocol or authentication scheme), or those which define selection criteria for web services (for example QoS characteristics or privacy mechanisms) [BBC06].

WS-Policy defines policies by using the combinations of two operators (XML tags): *wsp:ExactlyOne* and *wsp:All*. WS-Policy provides two types of expressions for defining policies: normal form and compact form. Listing 4.8 is a simple example of WS-Policy definition using the compact form. Line 1 in the listing is the policy definition. Line 2 to line 5 is the policy declaration, in which line 3 and line 4 are the assertions for the declarations. The compact form is more human readable and concise, and the normal form expression is verbose because it enforces a strict order to define policies. The normal form version of Listing 4.8 is shown in Listing 4.9. The normal form expression

Listing 4.8 WS-Policy example - Using compact form expression [Ley]

```
1 <wsp:Policy xmlns:fl="..." xmlns:wsp="...">
2       <wsp:ExactlyOne>
3             <fl:PaymentMethod Period="monthly"/>
4             <fl:PaymentMethod Period="perClick"/>
5       </wsp:ExactlyOne>
6 </wsp:Policy>
```

Listing 4.9 WS-Policy example - Using normal form expression [Ley]

```
1  <wsp:Policy xmlns:fl="..." xmlns:wsp="...">
2        <wsp:ExactlyOne>
3              <wsp:All>
4                    <fl:PaymentMethod Period="monthly"/>
5              </wsp:All>
6              <wsp:All>
7                    <fl:PaymentMethod Period="perClick"/>
8              </wsp:All>
9        </wsp:ExactlyOne>
10 </wsp:Policy>
```

enables interoperability by performing intersection on policies from a requester and a service provider, resulting in an effective policy [BBC06; Ley].

The policies could be referenced from the same document, a different document placed locally, or a document at a remote location. Listing 4.10 is an example of how this can be done using WS-Policy semantics.

WS-Policy framework allows referring to multiple policies using Policy Inclusion as shown in Listing 4.11. The policy is reffered using URI attribute, and the optional attribute Digest enables to check the integrity of a referred policy by, for example, calculating and comparing hashes. Policy Inclusion is specially useful in order to import a common policy in multiple policies, hence supporting reusability.

When resolving policies, assertions are intersected based on assertion type and delegate parameters. For example in Figure 4.4, Policy 1 have two alternatives, A1 and A2. Policy 2 also have two alternatives, A3 and A4. Alternatives A2 and A3 are compatible with each other because each assertion type in A2 (sp:SignedParts and sp:EncryptedParts) is compatible with that of assertions in A3. Parameters of the assertion types may be different when checking for the compatibility of alternatives, as in this case. The

Listing 4.10 WS-Policy example - Reffering policies using WS-Policy Attachment [Ley]

```
1 <wsp:Policy xml:Name="http://fabrikam123.com/policies" wsu:Id="audit" xmlns:my="...">
2       <my:Audit/>
3 </wsp:Policy>
```

Listing 4.11 WS-Policy schema - Including one policy in another [BBC06]

```
<wsp:Policy>
    ...
    <wsp:PolicyReference URI="xs:anyURI" ( Digest="xs:base64Binary" (
        DigestAlgorithm="xs:anyURI" )? )? .../>
    ...
</wsp:Policy>
```

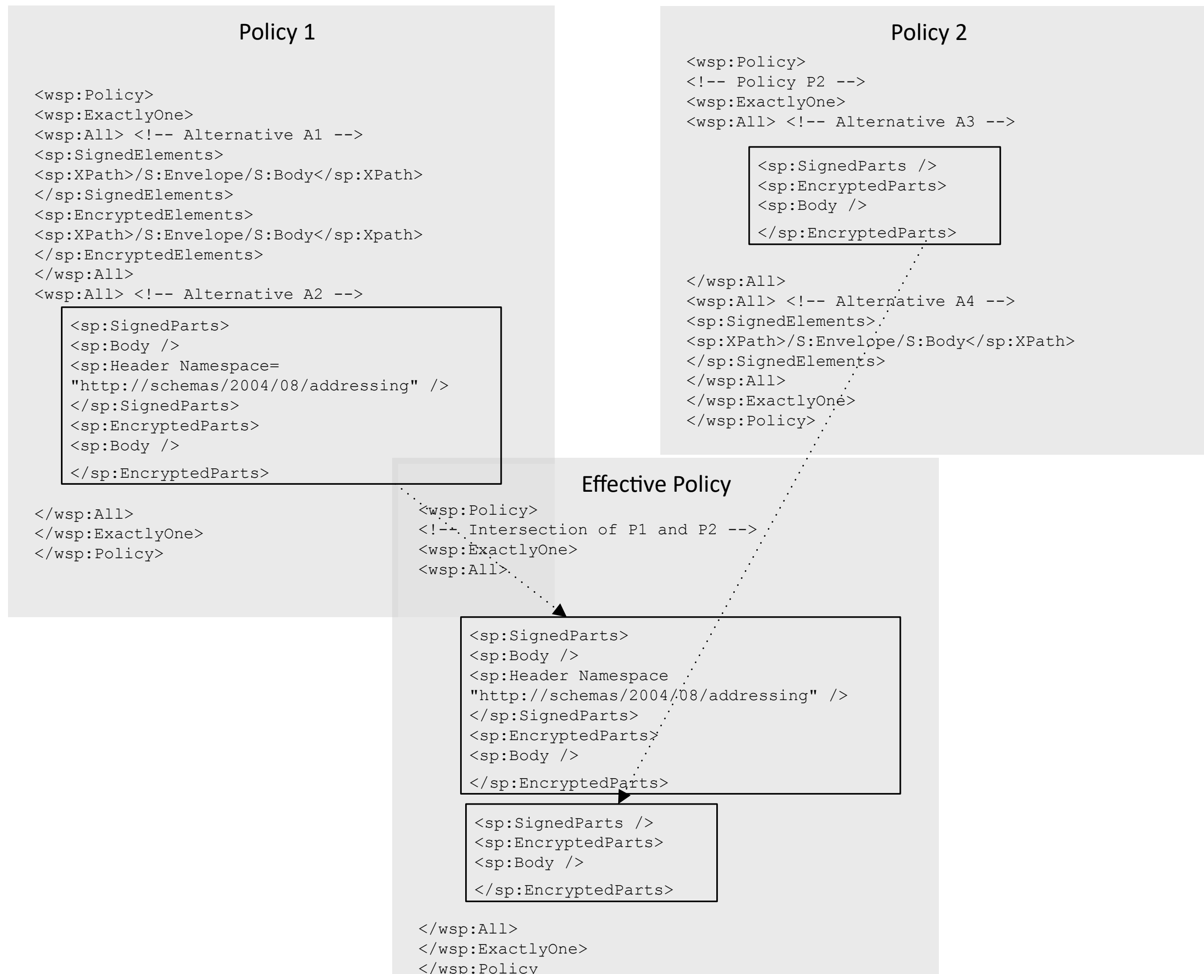

Figure 4.4: WS-Policy assertions intersection example [BBC06]

effective policy constitutes of an alternative having assertions in both of the policies A2 and A3. It must be noted here, that the effective policy now have two assertions of type sp:SignedParts. In such cases, both of the assertions must be calculated for correct behaviour [BBC06].

Domain specific extensions are available that make up the actual assertion. WS-Security (an OASIS standard) [OAS06] is one such specification to secure Simple Object Access

Listing 4.12 WS-Policy example - Attaching a policy to a webservice using WS-Policy Attachment [BBC06]

```
1 <businessService>
2       <name>...</name>
3       <description>...</description>
4       <bindingTemplates>...</bindingTemplates>
5       <categoryBag>
6             <keyedReference
                  tModelKey="uddi:schemas.xmlsoap.org:remotepolicyreference:2003_03"
                  keyName="Policy for my registered Web service"
                  keyValue="http://www.fl.com/myservice/policy"/>
7       </categoryBag>
8 </businessService>
```

Protocol (SOAP) messages in SOAP based web services, by providing a framework for confidentiality, authentication, authorization, and integrity. Discussion of SOAP messages is not under the scope of this thesis [Sos10]. WS-Security provides five types of token profiles to incorporate security in SOAP messages: Username, x509 (use of x509 certificates to secure the SOAP message), Security Assertion Markup Language (SAML), Kerberos, REL. The two token profiles are worth mentioning here are username token profile and x509 token profile. The username token profile uses username and password to authenticate the SOAP messages. The x509 token profile uses x509 certificates to encrypt, sign, verify, and decrypt the SOAP messages [OAS06]. WS-Trust [NGG+] is an extension of WS-Security which provides a framework for issuing and requesting security token for building trust relationship between involved parties [NGG+].

WS-Policy does not enforce how the policies need to be attached to web services endpoints, resources, messages, and entities. Other technologies specific specifications could be defined for this. WS-Policy Attachment [VOH07] is one such specification (a W3C recommendation) for attaching WS-Policies with Web Services Description Language (WSDL) artifacts and UDDI elements. Just to give a flavor of how the WS-Policy Attachment specification hooks policies to subjects, Listing 4.12 is an example for that [BBC06]. In this listing, `keyValue` is the URI where the policy is located.

5 Policy Based Approach to Secure TOSCA-based Cloud Services

This chapter starts off with a comparison of the shortlisted policy languages that were discussed in the previous chapter. One policy language is selected based on contextual requirements and the characteristics of policy languages that were discussed in Chapter 3. After that, an approach is formulated on how to define policies for cloud services running in TOSCA environment.

5.1 Evaluation of Discussed Policy Languages

This section determines the positive and weak points of the shortlisted policy languages based on the characteristics discussed in Chapter 3 and common scenarios in context of the distributed systems.

5.1.1 Characteristics Comparison

Table 5.1 summarizes the comparison for the full list of characteristics. All the languages except Ponder have *well-defined semantics*, and this is because Ponder is not XML based. In Chater 3, the assumption was made that only the languages that are based on logic programming or description logics have well-defined semantics [CO08]. *Monotonicity* means that fulfillment of more credentials only gives more privileges. So this characteristic is only for the languages which support a concrete credential or evidence handling mechanism. In this sense, DTPL is the only language which is monotonic. All the languages support *condition expressiveness* by means of functions and operators.

In *Type of evaluation*, WS-Policy is the only language which supports both distributed evaluation and distributed policies. This is because WS-Policy has *Negotiations* characteristics. TPL also supports *Negotiation* but does not support distributed evaluation because their semantics does not support breaking down the policies in chunks (distributed policies). Ponder is the only policy language which supports *ECA* paradigm because of

Language / Characteristic	TPL/DTPL	Ponder	EPAL	XACML	WS-Policy
Well-defined semantics	Yes	No	Yes	Yes	Yes
Monotonicity	TPL: Yes DTPL: No	Not applied	Not applied	Not applied	Not applied
Condition expressiveness	Yes	Yes	Yes	Yes	Yes
Action execution	No	Yes	Yes	Yes	Yes
External functions	Yes	Yes	Yes	Yes	Yes
Negotiation support	Yes	No	No	No	Yes
Type of evaluation	Local	Local	Local	Distributed polices local evaluation	Distributed policies distributed evaluation
Policy engine decision	Yes	Yes	Yes	Yes	Yes
ECA	No	Yes	No	No	No
Indexing	No	No	Yes	Yes	No
RBAC	Yes	Yes	Yes	No	No
Obligation	No	Yes	Yes	Yes	No
Extensibility	Yes	Yes	Yes	Yes	Yes
Conflict Resolution	Not applicable	Yes	Yes	Yes	Yes
Protecting sensitive policies	By default: No Extension: Yes	No	No	No	Yes
Evidences	Yes	Not applied	No	No	Yes
Credential chains	Yes	Not applied	Not applied	Not applied	Yes
Credential combinations	Yes	Not applied	Not applied	Not applied	Yes
Inter-credential constraints	Yes	Not applied	Not applied	Not applied	No

Table 5.1: Comparison of Policy Languages with respect to the Defined Characteristics

`"on"` element in their Obligation policies. XACML and EPAL do not support *ECA* because in PCIM, an event received by PEP and sent to PDP is only an access request, it does not include environment attribute changes [CO08].

