Protecting Deployment Models in Collaborative Cloud Application Development

Vladimir Yussupov, Ghareeb Falazi, Michael Falkenthal, and Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart, Stuttgart, Germany
email: [firstname.lastname]@iaas.uni-stuttgart.de

Abstract—**Profitability of industrial processes today depends on well-timed utilization of new technologies. Development of cloud applications combining cross-domain knowledge from multiple collaborating parties is one common way to enhance manufacturing. Often, such collaborations are not centralized due to outsourcing or rearrangements in organizational structures. Moreover, manual deployment inefficiency and intellectual property issues further tangle the development process of such applications. While the development of deployment models obviates the necessity to manually deploy applications, a way to protect sensitive data in exchanged deployment models is still needed. In this work, we describe the specifics of modeling and enforcement of security requirements for deployment models in the context of decentralized collaborative cloud application development. We provide a stepwise demonstration of how security requirements can be specified and enforced in a collaborative development scenario based on the TOSCA cloud standard. Furthermore, we conceptualize the system architecture, provide details about the implementation of certain approach-specific operations, and discuss the limitations of the approach. Finally, we show the feasibility of the presented concepts via an open-source prototype.**

Keywords–Collaboration; Security Policy; Confidentiality; Integrity; Deployment Model; Deployment Automation; TOSCA.

I. Introduction

In the recent years, processes and technologies fostering manufacturing automation gained a lot of attention from both, industry and academia. Often, an intricacy of industrial processes leads to the fact that desired automation goals can only be achieved using custom-tailored software solutions. Frequently, such software is the result of a teamwork involving multiple independent parties, e.g., representing different participating organizations. As a prerequisite for collective software development to be successful, often, various functional and non-functional system requirements have to be satisfied. Security and privacy of the data exchanged among involved parties are critically important requirements that have to be properly documented and enforced during the development lifecycle [1].

Numerous modern computing paradigms have great potential for accelerating the 4^{th} industrial revolution, often referred to as Industry 4.0 [2]. One notable example is the rapidly evolving field of cloud computing [3], which allows on-demand access to potentially unbounded number of computing resources. Combined together with ubiquitous sensors usage in the context of the Internet of Things (IoT) [4], cloud computing facilitates the development of composite, cross-domain applications tailored specifically for automation and optimization of manufacturing. Along with the clear advantages, such emerging technologies introduce additional challenges that need to be tackled. For instance, the overall complexity of development processes might become a significant obstacle for industries willing to benefit from cloud applications.

A typical cloud application today has a composite structure consisting of multiple interconnected and heterogeneous components [5]. Deploying such complexly-structured applications in a manual fashion is error-prone and inefficient [6]. Therefore, various deployment automation approaches exist. One well-established automation technique relies on the concept of *deployment models* that specify application structure along with the necessary deployment information. Automated processing of such models considerably reduces the deployment's complexity and minimizes required efforts. Another significant benefit, which improves portability and reusability aspects of the application development process, is that instead of separate application components, standardized models can be exchanged [7].

Complexity and heterogeneity of application's components are among the reasons why a common cloud application development scenario in the context of Industry 4.0 is a collaboration [8] involving several multidisciplinary partners responsible for separate parts of the application [5]. The final goal of this collaboration is to combine all parts into a complete and deployable cloud application. Collaborative development can significantly benefit from the portability and reusability properties of deployment models. However, since not all parties are known in advance, e.g., due to task outsourcing or changes in organizational structure, the issues of intellectual property protection in decentralized settings arise. For instance, confidential information like sensor measurements and proprietary algorithms might be subject to various *security requirements*, including protection from unauthorized access and verification of its integrity. Therefore, modeling and enforcement of such requirements aimed at specific parts of deployment models, have to be supported.

In our previous work [1], we introduced a method for modeling and enforcement of security requirements in deployment models that combines the ideas of sticky policies [9], policy-based cryptography [10], and Cryptographic Access Control (CAC) [11]. In this paper, we build upon our previous work and discuss in more details, how security requirements aimed at data protection in modeled cloud applications can be expressed using security policies and which parts of deployment models need to support the attachment of security policies. Focusing more on the practical aspects, we provide a stepwise demonstration of how the introduced approach can be applied to a collaborative and standardized process of deployment model development. To have a uniform way of deployment modeling, we use the existing OASIS standard, Topology and Orchestration Specification for Cloud Applications (TOSCA) [12], [13], which specifies an extensible, provider-agnostic cloud modeling language [14]. To validate our concepts, we implement them in OpenTOSCA [15], an opensource ecosystem that allows modeling and execution of TOSCA-compliant deployment models. Moreover, we include a detailed description of the system architecture and elaborate on the process of security requirements enforcement during import and export of deployment models using our prototype. The remainder of this paper is structured as follows. As in our previous work [1], we first describe the fundamentals underlying this work in Section II and discuss a motivational scenario in Section III. In Section IV, we present concepts for modeling and enforcement of security requirements in collaborative deployment models development. In Section V, we apply the concepts to a TOSCA-based deployment modeling process and provide a demonstration of a TOSCA-based collaborative development scenario using the example collaboration described in the motivational scenario. The details about the prototypical implementation in OpenTOSCA are discussed in Section VI. In addition, we discuss the system's architecture and describe the specifics of import and export processes of deployment models. Finally, in Section VII, we describe related work and conclude this paper in Section VIII.

II. Fundamentals

In this section, we provide an overview of several important concepts that serve as a basis for our work, namely: (i) deployment automation of cloud applications by means of deployment modeling approaches, (ii) usage of policies as means to specify non-functional system requirements, (iii) and a brief coverage of access control mechanisms.

A. Deployment Modeling

The compound application structure and increased integration complexity make it non-trivial to automate the deployment of modern cloud applications [6]. The concept of deployment modeling aims to tackle the automation problem, and there are several known approaches including imperative and declarative modeling [6], [16], [17]. Both paradigms are based on the idea of creating a description, or deployment model, sufficient enough for deploying a chosen application in an automated fashion. What makes these modeling approaches different is the way how corresponding deployment models are implemented.

In case of the declarative modeling [16], a deployment model is a structural model that conveys the desired state and structure of the application. Essential parts of the declarative deployment model include a specification of application's components with respective dependencies and necessary connectivity details. As a result, the model might contain binaries or scripts responsible for running some application's components, e.g., a specific version of Apache Tomcat, or a predefined Shell script for running a set of configuration commands. In addition, a description of non-functional system requirements in some form can be included into the model. Some examples supporting this type of modeling include Chef [18] and Juju [19] automation tools, as well as TOSCA. This type of models relies on the concept of deployment engines, which are able to interpret a provided description and infer a sequence of steps required for successful deployment of the modeled application.

Compared to the declarative approach, the imperative modeling [16] focuses on a procedure, which leads to automatic application deployment. More specifically, an imperative model describes (i) a set of activities corresponding to the required deployment tasks that need to be executed, and (ii) the control and data flow between those activities. One robust technique for this modeling style is to use a process engine, e.g., supporting standards like Business Process Execution Language (BPEL) [20] or Business Process Model and Notation (BPMN) [21], which can execute provided imperative models in an automated fashion.

A combination of declarative and imperative approaches is also possible. In general, creating both types of models requires efforts from the modeler. However, the imperative modeling approach is generally more time-consuming and error-prone, since multiple heterogeneous components need to be properly orchestrated. Moreover, the structure of the application might change frequently, which requires to modify imperative models. To minimize required modeling efforts, imperative models might be derived from the provided declarative models [6].

One important aspect of deployment models is that apart from valid descriptions they also need to include various files related to described software components and other parts of the application, e.g., scripts, binaries, documentation and license details. As a result, the term deployment model usually refers to a combination of all the corresponding metadata and application files required for automatically deploying a target application.

B. Policies

One well-known approach for separating non-functional requirements from the actual functionalities of a target system relies on the usage of policies [22]. Essentially, a *policy* is a semi-structured representation of a certain management goal [23]. The term management here is rather broad, as it might refer to different aspects of management, e.g., high-level corporate goals or more low-level, technology-oriented management goals. For instance, from the system's perspective, performance, configuration, and security are among the classes of non-functional requirements that can be described using policies. Additionally, various policy specification languages exist in order to simplify the process of describing such requirements in a standardized manner [22]. From the high-level view, policies only declare the requirements, which then have to be enforced using dedicated enforcement mechanisms [24].

The idea to specify security requirements in policies dates back to at least the 1970s [22]. Depending on the level of details security policies might specify, e.g., privacy requirements for the whole system or for particular data objects. In information exchange scenarios, security policies specified on the level of data objects have to be ensured during the whole exchange process [25]. For this reason, all receivers have to be aware of specified policies and enforcement must happen, e.g., by means of globally-available security mechanisms. Similarly, deployment models in collaborative application development are constantly exchanged and parts of them might be subject to security policies. So-called *sticky policies* [25] is an approach to propagate policies with the data they target. This approach can be combined with cryptography in order to ensure that data is accessed only when requirements specified in policies are satisfied. Multiple approaches to combine sticky policies with different cryptographic techniques such as public key encryption or Attribute-Based Encryption (ABE) exist [26].

C. Access Control

A secure information system must prevent disclosure (confidentiality) or modification (integrity) of sensitive data to an unauthorized party and ensure that data are accessible (availability) [24]. These requirements can be enforced by assuring only authorized access to the system and its resources. Commonly, this process is referred to as *access control* and there exist multiple well-established access control mechanisms. For example, in Discretionary Access Control (DAC) mechanism, the access is defined based on the user's identity. This results in access rules that are specified specifically for this identity, e.g., in the form of an access control matrix [27]. Another well-known access control mechanism is called Role-Based Access Control (RBAC) where access is granted or denied based on the user roles and access rules defined for these roles.

One disadvantage of the aforementioned access control mechanisms is that they commonly rely on some centralized trusted authority, making it difficult to implement them in large scale and open systems [11]. The idea of CAC is based on well-known cryptographic mechanisms and regulates access permissions based on the possession of encryption keys. In CAC, the stored data are encrypted and can only be accessed by those users who have the corresponding keys. One positive advantage of this approach is that the data owner can grant keys to other involved parties of his choice using established key distribution mechanisms, thus enforcing the access control without relying on the trusted third party.

III. Motivational Scenario

Developing distributed cloud applications and analytics applications in the context of Industry 4.0 typically requires combining numerous heterogeneous software components [28], [29]. Commonly, this process implies a collaboration among experts from various domains, such as data scientists, infrastructure integrators, and application providers. Furthermore, resulting applications are often required to be deployable on demand and, thus, are expected to be in the form of deployment models that allow automating application provisioning [5], [30].