WS-Policy is the only policy language which supports *sensitive policies* by default, the reason being they support distributed evaluation. This means WS-Policy can keep a policy confidential, and only disclose this policy if the previous negotiations have succeeded. TPL can support *sensitive policies* with an extension approach discussed in Chapter 4. EPAL, XACML, and Ponder do not support *sensitive policies* because they require the (distributed or local) policies to gather at one place before starting the evaluation.

Evidences are required if there is a need to prove identity in the requirements of policy. TPL supports shreds of evidence based on credentials, by exchanging digital certificates. *Evidences* are not applicable to Ponder because Ponder assumes that the subject (for example, a user, or an application) has already been authenticated, and its policies are mainly in regard to limiting the access of the subject [CO08]. WS-Policy also supports Pieces of evidence with its extension WS-Security. *Credential chains* and *Credential combinations* are supported by both TPL/ DTPL and WS-Policy. The last characteristic *Inter-credential constraints* is only supported by TPL/ DTPL, for this reason TPL/ DTPL is a good trust negotiation policy language.

The intent of this comparison is not to select the best policy language having the most features. Not a single language exists which is able to fulfill the requirements of all the environments and the best fit for every scenario. Having said that, it is also not the case that this comparison is meaningless. The comparison based on the characteristics can help to identify the weak and strong points of a policy language. One policy language can be selected based on the context where it is going to be used. In order to select such language, first it is required to identify the context of use, then the characteristics can be filtered which are applicable in the required context. Finally, the language which fulfills most of the characteristics can be selected.

5.1.2 Contextual Comparison

If there is a need to setup an access control system for devices and users in a network, with the assumption that the devices or users have already been authenticated by some other system, for example, username and password authentication. And if, there is a need to perform the audit of any access request made to the resources of the network. Further, the requirements also contain the need for a ready-made implementation for the policy engine supporting the policy language. In such a scenario, looking at Table 5.1, this can be deduced that Ponder is the best option. The reason that Ponder offers a ready-made policy engine [Pon13] and a set of tools, also vote for Ponder in this scenario.

EPAL, XACML, and WS-Policy are platform independent languages, to describe policies for exchanging information in a distributed environment. Though the purpose of all the three languages is to establish the exchange policies between applications or enterprises, the structure of the languages is different which considerably affect their usability and capabilities. It is practically feasible to identify which of the three policy languages should be used in which types of distributed systems.

XACML 2.0 is an OASIS standard, and EPAL 1.2 was submitted to W3C to be recognized as a standard in 2003, but no decision has yet been taken on that. Both XACML and EPAL are based on PCIM (Figure 3.2). Both are XML based and hence have well-defined semantics, but both have different language structure to exchange policies [And05; HKPS03; Ris13]. The correspondence of EPAL request and XACML decision request is shown in Figure 5.1.

Both XACML and EPAL use the concept of *Rules* in their semantics. The policy request query outcome could be one of the following in both of the languages: *allow*, *deny*, *Not-applicable*. EPAL have it indicated in *ruling* attribute of *rule* tag. XACML have it in *Effect* attribute of *Rule* tag. EPAL supports Obligations in Rules, where as XACML define it in Policies [HKPS03; Ris13].

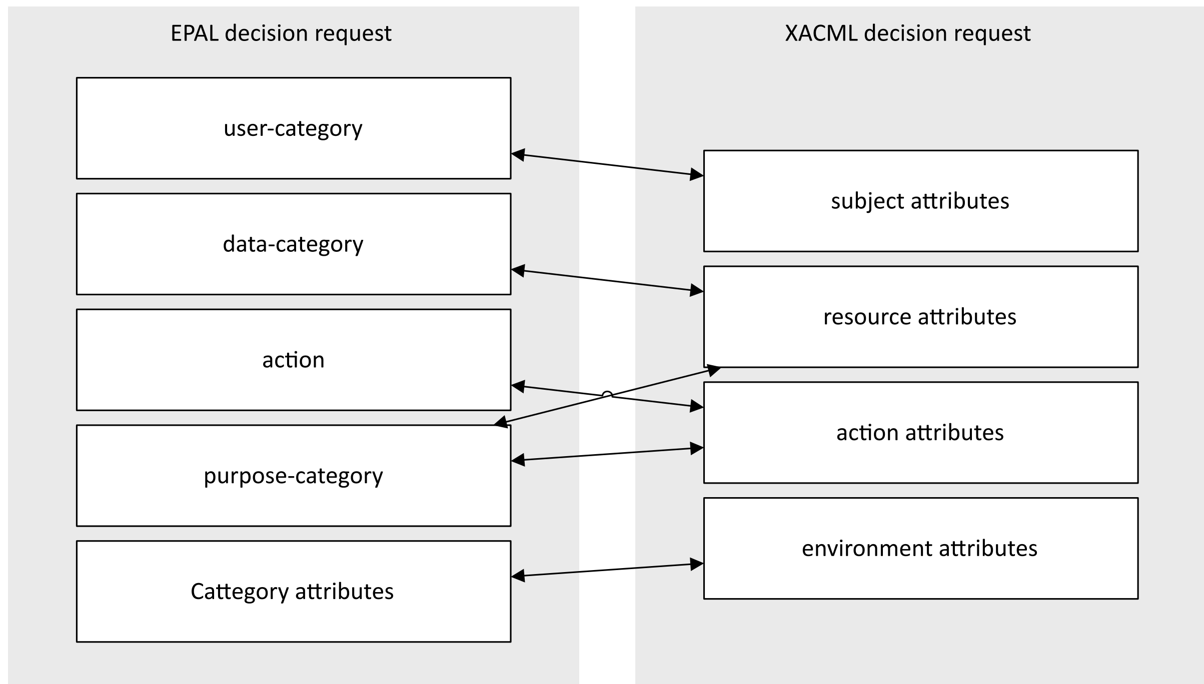

Figure 5.1: correspondence between the decision requests of XACML and EPAL [And04]

EPAL is not recommended for enterprise level security because EPAL authorization query consists of *purpose*. This is a mandatory part of the query, hence without knowing it, a policy decision cannot be calculated. XACML also have two purpose elements, one for data collected, and one for data accessed. But these elements are optional. The policy calculation does not depend on *purpose*. EPAL does not support nesting of the policies as XACML enables with the use of *PolicySet*, within *PolicySet*, within *Policy*, and so on. Further, XACML provides policy combining algorithms, for example, permit-overrides, first applicable, etc. to resolve conflicting policies. Own custom algorithms can also be defined. On other hands, EPAL resolves conflicting policies with precedence. It is the responsibility of policy writer to order set of rules in EPAL query in such a way that whichever rule is satisfied first, gets applicable. Further, EPAL does not provide distributed policies, XACML achieves this by using references to remote policies in PolicySet [And04; HKPS03; Ris13].

As XACML is an OASIS standard, it is already being used by many enterprises. XACML also have supporting frameworks, for example, ALFA [AXI12] is an eclipse plugin to author XACML 3.0 policies using a programming language. The ALFA program generates XACML 3.0 policies. Such plugins can simplify writing and managing policies. Considering the upsides of XACML over EPAL, XACML leads EPAL.

XACML query evaluation takes place in PDP. Combining algorithms are used while resolving policies, which returns boolean results. This can be used in web service calls, for example, if a (web) service provider has a set of policies, and wants to verify if a client request fulfills those policies before processing the request. Similarly, a client can verify that if the response from the service provider complies with its defined policies in order to process the response. The issue is that XACML is unable to deal with the scenarios where both a client and a service provider have policies. This is the Negotiation characteristic that we discussed in Chapter 3. Such scenarios are applicable when a client at the time of sending a request to a service provider, considers both its own policies and the policies of the service provider. Similarly, the service provider should send the response to the client with considering both its own policies and the client's policies. XACML only considers its own policies when a request is received. However, WS-Policy framework does support this characteristic. Negotiation is an important characteristic that we need to have when defining policies for a distributed system. Apart from WS-Policy, TPL/ DTPL also supports Negotiation, but TPL/ DTPL have several other features missing which exist in XACML and WS-Policy (as can be seen in Table 5.1).

WS-Policy framework also allows securing sensitive policies because it has the ability to perform distributed evaluation of policies. The negotiation can proceed in the WS-Policy framework by evaluating a subset of policy at each intermediary. If the policy evaluation is successful, then the negotiation could be carry on, otherwise terminated. This way, a confidential policy could be kept secret at a very last intermediary and can only be exposed if all previous negotiations get successful.

To verify, if WS-Policy has all other required features that are in XACML, Figure 5.2 shows the correspondence between elements of XACML and WS-Policy.

XACML Target specifies the resource to be protected. In the case of WS-Policy, there is no explicit need of a Target. This is because the target is indicated by the location where a WS-Policy is being attached. For example an operation or endpoint in WSDL (Listing 4.10), or the SOAP message with which a policy is being attached. XACML uses Target for indexing which is useful when a relevant policy needs to be referred from an external system, for instance, ODBC, LDAP, etc.

XACML enables attaching multiple policies to a single target. WS-Policy framework allows this by referring multiple policies in one policy through the usage of Policy Inclusion (Listing 4.12). XACML provides Obligation related policies, for example, auditing requests, deletion of a record after some time, etc. WS-Policy does not have any such built-in semantics for doing that. But this can easily be created as a domain specific assertion, the same way, WS-Security framework provides security assertions for WS-Policy.