An example of a collaborative cloud application development depicted in Figure 1 involves four participants responsible for distinct parts of the application. When joined together, all developed parts of the application, e.g., software components, datasets, and connectivity information, comprise a complete and provisioning-ready deployment model. In this scenario, the main beneficiary who orders the application from a set of partners and has exclusive rights on the resulting deployment model is called the *Application Owner*. The *Infrastructure Modeler* is responsible for integrating different components, such as analytics runtime environments, databases, or application servers. Moreover, two additional co-modelers are involved in the development process, namely a *Data Scientist* and a *Dataset Provider*. The former develops a certain proprietary algorithm, whereas the latter provides a private dataset, e.g., comprised of sensor measurements obtained from a combination of various cyber-physical systems used in production processes.

While the Application Owner has full rights on the resulting deployment model, other participants might be subject to security restrictions. For example, access to the dataset provided by the Dataset Provider might need to be restricted to some of the parties. Similarly, the Data Scientist might want to specify security requirements on the provided algorithm. Since the final infrastructure must include all corresponding sub-parts that were provided directly or indirectly by participants, the Infrastructure Modeler is responsible for preparation and shipping of the finalized deployment model to the Application Owner who is then able to create new instances of the application on demand.

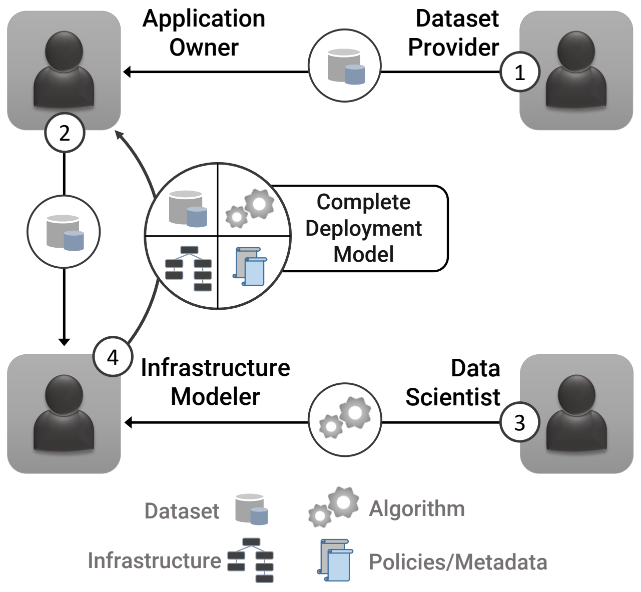

Figure 1. A collaborative application development scenario.

Generally, collaborative processes from various fields share some common characteristics. For instance, according to Wang et al. [31] such issues as (i) *dynamically changing sets of participants*, (ii) the *lack of centralization*, (iii) *intellectual property and trust management issues*, and (iv) *heterogeneity of exchanged data* are important in collaborative development of computer-aided design models. Likewise, the lack of knowledge about all participants involved in collaborative cloud application development makes it difficult to establish a centralized interaction among them. Possible reasons include outsourcing of development tasks and introduction of additional participants due to rearrangements in organizational structures. Since no strict centralization is possible, communication with known participants happens in a peer-to-peer manner. Another important aspect of collaborative cloud application development is its iterative nature. Since exchanged deployment models might be impartial or require several rounds of refinement, a potentially complicated sequence of exchange steps is possible for obtaining a final result. Therefore, deployment models need to be exchanged in collaborations in a way that simplifies the overall process and enforces potential security requirements.

A deployment model, generally, can be exchanged either in a self-contained form or on a per-participant basis. In the former case, the deployment model is self-contained and its content is the same for all participants, whereas in the latter case its content is fragmented according to some rules separately for each participant. Sometimes, however, exchanging deployment models on a per-participant basis interferes with the actual goals of the collaboration. For example, in the exchange sequence shown in Figure 1 the dataset is firstly passed directly to the Application Owner by the Dataset Provider. For the integration of the dataset into the final model, the Infrastructure Modeler needs to model the required infrastructure, e.g., a Database Management System (DBMS) and related tooling. As only the Application Owner has full rights on all parts of the application, the provided dataset has to be protected from unauthorized access. Intellectual property issues become even more complex in highly-dynamic scenarios when multiple parties continuously exchange partially-completed deployment models. Unfortunately, encrypting an entire deployment model does not solve the problem since models might be intended to remain partially-accessible by parties with limited access rights. Apart from confidentiality problems, the authenticity and integrity of passed deployment models and their parts might be subject to verification requirements. For instance, the Application Owner might need to check if an algorithm was actually provided by the Data Scientist and no changes were made by other parties. In such case, signing the hash value of an entire deployment model is not suitable as integrity of individual model's parts have to be verified. Hence, it should be possible to verify distinct parts of deployment models independently.

The aforementioned scenario highlights several important issues in collaborative development of deployment models, which need to be solved, namely (i) confidentiality, authenticity, and integrity requirements of each involved participant have to be reflected in the model, (ii) various levels of granularity for these requirements need to be considered, i.e., from full models to their separate parts, and (iii) a method to enforce modeled requirements in a peer-to-peer model exchange is needed.

IV. Modeling and Enforcement of Security Requirements

Intellectual property in collaborations has to be protected from both, external and internal adversaries with respect to their relation to the process. The former describes any attacker from outside of the collaboration, i.e., who is not participating and is not reflected in any kind of agreements, e.g., Service Level Agreements (SLAs). Conversely, the latter refers to a dishonest party involved in the process. We focus on internal adversaries and data protection issues involving known parties.

This section presents an approach to ensure the fulfillment of security requirements in the collaborative development of deployment models. Our approach relies on the well-established concept of representing non-functional requirements via policies [32], [33], [34], [35]. The semantics of security requirements is analyzed to derive a set of action and grouping policies. The former type represents cryptographic operations allowing to enforce confidentiality and integrity requirements, inspired by the idea of policy-based cryptography [10]. The latter type simplifies grouping parts of models that are subjects to action policies. Both policy types are data-centric and attachment happens with respect to a certain entity or a group of entities in the manner of sticky policies [25] to preserve the self-containment property of deployment models. The access control enforcement is inspired by the idea of CAC [11].

A. Assumptions

To focus on internal adversaries, we assume that participants establish bidirectional secure communication channels for data exchange and that the modeling environment of every involved participant is secure. By modeling environment we mean any software that simplifies the process of deployment modeling, e.g., by providing functionalities like loading (import) and packaging (export) of deployment models of different formats or performing various kinds of model validation. We employ an "honest but curious" [36], [37], [38] adversary model in which adversaries are interested in reading the data, but avoid modifications to remain undetected. Despite the absence of modifications made by adversaries, authenticity and integrity requirements still need to be modeled and enforced. For instance, participants might want to track changes or verify the origin of some specific part in the model.

When describing how data encryption can be modeled, we assume that no double encryption is needed for distinct parts of deployment models. We do not distinguish between read and write rights when discussing access control based on cryptographic key possession. Therefore, a participant with the required key is assumed to have full access rights on the corresponding entity. For efficiency reasons, we adopt symmetric encryption for ensuring the confidentiality of data.

B. Security Policies in Collaborative Deployment Models

An assumption that data is exchanged in a secure manner among the participants does not guarantee that all involved parties can be trusted. Therefore, security requirements are important even under the secure communication channels assumption. Security requirements we focus on are: (i) protection of data confidentiality in deployment models, and (ii) verification of data integrity and authenticity of deployment models. On the conceptual level, two distinct types of policies, namely *encryption policy* and *signing policy*, can be distinguished. The former is aimed to solve the confidentiality problem, whereas the latter targets integrity-related requirements. However, having a completely encrypted deployment model does not solve the confidentiality problem, since a party with limited rights will not be able to access the parts of the deployment model that were intended to remain accessible. A similar problem might arise for a signature of the complete packaged deployment model, e.g., in a form of an archive, since it will not be possible to check what exactly was changed unless all files are also signed separately as a part of the process. More specifically, if only the hash of an entire deployment model was signed, there will be no way to distinguish, which specific part of the model is invalid. Therefore, we need to model security policies on the level of atomic entities in deployment models to support collaborations similar to the scenario described in Section III.

Naturally, if only parts of deployment models are subject to confidentiality requirements, enforcement of encryption and signing policies must affect only respective entities. In our approach, an encryption policy attached to a certain entity of the deployment model signals that it has to be encrypted. In a similar manner, if a certain entity of the deployment model needs to be signed, a corresponding signing policy needs to be linked with it. In both cases, policies represent actual keys that are going to be used for encryption or signing. Since not all collaborations can rely on a centralized way to manage policies, the deployment model has to be transferred together with corresponding policies attached to its entities. The keys bound to policies, however, cannot be embedded, as deployment models will no longer remain suitable for sharing with all possible participants in a self-contained fashion. In such cases, either participants with proper access control rights can receive such models, or the models have to be split on a per-participant basis. Since not all scenarios favor participant-wise model splitting, a policy needs to be linked with a specific key in a decoupled manner to preserve self-containment of a deployment model. As a side effect of decoupling keys from policies, existing key distribution channels can be utilized independently from deployment model exchange channels.

For linking policies with particular keys, we need to maintain unique identifiers for every key involved in the collaboration. Since not all participants know each other, one simple solution is to compute a digest of the key and use it as an identifier or additionally combine it with several other parameters such as algorithm details, participant identifier, etc. Another option is to use identifiers, which include some partner-specific parts so that policies can be easily identified. Several important points have to be mentioned here. Linking the policy only with the unique key identifier is not enough for decryption since the modeler needs to know the algorithm details to perform decryption. Such information can be provided either as properties of a policy itself or be a part of the key exchange. Additionally, specifically for encryption there is no obvious way to distinguish if the policy was already applied and the data is in encrypted state when a deployment model is received. Although the data format after encryption will not be identical to the original entity's format, checking this difference for every modeled entity is not efficient. For this reason, a policy needs to have an attribute stating that it was applied. Due to the usage of symmetric encryption, generating a respective *decryption policy* is unnecessary as it is identical to the encryption policy.

Conversely, the verification of signing policies differs from the encryption process since private keys are used for signing and certificate chains of one or more certificates containing the public key and identity information are used for verification. As a result, there are two options: to follow the encryption approach and decouple certificates from policies, or, to embed certificates into policies to simplify the verification process. While certificates are meant for distribution, there is one caveat in the embedding of certificates approach, however. Certificates commonly have a validity period and verification must be able to deal with the cases when certificates embedded into policies are no longer valid. Since such verification is more an issue of a proper tooling, the certificates are embedded into policies.

Unlike file artifacts, e.g., software components or datasets, which are referenced from deployment models and supplied alongside with them, some sensitive information, e.g., model's properties, might be directly embedded into models. For instance, if user credentials for a third-party service have to be passed from one modeler to another and no other participant is allowed to see them, then these properties must be encrypted. Sometimes such properties also need to be verified, e.g., the Service Owner might want to check if the endpoint information for a third-party service was actually modeled by the Infrastructure Modeler. Therefore, an additional caveat one has to consider is that not only distinct artifacts, but also separate parts of artifacts might require encryption or signing. The corresponding artifact in this case has to store these properties with the modeled security requirements being enforced, e.g., encrypted or signed.

Hence, we need two more policy types: *encryption grouping policy* and *signing grouping policy*, which contain lists of properties within an artifact that have to be encrypted or signed, respectively. From the conceptual point of view, the discussed policies can be classified as *action* and *grouping* policies. The former includes policies representing an action, i.e., encryption or signing, whereas the latter identifies groups of entities that require the action. As a result, the corresponding grouping policies are linked with the desired action policies, i.e., with actual keys that will be applied to selected properties.

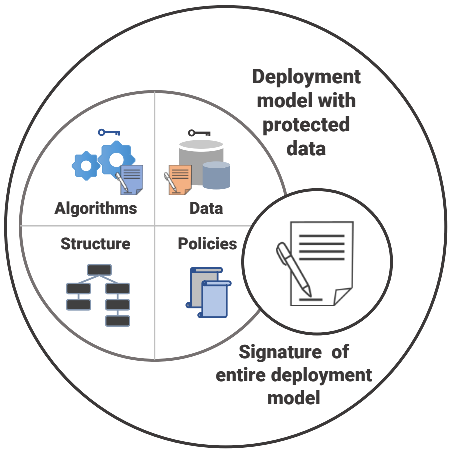
Deployment model with protected data
Algorithms
Data
Structure
Policies
Signature of entire deployment model

Figure 2. A conceptual model of the signed deployment model.

C. Integrity and Self-Containment of Deployment Models

When security policies are modeled and enforced, the resulting deployment model contains a combination of encrypted and signed artifacts and properties. Integrity check at this point allows to verify the state of modifications and authenticity of entities modeled by other participants. However, verification of the entire deployment model's integrity including modeled security policies and other attached metadata requires an additional signature on the level of deployment model.

For this purpose we adopt a technique analogous to signing of Java archives (JARs) [39]. Essentially, a packaged deployment model is some sort of an archive containing grouped artifacts. It is then possible to assume the presence of a meta file similar to manifest in JARs, which provides the list of all contents plus some additional information. In situations when such a *manifest* file does not exist, it can easily be generated by traversing the contents of a corresponding deployment model.

As both, integrity of the model's parts that are targeted by security requirements and integrity of the entire deployment model have to be considered, an enhanced packaging format is needed. The enhanced structure of a deployment model consists of its original content as well as the content's signature files. The latter is achieved via a combination of: (i) a manifest file with digests for every file, (ii) a signature file consisting of digests for every digest given in the manifest file plus the digest of the manifest file itself, and (iii) a signature block file consisting of a signature generated by the modeler and the certificate details. The resulting conceptual model is shown in Figure 2. To make a signed deployment model distinguishable from regular deployment models, the signature has to be generated in a standardized fashion, e.g., it can be stored in a predefined folder inside the package or entire deployment models can be archived along with the generated signature information.

One important issue is that, technically, there is no fixed concept of a deployment model in collaboration. Since parts of cloud applications might be exchanged separately or merged together, the definition of the exchanged deployment model is changing throughout the process. Thus, it is mandatory to preserve the self-containment of modeled security requirements on the level of atomic entities. Firstly, security policies are always included into the deployment model since they are tightly-coupled with target entities. With respect to actual

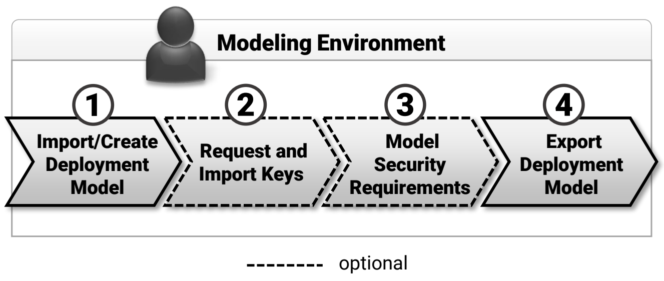
Modeling Environment
1 Import/Create Deployment Model
2 Request and Import Keys
3 Model Security Requirements
4 Export Deployment Model
-------- optional

Figure 3. Actions of a collaborating participant.

entities, the problem is trivial in case of encryption since locations of files or properties remain unchanged and only their state changes. In other words, whether the encrypted entity is exported from or imported into the modeling environment, the information about encryption is always available. Conversely, signatures of modeled entities have to be created as separate files since embedding them might not always work. For instance, embedding a signature into the application's source code might result in an incorrect behavior at runtime. This leads to a requirement of generating and storing signatures in a self-contained manner when signing policies enforcement happens.

In contrast, the signature of an entire deployment model reflects a snapshot of its state at a particular point in time, e.g., when the deployment model was packaged by a certain participant. Semantically, this signature does not mean that all content of the deployment model belongs to a signing party, but only captures the state of the deployment model at export time. In our approach, we use this external signature only for integrity verification at import time, but do not explicitly store it if verification was successful. However, if stored in a centralized or decentralized manner, this type of signature might form an expressive log of all export states, which can later be utilized for audit and compliance checking purposes.

D. Enforcement of Security Policies

As participants of a collaborative development process might not know all involved parties, every side has to maintain a set of permissions for known participants, e.g., in a form similar to the access matrix model [24]. In our case, permissions have to reflect, which policies are available to which participant and are, therefore, used for export and distribution of keys. One caveat is that in long sequences of steps there will be cases when a party does not know which rights with respect to the specific key have to be defined for some of the involved parties. The rules in such collaborations rely on various types of agreements, such as SLAs, which define the lists of trusted parties. Hence, we handle only explicitly mentioned access rights defined by participants and forbid transitive trust [40] propagation.

To enforce security policies in collaborations, participants have to follow a set of actions shown in Figure 3. A new or existing deployment model can be imported into the participant's modeling environment. Signatures are verified for an existing deployment model before import. An entire model's signature is verified first and if verification is successful, all signed entities are verified next. If certificate chains are embedded, all certificates must be valid. The import is aborted in case some signatures or certificates are invalid. A participant might request

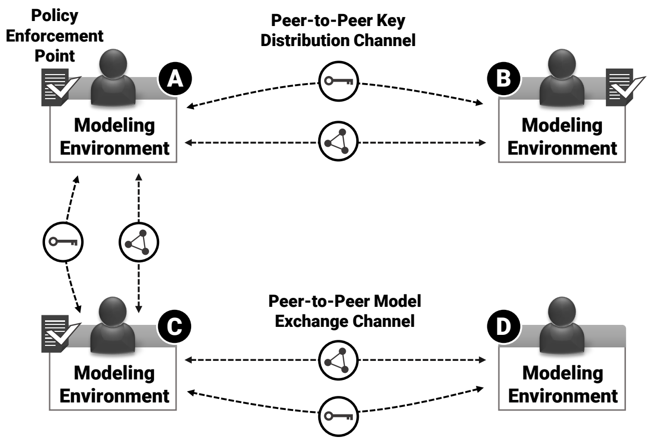

Figure 4. Model and key exchange in collaborations.

keys needed for encrypted entities and if access is granted by the key owner, keys can be imported into the modeling environment and used for decryption. The policy enforcement at export time happens transparently for participants as entities always get encrypted if the respective keys are present. Since decryption is only possible when the key is available, the encryption at export is ensured by the modeling environment.

Afterwards, participants can model additional security requirements and export a modified deployment model. One issue related to signatures and mutual modifications of the same entity is whether to keep the obsolete signature information. Since the original content of the entity has to be modified, we consider it being a new entity, which can be modeled separately eliminating the problem of handling several signatures altogether. At export time, all modeled requirements are enforced with respect to the keys available in the modeling environment. The decrypted data get encrypted again, in case the corresponding key is present and the entity was decrypted previously. Only signatures modeled by the participant who performs the export are generated. All entities that were signed by others remain in a self-contained state after import and thus are exported in a regular fashion. One important point here is that all enforced signing policies have to be verified before export. In case a violation is found, the export can no longer proceed in the regular way.

Generated signatures must be linked with corresponding modeling constructs. For instance, for every signed file the corresponding signature files must be added as additional linked references, e.g., following a predefined name format "filename#sigtype.sig". Signing properties requires a slightly different approach. Since properties are parts of artifacts and are subject to certain policies, their signatures have to be grouped with respect to the policy. This results in generation of the combined signature file and linking it with the artifact that holds the signed properties. Signature of this file is, again, generated similar to JAR files signing, but in this case the generated artifact contains the details about signed properties.

Figure 4 shows communication infrastructure for collaboration described in Section III. As key distribution is decoupled from the model exchange, two peer-to-peer channel types are distinguished. Generally, not all participants need to communicate with each other. For example, in an outsourcing scenario, a contractor might grant rights to the ordering party based on the contract rules and does not need to communicate with others. Therefore, access permissions of the ordering party have to also reflect access rules for the part of the deployment model provided by the contractor. The access to encrypted data is inquired by requesting a key using the corresponding policy identifier. Without having a centralized Policy Enforcement Point (PEP) [41], [42], every participant's modeling environment acts as a separate PEP, which regulates access control permissions based on inter-participant agreements. Participants are responsible for maintaining proper access control permissions including transitive cases.

V. Standards-based Secure Collaborative Development of Deployment Models

In this section we discuss the specifics of collaborative development of deployment models using TOSCA. We first start with a basic overview of TOSCA concepts and proceed with an analysis of which modeling constructs in TOSCA might require protection. As a next step, we describe how our concepts can be applied to this cloud standard and present a stepwise example demonstrating how collaborative TOSCA development process might look like.

A. *TOSCA Application Model*

TOSCA [12] is a cloud application modeling standard that allows to automate the deployment and management of applications. The structure of a TOSCA application is characterized by descriptions of its components with corresponding connectivity information, modeled as a directed, attributed, and not necessarily connected graph. In TOSCA terminology the entire application model is called a *Service Template*, whereas the connectivity information is a subpart of it and referred to as a *Topology Template*. The management information in TOSCA terms is called *Management Plans*. This information is necessary for execution and management of applications throughout their lifecycle and can be represented, e.g., as BPEL [20] or BPMN [21] models. A simplified TOSCA topology of a Python cloud service [5] is shown in Figure 5. It consists of several *nodes* representing software components that are connected with directed edges describing the *relationships* among them.

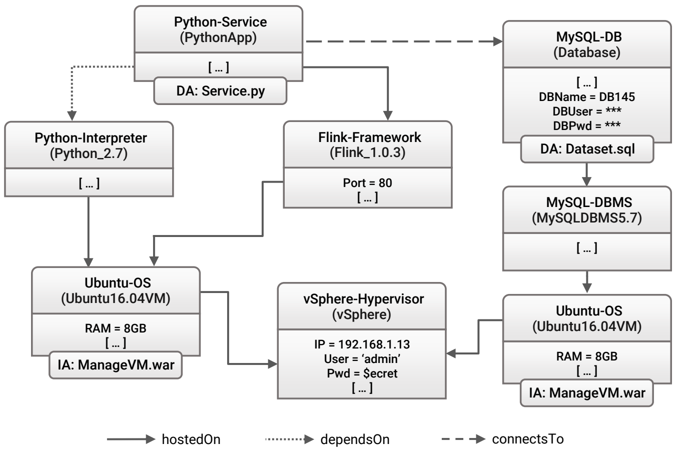

Figure 5. A simplified TOSCA model of a cloud application.