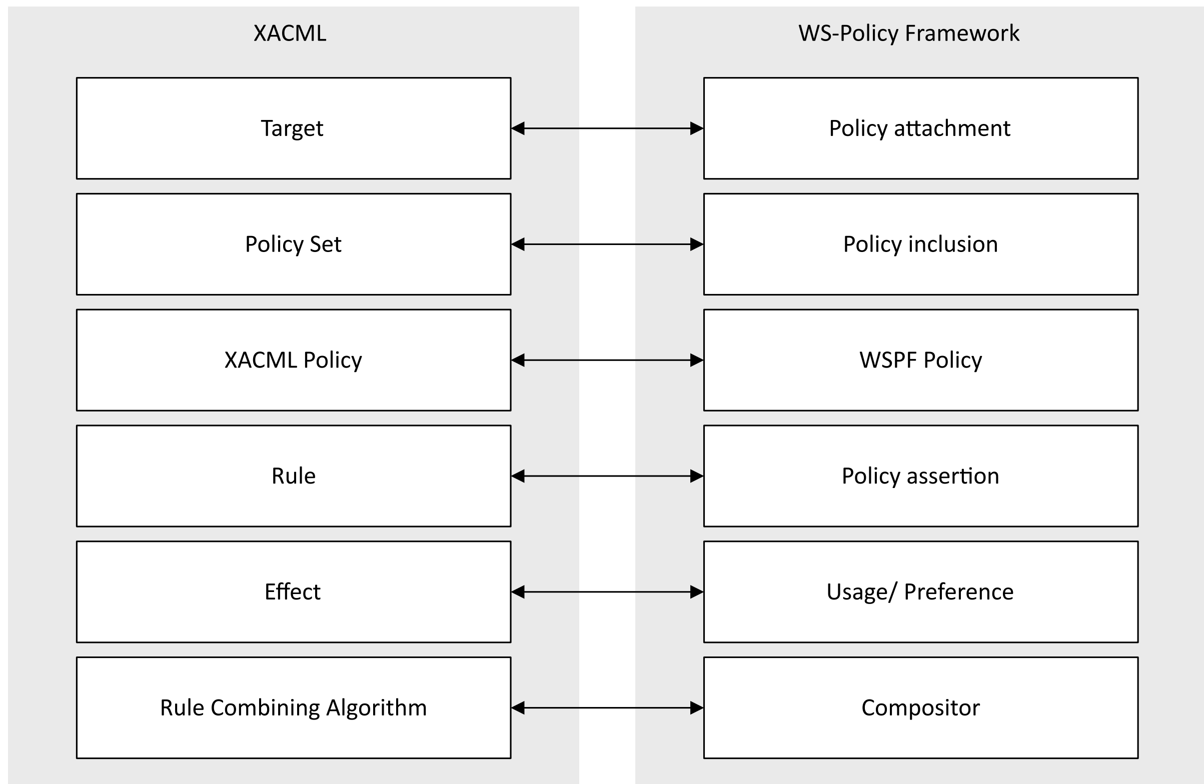

Figure 5.2: Correspondence between the elements of XACML and WS-Policy framework [Mos03]

XACML Policy consists of a set of rules and Rule Combining Algorithms are used to combine the rules. On other hands, WS-Policy contains a nesting of assertions, which are resolved by forming a hierarchy of "ExactlyOne", "All", or "OneOrMore" tags. The concept of Combining Algorithms is more formal and extensible and WS-Policy assertion nesting is complex, informal, and inextensible. A rule in XACML forms a conditional expression using custom or built-in functions for example, "and", "or", "time-greater-than-or-equal", etc. The assertions in the WS-Policy framework are declarative. For example, "I will only respond if you talk to me in German or English" is a conditional expression and, "I can communicate in German and English" is a declarative expression. However, there is no such feature difference here, because this is how both of the policy languages work. XACML use predicates because it was designed with the focus on access control. WS-Policy framework on another hand, forms an agreement based on requirements and capabilities.

Concluding from the discussion above, XACML should be used when the requirement really is to define access control in a distributed system, and WS-Policy framework should be used when there is a need to establish peer-to-peer communication between parties in a distributed system based on requirements and capabilities of the parties. This

is because XACML provides features including indexing support for policies, obligation policies, and extensible rules combining algorithms, which are useful for defining access control policies but are missing in the WS-Policy framework. On the other hand, WS-Policy framework supports Negotiation characteristic and protection of sensitive policies which are important for peer-to-peer communication, and are missing in XACML. Another common requirement for peer-to-peer communication is trust building between parties through the issuance, verification, and exchange of credentials. WS-Security has extensions including WS-Security and WS-Trust which fulfill these requirements.

5.2 Approach to Address Security Requirements of Smart Services

The end-to-end business process and security requirements of smart services are briefly recalled before defining an approach to fulfill the security requirements of smart services for function shipping and data shipping. After which, policy language characteristics and features are selected which are mandatory to address the security requirements in the context of smart services. Finally, an approach is defined to refine and enhance the TOSCA standard to the field of smart services in production environments through the usage of policies.

There should be *Restricted Access to Smart Services* as the smart services may contain critical intellectual property of a company, for example, proprietary analytics algorithms, or confidential data about manufacturing steps of a product, etc. *Data Security* both in terms of transport and persistency layer needs to be enforced especially in the case of data shipping approach. As we discussed in Chapter 2, smart service is just a smart (web) service with provided awareness of context, hence it can be deployed anywhere. It can be deployed at data source environment or at any distant environment. *Data Ownership* is a key requirement in this regard, as the data owner should be able to govern that where its data is stored or being processed. The *Algorithms as Intellectual Property* also need to be safeguarded as it is the property of the service developer. The customer who purchases smart services does not necessarily gets the rights to reuse, adapt, or manipulate the algorithm. This is especially critical when a smart service needs to be offered via platforms, for instance, public repositories or marketplaces. Therefore *Smart Service Integrity* need to be preserved [FBC+16].

Table 5.2 suggests possible solutions for each security requirement defined above. In the table, CSAR stands for Cloud Service Archive which contains the implementation of smart service, the structure of which we discussed in Chapter 2. Looking at the security requirements and their possible solutions, it is realized that the security requirement

No.	Security Requirement	Possible Solution
1	Restricted Access to Smart Services	Authentication of the smart service client using X509 certificates
2	Data Security (transport)	Encrypt and sign request/ response messages when sending. Verify and decrypt the messages when receiving.
3	Data Security (storage)	Database encryption to store the metered data
4	Data Ownership	Offer an interface to service consumer where it could be defined that where the data should be stored and processed
5	Algorithms as Intellectual Property	Encrypt, sign, verify, and decrypt CSAR
6	Smart Service Integrity	Sign and verify CSAR

Table 5.2: Possible solutions of smart services security requirements

number 3, 4, 5 and 6 are related to the deployment and management of the smart services and number 1 and 2 are related to dialogue or interaction between a smart service and its client begins. Thereby, the policies are suggested to be divided into two types: *Deployment and Management policies* and *Dialogue policies*. The *Deployment and Management policies* should cater all policy requirements whi, irrespective of whether the requirements are for QoS, access control, security, or obligation. The same way, *Dialogue policies* should deal with all types of policies related to negotiation between the two parties regardless of the type of policy requirements.

5.2.1 Dialogue Policies

For the dialogue requirements, for example, 1 and 2 in Table 5.2, WS-Policy framework should be used. The Systems based on the concept of IoT and Industry 4.0 involve peer-to-peer communication, both between a device to device, and device to a server. And so with the case, when the smart services are used in such systems. Especially in regard to function shipping and data shipping, when a negotiation may require exchanging of policies from both, the party acting as a client and the party acting as a server. So, there is a need for Negotiation support which are only provided by the WS-Policy framework, except TPL/ DTPL. But TPL/ DTPL misses many other features that are required to fulfill other security requirements of smart services. The other reason for choosing WS-Policy framework is that it enables protection of sensitive policies by supporting distributed evaluation of policies. The security requirements of smart services also include a requirement for two-way authentication for verification

of the identity of smart (web) service endpoint and to protect the service endpoint from attacks. This requirement can be catered with exchanging, for example, x509 certificates, for trust building. WS-Policy provides this mechanism with the usage of its extensions, WS-Security and WS-Trust. Having said that, an assumption has been taken that smart services are SOAP based web services. Only then, WS-Policy and accompanying frameworks are applicable. Otherwise, if smart services are, for example, REpresentational State Transfer (REST) based, then WS-Policy framework should not be used because REST web services have whole different concepts. In this case, XACML with some extensions of the language could be one option. Detailed discussion on web service technologies is not part of this thesis, but talking about security, it is recommended to use SOAP based web services for smart services. In the rest of the discussion, smart services are assumed to be SOAP based web services.

5.2.2 Deployment and Management Policies

The deployment related security requirements (3, 4, 5, and 6 in Table 5.2) should be defined using TOSCA Policy Type and Policy Templates. TOSCA is being used for automatic deployment and management of smart services and TOSCA already enables defining policies. These policies could define deployment and management policies, for example, configuring a particular region for cloud provider or data center selection, or setting up high availability of a service to support particular Quality-of-Service (QoS) defined in the SLAs. [WWB+13b], [BBK+13], and [WWB+13a] describe how to realize TOSCA policies in TOSCA runtime, and enable "Policy-Aware Provisioning and Mangement of Cloud Applications" [BBK+13]. All the policy languages shortlisted in this thesis after a consideration of different policy language characteristics and different policy models are meant for catering access control, trust negotiation, and interaction policy requirements, hence should not be used for deployment or management policy requirements.

5.2.3 An Approach to Secure Cloud Service Archives

Figure 5.3 illustrates an approach to secure cloud service archives for function and data shipping in industrial environments. Starting from **step 1** , a service developer implements a smart service, creates WSDL file, defines WS-Policy for the endpoints of the smart service. Listing 5.1 shows a sample WS-Policy used for authentication of the clients using X509v3 certificates. This means, whichever client needs to interact with the web service is required to send security tokens of the type specified in the policy. In this case, it is requiring the client to submit an X509v3 certificate. If the security token

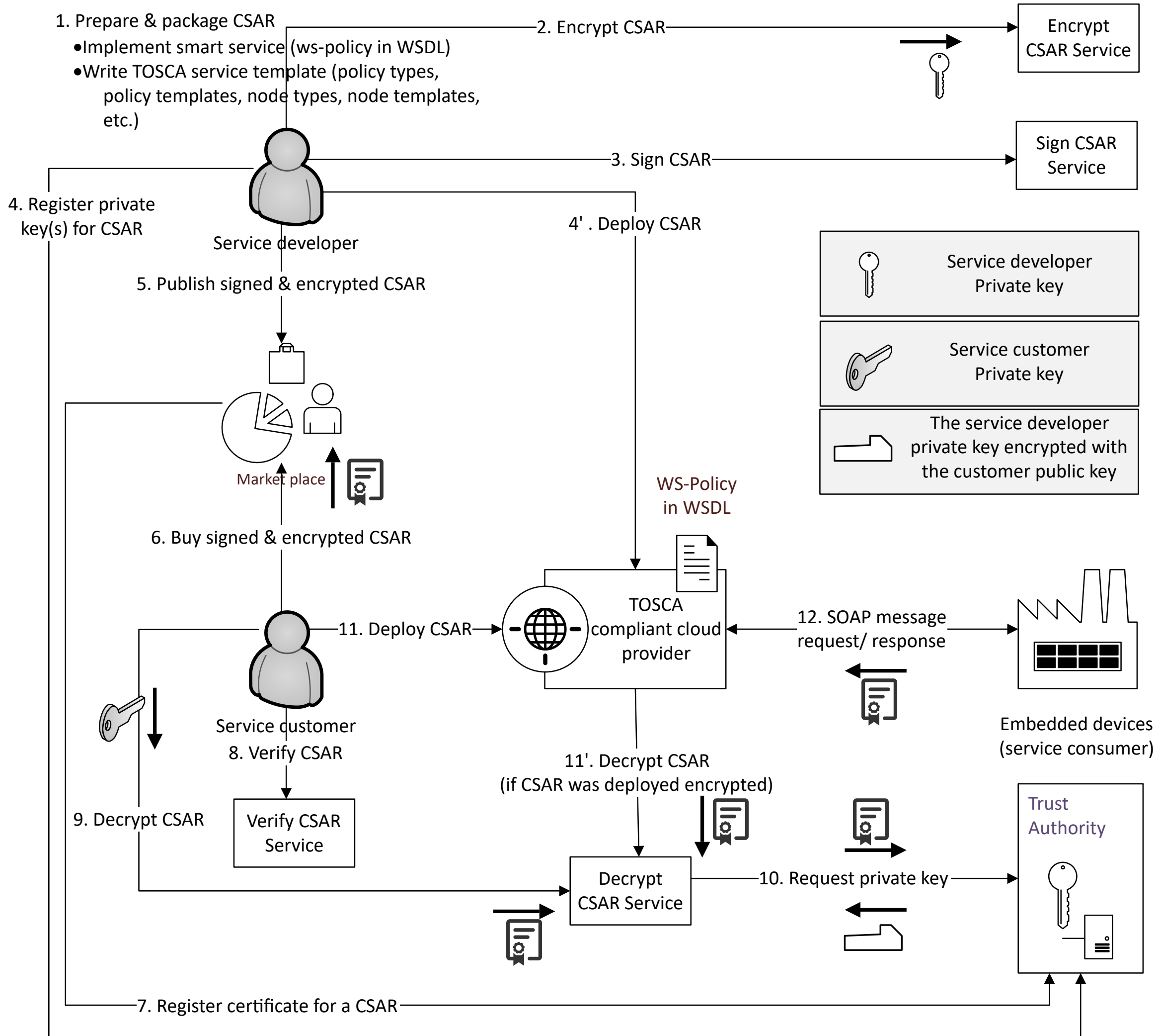

Figure 5.3: An approach to secure cloud service archives

is trusted by the smart service, only then the request should be catered. There are other security mechanisms supported by WS-Policy [OAS06] as well, for example, username/ password authentication, or mutual authentication with signing and encrypting the messages using X509 certificates, etc.