TOSCA differentiates between *entity types* and *entity templates*, where the term entity might refer to distinct TOSCA entities such as nodes, relationships, artifacts, or policies. Such separation eases reusing modeled TOSCA entities, since the semantics is always defined in the corresponding type. For instance, the node representing a vSphere hypervisor in Figure 5 is a template *of a certain type*, or, more specifically, a certain *Node Type*. The term "vSphere" written in braces in Figure 5 is the name of this node type. It describes a generic setup of a vSphere virtualization platform and defines all required configuration properties. Correspondingly, the term "vSphere-Hypervisor" written without braces represents a particular instance of the "vSphere" *Node Type* and in TOSCA terms is referred to as a *Node Template*. Apart from defining common properties, any Node Type might provide definitions of interface operations required for managing its instances. For example, a virtual machine node might need to implement management operations such as "start", "stop", and "restart" that allow controlling the state of the virtual machine. Implementations of the operations can be provided in various ways, e.g., in a form of Java web applications or shell scripts.

For deployment and management of the cloud service, all required artifacts have to be modeled, e.g., the application files and implementations of management interface operations. The artifact entity in TOSCA can be of two types, namely deployment artifacts (DA) and implementation artifacts (IA). The former defines an executable required for materialization of a node instance. The latter is a representation of an executable, which implements a certain interface management operation.

One of the main goals of deployment models is to make cloud applications portable and reusable. For this reason TOSCA introduces a self-contained packaging format called Cloud Service Archive (CSAR). Essentially, it is an archive containing all application-related data necessary for automated deployment and management, including, e.g., the model definitions, artifact files, policies and other metadata. In addition, it contains a TOSCA.meta file, which describes files in the archive similarly to a manifest file in JARs.

B. *Security Requirements for TOSCA Entities*

Several TOSCA modeling constructs can be associated with confidential information or be subject to integrity checks. Modeled application files, i.e., artifacts in TOSCA terms, are an obvious example. All artifacts are always modeled as Artifact Templates of desired Artifact Types in TOSCA, e.g., a Java web application artifact is a template of the Web application Archive (WAR) Artifact Type. While Artifact Types are generic entities, which do not store any sensitive data, Artifact Templates include actual application files. However, in the TOSCA specification there is no standard way to describe security requirements using policies for Artifact Templates. To provide such modeling capabilities, an extension to TOSCA is needed. Since properties are defined at the level of Types in TOSCA, e.g., Node Types, it is useful to have a mechanism allowing to enforce security requirements at this level. Semantically, this would mean that encryption or signing policies have to be applied to all Node Templates of a certain Node Type. TOSCA does not offer a standard way of attaching policies to specific properties, thus a proper way to enforce protection of properties at the level of Node Types is needed as well. In both cases, some form of extension to TOSCA is needed. In the next subsection, we demonstrate how such extensions can be made without introducing non-compliant changes to the standard.

C. TOSCA Policy Extensions

Due to the highly-extensible nature of TOSCA, we introduce several extension points to support the attachment of security policies to the aforementioned TOSCA entities. All policies are defined in a dedicated extension element, which belongs to a chosen entity. A simplified XML snippet in Figure 6 shows extension policies for Artifact Templates and Node Types from Figure 5. One important point here is to use a separate namespace for newly-introduced extensible elements, but for the sake of brevity we omit namespaces in demonstrated XML snippets. For Artifact Templates, a security policy is attached in a separate element directly to the Artifact Template. Essentially, an Artifact Template is a container grouping related files in a form of file references. We treat Artifact Templates as atomic entities meaning that policies are applied to all referenced files, which makes the semantics of modeled security requirements clearer. In cases when some referenced files need to be distributed without enforcement of policies, they can be modeled as separate Artifact Templates.

A combination of two policy types has to be defined in a dedicated extension element for encryption and signing of properties. A modeler has to specify a list of property names that must be encrypted or signed as well as to attach a corresponding action policy. These extensions allow participants to model desired security requirements for parts of the CSAR.

The introduced extensions, however, do not offer modeling capabilities for signing the entire CSAR. These two notions of integrity might contradict with each other, since a party having parts of the cloud service belonging to other parties is required to sign them as well. Hence, we separate the integrity check for a specific part of the model from an integrity check of the entire CSAR leaving the latter outside of TOSCA modeling.

Essentially, Policy Types and Templates representing action and grouping policies do not require significant modeling efforts. The Encryption Policy Type defines a key's hash value, an algorithm, and a key's size as its properties. In the corresponding Policy Template, these properties are populated using the respective key's data. Similarly, the Signing Policy Type has public key's hash and related certificate chain as its properties, filled in using the given key. Certificate chain can be embedded, e.g., in a form of a Privacy Enhanced Mail (PEM) encoded string in case of X509 [43] certificates. The only property defined in grouping policies is a space-separated list of property names. This Policy Type is abstract and is not directly bound to any specific entity. Therefore, the tooling is responsible for verification of the consistency of specified property names in attached policies.

D. Ensuring the Self-Containement Property of CSARs

Enforcement of modeled encryption requirements does not produce additional entities, but only modifies the existing ones. However, in case of enforcing signing requirements, new files are generated, i.e., the corresponding signature files for properties or files of Artifact Templates. These generated files have to be associated with the deployment model to ensure that the resulting CSAR contains all files and the deployment model represents these newly-generated files properly. Preservation of a CSAR's self-containment property after enforcement of modeled policies requires embedding the signature information for artifacts and properties into the corresponding entities. More specifically, when a signature for an artifact is created, it has to be placed along with other files referenced in the artifact. For the signature of properties, one artifact containing all properties' signatures needs to be generated and attached to the corresponding Node Template. Following this approach, modeled entities remain self-contained even in case they are being reused in other Service Templates.

```
<ArtifactTemplate name="Python-Service" ...>
  <Policies>
    <Policy applied="false" name="encryption"
         policyType="csar:EncryptionPolicyType"
         policyRef="csar1:c0e9a0e7".../>
  </Policies>
  <ArtifactReferences>
    <ArtifactReference ref=".../Service.py"/>
  </ArtifactReferences>
</ArtifactTemplate>
...
<NodeType name="vSphere" ...>
  <PropertiesDefinition>...</PropertiesDefinition>
  <Policies>
    <Policy ... name="signing" .../>
    <Policy ... name="signedprops" .../>
  </Policies>
</NodeType>
```

Figure 6. Example of TOSCA extension policies specification in XML.

E. Step by Step Collaborative Development Example in TOSCA

For a stepwise demonstration, we consider that the application topology shown in Figure 5 is being collaboratively developed following the same scenario and exchange sequence among the four participants as depicted in Figure 1. This example is not meant to be a thorough guideline for a given collaboration scenario, but rather it aims to demonstrate how certain security requirements can be modeled and enforced throughout the collective TOSCA-based model development.

As discussed in Section III, the four different participants involved in this scenario are: (i) an Application Owner who orders an application, (ii) a Dataset Provider who provides a private dataset, (iii) a Data Scientist who provides a proprietary algorithm, (iv) and an Infrastructure Modeler who defines the infrastructure. The output expected from this collaboration is a complete application topology, which is depicted in Figure 5. In the motivational scenario, essentially both the dataset provider and the data scientist are only responsible for single nodes in the topology that represent a private dataset and a proprietary algorithm, respectively. The participant-specific and infrastructure-specific nodes are combined by the infrastructure modeler in a complete and deployable application topology, which can afterwards be used by the application owner.

Prior to the beginning of the exchange sequence shown in Figure 1, an application deployment model does not exist yet, hence, one of the participants needs to instantiate it. Since the application owner is the main driver for this collaboration, we assume that the deployment model is first instantiated by the application owner. For the sake of brevity, we omit the discussion on how an empty deployment model is instantiated.

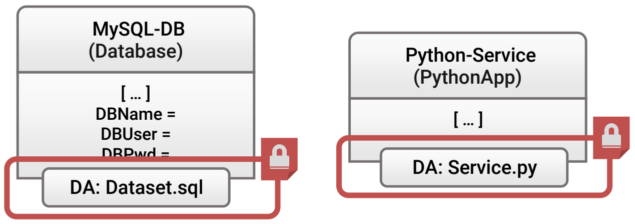

Figure 7. Application topology consisting of two decoupled Node Templates that represent the dataset and algorithm with encryption requirements.

In the *first part* of this collaboration, the dataset provider enriches the deployment model with the dataset-specific information and sends it back to the application owner, who then passes this part of the model to the infrastructure modeler. Since the dataset is private and must only be accessed by the application owner, the dataset provider must provide it in a secured form. Thus, to protect the dataset from unauthorized access, it must be encrypted. In addition, the dataset provider might want to sign the dataset to ease its integrity verification. The MySQL-DB Node Template shown in Figure 5 represents a MySQL database node that is hosted on the MySQL Database Management System (DBMS). In our example, the private dataset is provided as a SQL dump file and is attached as a DA to the node. Note that apart from the DA, the MySQL database node has properties such as a database name or access credentials, some of which might be set by different participants. For instance, a name for the database might be present in the dump file, therefore, the dataset provider might specify this property and, optionally, decide to protect it. We assume that only the artifact corresponding to the dataset has to be protected for the MySQL-DB node as depicted in Figure 7.

In the *second part* of collaboration, the data scientist models and enforces the encryption requirements for the Deployment Artifact of the Node Template, which represents the algorithm, and passes it to the infrastructure modeler. As the application owner is the only authorized user of the algorithm, the encryption requirements are similar to the ones of the dataset. At this point, the application topology consists of two decoupled Node Templates, namely *MySQL-DB* and *Python-Service* as shown in Figure 7. Since both steps are about enforcing encryption requirements for corresponding Deployment Artifacts, the process of modeling and enforcing these requirements is similar. A participant, i.e., the dataset provider or the data scientist, needs to: (i) create a TOSCA Policy Template of type “Encryption Policy”, (ii) attach a TOSCA Policy, i.e., an instance of the corresponding Policy Template, to the corresponding Artifact Template, (iii) optionally include other information, and (iv) export the resulting CSAR and pass it to the next participant. As encryption Policy Templates are uniquely associated with symmetric keys, the respective participants must generate keys first and afterwards generate corresponding Policy Templates. Moreover, the rules for sharing keys need to be stored and maintained, e.g., in the form of access control lists. In our example, though, the key sharing rules are trivial, as only the application owner is able to access the private contents of the application’s topology. Simplified examples of Encryption Policy Type and Template are shown in Figure 8. While the Encryption Policy Type only specifies the schema for properties, e.g., a property *keyHash* is of type *xsd:string*, the corresponding Policy Template references an actual key. Therefore, the properties in the Policy Template shown in Figure 8 contain the details about the key, i.e., its hash value and the algorithm name. Depending on the algorithm used and the desired level of details, the number of properties might be increased, e.g., to include the key size in bits. Moreover, the attachment of policies to respective Artifact Templates looks similar to the example shown in Figure 6. The modeled encryption requirements are enforced at export time, which results in encryption of the Artifact Template’s files using the referenced key and changing the value of the corresponding policy’s *applied* attribute to *true*.

```
<PolicyType name="EncryptionPolicyType" ...>
  <PropertiesDefinition>
    <properties>
      <key>keyHash</key>
      <type>xsd:string</type>
    </properties>
    <properties>
      <key>algorithm</key>
      <type>xsd:string</type>
    </properties>
    ...
  </PropertiesDefinition>
</PolicyType>

<PolicyTemplate name="4def0020237351e59...“
                type="EncryptionPolicyType"...>
  <Properties>
    <keyHash>4def0020237351e59...</keyHash>
    <algorithm>AES</algorithm>
    ...
  </Properties>
</PolicyTemplate>
```

Figure 8. Simplified example of TOSCA Encryption Policy Type and Template specification in XML.

```
<PolicyType name="SignedPropertiesPolicyType“ ...>
  ...
  <PropertiesDefinition>
    <properties>
      <key>propertyNames</key>
      <type>xsd:string</type>
    </properties>
  </PropertiesDefinition>
</PolicyType>

<PolicyTemplate name=“vSphereSignedProperties“
                type="SignedPropertiesPolicyType“...>
  <Properties>
    <propertyNames>IP User Pwd</propertyNames>
  </Properties>
</PolicyTemplate>
```

Figure 9. Example of TOSCA Signing Grouping Policy Type and its Template specification for vSphere-Hypervisor Node Template’s properties in XML.

In the *final part* of the collaboration scenario, the infrastructure modeler enriches the application topology with infrastructure-related nodes and sets up all the necessary properties required for deploying this application in an automated way. One important point here is that the model can be further refined in case additional requirements arise afterwards. This means that any participant might need to add or remove something

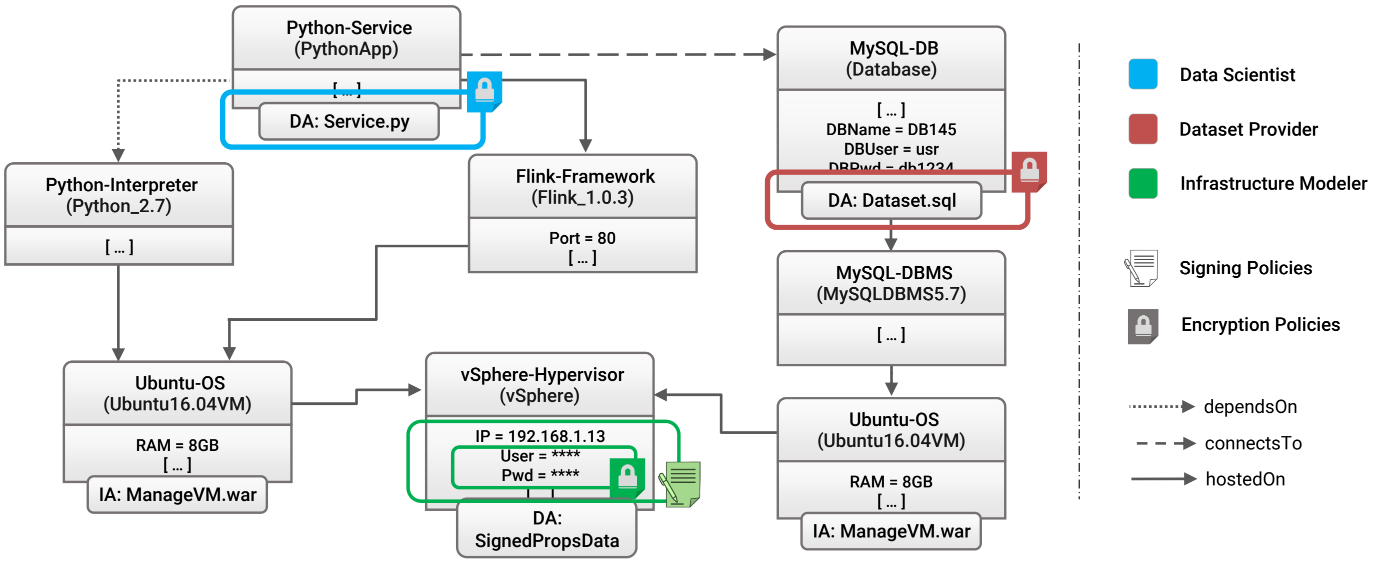

Figure 10. A complete deployment model with enforced security requirements.

from the model *after* the original exchange sequence. Therefore, all sensitive information, such as access credentials or endpoints, might require to be encrypted and signed by the infrastructure modeler to allow the application owner to ensure that the application will be deployed according to the SLAs.

To demonstrate how encryption and signing requirements can be combined, we assume that the infrastructure modeler must encrypt and sign the properties for the vSphere-Hypervisor Node Template, thus, ensuring that the application will be deployed to the correct location. For instance, the user credentials data must be encrypted and both, user credentials and IP address information need to be signed. With these requirements, the resulting set of policies related to the vSphere-Hypervisor Node Template consists of: (i) encryption policy, (ii) encryption grouping policy, (iii) signing policy, and (iv) signing grouping policy. This set of policies, as discussed in Section V-B, is attached to the vSphere Node Type to guarantee that all templates of this type will have these requirements enforced. The attachment of policies happens similar to the example Node Type policy specification snippet shown in Figure 6. Two attached policies represent respective keys, namely encryption and signing policies. The former is linked with a symmetric key used for encryption, and the latter is linked with a keypair. The purpose of grouping policies is to combine sets of properties targeted for the same action, i.e., encryption or signing. Figure 9 depicts a simple specification of a signing grouping Policy Type and the corresponding Policy Template for grouping the desired properties of vSphere-Hypervisor Node Template. Chosen property names are listed in the form of a space-separated list; therefore, the Policy Type contains only one property of type *xsd:string*. The Policy Template instantiates this property with a list of property names related to the vSphere-Hypervisor Node Template. Note that separate grouping policies are specified for encryption and signing requirements to allow distinguishing them and looking for overlaps. In cases when property lists overlap, the signing operations need to be performed twice, before and after encryption. This allows keeping the information about plain properties' hash values. This is one technical nuance that can simplify verification of properties at export time for parties authorized to access their values, since values can be checked prior to encryption. After enforcing modeled requirements, the infrastructure modeler possesses the complete version of the deployment model, which is now ready to be passed to the application owner. Figure 10 depicts the complete deployment model with enforced encryption and signing requirements from all involved participants. Generated signature files related to properties are attached to the model to keep it self-contained. Files that are required to be encrypted keep their original names, but are not accessible anymore without being decrypted.

VI. Prototype

In this section we describe the prototypical implementation of the presented concepts. The prototype is implemented during the course of SePiA.Pro [44], a project that tackles the issues of optimizing industrial automation processes in the context of Industry 4.0. Our prototype is based on the OpenTOSCA ecosystem, an open source toolchain for development and execution of TOSCA-compliant cloud applications. The OpenTOSCA ecosystem consists of such tools as Winery [45], [46], OpenTOSCA Container [15], and Vinothek [47]. Winery is a TOSCA-compliant modeling environment that supports graphical and text-based modeling of deployment models and offers additional functionalities including, e.g., GUI-based plan modeling capabilities. OpenTOSCA Container is an execution environment for TOSCA-based deployment models, which offers a rich set of functionalities including, e.g., deployment model execution, and a GUI for managing and monitoring the application instances that is based on Vinothek concepts.

A. *Overview*

Winery is the core part for implementation of the presented concepts, as most of them are coupled with the modeling process. Winery is a feature-rich modeling environment for TOSCA-compliant applications. The prototypical implementation adds extensions to both the backend and frontend of Winery, does not require further configuration, and can be used as an integral part of it. It is open source and available via Github [48]. As discussed in Section IV, in our approach every modeler is required to use a local Winery instance due to the absence of a centralized environment. Since keys are used for the enforcement of policies, Winery is extended to support key management functionalities. This includes storing, deletion, and generation of symmetric and asymmetric keys. For key storage we rely on the usage of Java's Java Cryptography Extension KeyStore (JCEKS) for storing all imported keys together. Assuming that Winery runs in a local and secure environment of a distinct party, registering keys in it is not problematic since keys never leave the modeler's environment. This approach, however, has to be extended to support multiple-owner Winery instances. Corresponding policies are generated based on the selected keys. For key distribution, a partner-wise specification of access control lists for security policies is added to Winery. Every participant needs to maintain a list of partner-specific rules negotiated by means of agreements in collaborations. Therefore, whenever a key is requested by some party, the key access rights are defined based on the local rules in Winery. All functionalities are accessible via the corresponding REST over HTTP endpoints.

The resulting prototype supports modeling of security requirements using Winery's built-in XML editors for the corresponding TOSCA entities. Winery stores the modeled TOSCA entities in a decoupled manner, which makes a concept of CSAR important only at export or import time. More specifically, at import time CSARs are disassembled into distinct entities to prevent storing duplicates. In a similar fashion, at export time CSARs are assembled from all the entities that are included in the chosen Service Template. This results in an issue that TOSCA meta files are not explicitly stored and are generated on-the-fly. As described in Section IV, the enforcement of modeled security policies at export time for selected TOSCA entities, e.g., Service Templates or Artifact Templates, happens in case specified keys are present in the system. Signatures for files in Artifact Templates are generated, and then referenced as additional files in the same Artifact Templates. If the files of Artifact Templates are subject to both, encryption and signing requirements, then the signatures of plain and encrypted files are attached. This allows verifying the integrity of target files to any involved participant independently of whether the files are encrypted or not. Signatures for properties are grouped as a separate Artifact Template of type "Signature", which is attached to the respective Node Template. This way of referencing newly-generated files ensures the self-containment property of deployment models. In addition, if policies were applied, the corresponding attribute is set to signify this fact. After encryption and signing requirements are enforced, an external signature of a CSAR is generated using a so-called *master key*, which is specified by the modeler for the whole environment as discussed in Section IV. The corresponding certificate or chain of certificates for this external signature is embedded into the CSAR and is used for verification at import time. The external signature is verified first at import time and is not stored if verification succeeds, since the CSAR is decomposed into distinct separately-stored entities. Furthermore, the import is aborted in case if the integrity check was not successful. In situations when keys requested by a modeler were provided, they can be imported and used for decryption of entities. Finally, only the participant who has an entire set of keys is able to decrypt and deploy the final application. Afterwards, the deployment and execution in the OpenTOSCA Container happens in a regular manner, since the CSAR contains the original deployment model.