After defining the Dialogue Policies in WSDL using WS-Policy, the service developer writes TOSCA service template for defining the deployment and management lifecycle of the smart service. While doing this, policy types and policy templates are also defined. This could include policies, for example, which regions should be selected for the deployment of a service, or what database encryption should be used for data storage. Listing 5.2 shows a simple policy type, with corresponding policy template shown in Listing 5.3, to setup database encryption on a database. In this example, *DatabaseProperties* defines the

Listing 5.1 Secure smart service example - X509 Authentication using WS-Policy

```
1 <wsp:ExactlyOne>
2    <wsp:All>
3       <sec:SecurityToken>
4          <sec:TokenType>sec:X509v3</sec:TokenType>
5       </sec:SecurityToken>
6    </wsp:All>
7 </wsp:ExactlyOne>
```

Listing 5.2 Secure smart service example - Policy Type definition database encryption for storage (Adopted from [OAS13b])

```
1 <Definitions id="MyPolicyTypes" name="My Policy Types"
      targetNamespace="http://www.example.com/SamplePolicyTypes"
      xmlns:bnt="http://www.example.com/BaseNodeTypes"
      xmlns:dbp="http://www.example.com/DatabaseProperties">
2
3       <Import importType="http://www.w3.org/2001/XMLSchema"
            namespace="http://www.example.com/SamplePolicyProperties"/>
4       <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
            namespace="http://www.example.com/BaseNodeTypes"/>
5
6       <PolicyType name="DatabaseEncryption">
7             <PropertiesDefinition element="dbp:DBEncProperties"/>
8             <AppliesTo>
9                   <NodeTypeReference typeRef="bnt:DBMS"/>
10            </AppliesTo>
11      </PolicyType>
12
13 </Definitions>
```

XML schema which is used by *DatabaseEncryption* policy type to hook with *bnt:DBMS* node type and instantiated by *MyDatabaseEncPolicy* template. After that, CSAR is packaged with all the relevant files discussed above.

In **step 2**, the service developer use *Encrypt CSAR Service* to encrypt the CSAR. In doing so, the service requires from the service developer to provide information including, encryption key, key algorithm, key size, etc. The service should be able to perform the encryption on artifact level. By default, all artifacts can get encrypted. Once the CSAR is encrypted, The service developer signs the CSAR using *Sign CSAR Service*, as shown in **step 3** in Figure 5.3. In this step, the service developer must provide the service with a key pair and certificate information. The other optional information could include, for example, signature algorithm, key size, etc. Once the CSAR is encrypted and signed, the service developer registers the private keys for the CSAR to the TA as illustrated in **step 4**. There could be multiple keys for each CSAR if the service developer uses different

Listing 5.3 Secure smart service example - Policy Template definition database encryption for storage (Adopted from [OAS13b])

```
1  <Definitions id="MyPolices" name="My Policies"
       targetNamespace="http://www.example.com/SamplePolicies"
       xmlns:spt="http://www.example.com/SamplePolicyTypes">
2
3        <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
             namespace="http://www.example.com/SamplePolicyTypes"/>
4
5        <PolicyTemplate id="MyDatabaseEncPolicy" name="My AES Database Encryption Policy"
             type="pt:DatabaseEncryption">
6              <Properties>
7                    <DBEncProperties>
8                          <EncryptionAlgorithm>AES</EncryptionAlgorithm>
9                          <KeyLength> 256 </KeyLength>
10                   </DBEncProperties>
11             </Properties>
12       </PolicyTemplate>
13
14 </Definitions>
```

keys for encrypting different artifacts in the CSAR. In **step 5**, the service developer publishes the CSAR to a marketplace or a public repository.

A smart service customer buys the CSAR from the market place, or download it from a public repository (**step 6**). The customer must provide its certificate to buy the CSAR, which is used to fetch encryption key in the later steps. In **step 7**, the market place registers this certificate and the CSAR information in the TA, confirming that the customer has bought the CSAR. A variation in this step could be that the market place does not send the customer certificate to the TA, instead it notifies the service developer with the customer certificate that a service has been bought. Now it becomes the responsibility of the service developer to register the customer certificate to the TA, approving her for the use of CSAR.

In **step 8** the service customer verifies that the contents of the CSAR has not changed and the CSAR is from the right service developer, by using *Verify CSAR Service*. Once the service confirms the authenticity and integrity of the CSAR, in **step 9**, the service customer use *Decrypt CSAR Service* to decrypt CSAR. The service customer must provide its certificate and private key when using this service. The *Decrypt CSAR Service* requests the TA for the private key of the CSAR by forwarding the certificate provided by the service customer to the TA (**step 10**). The TA sees if the certificate is registered for the CSAR. Optionally, the trustworthiness of the certificate can also be verified by traversing through the certificate chain to see if the root certificate of the customer's certificate is trusted. Once the eligibility of the certificate is confirmed, TA encrypts the CSAR private

key with the customer public key in the certificate, and send it to the *Decrypt CSAR Service*. The *Decrypt CSAR Service* decrypts the CSAR private key using the customer's private key. Now the *Decrypt CSAR Service* has gotten the private key of the CSAR, hence all the encrypted artifacts can be decrypted. If individual artifacts are encrypted with different private keys, then *step 10* needs to be performed multiple times for each artifact.

After the CSAR is decrypted, the service customer deploys the CSAR to a TOSCA compliant cloud provider (**step 11**). If the service developer does not want the service customer to see the code/ artifacts in the CSAR in any way, then the customer has to upload the encrypted CSAR to the TOSCA compliant cloud provider. And in that case, at the time of the service deployment, TOSCA runtime needs to call the *Decrypt CSAR Service* to decrypt the CSAR, as shown in **step 11'**. Also, in that case, the service developer needs to register the TOSCA compliant cloud provider's certificate to the TA.

Once, the deployment of the smart service is done, the Dialogue Policies come into play (**step 12**). The smart service endpoints and their corresponding policies are described in WS-Policy. As discussed in the previous sections, WS-Policy with the usage of its extension WS-Security and WS-Trust, provides several security mechanisms, for instance, username authentication, X509 certificate authentication, etc. Depending on that, the service consumer needs to send the required security tokens in its SOAP message. The smart service infrastructure verifies the validity of the tokens, and respond accordingly. An intermediary could be added between the service consumer and the smart service to audit every request. This can result in more security but can degrade the performance, especially in the case when the metering data from embedded devices need to be analyzed and processed in a timely manner. Moreover, WS-Policy framework does not provide audit policies, but the WS-Policy language could be extended to support this feature.

The approach defined above is not rigid in sense, that some of the components could take more responsibilities, for example, the encrypt CSAR, sign CSAR, verify CSAR, and decrypt CSAR services could be part of the TA. Similarly, the market place or the cloud provider may have their own TAs. A mandatory functionality of TA is to log all the requests made for the private keys, for an individual artifact, or the whole CSAR. The audit trail from the request logs should be available to the service developer. The TA must have a Graphical User Interface (GUI) for private key registration, customer certificate registration, and viewing of the audit trail. Similarly, the secure CSAR services must also have a GUI for use cases, for example, keystore creation, certificate creation, signature, encryption algorithm selection, etc. All the secure CSAR services must either be running on secure transport protocol or have their own endpoints' policies defined in WS-Policies with one of the WS-Security mechanisms as well.

6 Prototype to Secure Cloud Service Archives

The prototype includes the implementation of four services: Encrypt CSAR, Sign CSAR, Verify CSAR, and Decrypt CSAR. These four services are collectively called *Secure CSAR* services in the following discussion. As discussed in the previous chapter, the deployment and management policies are defined using TOSCA Policy Types and Policy Templates, hence the policies of the Secure CSAR services are also defined in TOSCA policy files.

6.1 Technologies

The Secure CSAR services are REST based web services, implemented on a well-known framework for modern Java-based enterprise web applications, known as Spring [SPR]. The version of Java used is 1.8. Bouncy Castle Crypto APIs [BOU] are used for encryption, signing, key generation, and keystore management. The services are packaged as Web application ARchive (WAR) file and are deployed on Tomcat Application Server. The prototype is tested with Tomcat version 9 [TOM]. JavaScript Object Notation (JSON) [JSO] is used as data interchange format between the server and the client which is using the Secure CSAR services. The services are exposed as REST endpoints, so they can be used for any type of client supporting HTTP. It must be noted, that the services are mandatory to run on a secure channel, for example, HTTPS, but the prototype only supports HTTP. For the packaging and dependency management of the Secure CSAR service implementation, Maven [MAV] is used. The prototype also includes a Web GUI front-end to call the Secure CSAR services. The technologies used to implement the front-end are: Grunt [GRU] for front-end web server and task runner, NodeJS [NOD] and Bower [BOW] for front-end package management, Restangular [res] for front-end Model View Controller (MVC) support, and finally Bootstrap [BOO] as a framework to design the look-and-feel of the web front-end. The technologies are chosen with the mission to ease the development and future extensions.

6.2 Secure CSAR Use cases and Algorithms

All the Secure CSAR services can be used independently of each other. Signing and verification scheme is inspired from JAVA JAR signing utilities [JAV] and from [BLS12].

6.2.1 Encrypt CSAR Service

The Encrypt CSAR service encrypts all the artifacts in a CSAR by default. However, individual artifacts can also be specified in the policy file for the encryption. The name of Encrypt CSAR Service REST endpoint is "EncryptCsarService". Figure 6.1 is the flow chart of the algorithm to encrypt a CSAR.

The request to the endpoint must contain the following mandatory attributes: CSAR file, new keystore information or an existing keystore file. The following information is also required, but if not provided, default parameters are used: Encryption algorithm (default value is AES), Key algorithm which is required if a new keystore is being created (default value is AES), Key size is also required a new keystore is being created (default value is 128). Other optional parameters are Encrypted By and Encryptor contact information.

Once a request is received, the request is validated. The validation checks the existence of mandatory attributes and validation of keystore credentials, for example, if the alias exists, or the password of the keystore is correct, etc. Validation also checks if the encryption algorithm is compliant with the encryption key and if the key size is correct. The prototype provides the following options for encryption algorithms and the corresponding key algorithms: AES, DES, DESede. All the possible key sizes for the key algorithms are supported, for example, for AES, the key size could be one of 128, 192, and 256. It must be noted that encryption of CSAR can take a lot of time if the encryption algorithm is asymmetric. In one run, that was performed while testing the prototype, it took 26 minutes to encrypt a CSAR of size 25MB, using RSA encryption, on a machine with 2.59GHz processor, 8GB RAM and 64-bit operating system. So considering this, the prototype is only supporting symmetric keys currently. The decision for asymmetric or symmetric encryption can be left for the client.