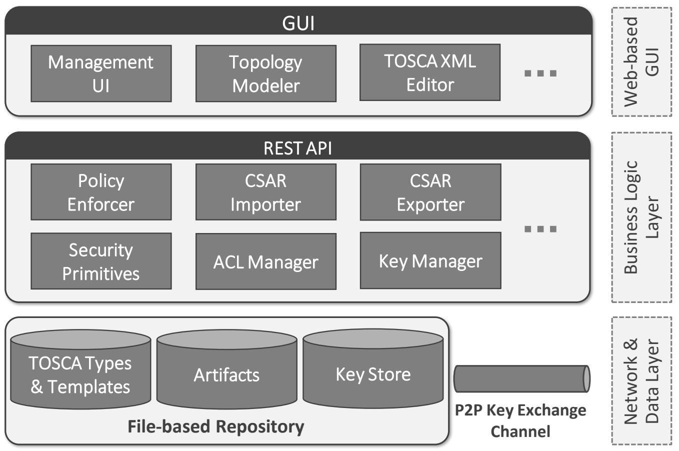

Figure 11. Architectural overview of the prototypical implementation.

To provide readers with a better view, in the following subsection we discuss an architecture of the prototype and explain several specific details related to the described concepts.

B. System Architecture

Figure 11 shows an architectural overview of Winery focusing on the components relevant to the prototype. At the topmost layer, Winery provides a Web-based graphical user interface (GUI) that consists of multiple sub-modules among which the following are of interest to the prototype: (i) The *Topology Modeler* is a sub-module that allows users to visually design various nodes and their interconnections as parts of a desired TOSCA deployment model. Using this component one is able to instantiate Node Types into Node Templates and populate their properties appropriately. Furthermore, this sub-module allows attaching various artifacts to these nodes and form a topology out of them using suitable Relationship Templates, all with simple UI operations such as Drag and Drop (c.f. Section V-A). On the other hand, (ii) the *TOSCA XML Editor* allows manually altering declarations of various TOSCA constructs. This editor is used, for example, to define the necessary Encryption and Signing Policy Types and attach instances of them to Node Types and Artifact Templates (c.f Section V-C). Finally, (iii) the *Management UI* allows users, among other things, to request the export of CSARs, which would enforce the corresponding policies, and the import of CSARs, which would trigger the process of verifying signed entities against the embedded certificate chains.

The GUI is merely an interface for the actual operations residing in the Winery backend, which are accessible via a REST API. In Figure 11, these operations are grouped into the Business Logic Layer, which contains an extensive set of sub-modules: (i) The *CSAR Exporter* is responsible for assembling the resulting portable CSAR out of the corresponding TOSCA templates and the artifacts attached to them. It is also responsible for enforcing the policies attached to these constructs. To this end, the CSAR Exporter utilizes another sub-module, namely (ii) the *Policy Enforcer*, which applies policies by

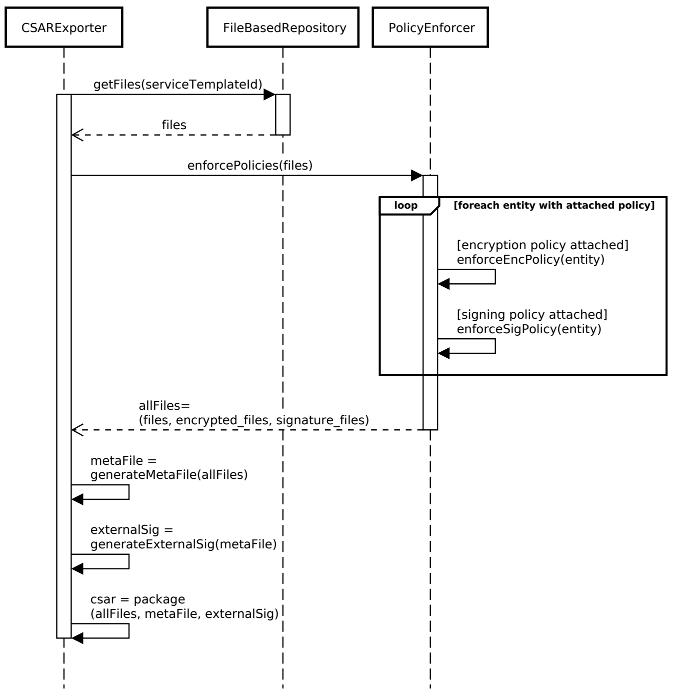
CSARExporter
FileBasedRepository
PolicyEnforcer
getFiles(serviceTemplateId)
files
enforcePolicies(files)
loop
[foreach entity with attached policy]
[encryption policy attached]
enforceEncPolicy(entity)
[signing policy attached]
enforceSigPolicy(entity)
allFiles=
(files, encrypted_files, signature_files)
metaFile =
generateMetaFile(allFiles)
externalSig =
generateExternalSig(metaFile)
csar = package
(allFiles, metaFile, externalSig)

Figure 12. A simplified process of securely exporting a CSAR.

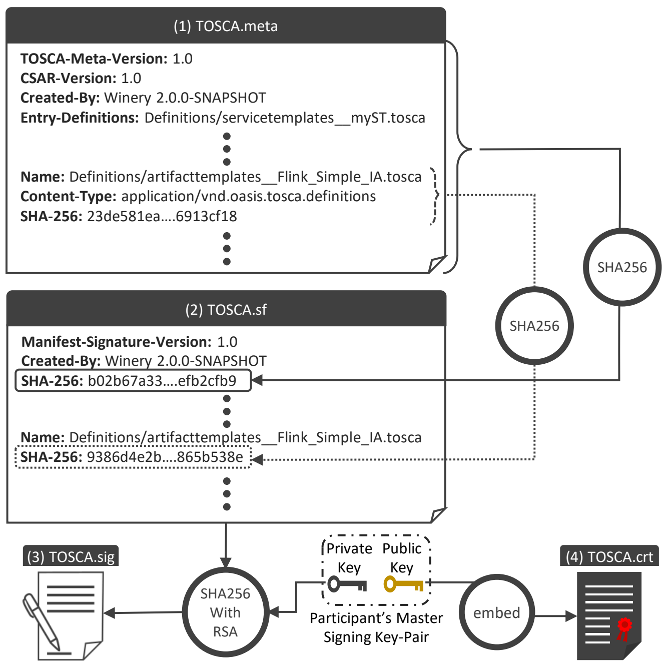
(1) TOSCA.meta
TOSCA-Meta-Version: 1.0
CSAR-Version: 1.0
Created-By: Winery 2.0.0-SNAPSHOT
Entry-Definitions: Definitions/servicetemplates__myST.tosca
Name: Definitions/artifacttemplates__Flink_Simple_IA.tosca
Content-Type: application/vnd.oasis.tosca.definitions
SHA-256: 23de581ea....6913cf18
SHA256
SHA256
(2) TOSCA.sf
Manifest-Signature-Version: 1.0
Created-By: Winery 2.0.0-SNAPSHOT
SHA-256: b02b67a33....efb2cfb9
Name: Definitions/artifacttemplates__Flink_Simple_IA.tosca
SHA-256: 9386d4e2b....865b538e
(3) TOSCA.sig
SHA256 With RSA
Private Key
Public Key
Participant's Master Signing Key-Pair
embed
(4) TOSCA.crt

Figure 13. The process of generating the external CSAR signature files at export time.

encrypting the properties and artifacts that have an Encryption Policy attached, and generating a signature for them if a Signing Policy is attached. On the other hand, the provisioning of cryptographic keys used for encryption and signing is done by (iii) the *Key Manager*. Moreover, (iv) the *CSAR Importer* is responsible for disassembling a CSAR file and importing its constituent components into the local repository. It is also responsible for verifying the integrity and authenticity of the imported CSAR by comparing the contents of the signed entities with the associated signatures and only performing the import if no violations are detected. As a supporter for the other components, (v) the *Security Primitives* sub-module utilizes the Bouncy Castle Crypto APIs [49] to provide various necessary cryptographic operations, such as symmetric and asymmetric encryption, key and certificate generation and content signing and verification. Finally, as mentioned earlier, (vi) the *Access Control List (ACL) Manager* specifies the rules that govern the relationships the local participant has with the other participants. This is necessary to determine who is authorized to obtain secret keys used to encrypt sensitive parts of the deployment model.

As discussed earlier, the actual exchange of keys happens at a lower level using a secure *P2P Channel*, which is not part of the actual prototype. Besides key exchange, the bottom layer of Figure 11 is responsible for the permanent storage of the modeled and imported TOSCA entities, as well as the associated implementation and deployment artifacts. The JCE KeyStore also resides at this layer. Storage of these objects happens using a *File-based Repository*.

In the following subsection, we see how the introduced sub-modules interact together to achieve the operations of exporting and importing a CSAR that utilizes the described TOSCA Policy Extension (c.f. Section V-C).

C. Secure Export of CSARs

As we have discussed earlier, the enforcement of the introduced policies happens during the creation of a CSAR, i.e., the CSAR export operation. Moreover, during this operation the CSAR's external signature is also generated. Figure 12 shows a simplified UML sequence diagram that explains how involved modules from the introduced system architecture interact together in order to perform the export operation: after receiving a request from the *Management UI* (not shown in this figure) the *CSAR Exporter* starts the process via traversing the TOSCA Topology specified by the Service Template and requesting pointers to the corresponding TOSCA definition files, as well as the files associated with them, from the *File-based Repository*. Afterwards, a request is sent to the *Policy Enforcer* to perform the enforcement. It does so by searching the set of TOSCA definition files for Node Types or Artifact Templates with attached policies, then encryption and signing policies are enforced as discussed earlier. To this end, the *Policy Enforcer* utilizes the functionalities provided by the *Security Primitives* sub-module (not shown in the figure). The result of policy enforcement is a changed set of files that has some encrypted content and/or additional signature files. In the next step, the *CSAR Exporter* populates the TOSCA.meta file by traversing the set of files and creating an entry for each of them. Finally, the external signature is generated starting from the TOSCA.meta file and the resulting CSAR file is created by packaging all files in a single compressed archive.

Figure 13 demonstrates the overall procedure of generating the external signature of the CSAR, which starts from processing the TOSCA.meta file and which, as mentioned earlier, follows the JAR signing process. The TOSCA.meta file is used to describe the contents of a CSAR and consists of: (i) a header section, which provides general information about the

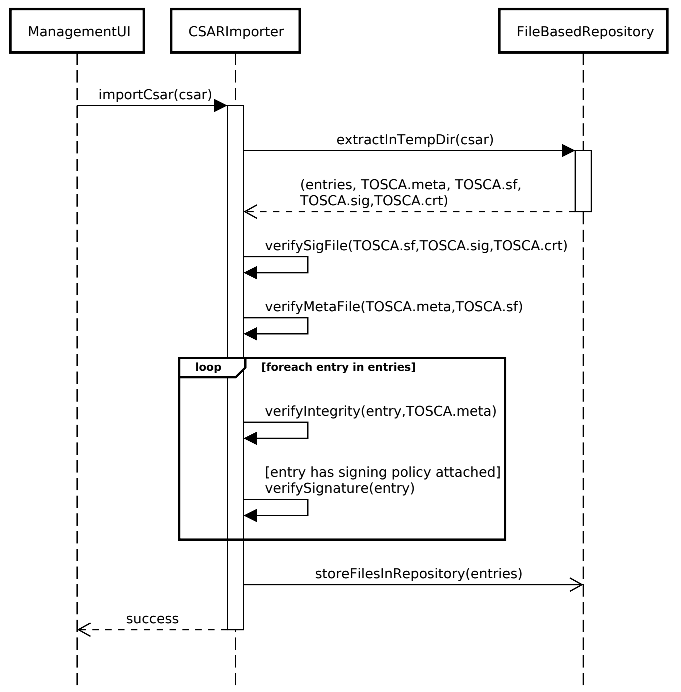
ManagementUI
CSARImporter
FileBasedRepository
importCsar(csar)
extractInTempDir(csar)
(entries, TOSCA.meta, TOSCA.sf, TOSCA.sig,TOSCA.crt)
verifySigFile(TOSCA.sf,TOSCA.sig,TOSCA.crt)
verifyMetaFile(TOSCA.meta,TOSCA.sf)
loop
[foreach entry in entries]
verifyIntegrity(entry,TOSCA.meta)
[entry has signing policy attached] verifySignature(entry)
storeFilesInRepository(entries)
success

Figure 14. A simplified process of securely importing a CSAR showing only the successful control flow.

deployment model such as who created it and the path of the TOSCA definition file associated with the Service Template, and (ii) a body section that lists all files contained in the CSAR along with their exact location and MIME file type. We enhance the body section of the TOSCA.meta file with a digest subsection for each contained file that helps in guaranteeing its integrity. In this prototype, we use the SHA-256 [50] function to calculate digests. In the next step, a signature file (TOSCA.sf) is generated using the TOSCA.meta file. While the purpose of the TOSCA.meta file is to guarantee the integrity of the CSAR, the purpose of TOSCA.sf file is guaranteeing the integrity of the TOSCA.meta file itself. To this end, the TOSCA.sf file has a header section that includes a digest of the whole TOSCA.meta file, and a body section with subsections corresponding to the ones in the TOSCA.meta file. However, in this case each subsection contains a digest of the whole matching subsection in the TOSCA.meta file. Furthermore, to ensure the overall integrity, the TOSCA.sf file is signed using the master signing private key of the participant who issued the CSAR export. As a result, a block signature file (TOSCA.sig) is generated. Finally, a certificate (TOSCA.crt), which contains the master signing public key of the participant, is also included in the archive to provide future importers with the possibility to verify the signature. The participant's master signing key-pair resides, like other keys, in the *Key Store* repository that can be accessed by other components via the *Key Manager* sub-module.

D. Secure Import of CSARs

The import of a protected CSAR into the modeling environment comprises two steps: (i) firstly, it must trigger verifications of CSAR's integrity and authenticity, and afterwards (ii) the regular task of storing constituent TOSCA entities and other artifacts in the local repository must be accomplished. This is depicted in Figure 14: upon a request from the *Management UI* the *CSAR Importer* module begins the process of importing the CSAR by asking the *File-based Repository* to create a temporary directory in which the archive content is extracted. As we have seen in the previous section, in addition to the regular entries, the contents of the CSAR include the TOSCA.meta file as well as the external signature files, i.e., TOSCA.sf, TOSCA.sig, and TOSCA.crt. The importer then performs a check to verify the integrity and authenticity of the signature file (TOSCA.sf) by performing the digital signature verification process using the TOSCA.sig and TOSCA.crt files. Next, the importer verifies the integrity of the TOSCA.meta file by comparing its overall digest as well as the digests of its subsections to the corresponding entries in the TOSCA.sf file. Afterwards, the regular CSAR entries are enumerated, and an integrity check is performed for each one of them by calculating its digest and comparing it to the corresponding digest listed in the TOSCA.meta file. A further authenticity check is performed if the entry is a Node Type, or an Artifact Template with an associated Signing Policy in which case the included signature is verified. If all previous validity checks pass, the contents of the CSAR are imported into the *File-based Repository* and the *Management UI* is notified about the success of the operation. Otherwise, the import is aborted, and a proper error message is returned to the *Management UI*. Similar to what we have seen in the export process, the *Security Primitives* sub-module is utilized by the *CSAR Importer* to perform required cryptographic tasks.

VII. Related Work

The problem of data protection in outsourcing and collaboration scenarios appears in works related to different domains. Multiple works attempt to tackle security-related problems using centralized approaches. Wang et al. [31] present a method for protecting the models in collaborative computer-aided design (CAD), which extends RBAC mechanism by adding notions of scheduling and value-adding activity to roles. Authors propose to selectively share data to prevent reverse engineering. However, no clear description how to enforce the proposed model is given. Cera et al. [51] introduce another RBAC-based data protection approach in collaborative design of 3D CAD models. Models are split into separate parts based on specified role-based security requirements to provide personalized views using a centralized access control mechanism. Li et al. [32] propose a security policy meta-model and the framework for securing big data using sticky policies concept. Policies are loosely-coupled with the data and the framework relies on a trusted party, which combines policy and key management functionalities, and enforces the access control. Huang et al. [52] introduce a set of measures allowing to protect patients data in portable electronic health records (EHRs). Authors propose a centralized system, which combines de-identification, encryption, and digital signatures as means to achieve data privacy. Li et al. [36] describe an approach based on the Attribute-Based Encryption that helps to protect patient's personal health records in the cloud. In this approach, data is encrypted using keys that are generated based on the owner-selected set of attributes and then published to the cloud. Users can only access the data in case they possess corresponding attributes, e.g., profession or organization. More specifically, users are divided into several security domains and the attributes for these domains are managed by corresponding attribute authorities. Decryption keys, therefore, can be generated independently from data owners by the respective attribute authorities. Essentially, the described approaches are domain-specific and rely on a trusted party, which makes it problematic to apply them to the problem of protecting collaborative development of deployment models discussed in Section III.

A number of approaches focus on the data encryption in outsourcing scenarios. Miklau and Suciu [53] introduce an encryption framework for protecting XML data published on the Internet. Contributions of the work include a policy specification language available in the form of queries and a model allowing to encrypt single XML documents. Access control is enforced based on key possession. Vimercati and Foresti [54] discuss fragmentation-based approaches for protecting outsourced relational data. The authors elaborate on several techniques allowing to split up the given data based on some constraints into one or more fragments and store them in a way to protect confidentiality and privacy. For instance, data can be split into two parts and stored on non-communicating servers. Whenever constraints cannot be satisfied for some attributes, the encryption is used. In the follow-up work, Vimercati et al. [55] present a way to enforce selective access control using the cryptography-based policies. Authors propose to use key derivation mechanisms to simplify the distribution of keys.

To the best of our knowledge, there is no approach that successfully tackles our problem of deployment models protection in collaborative application development scenarios. While the described related work provides useful insights on the usage of well-established concepts such as the usage of sticky policies and access control enforcement by means of key possession, it is not applicable to our problem due to several reasons. One of the main issues is that most of the discussed approaches rely on the idea of a trusted party, which can regulate the access control. While it is desirable to have a central authority, in many cases it is unrealistic, leading to a need for peer-to-peer solutions. Moreover, having focus only on separate security requirements like encryption or strong assumptions about the underlying data makes these approaches not suitable for the described problems.

VIII. Conclusion and Future Work

In this work, we showed how security requirements can be modeled and enforced in collaborative development of deployment models. We identified sensitive parts in deployment models and proposed a method, which allows protecting them based on a combination of existing research work. For validation of the presented concepts, we applied them to TOSCA, an existing OASIS standard, which specifies a provider-agnostic cloud modeling language. The resulting prototypical implementation is based on Winery, the modeling environment, which is a part of the OpenTOSCA ecosystem, an open source collection of applications supporting TOSCA.

One issue in our approach that has to be optimized is the way keys are distributed. We rely on the fact that not all participants need to exchange keys, which, however, does not solve the scalability problem. If N keys were used for encryption, eventually all of them will be used in key distribution. For improving the efficiency, the key derivation techniques, e.g., described by Vimercati et al. [55], can be used to reduce the number of keys that need to be exchanged. Another problem for future work is the generalization of the adversary model. Since deployment models can be intentionally corrupted by an adversary, there is a strong need to store the provenance information, which describes deployment model's states at every export with respect to a certain collaboration. Having such provenance information stored in some accessible form makes it possible to track the entire collaboration history with all the deployment model states that were existing throughout the process. For this reason, one might employ a centralized system, which will also simplify the policy enforcement and key distribution processes, or store the provenance in a decentralized fashion, e.g., by utilizing the blockchain technology [56]. In addition, we plan to analyze the existing deployment technologies and identify the optimal mapping constructs for enabling our approach to them. For instance, not every technology allows using policies as explicit modeling constructs, which requires attaching security and privacy requirements using other constructs, e.g., by means of annotations or descriptors. Moreover, the modeling-related tooling varies for different deployment technologies, which imposes additional hurdles on enforcing modeled requirements for the chosen technology.

There are several issues in our prototype that can be optimized. Currently, the grouping policies in the prototype cannot be linked with the corresponding encryption or signing policies, hence it is not possible to model several groups of properties signed or encrypted using different keys. In future work, we plan to extend the implementation to support more complex scenarios and increase the overall usability of the modeling process using Winery and its respective components. Finally, there is a pitfall for cases when files are modeled in the form of references, e.g., if they reside on a remote server. Encrypting and signing such files completely changes the verification semantics as only the references are checked. This is not safe since the actual content behind the reference can be changed multiple times by the data owner without changing the reference itself. Moreover, the usage of references invalidates the self-containment property of deployment models. In future work, referenced files need to be materialized at export time, which solves this problem and preserves deployment models in a self-contained state.

In our previous work [7], we tackled the issue of guaranteeing accountability of collaborative development of deployment models by registering fingerprints of the various states a deployment model goes through while being developed in the blockchain. These fingerprints serve as a guarantee for the integrity and authenticity of the model when being passed from one participant to the next. Furthermore, we stored the actual contents of these models in a decentralized file storage addressable using the aforementioned fingerprints. This creates an immutable history of verifiable deployment model states. A problem with that approach is the public nature of the decentralized storage where the contents of deployment models are stored. This makes the sensitive information that might exist within these models accessible to the public. The obvious solution to this is encryption, thus in a future work, we plan to combine both approaches in order to have a collaboration process, which guarantees both security and accountability.

Acknowledgment

This work is partially funded by the BMWi project *SePiA.Pro* (01MD16013F) and the European Union's Horizon 2020 research and innovation project *RADON* (825040).