If the validation gets failed then an error response is created, having validation failure reason, and is sent to the client. If the validation is successful, the csar file in the request is extracted using Apache Commons Compress utility [Com]. A list of the artifacts in the CSAR is created. A Policy object is created from the Policy Template in the CSAR. The Policy Template, the corresponding Policy Type, and Policy schema for CSAR encryption information are shown in Listing 6.5, Listing 6.7, and Listing 6.6 respectively. The policy object contains the list of artifacts which need to be encrypted, as defined in the Policy Template. The location of the artifact is defined by the path of the artifact in CSAR. If

Listing 6.1 CSAR TOSCA.meta file - Encrypted entry

```
1  .
2  .
3  Name: artifacttemplates/path/to/a/deploymentartifact.ear
4  Content-Type: application/zip
5  Encrypted-By: Muhammad Ali Haider
6  Encryption-Algorithm: AES
7  Encryptor-Contact: smalihaider@gmail.com
8  Key-From: KEYSTORE
9  .
10 .
```

there exist `TOSCA-Metadata/TOSCA.meta` file, CSAR manifest object is created out from it. If there is no such file exists, then the manifest object is created, using the artifact list in the CSAR object. If there is a request to create new keystore, then a new JCEKS type keystore is created, with provided alias name, password, and keystore name. A new symmetric key is created in the keystore. If the client provides an existing keystore, then JCEKS keystore object is created from the provided keystore file and keystore password. The symmetric key is extracted from the keystore using the provided alias name and alias password.

All the artifacts listed in the Policy object are encrypted. For each encrypted artifact, its entry in the manifest object is updated. Listing 6.1 shows how an encrypted entry may look like. Here, Encryptor-Contact (Line 5 in Listing 6.1) and Key-From (Line 6 in Listing 6.1) can be used to contact the service developer for the private key, for decryption. This is not the approach defined in the previous chapter. In actual, the process of exchanging private key for decryption is automatic, via Trusted Authority (TA). But in the prototype, a manual process is being followed, it does give a general idea though, that how private/ public keys are used for encryption, signing, verification, and decryption. In fact, this process is practical as well, in cases where there is no such need to automize the key exchange process. The value of Key-From, `"KEYSTORE"` comes from policy template as can be seen on Line 14 in Listing 6.5. There could be URL instead of `"KEYSTORE"` if the process needs to be automized. The policy engine can read this URL and request, for example, the TA for the private key for decryption. Once all the artifacts are encrypted and TOSCA.meta is updated for all the encrypted artifacts, encrypted CSAR object is created out of it. If the client had requested for the creation of a new keystore, then the created keystore and the encrypted CSAR are zipped and sent to the client, otherwise, only the encrypted CSAR is sent to the client.

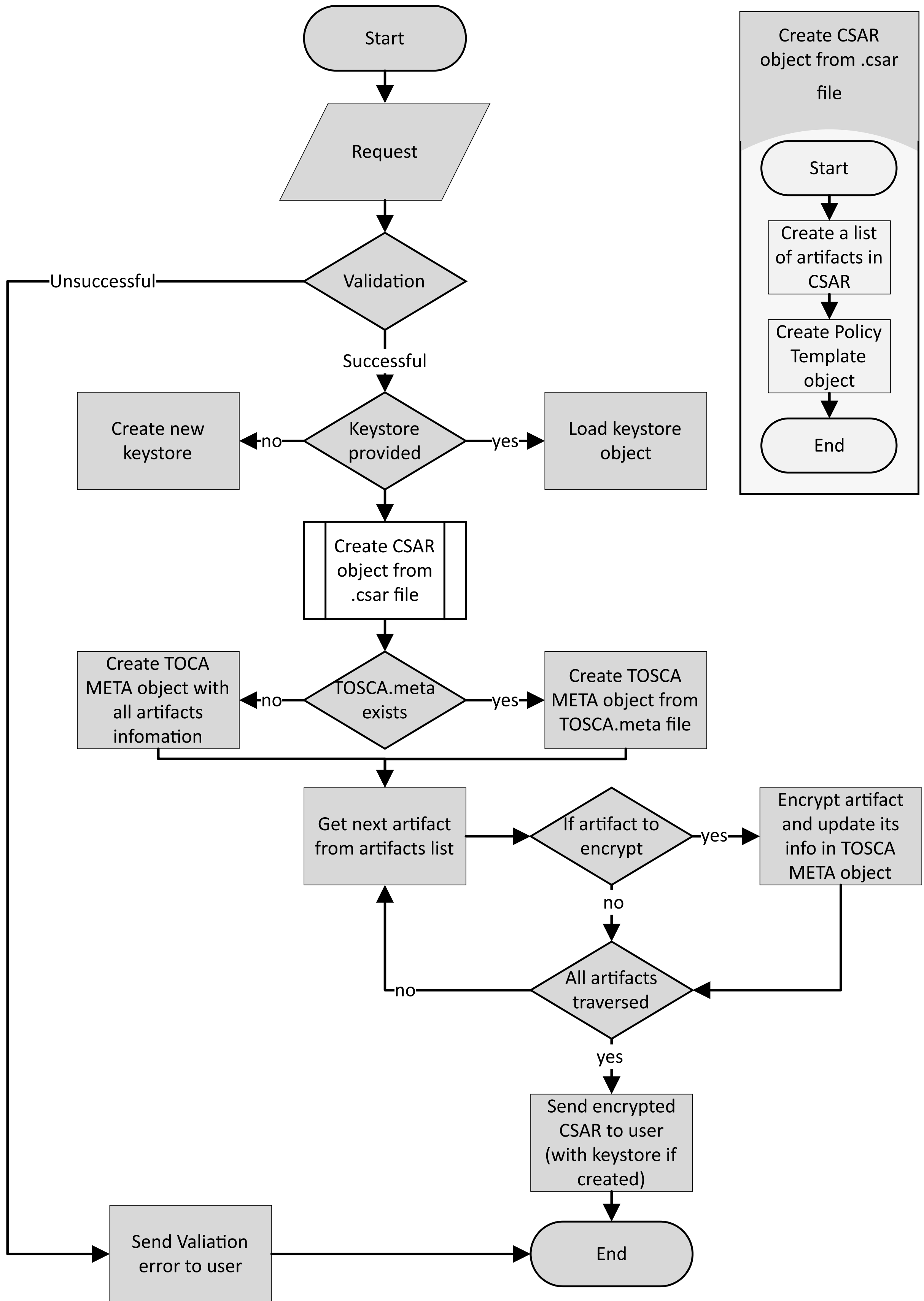

Figure 6.1: Encrypt CSAR Service - Algorithm Flowchart

6.2.2 Sign CSAR Service

The Sign CSAR service signs all the artifacts in a CSAR by default. However, individual artifacts can also be specified in the policy file for the signing. The name of Sign CSAR Service REST endpoint is "SignCsarService". Figure 6.2 is the flow chart of the algorithm to sign a CSAR.

The request to the endpoint must contain the following mandatory attributes: csar file, new keystore information or an existing keystore file. The following information is also required, but if not provided, default parameters are used: digest algorithm (default value is SHA-256), signature algorithm (default value is SHA1withDSA), key algorithm (default value is DSA), key size (default value is 1024), certificate Information, certificate signature algorithm (default value is SHA1withDSA), certificate validity, which are only required if a new keystore is being created. If certificate information is not provided, all the certificate fields, for example, Name of Organization, Name of City or Locality, are set to "Unknown". Other optional parameters are the Signature file name. If the signature file name is not provided, keystore alias name is used as the signature file name.

Once a request is received, the request is validated. The validation checks the existence of mandatory attributes, validation of keystore credentials, for example, if the alias exists, or the password of the keystore is correct, etc. Validation also checks if the signature algorithm is compliant with the signature key and if the key size is correct. The prototype provides the following options for signature algorithms: MD2withRSA, MD5withRSA, SHA1withRSA, SHA256withRSA, SHA384withRSA, SHA512withRSA, SHA1withDSA, with the corresponding key algorithms: RSA and DSA. All the possible key sizes for the key algorithms are supported, for example, for AES, full range from 512 to 1024 is supported.

If the validation gets failed then an error response is created, having validation failure reason, and is sent to the client. If the validation is successful, the csar file in the request is extracted. A list of the artifacts in the CSAR is created. The Policy object is created from the policy template file in the CSAR. The Policy Template, the corresponding Policy Type, and Policy schema for CSAR encryption information are shown in Listing 6.5, Listing 6.7, and Listing 6.6 respectively. The policy object contains the list of artifacts which need to be signed, as defined in the Policy Template. The location of the artifact is defined by the path of the artifact in CSAR. If there exist `TOSCA-Metadata/TOSCA.meta` file, CSAR manifest object is created out from it. If there is no such file exists, then the manifest object is created, using the artifact list in the CSAR object. If there is a request to create new keystore, then a new JCEKS type keystore is created, with provided alias name, password, and keystore name. A key pair is created using the provided key algorithm and key size. The certificate associated with the key pair is also created in

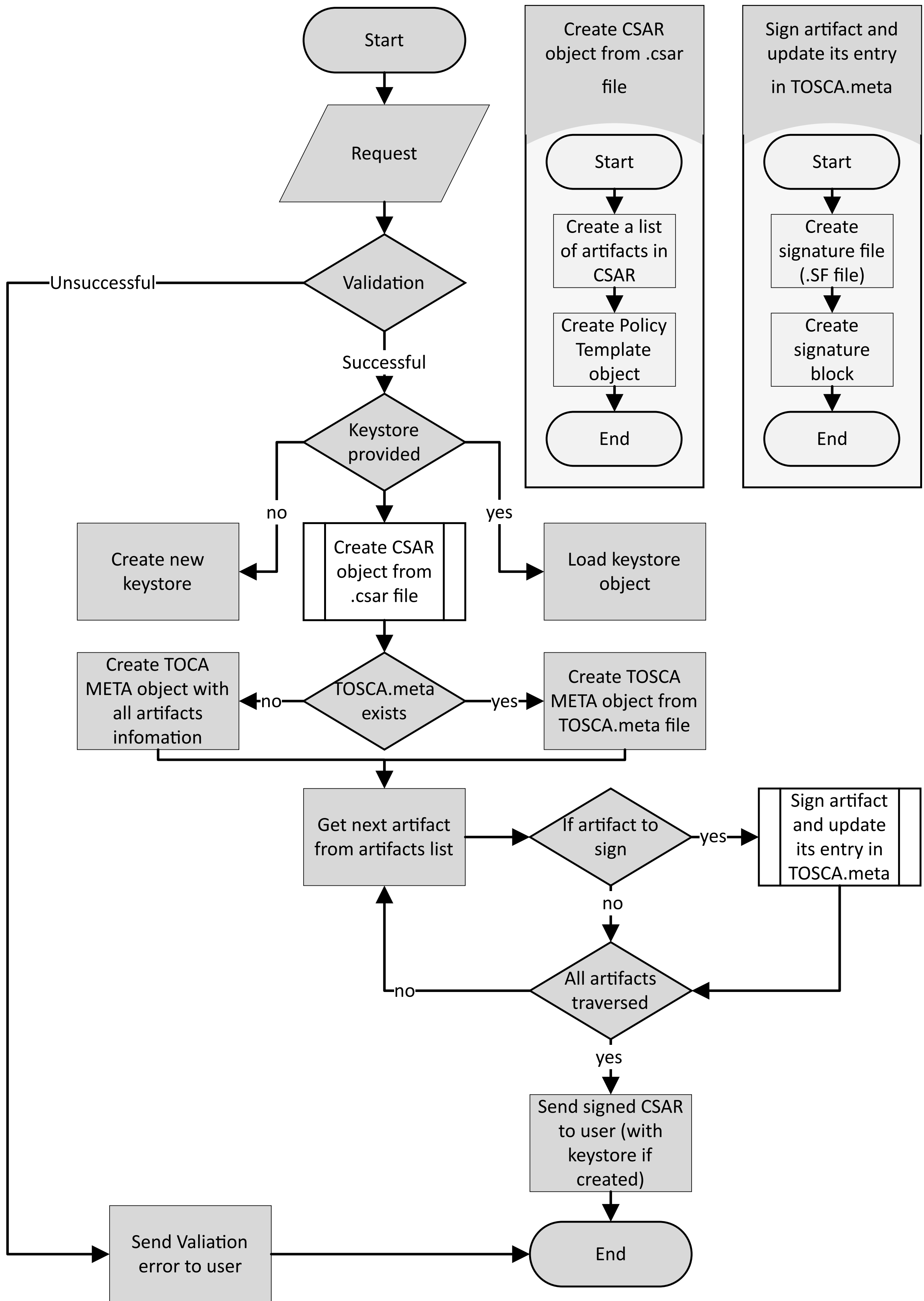

Figure 6.2: Sign CSAR Service - Algorithm Flowchart

Listing 6.2 CSAR TOSCA.meta file - Signed entry

```
1  .
2  .
3  Name: Definitions/path/to/a/artifact.tosca
4  SHA-256-Digest: 5r042XJ1ILzM4bMX2fv3yZcH0TYLkV7DRgG8rQjrrkc=
5  Content-Type: application/vnd.oasis.tosca.definitions
6  .
7  .
```

the keystore. If the client provides an existing keystore, then JCEKS keystore object is created from the provided keystore file and keystore password. The key pair is extracted from the keystore using the provided alias name and alias password.