References

[1] V. Yussupov, M. Falkenthal, O. Kopp, F. Leymann, and M. Zimmermann, “Secure Collaborative Development of Cloud Application Deployment Models,” in Proceedings of The 12th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2018). Xpert Publishing Services, September 2018, pp. 48–57.

[2] M. Hermann, T. Pentek, and B. Otto, “Design principles for industrie 4.0 scenarios,” in 2016 49th Hawaii International Conference on System Sciences (HICSS). IEEE, 2016, pp. 3928–3937.

[3] P. M. Mell and T. Grance, “Sp 800-145. the NIST definition of cloud computing,” Gaithersburg, MD, United States, Tech. Rep., 2011.

[4] L. Atzori, A. Iera, and G. Morabito, “The internet of things: A survey,” Computer networks, vol. 54, no. 15, 2010, pp. 2787–2805.

[5] M. Zimmermann, U. Breitenbücher, M. Falkenthal, F. Leymann, and K. Saatkamp, “Standards-based function shipping – how to use tosca for shipping and executing data analytics software in remote manufacturing environments,” in Proceedings of the 2017 IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC 2017). IEEE Computer Society, 2017, pp. 50–60.

[6] U. Breitenbücher et al., “Combining declarative and imperative cloud application provisioning based on tosca,” in Proceedings of the IEEE International Conference on Cloud Engineering (IEEE IC2E 2014). IEEE Computer Society, March 2014, pp. 87–96.

[7] G. Falazi, U. Breitenbücher, M. Falkenthal, L. Harzenetter, F. Leymann, and V. Yussupov, “Blockchain-based Collaborative Development of Application Deployment Models,” in On the Move to Meaningful Internet Systems. OTM 2018 Conferences (CoopIS 2018), ser. Lecture Notes in Computer Science, vol. 11229. Springer, 2018, pp. 40–60.

[8] T. Kvan, “Collaborative design: what is it?” Automation in construction, vol. 9, no. 4, 2000, pp. 409–415.

[9] G. Karjoth, M. Schunter, and M. Waidner, “Platform for enterprise privacy practices: Privacy-enabled management of customer data,” in International Workshop on Privacy Enhancing Technologies. Springer, 2002, pp. 69–84.

[10] W. Bagga and R. Molva, “Policy-based cryptography and applications,” in International Conference on Financial Cryptography and Data Security. Springer, 2005, pp. 72–87.

[11] A. Harrington and C. Jensen, “Cryptographic access control in a distributed file system,” in Proceedings of the 8th ACM symposium on Access control models and technologies. ACM, 2003, pp. 158–165.

[12] OASIS, Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0, Organization for the Advancement of Structured Information Standards (OASIS), 2013.

[13] OASIS, Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0, Organization for the Advancement of Structured Information Standards (OASIS), 2013.

[14] A. Bergmayr et al., “A systematic review of cloud modeling languages,” ACM Comput. Surv., vol. 51, no. 1, Feb. 2018, pp. 22:1–22:38.

[15] T. Binz et al., “Opentosca – a runtime for tosca-based cloud applications,” in Service-Oriented Computing. Berlin, Heidelberg: Springer, 2013, pp. 692–695.

[16] C. Endres et al., “Declarative vs. imperative: Two modeling patterns for the automated deployment of applications,” in Proceedings of the 9th International Conference on Pervasive Patterns and Applications. Xpert Publishing Services (XPS), Feb. 2017, pp. 22–27.

[17] U. Breitenbücher, K. Képes, F. Leymann, and M. Wurster, “Declarative vs. imperative: How to model the automated deployment of iot applications?” in Proceedings of the 11th Advanced Summer School on Service Oriented Computing. IBM Research Division, Nov. 2017, pp. 18–27.

[18] Chef. [Online]. Available: https://www.chef.io/ [retrieved: July, 2018]

[19] Juju. [Online]. Available: https://jujucharms.com/ [retrieved: July, 2018]

[20] OASIS, Web Services Business Process Execution Language (WS-BPEL) Version 2.0, Organization for the Advancement of Structured Information Standards (OASIS), 2007.

[21] OMG, Business Process Model and Notation (BPMN) Version 2.0, Object Management Group (OMG), 2011.

[22] R. Boutaba and I. Aib, “Policy-based management: A historical perspective,” Journal of Network and Systems Management, vol. 15, no. 4, Dec 2007, pp. 447–480.

[23] R. Wies, “Using a classification of management policies for policy specification and policy transformation,” in Integrated Network Management IV. Springer, 1995, pp. 44–56.

[24] P. Samarati and S. C. di Vimercati, “Access control: Policies, models, and mechanisms,” in International School on Foundations of Security Analysis and Design. Springer, 2000, pp. 137–196.

[25] S. Pearson and M. Casassa-Mont, “Sticky policies: An approach for managing privacy across multiple parties,” Computer, vol. 44, no. 9, 2011, pp. 60–68.

[26] Q. Tang, On Using Encryption Techniques to Enhance Sticky Policies Enforcement, ser. CTIT Technical Report Series. Netherlands: Centre for Telematics and Information Technology (CTIT), 2008, no. WoTUG-31/TR-CTIT-08-64.

[27] B. W. Lampson, “Protection,” ACM SIGOPS Operating Systems Review, vol. 8, no. 1, 1974, pp. 18–24.

[28] M. Falkenthal et al., “Opentosca for the 4th industrial revolution: Automating the provisioning of analytics tools based on apache flink,” in Proceedings of the 6th International Conference on the Internet of Things, ser. IoT’16. New York, NY, USA: ACM, 2016, pp. 179–180.

[29] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, TOSCA: Portable Automated Deployment and Management of Cloud Applications. New York, NY: Springer New York, 2014, pp. 527–549.

[30] M. Zimmermann, F. W. Baumann, M. Falkenthal, F. Leymann, and U. Odefey, “Automating the provisioning and integration of analytics tools with data resources in industrial environments using opentosca,” in Proceedings of the 2017 IEEE 21st International Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOCW 2017). IEEE Computer Society, Oct. 2017, pp. 3–7.

[31] Y. Wang, P. N. Ajoku, J. C. Brustoloni, and B. O. Nnaji, “Intellectual property protection in collaborative design through lean information modeling and sharing,” Journal of computing and information science in engineering, vol. 6, no. 2, 2006, pp. 149–159.

[32] S. Li, T. Zhang, J. Gao, and Y. Park, “A sticky policy framework for big data security,” in 2015 IEEE First International Conference on Big Data Computing Service and Applications (BigDataService). IEEE, 2015, pp. 130–137.

[33] T. Waizenegger et al., “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, Sep. 2013, pp. 360–376.

[34] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and M. Wieland, “Policy-aware provisioning of cloud applications,” in Proceedings of the 7th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE). Xpert Publishing Services (XPS), 2013, pp. 86–95.

[35] A. A. E. Kalam et al., “Organization based access control,” in IEEE 4th International Workshop on Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE, 2003, pp. 120–131.

[36] M. Li, S. Yu, K. Ren, and W. Lou, “Securing personal health records in cloud computing: Patient-centric and fine-grained data access control in multi-owner settings,” in International conference on security and privacy in communication systems. Springer, 2010, pp. 89–106.

[37] F. Li, B. Luo, and P. Liu, “Secure information aggregation for smart grids using homomorphic encryption,” in 2010 First IEEE International Conference on Smart Grid Communications (SmartGridComm). IEEE, 2010, pp. 327–332.

[38] S. Ruj and A. Nayak, “A decentralized security framework for data aggregation and access control in smart grids,” IEEE transactions on smart grid, vol. 4, no. 1, 2013, pp. 196–205.

[39] Oracle. Understanding signing and verification. [Online]. Available: https://docs.oracle.com/javase/tutorial/deployment/jar/intro.html [retrieved: July, 2018]

[40] J. Huang and M. S. Fox, “An ontology of trust: formal semantics and transitivity,” in Proceedings of the 8th international conference on electronic commerce: The new e-commerce: innovations for conquering current barriers, obstacles and limitations to conducting successful business on the internet. ACM, 2006, pp. 259–270.

[41] M. Falkenthal et al., "Requirements and Enforcement Points for Policies in Industrial Data Sharing Scenarios," in Proceedings of the 11th Advanced Summer School on Service Oriented Computing. IBM Research Division, 2017, pp. 28–40.

[42] F. W. Baumann, U. Breitenbücher, M. Falkenthal, G. Grünert, and S. Hudert, "Industrial data sharing with data access policy," in Cooperative Design, Visualization, and Engineering. Springer International Publishing, 2017, pp. 215–219.

[43] M. Cooper et al. Internet X.509 Public Key Infrastructure: Certification Path Building. [Online]. Available: https://tools.ietf.org/html/rfc4158 [retrieved: July, 2018]

[44] SePiA.Pro. [Online]. Available: http://projekt-sepiapro.de/en/ [retrieved: May, 2019]

[45] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery – A Modeling Tool for TOSCA-based Cloud Applications," in Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013). Springer, Dec. 2013, pp. 700–704.

[46] Winery. [Online]. Available: https://eclipse.github.io/winery/ [retrieved: July, 2018]

[47] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, "Vinothek – a self-service portal for tosca," in Proceedings of the 6th Central-European Workshop on Services and their Composition (ZEUS 2014). CEUR-WS.org, Feb. 2014, Demonstration, pp. 69–72.

[48] Prototypical implementation of the secure csar concepts. [Online]. Available: https://github.com/OpenTOSCA/winery/releases/tag/paper%2Fvy-secure-csar [retrieved: July, 2018]

[49] Bouncy Castle Crypto APIs. [Online]. Available: http://bouncycastle.org/ [retrieved: Dec., 2018]

[50] W. Penard and T. van Werkhoven, On the Secure Hash Algorithm family, 2008, ch. 1, pp. 1–18.

[51] C. D. Cera, T. Kim, J. Han, and W. C. Regli, "Role-based viewing envelopes for information protection in collaborative modeling," Computer-Aided Design, vol. 36, no. 9, 2004, pp. 873–886.

[52] L.-C. Huang, H.-C. Chu, C.-Y. Lien, C.-H. Hsiao, and T. Kao, "Privacy preservation and information security protection for patients' portable electronic health records," Computers in Biology and Medicine, vol. 39, no. 9, 2009, pp. 743–750.

[53] G. Miklau and D. Suciu, "Controlling access to published data using cryptography," in Proceedings of the 29th international conference on Very large data bases-Volume 29. VLDB Endowment, 2003, pp. 898–909.

[54] S. D. C. di Vimercati and S. Foresti, "Privacy of outsourced data," in IFIP PrimeLife International Summer School on Privacy and Identity Management for Life. Springer, 2009, pp. 174–187.

[55] S. D. C. di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Encryption policies for regulating access to outsourced data," ACM Transactions on Database Systems (TODS), vol. 35, no. 2, 2010, p. 12.

[56] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: http://bitcoin.org/bitcoin.pdf [retrieved: July, 2018]