For each artifact to sign, the hash of the artifact is created using the digest algorithm. The artifact entry in TOSCA.meta file is updated as shown in Listing 6.2. "SHA-256-Digest" key name tells about the hashing algorithm used, and the key value is the hash of the contents of the artifact. Once the hash of each artifact is calculated, the hash of each entry of the manifest is also calculated and is written to the signature file (Listing 6.3). The name of the signature file is either provided by the client or otherwise, the name is taken from the keystore alias. The extension of the signature file is always ".SF". Once the hash of each manifest is created, the hash of whole the TOSCA.meta file (Line 3 in Listing 6.4) and hash of the header of TOSCA.meta (Line 2 in Listing 6.4) are created. The hash of the header is calculated to verify, for example, the entry point of service template is not changed. The purpose of the hash of the whole manifest is to verify, that neither a new file is added in the CSAR, nor any of the existing files are deleted. Once the signature file is created, signature block file is created. The signature block file is ASN1 encoded file and is created with signing the content of the signature file with the private key part of key-pair, using the provided signature algorithm. The name of the signature block file is either provided by the service client or if not provided, alias name of the keystore is used. The extension of the signature block file depends on the signature algorithm. So if the signature algorithm used is "SHA1withDSA", the extension is ".DSA".

Multiple signing of a CSAR is also possible, and in that case, there are multiple signature and signature block files, each having a different name. If the CSAR is signed the second time, with the same signature name and the same signature algorithm, signature file and signature block are overwritten. If different digest algorithms are used in each signing, then there are multiple digest entries in TOSCA.meta file as shown in Listing 6.3. The signature file, signature block file, and updated TOSCA.meta files are put into the CSAR. If the client had requested for the creation of a new keystore, then the created keystore and the signed CSAR are zipped and sent to the client, otherwise only the signed CSAR is sent to the client.

Listing 6.3 CSAR TOSCA.meta file - Signing a CSAR multiple times with different digest algorithms

```
.
.
Name: Definitions/path/to/a/artifact.tosca
SHA-256-Digest: 5r042XJ1ILzM4bMX2fv3yZcH0TYLkV7DRgG8rQjrrkc=
MD5-Digest: nwb0El4x3EUPjCuK7ZzgpA==
Content-Type: application/vnd.oasis.tosca.definitions
.
.
```

Listing 6.4 Signature file (with .SF extension) of a signed CSAR

```
Signature-Version: 1.0
SHA-256-Digest-Manifest-Main-Attributes: c38rAc98FtZ9Hc0kKyao5wXylOW8etEYEkcoAt8taTY=
SHA-256-Digest-Manifest: YWaVexRuQql7FAvY/IPj8oQ1Y5V+fdV3awFWgOeZ1rA=
Created-By: University of Stuttgart - IAAS Department

Name: Definitions/path/to/a/artifact1.tosca
SHA-256-Digest: OgU/KurG0cXPNblElX+y82vOKY7MXB2ig3GfJUbNlDg=

Name: Definitions/path/to/a/artifact2.tosca
SHA-256-Digest: N3fm3LmFE7jYPOzRhTO4/2PE689HkQ5Hnu/HTx47bNo=
.
.
```

6.2.3 Verify CSAR Service

The Verify CSAR service verifies only those artifacts which are requested for, by providing the signature name. The name of Verify CSAR Service REST endpoint is "VerifyCsarService". Figure 6.3 is the flow chart of the algorithm to verify a CSAR. This service maintains a list of response messages. The purpose of this list is to present the verification status of each requested artifact. Overall CSAR verification result, missing mandatory files, etc. are also added to this response message list. The request to the endpoint must contain the following mandatory attributes: CSAR file and signature file name. Once a request is received, the request is validated. The validation checks for the existence of mandatory attributes. If the validation gets failed then an error response is created, having validation failure reason, and is sent to the client. If the validation is successful, the csar file in the request is extracted.

If TOSCA.meta file, signature file (.SF) against the provided signature file name, and signature block file against the signature file name, do not exist in the CSAR, corresponding messages are populated in the response message list. Otherwise, the signature file (.SF) file is read, and all the entries are traversed for verification.

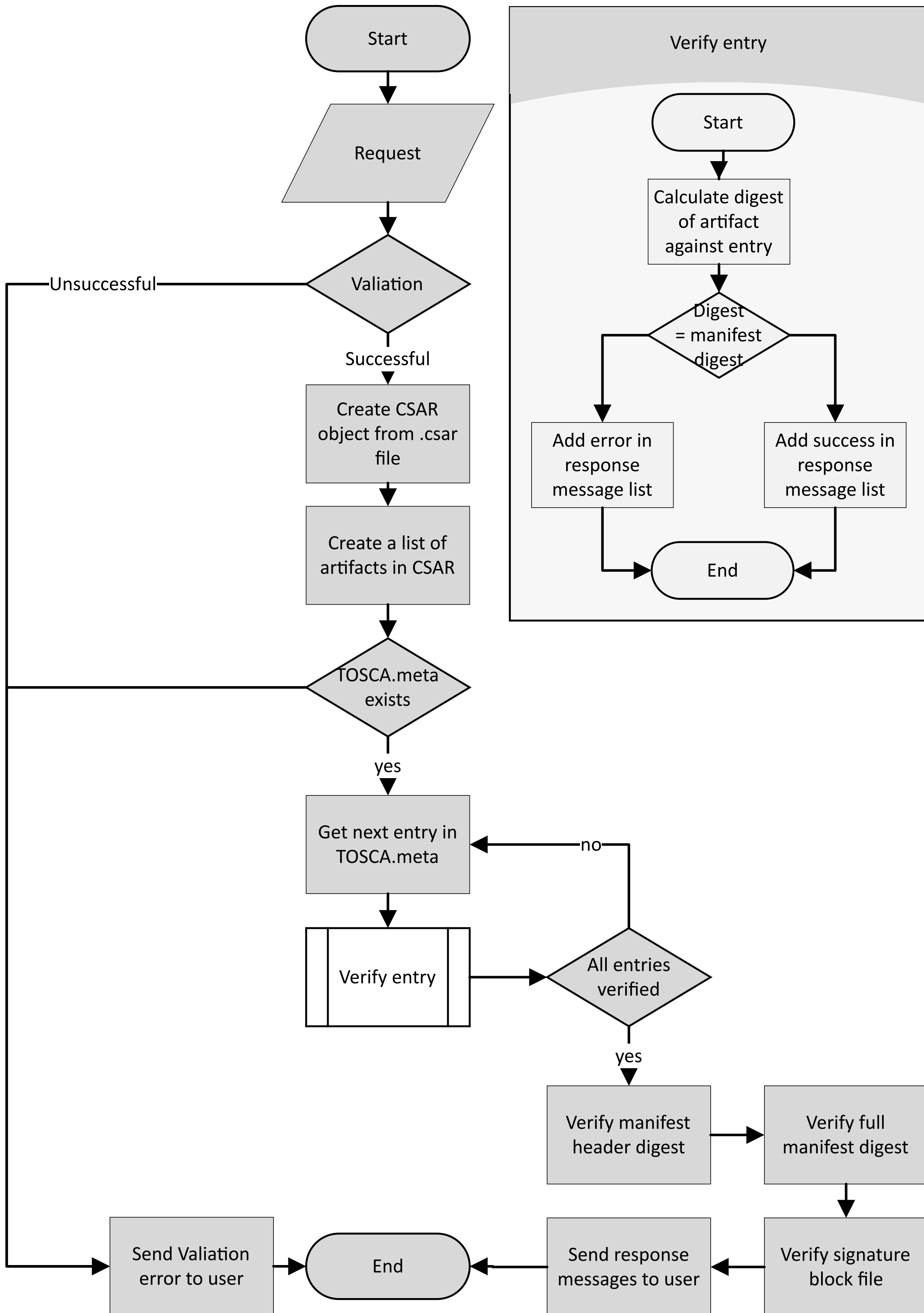

Figure 6.3: Verify CSAR Service - Algorithm Flowchart

For verification, the entry in TOSCA.meta corresponding to the ".SF" file is located. Then the artifact corresponding to the TOSCA.meta entry is loaded. Further, the hashing algorithm is also read, which is mentioned in one of the meta file entry (Line 4 in Listing 6.2). The digest is calculated for the artifact. The digest is compared with the one mentioned in the TOSCA.meta entry. If both of the digests are equal, which means that the content of the artifact has not changed, the verification for the artifact gets successful. If the digests are not equal, verification fails message is added to the response message list.

Once all the artifacts are verified. Hash of the whole TOSCA.meta is calculated and compared with the hash mentioned in the signature file (Line 3 in Listing 6.4). I both of the hashes are equal, this means that neither a new file got added nor deleted in the CSAR. The corresponding verification message is added to the response message list. In the end, signature block file is loaded and verified using BouncyCastle CMSSignedData Crypto API. Finally, the response message list is sent to the client.

6.2.4 Decrypt CSAR Service

The Decrypt CSAR service decrypts all the encrypted artifacts in a CSAR. The name of Decrypt CSAR Service REST endpoint is "DecryptCsarService". Figure 6.4 is the flow chart of the algorithm to decrypt a CSAR. The information about which artifacts need to decrypted comes from TOSCA.meta file. The structure of an encrypted entry (Listing 6.1) tells about the encrypted algorithm and from where to get the key to decrypt the artifact. This service maintains a list of response messages. The purpose of this list is to present the decryption status of each encrypted artifact. Overall CSAR decryption result is also added to this response message list. The request to the endpoint must contain the following mandatory attributes: CSAR file, keystore file, alias name, alias password. Once a request is received, the request is validated. The validation checks for the existence of mandatory attributes and credentials of keystore and alias. If the validation gets failed then an error response is created, having validation failure reason, and is sent to the client. If the validation is successful, the csar file in the request is extracted.

If TOSCA.meta file does not exist in the CSAR, error message is added to the response message list. Otherwise, all entries in TOSCA.meta files are traversed. If an entry needs to be decrypted then the symmetric private key is extracted from the keystore and de-cypher is performed on the content of the artifact. If the artifacts gets decrypted successfully, its encryption information is removed from the TOSCA.meta file. Once all the artifacts are decrypted, all the artifacts are packaged back in CSAR, and this decrypted CSAR is sent to the client.

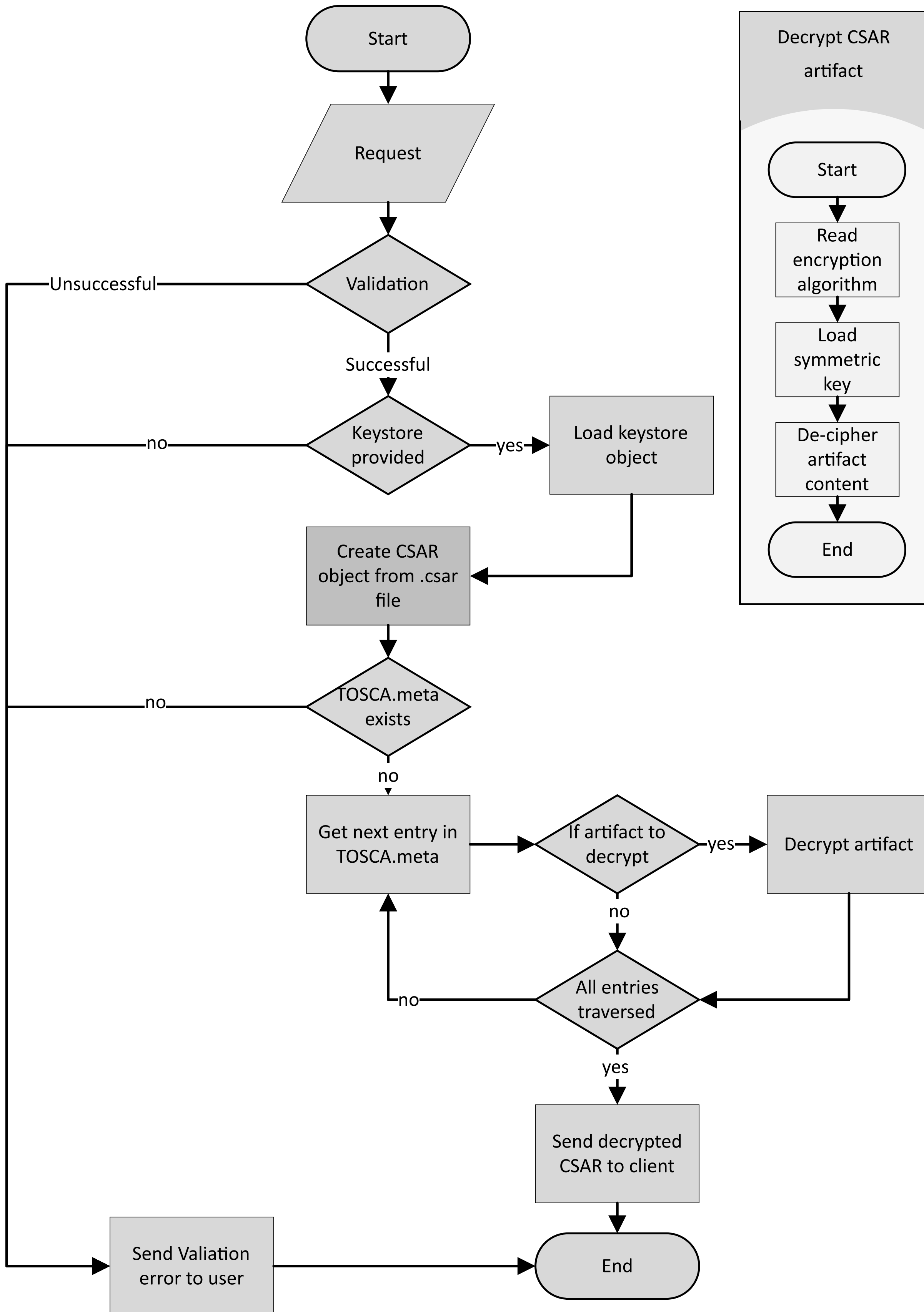

Figure 6.4: Decrypt CSAR Service - Algorithm Flowchart

Listing 6.5 Secure CSAR Policy Template for Signing and Encryption

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <tosca:Definitions id="SecureCsarPolicyTemplate"
       xmlns:tosca="http://docs.oasis-open.org/tosca/ns/2011/12"
       targetNamespace="http://opentosca.org/policytemplates"
       xmlns:pt="http://opentosca.org/policytypes"
       xmlns:pp="http://opentosca.org/policyproperties">
3
4     <tosca:Import namespace="http://opentosca.org/policytypes"
          location="SecureCsarPolicyTpes.tosca"
          importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>
5
6     <tosca:PolicyTemplate id="EncryptCsarPolicy" name="Encrypt Csar Policy"
          type="pt:EncryptCsar">
7        <tosca:Properties>
8           <pp:EncryptCsarProperties>
9              <pp:ArtifactsToEncrypt>
10                <tosca:ArtifactReference
                      reference="artifacttemplates/path/to/a/deploymentartifact.ear"/>
11                <tosca:ArtifactReference
                      reference="artifacttemplates/path/to/a/scriptfile.py"/>
12             </pp:ArtifactsToEncrypt>
13             <pp:DecryptionMode>
14                <pp:Name>KEYSTORE</pp:Name>
15             </pp:DecryptionMode>
16          </pp:EncryptCsarProperties>
17       </tosca:Properties>
18    </tosca:PolicyTemplate>
19
20    <tosca:PolicyTemplate id="SignCsarPolicy" name="Sign Csar Policy" type="pt:SignCsar">
21       <tosca:Properties>
22          <pp:SignCsarProperties>
23             <pp:SignAllArtifacts>true</pp:SignAllArtifacts>
24          </pp:SignCsarProperties>
25       </tosca:Properties>
26    </tosca:PolicyTemplate>
27
28 </tosca:Definitions>
```

Listing 6.6 Secure CSAR Policy Schema for Signing and Encryption

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <schema attributeFormDefault="unqualified" elementFormDefault="qualified"
       targetNamespace="http://opentosca.org/policyproperties"
       xmlns:tosca="http://docs.oasis-open.org/tosca/ns/2011/12"
       xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4     <xs:element name="EncryptCsarProperties">
5        <xs:complexType>
6           <xs:sequence>
7              <xs:choice>
8                 <xs:element name="EncryptAllArtifacts" type="xs:boolean" default="true"/>
9                 <xs:element name="ArtifactsToEncrypt" type="tosca:ArtifactReferences"/>
10             </xs:choice>
11             <xs:element name="DecryptionMode">
12                <xs:complexType>
13                   <xs:sequence>
14                      <xs:element name="Name"/>
15                      <xs:element name="Url" minOccurs="0"/>
16                   </xs:sequence>
17                </xs:complexType>
18             </xs:element>
19          </xs:sequence>
20       </xs:complexType>
21    </xs:element>
22
23    <xs:element name="SignCsarProperties">
24       <xs:complexType>
25          <xs:sequence>
26             <xs:choice>
27                <xs:element name="SignAllArtifacts" type="xs:boolean" default="true"/>
28                <xs:element name="ArtifactsToSign" type="tosca:ArtifactReferences"/>
29             </xs:choice>
30          </xs:sequence>
31       </xs:complexType>
32    </xs:element>
33
34 </schema>
```

Listing 6.7 Secure CSAR Policy Type for Signing and Encryption

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <tosca:Definitions id="EncryptCsarPolocyTypes" name="Encrypt Csar Policy Types"
       targetNamespace="http://opentosca.org/policytypes"
       xmlns:tosca="http://docs.oasis-open.org/tosca/ns/2011/12"
       xmlns:pp="http://opentosca.org/policyproperties">
3
4  <tosca:Import namespace="http://opentosca.org/policyproperties"
       location="schemas/SecureCsarProperties.xsd"
       importType="http://www.w3.org/2001/XMLSchema"/>
5
6  <tosca:PolicyType name="EncryptCsar">
7  <tosca:documentation>EncryptCsar policy type defines list of artifacts to be encrypted in
       CSAR</tosca:documentation>
8  <tosca:PropertiesDefinition element="pp:EncryptCsarProperties"/>
9  </tosca:PolicyType>
10
11 <tosca:PolicyType name="SignCsar">
12 <tosca:documentation>SignCsar policy type defines list of artifacts to be signed in
       CSAR</tosca:documentation>
13 <tosca:PropertiesDefinition element="pp:SignCsarProperties"/>
14 </tosca:PolicyType>
15
16 </tosca:Definitions>
```

6.2.5 Prototype Usage Instructions

The implementation of the prototype is consisted of two projects: *securecsar-frontend* (accessible at https://github.com/smalihaider/securecsar-frontend.git), which contains Web based GUI to call REST services, and *securecsar* (accessible at https://github.com/smalihaider/securecsar.git), which contains the REST web services. The respective URLs contains instructions to setup and use the *Secure CSAR Service*.

7 Conclusion and Future Work

The goal of this thesis is to define a concrete security policy approach for smart services with the refinement and enhancement of TOSCA standard. Thereby, various policy languages and their applicability for the field of Industry 4.0 are researched. In total, 19 policy language characteristics and 3 policy frameworks are analyzed to set selection criteria for the policy languages.

The 5 policy languages shorlisted and included in this thesis are TPL/ DTPL, EPAL, XACML, Ponder, and WS-Policy. The semantics, features, underlying policy engines, and characteristics of each of the languages are discussed. Various language snippets are presented to define common security requirements. Strong and weak points of the languages are identified using the defined characteristics and the context of use. Different security requirements in the area of Industry 4.0 are discussed with the recommendation for a policy language given for specific scenarios. It is identified that the security requirements of the smart services need to be satisfied with policies of two types: deployment and management policies, and dialogue policies. The deployment and management policies cater policy requirements, such as, availability of an application, database encryption, cloud provider region selection, etc. which are enforced when an application is being deployed and when the management aspects of an application are being configured. On the other hand, the dialogue policies deal with the policy requirements, for instance, a secure transport channel, client server mutual authentication using X509 certificates, etc. which are supposed to be enforced during a dialogue between a smart service and a client.

It is suggested that the deployment and management policies should be defined using TOSCA language constructs, Policy Types and Policy Templates, to get the full use of TOSCA automatic deployment and management support. For the dialogue policies of the smart services, WS-Policy framework is selected. WS-Policy with its accompanying security policy extensions WS-Security and WS-Trust provide end-to-end security to SOAP based web services. The ability of WS-Policy framework to support distributed policy evaluation, protection of sensitive policies, and being an approved W3C standard also contributed to the selection. On a downside, WS-Policy does not provide obligation policies and rule combining algorithms, which can be areas of WS-Policy extension.

Finally, an approach is formulated to secure cloud service archives for function and data shipping in industrial environments through the usage of policies. The approach also covers the end-to-end security of smart services. The approach requires no extension for the TOSCA metamodel, except the addition of a Trust Authority component in TOSCA processing environment for credentials management. A prototype, realizing the use case of encrypting, signing, verifying, and decrypting a CSAR is also implemented and discussed in this thesis.

Future work is required to enhance WS-Policy framework to provide the missing features which exist in another popular policy language, XACML. It is proposed to introduce a new extension for the WS-Policy framework with the name WS-Obligation, for provisioning of obligation policies, for example, auditing of SOAP request and response messages, deletion of records, emailing of messages, etc. Further, it is recommended that the Secure CSAR services should be part of the TOSCA modeling tools. OpenTOSCA [Stu] which is an open source ecosystem for TOSCA can be extended to incorporate the Trust Authority and Secure CSAR services to demonstrate the proposed approach.

Bibliography

[AKN17] B. Abendroth, A. Kleiner, P. Nicholas. "CYBERSECURITY POLICY FOR THE INTERNET OF THINGS." In: (2017). URL: https://mscorpmedia.azureedge.net/mscorpmedia/2017/05/IoT_WhitePaper_5_15_17.pdf (cit. on pp. 15, 25).

[AL05] G. Allmendinger, R. Lombreglia. *Four Strategies for the Age of Smart Services*. Oct. 2005. URL: https://hbr.org/2005/10/four-strategies-for-the-age-of-smart-services (cit. on p. 26).

[And04] A. Anderson. *Privacy Policy Languages: XACML vs EPAL*. 2004. URL: http://cacr.uwaterloo.ca/conferences/2004/isw/slides/AnneAndersonpresslides.pdf (cit. on pp. 41, 56).

[And05] A. Anderson. "A Comparison of Two Privacy Policy Languages: EPAL and XACML Anne." In: (Sept. 2005) (cit. on p. 55).

[AXI12] AXIOMATICS, ed. *Axiomatics releases free plugin for the Eclipse IDE to author XACML3.0 policies*. July 16, 2012. URL: https://www.axiomatics.com/news/axiomatics-releases-free-plugin-for-the-eclipse-ide-to-author-xacml3-0-policies/ (cit. on p. 56).

[BBC06] S. Bajaj, D. Box, D. Chappell. *Web Services Policy 1.2 - Framework (WS-Policy)*. Apr. 25, 2006. URL: https://www.w3.org/Submission/WS-Policy/ (cit. on pp. 48–51).

[BBK+13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, M. Wieland. "Policy-Aware Provisioning of Cloud Applications." In: *Proceedings of the Seventh International Conference on Emerging Security Information, Systems and Technologies (SECURWARE)*. Xpert Publishing Services (XPS), 2013, pp. 86–95 (cit. on p. 61).

[BBKL14] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. In: *Advanced Web Services*. New York: Springer, Jan. 2014. Chap. TOSCA: Portable Automated Deployment and Management of Cloud Applications, pp. 527–549. ISBN: 978-1-4614-7534-7. DOI: 10.1007/978-1-4614-7535-4_22 (cit. on pp. 15, 24).

[BLS12] G. Breiter, F. Leymann, T. Spatzier. *Topology and Orchestration Specification for Cloud Applications (TOSCA): Cloud Service Archive (CSAR) – Version 0.1 –*. May 2012 (cit. on p. 68).

[BOO] BOOTSTRAP. *Bootstrap: The world's most popular mobile-first and responsive front-end framework*. URL: http://getbootstrap.com/ (cit. on p. 67).

[BOU] BOUNCYCASTLE. *The Legion of the Bouncy Castle*. URL: https://www.bouncycastle.org/ (cit. on p. 67).

[BOW] BOWER. *Bower A package manager for the web*. URL: https://bower.io/ (cit. on p. 67).

[CO08] J. L. D. Coi, D. Olmedilla. *A REVIEW OF TRUST MANAGEMENT, SECURITY AND PRIVACY POLICY LANGUAGES*. July 2008 (cit. on pp. 31–34, 48, 53, 54).

[Com] A. Commons. *Apache Commons Compress*. URL: https://commons.apache.org/proper/commons-compress/ (cit. on p. 68).

[DDLS01] N. Damianou, N. Dulay, E. Lupu, M. Sloman. "The Ponder Policy Specification Language." In: Jan. 31, 2001. URL: https://spiral.imperial.ac.uk:8443/bitstream/10044/1/18346/2/Policy2001.pdf (cit. on p. 46).

[DHS07] C. Duma, A. Herzog, N. Shahmehr. "Privacy in the Semantic Web: What Policy Languages Have to Offer." In: 2007 (cit. on p. 31).

[FBC+16] M. Falkenthal, U. Breitenbücher, M. Christ, C. Endres, A. W. Kempa-Liehr, F. Leymann, M. Zimmermann. "Towards Function and Data Shipping in Manufacturing Environments : How Cloud Technologies leverage the 4th Industrial Revolution." In: *Proceedings of the 10th Advanced Summer School on Service Oriented Computing*. IBM Research Division, 2016, pp. 16–25 (cit. on pp. 16, 26, 59).

[GRU] GRUNT. *GRUNT The JavaScript Task Runner*. URL: https://gruntjs.com/ (cit. on p. 67).

[HKPS03] P. A. S. Hada, G. Karjoth, C. Powers, M. Schunter. *Enterprise Privacy Authorization Language (EPAL 1.2)*. Ed. by C. Powers, M. Schunter. Nov. 10, 2003. URL: https://www.w3.org/Submission/2003/SUBM-EPAL-20031110/ (cit. on pp. 40, 55, 56).

[HL12] W. Han, C. Lei. "A survey on policy languages in network and security management." In: *The International Journal of Computer and Telecommunications Networking* (Mar. 2012) (cit. on pp. 29, 31, 33).

[HP] HP, ed. *The EnCoRe Project*. URL: http://www.hpl.hp.com/breweb/encoreproject/ (cit. on p. 29).

[IBM] R. IBM, ed. *Policy Language*. URL: https://www.research.ibm.com/haifa/projects/software/e-Business/TrustManager/PolicyLanguage.html (cit. on p. 37).

[JAV] JAVA. *Signing and Verifying JAR Files*. URL: https://docs.oracle.com/javase/tutorial/deployment/jar/signindex.html (cit. on p. 68).

[JSO] JSON. *Introducing JSON*. URL: http://www.json.org/ (cit. on p. 67).

[KCLC] P. Kumaraguru, L. F. Cranor, J. Lobo, S. B. Calo. *A Survey of Privacy Policy Languages*.

[Ley] "Policies For Web Services (University of Stuttgart IAAS - Service Computing course lecture slides)" (cit. on p. 49).

[Lip13] P. Lipton. "Escaping Vendor Lock-in with TOSCA, an Emerging Cloud Standard for Portability." In: (2013). URL: http://www.arcserve.com/us/~/media/Files/About%20Us/CATX/escaping-vendor-lock-in-with-tosca-an-emerging-cloud-standard-for-portability.pdf (cit. on p. 20).

[MAV] MAVEN. *Apache Maven Project*. URL: https://maven.apache.org/ (cit. on p. 67).

[MG11] P. Mell, T. Grance. *The NIST Definition of Cloud Computing. Recommendations of the National Institute of Standards and Technology*. Sept. 2011, p. 2. 3 pp. URL: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (cit. on pp. 15, 19).

[MN] O. M. G. B. P. Model, Notation. *Object Management Group Business Process Model and Notation*. URL: http://www.bpmn.org/ (cit. on p. 20).

[Mos03] T. Moses. *A technical comparison of WS-Policy framework and XACML*. June 29, 2003. URL: https://lists.oasis-open.org/archives/xacml/200301/msg00021.html (cit. on p. 58).

[NGG+] A. Nadalin, M. Goodner, M. Gudgin, D. Turner, A. Barbir, H. Granqvist, eds. *WS-Trust 1.4*. URL: http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html (cit. on p. 51).

[NOD] NODEJS. *NODEJS*. URL: https://nodejs.org/en/ (cit. on p. 67).

[OAS06] OASIS, ed. *OASIS Web Services Security (WSS) TC*. Nov. 28, 2006. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss (cit. on pp. 50, 51, 62).

[OAS13a] OASIS, ed. *A Brief Introduction to XACML*. Mar. 14, 2013. URL: https://www.oasis-open.org/committees/download.php/2713/Brief_Introduction_to_XACML.html (cit. on p. 42).

[OAS13b] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. OASIS Standard, Nov. 25, 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (cit. on pp. 15, 21–25, 63, 64).

[PM11] S. Pearson, M. C. Mont. "Sticky Policies: An Approach for Managing Privacy across Multiple Parties." In: (Sept. 2011) (cit. on pp. 29–31).

[Pon13] Ponder2Project, ed. *Ponder2 Wiki*. Nov. 25, 2013. URL: http://ponder2.net/ (cit. on pp. 48, 55).

[res] restangular. *AngularJS service to handle Rest API Restful Resources properly and easily*. URL: https://github.com/mgonto/restangular (cit. on p. 67).

[Ris13] E. Rissanen, ed. *eXtensible Access Control Markup Language (XACML) Version 3.0*. Jan. 23, 2013. URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html#_Toc325047070 (cit. on pp. 42–44, 55, 56).

[SAL15] SALESFORCE-UK. *Why Move To The Cloud? 10 Benefits Of Cloud Computing*. Nov. 17, 2015. URL: https://www.salesforce.com/uk/blog/2015/11/why-move-to-the-cloud-10-benefits-of-cloud-computing.html (visited on 06/25/2017) (cit. on p. 15).

[Slo] M. Sloman. *Ponder: Policy Specification for Large-Scale Distribute Computing*. URL: http://www.hpl.hp.com/news/events/csc/2003/sloman_slides.pdf (cit. on pp. 46, 47).

[Sos10] D. Sosnoski. *Understanding WS-Policy*. Nov. 2, 2010. URL: https://www.ibm.com/developerworks/library/j-jws18/index.html (cit. on p. 51).

[SPR] SPRING. *Spring Framework*. URL: https://projects.spring.io/spring-framework/ (cit. on p. 67).

[Stu] U. of Stuttgart. *OpenTOSCA Research Prototype*. URL: http://www.opentosca.org/ (cit. on p. 82).

[SWY+] K. E. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, L. Yu. *Requirements for Policy Languages for Trust Negotiation* (cit. on pp. 27, 28, 31, 32, 34, 35, 38, 39).

[SY01] K. Seamons, T. Yu. "Limiting the Disclosure of Access Control Policies during Automated Trust Negotiation." In: Jan. 2001. URL: https://www.researchgate.net/publication/277291102_Limiting_the_Disclosure_of_Access_Control_Policies_during_Automated_Trust_Negotiation (cit. on p. 39).

[TC] O. W. S. B. P. E. L. (TC. *OASIS Web Services Business Process Execution Language (WSBPEL) TC*. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel (cit. on p. 20).

[TOM] TOMCAT. *Apache Tomcat*. URL: https://tomcat.apache.org/ (cit. on p. 67).

[VOH07] A. S. Vedamuthu, D. Orchard, F. Hirsch. *Web Services Policy 1.5 - Attachment*. Sept. 4, 2007. URL: https://www.w3.org/TR/ws-policy-attach/ (cit. on p. 51).

[WIKa] WIKIPEDIA, ed. *Industry 4.0*. URL: https://en.wikipedia.org/wiki/Industry_4.0 (cit. on pp. 16, 25).

[WIKb] WIKIPEDIA. *OASIS (organization)*. URL: https://en.wikipedia.org/wiki/OASIS_(organization) (visited on 07/01/2017) (cit. on p. 20).

[WIK17] WIKIPEDIA, ed. *Internet of things*. 2017. URL: https://en.wikipedia.org/wiki/Internet_of_things (cit. on p. 15).

[WWB+13a] T. Waizenegger, M. Wieland, T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, B. Mitschang, A. Nowak, S. Wagner. "Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing." In: *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*. Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-41029-1. DOI: 10.1007/978-3-642-41030-7_26 (cit. on p. 61).

[WWB+13b] T. Waizenegger, M. Wieland, T. Binz, U. Breitenbücher, F. Leymann. "Towards a Policy-Framework for the Deployment and Management of Cloud Services." In: 2013 (cit. on p. 61).

All links were last followed on July 27, 2017.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature