CORRECTNESS CONTROL OF PRIVACY-PRESERVING
DECENTRALIZED CONSTRAINED WEB SERVICE COMPOSITIONS
THROUGH AUDITING

[Leila Bahri, Barbara Carminati, Elena Ferrari, Ngoc Hong Tran]
[University of Insubria, Italy]
[firstname.lastname@uninsubria.it]

Abstract
Deploying web service (WS) compositions following a choreography has been attracting considerable interest as it brings the advantages of decentralization in terms of performance and privacy preservation. However, in the absence of a central entity, the choreography model suffers from lack of control regarding the selection of appropriate WSes. This is further aggravated in the case of constrained compositions, where service selection is bounded by a set of requirements imposed either by the user requesting the composite service or by the WSes involved in the composition. In this scenario, it is essential to provide assurance that all selected services satisfied the posed constraints. This requirement might constitute a serious limitation to the acceptance of the choreography model, compared to the central orchestration driven one, as entities might collude to select each other even if they are not enough qualified for the task. To save the benefits of the choreography model all while ensuring control against collusion, we suggest an audit-based correctness control combined with a reputation system. Our approach detects anomalous behaviors, it deters entities from collusion with a punish-based reputation system, and it ensures the audit in a privacy preserving manner without subverting the performance of the choreography approach.

Keywords: Constrained Web Service Composition, Choreography, Audit-based control.

1. Introduction
Web service compositions (WSC) are mash-ups of different independent web services that, in accordance with an intended workflow, achieve a desired more complex service. These compositions are achieved in one of two majorly adopted approaches. The first approach, known as orchestration, dictates the deployment of a mediator that controls the formation of the composition by invoking selected services according to a desired workflow. The second approach, referred to as choreography, suggests a decentralized management whereby web service providers collaborate in a dynamic manner to achieve the desired WSC. Given that the first approach requires the existence of a centralized management entity, that might represent a single point of failure and create scalability limitations, much more interest is being given to the more flexible and decentralized choreography approach [5]. In a choreography, the workflow of activities required to achieve the desired WSC is predefined and its execution starts by invoking the service that will perform the first activity. Upon successful termination, this service takes the duty of searching for the subsequent web service that has to perform the following activity as dictated by the workflow. The same process is repeated at each invoked service until achieving the last required activity. This choreography approach brings the benefits of decentralization as related to enhanced performance, non-reliance on single point of failure, flexibility of execution, etc. However, it also introduces new problems, especially in the scenario of constrained web service compositions (CWSC).

In fact, WSC requests are often associated with constraints on the activities they demand. These constraints are requirements that can be imposed either by end-users, regarding their personal service preferences, or by web service providers, regarding some quality of service (QoS) criteria that they might require in potential collaborators [17]. The scenario of CWSCs has attracted considerable attention from researchers considering both the orchestration [19] and the choreography deployment approaches [2], [13]. The crucial issue being addressed is the selection of the appropriate web service to be invoked for a given activity in the workflow such that the composite service satisfies all constraints and gives the best overall performance. Under the orchestration approach, the reliance on a trusted central management entity cuts down the problem to the optimization of the selection process, without posing further challenges on the management and the control of this process. The choreography approach, on the other hand, in addition to issues related to the achievement of the selection process in a decentralized manner, raises more serious challenges, especially as related to the management and control of the established service selection protocol.

One of the inherent problems in decentralized architectures in general, and in choreographed CWSCs in particular, is related to the management of risks of collusion. In an ideal trusted environment, all peers are assumed to be trustworthy and expected to behave correctly and in accordance with the protocol managing the selection process without cheating. However, real environments dictate that
such trust cannot be passively and blindly assumed [3]. In fact, peers might cheat and collude with other cheaters to invoke their preferred web services even if they do not satisfy the required constraints. This might introduce serious limitations to the success and adoption of the choreography approach for CWSCs. For this, we believe that there is need for a solution that saves the benefits of the choreography approach, all while bringing the advantages of the orchestration paradigm in terms of installing a unit of control.

To achieve this, we suggest, in this paper, the use of deterring a posteriori correctness control over deployed CWSCs in an open environment. That is, entities are allowed to choose how to behave (i.e., to be honest or to collude) without being interfered at execution time. However, they are aware that there is an auditing mechanism taking place to perform correctness control for all deployed CWSCs in the choreography based system, considering both end-user and provider requirements. The aim is to obtain a reward-punish environment by which dishonest entities are post-detected and disqualified accordingly, without subverting the decentralized spirit of a choreography. Indeed, auditing does not compromise the flexibility of the process on top of which it operates (i.e., that it controls), as it does not interfere with its operational transactions. Instead, it installs a spirit of transparency and accountability, such that entities in the process are allowed to act freely but are also made aware that they are kept under watch, and that any malicious behavior that they engage in will eventually be detected and incur downgrades to their reputation. This reputation could be then used by both users or other WSes as additional constraints on future composition requests (e.g., requiring to collaborate with only WSes that have a reputation score higher than a given value); or it could be used to remove from the system those WSes whose reputation levels get lower of a given threshold.

Auditing consists of an a-posteriori form of control that comprises inspection and thorough verification of historical records with the intention of detecting and correcting errors or misbehavior, if any. This a-posteriori control approach has proved both effectiveness and efficiency across number of domains and systems, such as accounting [15], access control to health records in emergency health systems [1], etc. Moreover, it is especially convenient and naturally fitting to systems wherein different entities collaborate to perform a task, and it is an efficient mechanism against collusion in decentralized systems [9].

As such, we suggest an audit-based architecture for post control of choreographed CWSCs, that incorporates a trusted third-party auditor entity. The auditor performs a-posteriori control by listening to complaints from end-users, reporting instances of suspected collusion or incorrect behavior in a WSC they requested. The auditor receives these complaints, performs security checks to verify their correctness, and punishes the concerned entities according to a defined reputation management scheme. To compensate for the fact that not all users would report suspected malicious behavior, we suggest that the auditor runs periodic audit cycles through which it requests and audits all records of deployed compositions in the system. As we detail in the paper, all these operations are performed in a privacy preserving manner such that the audited data are not revealed to any entity including the auditor.

Since we need to assume an underlying service selection protocol in place, and given that we require such a protocol to support decentralized architectures and to provide privacy preservation guarantees, we exploit the framework proposed in [8]. This work provides an underlying protocol that solves, in a privacy preserving manner, the problem of service selection for a CWSC, providing a proofing mechanism by which involved web services justify their selections. As we explain later in the paper, although this proofing mechanism allows users to verify the correctness of a deployed composition, it cannot be exploited as is for auditing purposes. As such, we suggest a new verification protocol that uses this proofing mechanism and adapts it for auditing tasks. In addition to that, the proposal in [8] does not guarantee that the obtained composition has been deployed by following the activity order specified in the requested workflow. Therefore, before establishing the audit process, we first suggest an extension of the work in [8] with a protocol that allows the verification of the correctness of the deployment order with regard to the imposed workflow. We guarantee all these operations in a privacy preserving manner, as we detail in the paper.

The rest of the paper is organized as follows: we start by summarizing and discussing related work in Section 2, then, in Section 3, we describe the suggested architecture and detail the overall audit process. In Section 4, we recall and summarize the framework in [8], and we suggest an extension to it. In Section 5, we detail the three phases of the audit process and in Section 6, we present our secure audit verification protocols. We present and discuss experimental results in Section 7. We present and discuss the security properties in Section 8. Finally, we conclude the paper in Section 9.

2. Related Work

The problems related to achieving WSCs in a choreography approach have attracted considerable interest from the research community [5]. One of the most important investigated issues is related to ensuring CWSCs without subverting the decentralized nature of the choreography approach. Deploying CWSCs in a choreography requires ensuring two crucial tasks. First, it is compulsory to answer how the selection of appropriate web services in the deployment can be performed and guaranteed with respect to the imposed requirements in a fully decentralized system. Second, it is needed to provide control on the deployment solution and to provide guarantees that collaborating entities would abide by the protocol and would not collude to perform malicious tasks, or they are detected if they do so.
Revising the literature, we find that a considerable number of works have focused on addressing issues related to the first task. For instance, we find proposals of deployment protocols that perform CWSCs focusing on different types of targeted requirements, such as reliability [10], privacy preservation [7], quality of service [14], fault-tolerance using soft constraints that might release some conditions in order to achieve otherwise impossible compositions [22], etc. Some other works considered users requirements as well, such as the proposal in [11]. The first problem that can be noticed on almost all these works is that most of them do not consolidate both user and provider requirements in one solution and often rely on exposing private information about the entities in the system. At this level, we find [8] where a privacy preserving framework that ensures the evaluation of both user and provider requirements in a fully decentralized manner has been proposed. Whilst all these proposals address the deployment issue of CWSCs in a choreography model, none of them answers the second task regarding control and compliance insurance. In fact, these proposals often stop at providing mechanisms to evaluate the compliance of a deployed CWSC to its imposed constraints, without proving any follow-up or control on possibilities of collusion or of malicious activity from the involved entities. Typically, an interface, at end-user side, is exploited to evaluate deployed composition requests and noncompliant executions are simply dropped, without supporting any form of follow up or corrective control (e.g., [8]). This makes it hard to maintain a reliable environment in the sense that colluding web services would continue their dishonest behavior, and possibly enlarge its scope, as this will have no impact on their functionality or reputation.

There are some proposals to enforce control in choreographed compositions, but only regarding some non-functional requirements such as execution time and cost [4], or ensuring resilience to faults by detecting and correcting them during the deployment process using pattern based check-pointing [20]. Moreover, there are some works that considered the computation of reputation scores for web services, but mainly through feedback from end users, either explicitly solicited or predicted based on historical rating (e.g., the work in [23]). To the best of our knowledge, this is the first work that suggests a controlled choreography environment that targets CWSCs, considering both user and involved web services imposed requirements, using auditing techniques.

3. AUDITING OF CHOREOGRAPHED CWSCS

To achieve an audit-based choreography for CWSCs, we suggest the architecture depicted on Figure 1, where end-users specify their requirements and make their composition requests via a web based GUI, called Execution Protocol Application (EpApp).

This request triggers the execution of a protocol, among involved services, for the deployment of choreographed CWSCs, to which we refer to Choreography Driven Protocol (CDP). We consider both constraints posed by users at the time of initiating a composition request (hereafter referred to as user requirements), and constraints required by the involved web services regarding the entities that they accept to collaborate with (i.e., provider requirements). We assume that the adopted CDP offers a proofing mechanism [8] making each web service able to generate a proof about the correct service selection, regarding both user and provider requirements. Upon successful execution of the CDP, the resulting service composition, together with the generated proofs, are returned to the user, via the EpApp interface. As such, requesting users can verify, via the EpApp, these provided proofs, and detect abnormalities, if any. In case of a missing or an incorrect proof, EpApp reports it to an Audit Trusted Server (ATS) that is responsible for audits and reputation management (Section V-A). The ATS verifies the reported complaint and takes appropriate actions towards the detected dishonest entities, by recording the dishonesty and affecting their reputation scores accordingly (Section 5.3). However, as we show in Section 5.2, an end-user could fake a complaint as a trial to subvert the reputation of some target web services. To cope with this, we suggest a double verification of the complaint by considering the user and the providers proofs, both in a privacy preserving manner (Section 6.1).

In addition to listening to and verifying complaints from users side, the ATS also runs periodic audits through which it requests and verifies records of deployed compositions at users side. These periodic audits are performed to compensate for the fact that not all users would collaborate and deliberately report dishonest behavior that they might detect, as to verify possible collusion instances regarding the imposed provider requirements (Section 6.2).

As mentioned above, an audit system operates according to a CDP based on which CWSCs are deployed. The
literature offers numerous proposals for CWSC models to be exploited; however, we require that the exploited protocol meets some minimum qualifications. First, it has to be fully decentralized as we work under a choreography model. Second, it has to fit our scenario of CWSCs considering both user and provider requirements. Moreover, this protocol has to ensure privacy preservation of these requirements in its execution. This is because these requirements might contain private information that the requesting users and/or involved web services might require to keep secure during the deployment process. Finally, a suitable CDP has also to guarantee some form of proving mechanism based on which web services demonstrate their selection process for the subsequent worker in the execution of a CWSC. The framework we have suggested in [8] perfectly qualifies to meet these audit needs. As such, we exploit it as the underlying protocol for the execution of CWSC in our suggested audit-based choreography model. However, we note that this model does not provide any mechanism for verifying the correctness of the deployment order of a composition w.r.t its workflow. As this is one of the correctness elements we would like to include in the audit process as well, we first propose an extension of the model in [8] to account for execution order verification. We then demonstrate how to utilize it for auditing purposes (Section 6.3).

4. EXTENDED FRAMEWORK FOR SECURED CHEROGRAPHED CWSCs

In this section, we first recall the framework suggested in [8], then we detail its suggested extension w.r.t the verification of the execution order of a deployed composition.

4.1 SECURE CHEROGRAPHING OF CWSCs

In [8], we proposed a framework to enforce user and provider requirements in the scenario of service choreography in a privacy-preserving way. We considered the scenario of sequential workflows, in the form \( \text{WF} = \{a_1, a_2, \ldots, a_n\} \); where \( a_i \) is an atomic activity. Applying a user requirement to a workflow activity \( a_i \) implies that properties of services that will carry on \( a_i \) have to satisfy the corresponding conditions.

Typically, each WS, carrying on the activity \( a_j \) in the choreography has to perform two essential steps. First it has to select a service eligible for executing the next activity, \( a_j+1 \), in the workflow. Second it has to invoke the selected service locally and securely. To be selected, a web service \( \text{WS}_{a_j+1} \) has to satisfy the requirements the user has specified for the target activity, i.e., \( a_{j+1} \), as well as the conditions posed by the providers of service \( \text{WS}_j \).

Example 1. Let us consider as example the illustration on Figure 1, where we assume that an end-user is requesting a composite service to prepare a house, starting from renting it to assuring related gardening services. We assume the following workflow is defined: \( \text{WF} = \{a_1 = \text{houseRental}, a_2 = \text{interiorDesign}, a_3 = \text{exteriorDecoration}, a_4 = \text{Gardening}\} \).

In order to model user requirements, in [8], we defined the following:

Definition 1. Atomic condition [8]. Let \( P \) be the set of WS property names. An atomic condition \( \text{cond} \) on \( P \) is defined as:

\[ \text{cond} \in \{ \text{prop}, \text{op}, \text{thresh} \} \]

where \( \text{prop} \in \{\text{price}, \text{area}, \text{grade}\} \) is a property name, \( \text{op} \in \{<, >, \leq, \geq, =, \neq\} \) is a comparison operator, and \( \text{thresh} \) is a value in the domain of \( \text{prop} \).

Based on the above definition, we model user requirements as follows:

Definition 2. User requirements [8]. Let \( a_i \) be an activity specified in a workflow \( \text{WF} \), modeling the composite web service required by user \( u \). The requirements specified by \( u \) for \( a_i \), referred to as \( \text{UR}_{a_i} \), are defined as a Boolean expression over atomic conditions. We express \( \text{UR}_{a_i} \) in Disjunctive Normal Form denoted as:

\[ \text{UR}_{a_i} = \{C_1, C_2, \ldots C_{m-1}, C_m\} \]

where \( \{C_1, C_2, \ldots C_{m-1}, C_m\} \) are DNF clauses, such that

\[ C_j = \{\text{cond}_1, \text{cond}_2, \ldots, \text{cond}_{n-1}, \text{cond}_n\} \]

\[ \forall j = \{1, \ldots, m\} \]

\[ \forall l = \{1, \ldots, n\} \]

Example 2. Let us consider the scenario set in Example 1, and assume that the end-user specifies the following requirements regarding activity \( a_1 \) for the house she wants to rent: house grade is 4, the area is \( \geq 50\text{m}^2 \), the price is \( \leq \$2,000,000 \); or city center is "yes", the area is \( \leq 50\text{m}^2 \). These requirements are modeled as:

\[ \text{UR}_{a_1} = \{C_1, C_2\} \]

where

\[ C_1 = \{\text{cond}_1 = \{\text{grade} = 4\}; \text{cond}_2 = \{\text{area} \geq 50\text{m}^2\}; \text{cond}_3 = \{\text{price} \leq \$2,000,000 \}\} \]

and

\[ C_2 = \{\text{cond}_1 = \{\text{city} = \text{yes}\}; \text{cond}_2 = \{\text{area} \leq 50\text{m}^2\}\} \]

In [8], we have considered three privacy requirements in selecting and invoking services: (R1) user’s requirements have to be privately evaluated over service properties without revealing to the service carrying on the evaluation any of the requirement-related information (e.g., parameters’ values, comparison values, etc.); (R2) provider requirements have to be privately evaluated over service properties without revealing any of the requirement-related information (e.g., parameters, comparison values, etc.). Further, delegating the evaluation of user and providers requirements to each single service might bring to harmful situations, as the delegated service as well as the service
providing the properties might cheat the system. For an example, by skipping the evaluation or by unfairly evaluating conditions on fake property values so as to invoke a colluding service rather than the one satisfying the considered requirements. Hence, the proposed framework has to (R3) provide proofs of the correct privacy-preserving evaluation of user and provider requirements.

In order to satisfy R1, in [8] we adopted the protocol proposed in [6], which exploits asymmetric encryption to privately evaluate combinations of predicates (see [8] for more details of the proposed protocol). We assumed that, for each service request submitted by a user, EPApp generates a distinct pair of keys: a public key \( PK \), available to everyone, and a secret key \( SK \), that is held only by the user requiring the composition. Then, \( SK \) is used by the user to generate a distinct data structure, called token, for each of his/her requirements. This data structure contains the encryption of the information related to the user requirement (e.g., parameters, threshold values). A token is defined in such a way that only the service carrying on the activity to which the user requirement applies is able to decrypt and securely evaluate it over its property values. As an example, the user requirement asking for a house size less than 50 \( m^2 \) has to be evaluated only by the housing agent service. We defined the following:

**Definition 3** User requirement tokenset [8]. Let \( UR_{ai} \) be the requirements of a user \( u \) for activity \( a_i \) in a workflow \( WF \) (see Definition 2). Let \( (PK, SK) \) be a pair of public and secret keys generated by \( u \) according to [6]. We define the user requirement tokenset associated with \( UR_{ai} \), denoted as \( TK_{UR_{ai}} \), as follows:

\[
\{TKC_1, TKC_2, \ldots, TKC_m\}, \text{ where } TKC_j = \{\text{ATK}(\text{cond}), \forall \text{cond}\} \\
\text{is the } j\text{-th DNF clause of } UR_{ai}, \quad \forall j = \{1, \ldots, m\}, \text{ and} \\
\text{ATK}(\text{cond}) = (\text{Enc}_{PK}(\text{cond}.\text{propname}), \text{GenToken}(SK, \text{cond})), \\
\text{with } \text{Enc}_{PK}() \text{ and } \text{GenToken}() \text{ defined based on [6] and} \\
\text{cond}.\text{propname} \text{ the property name included in } \text{cond}.
\]

**Example 3** Let us continue with Example 2. The user requirement tokenset corresponding to the requirements modeled in \( UR_{house} \) is given as follows:

\[
\{TKC_1, TKC_2, \ldots, TKC_m\}, \text{ where } TKC_j = \{\text{Enc}_{PK}(\text{"grade"}), \text{GenToken}(SK, sCond_1)\}, \\
\{\text{Enc}_{PK}(\text{"area"}), \text{GenToken}(SK, sCond_2)\}, \\
\{\text{Enc}_{PK}(\text{"price"}), \text{GenToken}(SK, sCond_3)\}\}
\]

\( sCond_1, sCond_2, sCond_3 \) are respectively the bit strings encoding the threshold ‘grade’ and the operator ‘\( \leq \)’ in the predicate \( cond_1 \), the threshold ‘area’ and the operator ‘\( \geq \)’ in the predicate \( cond_2 \), the threshold ‘price’ and the operator ‘\( \leq \)’ in the predicate \( cond_3 \).

\[
\begin{align*}
\text{price} & \leq \text{cost} \\
\text{price} & \leq \text{cost}
\end{align*}
\]

Providing the properties might cheat the system. For an example, by skipping the evaluation or by unfairly evaluating conditions on fake property values so as to invoke a colluding service rather than the one satisfying the considered requirements. Hence, the proposed framework has to (R3) provide proofs of the correct privacy-preserving evaluation of user and provider requirements.

In order to satisfy R1, in [8] we adopted the protocol proposed in [6], which exploits asymmetric encryption to privately evaluate combinations of predicates (see [8] for more details of the proposed protocol). We assumed that, for each service request submitted by a user, EPApp generates a distinct pair of keys: a public key \( PK \), available to everyone, and a secret key \( SK \), that is held only by the user requiring the composition. Then, \( SK \) is used by the user to generate a distinct data structure, called token, for each of his/her requirements. This data structure contains the encryption of the information related to the user requirement (e.g., parameters, threshold values). A token is defined in such a way that only the service carrying on the activity to which the user requirement applies is able to decrypt and securely evaluate it over its property values. As an example, the user requirement asking for a house size less than 50 \( m^2 \) has to be evaluated only by the housing agent service. We defined the following:

**Definition 3** User requirement tokenset [8]. Let \( UR_{ai} \) be the requirements of a user \( u \) for activity \( a_i \) in a workflow \( WF \) (see Definition 2). Let \( (PK, SK) \) be a pair of public and secret keys generated by \( u \) according to [6]. We define the user requirement tokenset associated with \( UR_{ai} \), denoted as \( TK_{UR_{ai}} \), as follows:

\[
\{TKC_1, TKC_2, \ldots, TKC_m\}, \text{ where } TKC_j = \{\text{ATK}(\text{cond}), \forall \text{cond}\} \\
\text{is the } j\text{-th DNF clause of } UR_{ai}, \quad \forall j = \{1, \ldots, m\}, \text{ and} \\
\text{ATK}(\text{cond}) = (\text{Enc}_{PK}(\text{cond}.\text{propname}), \text{GenToken}(SK, \text{cond})), \\
\text{with } \text{Enc}_{PK}() \text{ and } \text{GenToken}() \text{ defined based on [6] and} \\
\text{cond}.\text{propname} \text{ the property name included in } \text{cond}.
\]

**Example 3** Let us continue with Example 2. The user requirement tokenset corresponding to the requirements modeled in \( UR_{house} \) is given as follows:

\[
\{TKC_1, TKC_2, \ldots, TKC_m\}, \text{ where } TKC_j = \{\text{Enc}_{PK}(\text{"grade"}), \text{GenToken}(SK, sCond_1)\}, \\
\{\text{Enc}_{PK}(\text{"area"}), \text{GenToken}(SK, sCond_2)\}, \\
\{\text{Enc}_{PK}(\text{"price"}), \text{GenToken}(SK, sCond_3)\}\}
\]

\( sCond_1, sCond_2, sCond_3 \) are respectively the bit strings encoding the threshold ‘grade’ and the operator ‘\( \leq \)’ in the predicate \( cond_1 \), the threshold ‘area’ and the operator ‘\( \geq \)’ in the predicate \( cond_2 \), the threshold ‘price’ and the operator ‘\( \leq \)’ in the predicate \( cond_3 \).

As depicted in Figure 1, to start the composition, by means of EPApp the user searches for service \( WS_j \) which has to execute the first activity \( a_j \).\(^2\) Afterward, EPApp passes all the generated tokens along with the user’s PK to \( WS_j \). In turn, \( WS_j \) retrieves the token(s) applying to \( a_j \) and, adopting the protocol in [6], it privately evaluates whether its properties satisfy the stated conditions. Basically, \( WS_j \) encrypts the list of its provided properties values using \( PK \) to generate its set of encrypted properties denoted as \( SEP_j \). Then it evaluates \( SEP_j \) against the secured tokenset of user requirements on the activity it would be expected to perform (i.e., \( TK_{UR_{ai_j}} \)), and generates a proof for the evaluation result. The evaluation is performed based on the PrivateEvaluation algorithm, as in [8], that would produce a two dimensional array of boolean values, \( arRes \), that contains the result of the evaluation. Basically, the evaluation is considered positive if \( arRes \) contains at least one record of all true values, indicating that the user requirements are answered by the properties of the web service. The result of this evaluation, along with \( SEP_j \), is communicated back to EPApp that can use them to check whether the requirements evaluation has been correctly executed (cfr. the PrivateEvaluation algorithm in [8] for full details on the proof verification). If this is the case, EPApp invokes \( WS_j \). A similar process is done for each activity \( a_j \) in the workflow by the previous web service \( WS_{j-1} \) that was invoked for activity \( a_{j-1} \). Therefore, once activity \( a_{j-1} \) has been executed, the corresponding service searches for a new service \( WS_j \) to be assigned with \( a_j \). \( WS_{j-1} \) sends to \( WS_j \) all tokens along with the user’s PK. \( WS_j \) is then invoked only if \( WS_j \) receives a proof that \( WS_j \) satisfies the user requirements.

Regarding provider requirements, for their privacy-preserving evaluation, [8] adopts the privacy preserving matching protocol presented in [12], which enables two entities to privately compute the intersection of two sets.

\(^1\) We assume that technically the threshold in the string type, e.g., the place name, the area name, etc., is mapped onto a number. For an example, ‘City Center’ is mapped onto 5.

\(^2\) This service is retrieved by inquiring Universal Description, Discovery and Integration (UDDI) as well as other search methods.
The key idea is that, to check if a given value satisfies a condition, we can verify whether it belongs to the predefined set of values satisfying the condition. As an example, given the condition “ExpectedTimeResponse < 10” the value 9 does satisfy the condition as it belongs to the set [0,9). Thus, [8] exploits the protocol defined in [12] to make the invoking service and the service to be invoked able to privately verify whether the requirements of the invoking service are satisfied. More precisely, when a service carrying on the generic activity $a_i$ ended its execution, it first verifies user requirements on $WS_{i+1}$, i.e., the candidate service for activity $a_{i+1}$. Then, it verifies if $WS_{i+1}$ is compatible with its requirements. Also in this case, the protocol execution generates a proof that can be used to check the correct evaluation.

To satisfy requirement $R3$, all proofs generated during the composite service deployment are collected and returned to EPApp together with the final results. By exploiting these proofs, EPApp can further verify that all privacy-preserving evaluations have been done correctly, that is, that the services evaluating the requirements have not colluded with the service under evaluation.

### 4.2 Execution Order Verification

In this section, we extend the protocols introduced in [8] in order to provide assurance about the execution order according to which the services have been invoked. Indeed, the execution order of activities, in a requested WSC, needs to be performed following a predefined workflow. In order to be able to verify that, we need to log the identity of all involved services along with the activities they performed. However, it is possible that end-users as well as some web service providers do not want to reveal their information or their identities to all the other web services involved in the composition’s deployment. To answer these requirements, we propose a protocol for the execution order verification in a private way. We achieve this by chaining the IDs of web services involved in a composition, in a secret way, to generate a unique secret chain representing the execution order, that users can verify at the end of the deployment. As also suggested in [8], we assume that for each composite service deployment, EPApp generates a pair of keys, including a public key $PK$ and a private key $SK$. Every service receives $PK$ along with the required workflow. In order to generate the execution order chain, we suggest a data structure, called execution order token, that is updated at the level of each involved web service and that is forwarded to each invoked web service. A web service updates the execution order token by inserting its ID into it. The token ensures that all inserted IDs are correctly ordered by means of hashing techniques. We formally define the execution order token as follows:

**Definition 4.** Execution Order Token. Let $WF = \{a_1, a_2, \cdots, a_n\}$ be a workflow corresponding to a composition request made by user $u$. Let $PK$ be the public key generated by $u$ for this request. Let $comp = \{WS_1, WS_2, \cdots, WS_n\}$ be the web services involved in the deployment of the request corresponding to $WF$. Let $WS_i.id$ be the ID of $WS_i \in comp$. When $u$, via the EPApp, invokes $WS_i$ for activity $a_i$, it initializes an execution order token, $eo_0$, that it sends to $WS_i$. $eo_0$ is defined as follows:

$$eo_0 = Enc_{PK}(\phi), \text{ where } Enc_{PK}(\cdot) \text{ is an encryption function using } PK, \text{ and } \phi \text{ is a random number generated by EPApp.}$$

Subsequently, an execution order token, $eo_i$, generated by $WS_i \in comp, i \in [1,n]$ is generally defined as follows:

$$eo_i = [\delta || Hash(\delta)]$$

where $\delta = Enc_{PK}(eo_{i-1} || WS_i.id)$, $Hash(\cdot)$ is a hash function, $||$ refers to string concatenation, and $eo_{i-1}$ is the execution order token received by $WS_i$ from EPApp if $i = 1$, or from $WS_{i-1}$, otherwise.

As per Definition 4, the execution order token is initialized at the user side by the EPApp and is passed to the first web service invoked for the first activity in the workflow. Afterwards, the execution order token is subsequently updated at the level of each invoked service by concatenating its own ID, after securing it with $PK$, and applying a hash function to the result that ensures the chronological chaining of previously inserted web service IDs. As such, by the end of a composition, the last version of the execution token, as updated by the last web service in the composition, contains a secured chain of the IDs of all involved web services. This is sent back to the requesting user along with the result.

**Example 4.** Let us consider the scenario set in Example 1. The requesting user, via the EPApp, initializes an order execution token $eo_0 = Enc_{PK}(\phi)$, where $\phi$ is a random number that it generates. EPApp sends $eo_0$ to the selected housing agent service, $WS_1$. After executing activity $a_1$ and deciding to invoke the interior design agent, $WS_2$, it updates the order execution token by computing $eo_1 = [Enc_{PK}(eo_0 || WS_1.id || Hash(Enc_{PK}(eo_0 || WS_1.id))]$ $eo_1$ is communicated to $WS_2$ that, at its turn, generates $eo_2$ in a similar way based on $eo_1$, and so on.

Figure 2 depicts the execution order of Example 1 with the corresponding execution order tokens.

![](image)

**FIGURE 2 AN EXAMPLE OF EXECUTION ORDER PROTOCOL**

It is to be noted that the intermediate web services in the deployment of a composition request cannot learn the
identities of the previously involved providers, as all IDs in this latter are secured using the $PK$ of the requesting user. Therefore, only the user, being the only entity holding the corresponding $SK$, can decipher the content of the final received execution order token (i.e., $eo_o$) and verify the chain of IDs it contains. The verification of $eo_o$ happens, at the level of the EPApp, according to Algorithm 1.

Algorithm 1 verify Order Token
\begin{algorithmic}[1]
\State \textbf{Input}: $eon; SK; WF$
\State \textbf{Output}: $incorrectOrderList$
\State 1. $incorrectOrderList = \emptyset$
\State 2. $i = n$
\While{$(eon! = null)$}
\State 3. $eo = eon.extractLastElement()$
\State 4. $\delta = eo:get\delta()$
\State 5. $dec = decrypt(\delta; SK)$
\State 6. $ID = getID(dec)$
\State 7. $H = Hash(ID)$
\State 8. $eo = eon.extractLastElement()$
\If{$(H != eo.getH())}$
\State 9. $incorrectOrderList.add(ID)$
\Else
\State 10. $incorrectOrderList.add(ID)$
\EndIf
\EndWhile
\State 11. $i = i - 1$
\State 12. $return incorrectOrderList$
\end{algorithmic}

Algorithm 1 takes as input the execution order token returned at the end of the composition’s deployment, $eo_o$, the corresponding workflow, $WF$, and the secret key associated with the request in question, $SK$, and generates as output the list of IDs of web services that did not respect the execution order (i.e., $incorrectOrderList$). The algorithm starts by initializing the $incorrectOrderList$ as an empty set, and by initializing an integer $i=n$ to reference the activities in $WF$. The algorithm then iterates over all the elements contained in $eo_o$, that is, the blocks added by each web service involved in the composition when updating the received execution order token. At every iteration, the last element of $eo_o$ is extracted (i.e., $eo$) (line 4). The $\delta$ value is extracted from $eo$, it is decrypted using $SK$, and the ID of the web service that created it is retrieved as $ID$ (lines 4 - 7). Afterwards, the integrity of $eo$ is verified, by recomputing the hashed value of the extracted $\delta$ using the $Hash()$ function, as in Definition 4, to get a value $H$ (line 3), and comparing it to the extracted hash value as available in the element $eo$ (line 9). If the two values do not match, then this means that this element of $eo_o$ is corrupt. As such, the ID of the corresponding service is added to the $incorrectOrderList$ (line 10). In the other case, the algorithm verifies if the web service that performed the corresponding activity in the workflow (i.e., $WF: a_i$) does in fact provide the functionality required by that activity. This is done by inquiring UDDI (the function call $getUDDI(ID)$) to retrieve the functions provided by the web service with ID $ID$. Then, function $check()$ is used to verify if these functions match the corresponding activity $WF: a_i$ (line 12). In case they match, the algorithm simply iterates over the next element of $eo_o$. In case they do not, $ID$ is added to $incorrectOrderList$ (line 13). When there are no more elements in $eo_o$, the algorithm terminates returning $incorrectOrderList$ (line 13).

In Section 6.3 we explain how the ATS can audit the execution order based on the results obtained by the user, using Algorithm 1, via the EPApp.

5. Audit and Reputation Management

The purpose of incorporating an audit control is to provide a deterrent and corrective mechanism that detects and punishes dishonest entities. This is generally achieved by considering an accompanying reputation management model in the system. The literature offers number of reputation management models [21]. The main problems studied in reputation management are related to how to set the starting reputation score, how to update it, and how to disseminate it. In our scenario, we consider that reputations update happens at the level of the ATS based on the audit functions it performs, as we explain later in this section. ATS, as a trusted entity in the system, is also responsible for disseminating reputation scores or providing them upon request. Regarding the entry reputation score, there are two main approaches, as discussed in [18]. The first one suggests starting with a very low entry reputation score that increases as the entity behaves well in the system. This approach focuses more on reward than on punishment with the rationale that new entities cannot be trusted until they demonstrate, with evidence, that they deserve it. The advantage of this approach is that it makes it costly for malicious entities to re-initiate their reputation in the system by simply creating new identities or new accounts. The second approach bases on trusting new entities until they show the opposite. As such, entities start with high entry reputation scores that are decreased with every detected misbehavior.

Given that we deploy an a-posteriori form of control that bases on the concept of trust-and-keep-under-watch, and that our model is punishment oriented, we assume that, in our system, a full reputation score is assigned at first entry.

---

3 We assume that the types of functions provided by a web service can be inferred from its corresponding ID, by checking some global public repositories such as the Universal Description, Discovery and Integration (UDDI) registry.
For simplicity, we consider that every entity in the system’s environment (every user $u$ and every web service $WS_i$) has a reputation score denoted as $u.rep$ or $WS_i.rep$, respectively. As such, users and web services start with full reputation scores that are decreased with every detected misbehavior from their side. We note that, in our scenario, reputation re-initiation attacks would not be harmful. First, the only applicable motivation of a malicious user is to claim dishonest behavior on honest web services with the aim of ruining their reputation. However, the system is immune to such attacks as it verifies users claims from a provider perspective as well, as we detail in Section 6.1. Concerning web service providers, their invocation depends first on their satisfaction of user requirements and not solely on their reputation. Moreover, it would be costly for a web service provider to each time change their identity and appear as a new entity as this goes against their business model.

5.1 Audit Notifications

ATS receives audit notifications to start an auditing process on suspected web services in one of two ways: 1) a user reports a complaint to the ATS, or 2) the ATS requests records of deployed composition requests from a user during a periodic audit. We assume that the ATS triggers a periodic audit process following a time period $\tau$, and that the EPApp keeps a log of all deployed composition requests that the user performed between each two periodic audit cycles; that is, for at least a time period equals to $\tau$. Regardless of whether an audit notification is generated by a complaint deliberately reported by the user or as an answer to the periodic audit, the ATS receives an audit report (ar) from user $u$ via the EPApp concerning a deployed composition. An audit report contains the list of the web services that were involved in the deployment of a CWSC request made by the user, along with the corresponding elements that the user received as verification proofs for that composition’s result. We assume that every web service $WS_i$ is known by a unique identifier denoted as $WS_i.id$, and we define an audit report as follows:

**Definition 5. Audit Report (ar).** Let $comp = \{ WS_1, .., WS_n \}$ be the composition of web services that answered a composition request made by user $u$ corresponding to a workflow $WF = \{ a_1, .., a_n \}$. Let $proof_i = (SEP_i, TK_{UrGWard}, arRes_i, WS_i.id)$ be the proof provided by service $WS_i$ whose ID is $WS_i.id$ in the composition, where $SEP_i$ is its set of encrypted properties, $TK_{UrGWard}$ is the secure token encoding user requirements for activity $a_i$, and $arRes_i$ is the result of the proof. An audit report $ar$ regarding the deployed composition $comp$ is defined as follows:

$$ar = \{(WS_i.id, proof_i_i) | \forall WS_i \in comp\}$$

where $proof_i_{k+1}$ is the proof provided by the service that was invoked by $WS_i$ for activity $a_{k+1}$ in the composition, and $k \in [1, (n-1)]$.

**Example 5** Let us consider the example depicted in Figure 1. For simplicity, assume that the user had only one constraint on activity $a_4$ about the gardening requiring the price to be less than 100S. That is, $UR_{gardening} = \{cond_i = [\text{price} \leq 100S]\}$. Assume that the user suspects that $WS_4$ has colluded with $WS_3$ and selected it even if it does not satisfy the imposed user requirement. That is, assume that the encrypted set of properties, $SEP_i$, returned by $WS_3$ to $WS_4$ during the selection process is an encryption of the set $\{\text{property} = [\text{price} > 200S]\}$. This means that the cheapest price offered by $WS_4$ is 200S. The user will make an audit report $ar = \{ (WS_3.id, (null, null, null, null, WS_3.id), WS_3.id, (SEP_i, TK_{Urgardening} [true], WS_3.id)), TK_{Urgardening} \}$ is the secured tokenset for $UR_{gardening}$.

5.2 Verifying an Audit Report

Upon receiving an audit report, the ATS needs to verify the correctness of the provided proofs. To perform this, an option is to rely on the private evaluation algorithm suggested in the protocols proposed in [8]. In fact, using this algorithm, the ATS could recompute an $arRes_i$ for any received $proof_i$ based on $SEP_i$ and $TK_{UrGWard_i}$. ATS could then simply compare the obtained $arRes_i$ to the one included in the proof (i.e., $arRes_i$). If the two do not match, then the proof is incorrect and a collusion is confirmed. In such a case, ATS could decrease the reputation score of the dishonest web service (cf. Section 5.3). However, it might be that the user has faked the reported proofs so as to affect the reputation of innocent web services. In fact, and as detailed earlier, the elements making a proof are all generated based on encryptions using the public key of the user that initiated the request. That is, the user, given that she holds her secret key, can use it to corrupt the provided elements in a proof in an attempt to ruin the reputation of attacked web services. To go around this problem, we suggest a new double edged secure verification protocol for the ATS to reliably evaluate a received audit report both from a user and a provider perspectives. We detail this verification protocol in Section 6. Once a collusion is confirmed, the ATS activates a function for the verification of compliance to provider imposed requirements as well. We detail this function in Section 6.2.

5.3 Reputation Update

---

* The purpose for considering users’ reputations as well is because they might fake report complaints against honest web services.

5 We assume that the elements of a proof are equal to value null if the user had no requirements for the activity it concerns.
As mentioned earlier, we assume a full reputation score is assigned upon entry to the system, both to users and to web services. Upon every confirmed dishonesty against a web service, or any false report from a user, ATS needs to adjust the reputation scores accordingly. Decreasing reputation scores might be uniform, such as taking off a fixed amount of points per detected dishonesty, as it might happen following an increasing scale such that more points are taken off as the frequency of wrongdoing increases. Regarding users, and since a dishonest user would not have serious effects on the results of requested compositions, we consider a simple update of their reputation scores that consists at taking off fixed number of points with every phony complaint they make. However, this cannot be adopted for web services as they are the entities that are subject to collusion and are the target of the auditing control. As such, and to have a meaningful reputation scores management, we suggest punishing web services according to their dishonesty history. Moreover, we also consider the type of the dishonesty as whether the dishonest web service provided an invalid proof or invoked an unqualified web service for a task. To account for these elements, the ATS keeps an audit record for each web service in the system. This record tracks the count of the dishonest behaviors of a service along with their types. We differentiate between two types of misbehavior: an unqualified invocation (unqInv), and an invalid proof provision (invPr). We refer to such a record as WS audit record and we define it as follows:

Definition 6. WS Audit Record. Let WS be a web service in the system. The WS audit record of WS, is denoted as \( WSAR = (WS.id, unqInvCount, invPrCount) \), where \( WS.id \) is the unique identifier of WS, in the system, and \( invPrCount \) are two integers referring to the number of dishonest records of types unqInv and invPr, respectively, that ATS has processed against WS.

A web service, WS, that has a full reputation score, would have no recorded dishonesty of any type. As such, and according to Definition 6, its corresponding audit record would be \( WSAR_i = (WS.id, 0, 0) \). When ATS receives an audit report that contains WS, and confirms a dishonesty against it, WSAR, is updated applying an increment by 1 to either \( WSAR_i.unqInvCount \) or to \( WSAR_i.invPrCount \), depending on the confirmed dishonest type.

Example 6 Let us continue with the scenario in Example 5. Let us assume that the reported suspected collusions has been confirmed by the ATS. Let us also assume that this is the first dishonesty recorded against both \( WS_i \) and \( WS_e \). The WS audit records for these two services would respectively be: \( WSAR_i = (WS_i.id, 1, 0) \), and \( WSAR_e = (WS_e.id, 0, 1) \).

Once the WS audit records are updated, the ATS also updates the reputation of WS. This is done based on Equations 1 or 2 depending on whether the dishonesty is of type unqInv or invPr, respectively.

\[
WS_{rep} = WS_{rep} - (WSAR.unqInvCount \times 2 \times \theta) \quad [1] \\
WS_{rep} = WS_{rep} - (WSAR.invPrCount \times \theta) \quad [2]
\]

We assume a \( \theta \) punishment coefficient that, in both Equations 1 and 2, defines the slope by which a reputation score is decreased. This is decreased based on the number of dishonesty records already logged in the audit record of the corresponding web service, \( WSAR_i \), that are of the same type as the newly detected dishonesty. In the case of a dishonesty of type unqInv, the reputation score is decreased based on two times the \( \theta \) coefficient (Equation 1); whereas it is decreased based on only one \( \theta \) for the other type (Equation 2). This is because colluding with an unqualified web service by invoking it for an activity against the required conditions is what actually affects the overall result of the composition. Providing an invalid proof, on the other hand, might not have any effect on the composition given that no entity colludes with the dishonest service.

Once the ATS has updated the reputation scores, it checks if they fall below a set tolerance threshold \( \theta \), that we consider as a variable that could be set at the level of the ATS by the system administrator. The ATS disqualifies any affected web service whose reputation score is below \( \theta \), by removing it from the system.

Moreover, reputation scores are meant to be used as possible further constraints required either by end users or by involved web services.

Example 7 Let us consider Example 2, by which the end-user would have specified these requirements for the house activity: \( UR_{house} = \{Cl_1, Cl_2\} \). The end-user could also incorporate an additional clause that requires selecting only those housing service providers that have a reputation score higher than value 10, for instance. As such, \( UR'_{house} = \{Cl_1, Cl_2, Cl_4\} \), where \( Cl_i = [\text{cond} = \{\text{reputation} \geq 10\}] \).

We also note that we consider a punish-based system only that does not incorporate rewards, as honest behavior is the one expected by default. As such, reputations could only be decreased with every detected misbehavior. Honest entities should be able to maintain their initial entry full reputation score.

6. **Proof Verification Protocols**

Once all activities of the workflow are invoked, an end-user can verify their compliance to the imposed user constraints by validating all the received proofs. As mentioned earlier, via the EPApp interface, she/he can report to the ATS any received composition result in which she/he found invalid or missing proofs for her/his required constraints. This is done by sending an audit report, as per Definition 5, to report detected dishonest web services. In addition to that, ATS could also receive an audit report as an answer to a periodic audit request it makes to a user. As discussed earlier, ATS cannot blindly punish services based on the received audit report from user side only, as a user can make false complaints to blame incorrect things on services to fake their reputation. For this, when receiving an
audit report from a user, ATS activates the function user requirements double proof test (UR double proof test) through which it determines whether the user or the complained services are dishonest. Moreover, it is to be noted that reported complaints to ATS concern suspected collusion regarding user requirements only. For this, the ATS activates also a provider requirements proof test (PR proof test) to check if the complained services colluded regarding provider imposed requirements as well (Section 6.2).

6.1 UR DOUBLE PROOF TEST

As per Definition 5, an audit report communicated to the ATS contains the list of the web services involved in a deployed composition along with the proofs they have provided to justify their selections. We assume that when a user u sends an audit report ar to the ATS, it also sends the corresponding PK. To audit ar, the ATS executes a UR double proof test, through which it communicates with the concerned web services, to make sure that the user has not faked the reported elements of the proofs. This is achieved according to Algorithm 2.

Before describing the algorithm, we note that we assume that all web services in the system keep a log of all tokens of user requirements they have used in the deployed compositions they were involved in for at least a time period of values \( \tau \). We remind that \( \tau \) is the cycle of the periodic audits performed by the ATS.

Algorithm 2 takes as input the audit report ar and the corresponding PK. The algorithm returns a boolean value true if a collusion was indeed detected in ar or if u is found to be dishonest by faking the proofs elements. If all the proofs in ar are found to be correct, the algorithm returns a false value to show that no dishonest behavior was detected. The algorithm iterates through all the records contained in ar (i.e., in line 2). For each record rec, the contained proof is extracted as proof, (line 3).

Algorithm 2 UR_Double_Proof_Test()

Input: \( \text{ar} = \{(WS_i, proof_{i,j}) | i \in \{1..n\}; \text{PK} \) \)
Output: true/false

1.  \( i = 2; \text{found} = \text{false} \)
2.  while (rec = ar.getNext() != null) do
3.      proof, = rec.getProof();
4.      SEP, = proof, getSEP();
5.      TK_{UR_{qi}} = proof, getUR();
6.      if (PrivateEvaluation(SEP, , TK_{UR_{qi}} ) ==
7.          proof, getarRes()) then
8.          if (validate_user_proof(proof, getarRes()) ==
9.              true) then
10.             i += 1;
11.         else
12.         WS_i, = rec.getWS();
13.         WS_i, = proof, getWS();
14.         TK_{WS_{i-1}} = requestUR(WS_i,
15.         SEP_i = requestSEP(WS_i, PK);
16.         if (TK_{UR_{qi}} != TK_{WS_{i-1}} or SEP_i != SEP_i ) then
17.             decrease(u.rep);
18.         else
19.             WSAR_i.updateRep(WS_i,
20.             WSAR_i.updateInvCount + = 1;
21.             updateRep(WS_i, rep; WSAR_i); Equation1);
22.             updateRep(WS_i, rep; WSAR_i; Equation2);
23.         end if
24.         i += 1;
25.         end if
26. end while
27. return found;

Afterwards, the SEP, and the TK_{UR_{qi}} elements are extracted (lines 4-5). These two elements are re-evaluated using the same function, suggested in [8], that was used by involved web services to evaluate proofs for the selection process. This is done by the call to PrivateEvaluation() function that is referring to Algorithm 1 as suggested in [8] (line 6).\(^6\) If it is found that the arRes value included in proof (i.e., proof, getarRes) is equal to the newly obtained value using PrivateEvaluation(), the algorithm checks if this arRes value represents a positive evaluation of the proof or not. This is done by the call to the function validate_user_proof() in line 7 that returns a boolean value reflecting the result of the evaluation. Basically, this function iterates over all the entries in arRes and checks if one of them is all true (i.e., the evaluation is positive), or not. If the function returns a true value, it means that the evaluation is positive; hence it is concluded that the record in question, rec, contains a valid proof and does not reflect any dishonesty of any type. As such, the algorithm jumps to evaluate the next record in ar. However, in case this evaluation is negative, or the obtained arRes does not match proof, getarRes, then a dishonest behavior is detected and it is needed to verify whether it is the user who is dishonest, by faking the elements of the proofs, or the involved web services colluded against user requirements. This verification requires communicating with the two involved web services in the selection related to rec. As such, both the selecting web service (i.e., WS_i, ) and the selected

\(^6\)As in [8], the PrivateEvaluation() inputs a UR secure token and a set of encrypted provider properties, and outputs an arRes double array of booleans that, if it has a record of all true values, demonstrates that the evaluation is positive; otherwise it is not.
service (WSi) are retrieved from rec and proofi, respectively (lines 12-13). Afterwards, the copy corresponding to TKi, as was used by WSi,i, is requested directly from WSi (getting TKWi,i in line 14). Moreover, WSi is requested to send its set of properties after encrypting them with PK (getting SEPWi in line 15). It is to be noted that SEPWi is requested directly from WSi without specifying any information on which composition is being audited. As such, WSi is simply asked to provide a secure version of its local properties. This is done instead of requesting WSi,i to send its logged version as it might send a fake one if it is dishonest and colluded when selecting WS. After receiving the other versions of the requested values, it is checked if the versions match. As in line 16, if one of either SEP or TKi does not match its corresponding version, then it is concluded that the user i is dishonest and has changed the elements of the proof. As such, the reputation of i is decreased (line 17). However, if both of them match, then it means that a collusion between WSi and WSi,i took place. Therefore, their corresponding WS audit records and their reputation scores are updated. More precisely, WSi is punished for an invalid proof provision and WSi,i for an unqualified invocation. Therefore, their corresponding audit reports are updated accordingly (lines 21-22). The algorithm terminates by returning false, if all records of ar are found to be legitimate, or true if at least one record of ar contained a dishonesty.

### 6.2 PR Proof Test

Although respecting user requirements in the deployment of a composition is what would matter to the requesting user, the system needs also to ensure that involved web services respect the requirements they claim. Hence this protocol is run by the ATS, periodically, to randomly verify if provider requirements have been correctly enforced. In general, a web service might have a set of non-functional requirements regarding the standards, usually related to security or QoS, based on which it can collaborate with other web services. Moreover, web services also have a set of non-functional properties that they support, such as meeting a given quality standard. Therefore, a set of provider requirements, PRi, imposed by web service WSi formulates the general conditions based on which WSi can collaborate with another web service, WSi,i. Moreover, we assume that WSi,i is characterized by a set of properties, PPi,i, that it supports and offers to its collaborators. Clearly, PPi and PPi,i are independent from user composition requests. For this, we suggest that whenever the ATS audits an ar for compliance to provider requirements, it directly communicates with every two directly collaborating web services, WSi and WSi,i, in the corresponding composition and audits whether PPi,i answers the requirements set in PRi or not. As PPi and PRi,i might contain private information, we suggest performing this audit in a secure way, such that the ATS can validate the requirements without being able to know their content. We achieve this using homomorphic encryption techniques. Since the ATS receives PK of the end-user with every received ar, we suggest making use of this key to ensure the secure audit of provider requirements. That is, the ATS communicates PK to both WSi and WSi,i and asks them to encrypt their required properties values. Before explaining how this is achieved, we first define provider requirements and provider supported properties. Let us consider that WSi and WSi,i are under audit by the ATS for meeting provider requirements, with WSi,i being the service that was invoked by WSi in the corresponding audited composition. Hence, we call WSi the selecting provider and WSi,i the selected provider. WSi has a set of requirements, PRi, and claims being able to collaborate with WSi,i only if this latter satisfies all of its requirements. We define provider requirements, similarly to user requirements (cfr. Definition 2), in DNF form as a boolean expression over atomic conditions. An atomic condition (i.e., Definition 1) encompasses a threshold, an operator, and a comparable value. We consider the basic operators (i.e., ‘≤’, ‘<’, ‘≥’, ‘>’, ‘=’, ‘≠’), the more complex operators, such as ‘∈’ or ‘∉’, can be established from basic operators. Relying on the threshold and the operator in an atomic condition cond, we formulate its atomic condition satisfying values, i.e., arVal(cond) as the finite ordered set of all values that satisfy cond. We define the following:

**Definition 7** Requirement satisfying values. Let \( PR = \{ C_i, \cdots, C_m \} \), such that \( C_j = \{ cond_1, \cdots, cond_n \}, j \in [1, m] \), be a DNF boolean expression over atomic conditions. We define the requirement satisfying values of \( PR, SV(PR) \), as the super set of sets of values satisfying each of its clauses. More precisely, given a clause \( C_i \in PR \), we define its set of satisfying values \( SV(C_i) \) as the union of the satisfying values of each of its conditions. That is,

\[
SV(C_i) = \{ \forall cond_1 \in C_i \mid \forall cond, \text{arVal}(cond) \} ,
\]

where \( \text{arVal}(cond) \) denotes the finite ordered set of all values in the domain of the property referred in cond, that satisfy cond. Let \( SV(C_i) = \{ a \in \text{Dom}(cond) \mid \text{arVal}(cond) \} \) for any property \( cond \) is true.

Therefore, \( SV(PR) = \bigcup SV(C_i) \).

**Example 8** Let us consider the illustration in Figure 1, and assume WSi, a house agent service, has a set of requirements \( PR_i \) that must be met by any service it can collaborate with. Assume that, \( PR_i = \{ C_i = \{ cond_1 = (\text{response\_time} \leq 10ms) \} \} \) or \( C_i = \{ cond_i = \{ (\text{fault\_tolerance} = 1) \} \} \). The requirement
satisfying values of $PR_j$ are generated as: $SV(PR_j) = \{arVal(Cl_i.cond) = [1,2,\cdots,9,10] \cup arVal(Cl_i.cond) = [1]\}$.

When $WS_i$ is under audit as a selecting provider, it returns to the ATS a secured data structure corresponding to the requirements satisfying values of its $PR_i$. We refer to this as the selecting provider proof, and we define it as follows:

**Definition 8 Selecting Provider Proof.** Let $WS_i$ be a selecting provider that is being audited by the ATS and let $PR_i = \{Cl_i, \cdots, Cl_m\}$ be the DNF boolean expression of its requirements. Let $SV(PR_i)$ be the requirement satisfying values of $PR_i$. Let $SV(Cl_i) \in SV(PR_i)$ be the satisfying values of $Cl_i \in PR_i$. Let $APS_{cond_i} = \{Enc_{pk}(\alpha) \mid \alpha \in arVal(cond_i) \in SV(Cl_i)\}$, such that $arVal(cond_i) \in SV(Cl_i)$, where $Enc_{pk}(\cdot)$ is an encryption function with $PK$ that $WS_i$ received from ATS. $WS_i$ communicates to the ATS a selecting provider proof, denoted as $PrSr_i$, and expressed as follows:

$$PrSr_i = [Pr_{i_1}, \ldots, Pr_{i_n}], \text{ with } Pr_{i_j} = \{APS_{cond_i} \mid \forall arVal(cond_i) \in SV(Cl_i)\}.$$

Every service provider, $WS_i$, besides having a set of requirements, also holds a set of non-functional properties that it provides (i.e., $PP_i$). For the sake of simplicity, we assume that each property supported by a web service is expressed as a property name and a property value. For example, if a web service supports a RAM of size 1GB, then this property would correspond to a property name = RAM, and property value = 1GB. As such, $PP_i = \{prop_k \mid k \in [a,n]\}$, where $prop_k = (prop.name, prop.val)$ with $prop.name$ and $prop.val$ are a property name and its corresponding value, respectively.

A selected web service, $WS_{i+1}$, that is being audited, is contacted by the ATS and is required to return the set of its supported properties (i.e., $PP_{i+1}$), that has to be first secured with the communicated $PK$. We refer to the result of this encryption as selected provider proof, and we define it as follows:

**Definition 9 Selected Provider Proof.** Let $WS_{i+1}$ be a selected provider being under audit. Let $PP_{i+1} = \{prop_{i_1}, \ldots, prop_{i_m}\}$ be its set of supported properties.

$WS_{i+1}$ returns to the ATS a selected provider proof, $PrSe_{i+1}$, that is defined as:

$$PrSe_{i+1} = \{Enc_{pk}(prop_{i_1}.val), \cdots, Enc_{pk}(prop_{i_m}.val)\}$$

where $Enc_{pk}(prop_{j}.val), \forall j \in [0,m]$ is an encryption of $prop_{j}.val \mid prop_{j} \in PP_{i+1}$, the $j$-th property value of $WS_{i+1}$, where $PK$ is the key that $WS_{i+1}$ received from the ATS.

After receiving the selecting provider proof $PrSr_i$ and the selected provider proof $proof(PrSe_{i+1})$, from $WS_i$ and $WS_{i+1}$, respectively, the ATS verifies them for compliance. This is performed according to Algorithm 3.

Algorithm 3 validate_Provider_Proof()

**Input:** $PrSr_i, PrSe_{i+1}$

**Output:** true/false

1. res = false;
2. for $Pr \in PrSr_i$ do
3. for $APSr \in Pr$ do
4. for $prop \in PrSe_{i+1}$ do
5. for atom $\in APSr$ do
6. if $prop \oplus atom == ZERO$ then
7. res = true;
8. break;
9. end if
10. end for
11. if (res == true) then
12. break;
13. end if
14. end for
15. end for
16. end for
17. return res;

Algorithm 3 has two inputs, that is, the selecting provider proof $PrSr_i$ and the selected provider proof $PrSe_{i+1}$, and outputs a boolean value true or false, showing that the validation is successful or failed. ATS iteratively traverses each set of satisfying clause $Pr$ in $PrSr_i$ and each secure atomic provider condition in $Pr$, i.e., $APSr$ (cfr. the for loops in lines 2-4). Then, for each $prop$ in $PrSe_{i+1}$ (line 4) and for each $atom$ in $APSr$, ATS checks which sum of $prop$ and $atom$ is equal to $ZERO$ (line 6). $ZERO$ is the encryption of '0' with $PK$ by ATS. The addition operator hereafter is understood explicitly as or-exclusive operator $\oplus$ of two bit strings. It means that $\gamma = prop \oplus atom$, assuming that $prop = Enc_{pk}(\alpha_q)$, $atom = Enc_{pk}(prop.val_q)$, hence $\gamma = Enc_{pk}(\alpha_q) \oplus Enc_{pk}(prop.val_q) = Enc_{pk}(\alpha_q \oplus prop.val_q)$.

In case $\alpha_q = prop.val_q$, the or-exclusive operator results in a string of bits '0' and the encryption result is equivalent to $ZERO$. It means that the corresponding property of $WS_{i+1}$ passes the condition of a corresponding atomic condition of $WS_i$; thus, the returned result is set to true (line 7). If all loops are done successfully, ATS receives the number of successful results equal to the number of elements in $PrSr_i$, it means that the providers followed correctly the service selection protocol before. Otherwise, ATS can conclude that $WS_i$ and $WS_{i+1}$ colluded regarding provider requirements satisfaction. At such an instance, these two WSes are punished for an unqualified invocation and an invalid proof, respectively.

We consider that the ATS keeps a log record of any two web services that it previously audited for compliance to...
provider requirements, along with the result of the audit. That is, when the ATS audits \( W_S \) and \( W_{S+i} \) for PR compliance, it keeps a record, \( PR_{Audit}(i+1) = [W_S, ith, W_{S+i}.id, matched] \), where \( matched \) is a boolean equal to true if \( W_{S+i} \) has been found to answer \( PR_i \) of \( W_S \) and, or is equal to false otherwise. Once the ATS has such record, it can use it to decide on any other instance where \( W_S \) and \( W_{S+i} \) collaborate. This is because the PRs of a web service are not expected to be quickly changing. However, to account for possible changes to PRs, we consider that these records have an expiry point after which the ATS needs to perform the audit again.

### 6.3 Execution Order Proof Test

When the ATS receives an audit report \( ar \), it does not only verify the compliance of the corresponding deployed composition to its user and provider requirements, but it can also verify its correctness to the required workflow. As this verification requires revealing workflow details (cfr. Algorithm 3), ATS performs the correctness check only if directly required by the edn-user. In such a case, she/he will be informed by EPApp that this verification process will let the ATS able to learn workflow related data. More precisely, these would be the workflow, the execution order token, as in Definition 4, and the secret key, \( SK \), that the end-user generated for that composition deployment. Having these elements, the ATS can execute Algorithm 3 to recompute the set of IDs of web services that did not respect the imposed workflow. Given this set, i.e., \( incorrectOrderList \), the ATS can punish the web services that did not respect the workflow. As this is considered as an equivalent to an unqualified invocation, the corresponding punishment happens by updating the reputation based on Equation 1. It is to be noted that the list \( incorrectOrderList \), as output by Algorithm 3, contains the IDs of the web services that were selected for an activity that does not match what is required in the workflow. Therefore, the ATS reconstructs the corresponding composition from the received \( ar \) and updates the reputation of all the services that invoked one of the those whose IDs are contained in \( incorrectOrderList \).

### 7. Experiments

We ran several experiments to demonstrate the efficiency of the proposed protocols. We measure the overhead, including time and space, consumed in validating user and provider requirement proofs in a private way, between two service providers. Our experiments are performed based on a simulated dataset. According to definitions 1 and 7, we randomly generate requirements of random numbers of clauses. For each clause, we keep on randomly generating the conditions. As such, the numbers of clauses and conditions are different in the simulated requirements for composition requests. We developed a program instance that simulates a web service, accepting a simulated composition request, and following the suggested protocols to generate the required provider proofs. Similarly, we developed an instance that simulates the operations of the ATS. The focus of our experiments is to study the performance, in terms of time and space usage, of the verification operations performed by the ATS and of the generation of the required proofs by providers. For each simulated composition request, we run the experiment 10 times, showing then the average overhead. All experiments have been run on a PC Intel Duo Core CPU 3GHz, 4GB RAM, and 64-bit Windows 7 Professional OS. We expect the overhead to be more optimal when the proposed protocols are deployed in real servers with a more powerful configuration.

In particular, as the key size of encryption algorithm and the digest size of hash function greatly impact the overhead of the cryptographic schemes, we ran different experiments by varying the size of the adopted keys. In addition to that, the number of services and the complexity of requirements (i.e., more specifically, the number of atomic conditions), affect the overhead as well. More precisely, we have assumed that user and provider requirements consist of a single DNF clause, but we varied the number of atomic conditions.

Regarding the user requirement proof verification by the ATS (i.e., running Algorithm 2), the most costly operation in Algorithm 2 is the call to the PrivateEvaluation function that refers to the corresponding algorithm in [8]. As such, when implementing and running Algorithm 2, the obtained results were closely similar to the ones achieved for the evaluation of user requirements at an involved web service side in [8].

### 7.1 Provider Proof Test

In this experiment, we measured the time and space used for provider proof generation and validation at service providers and ATS sides, respectively. As ECC algorithm is adopted in the protocol, we keep varying key sizes of ECC from 163, 283, 409 to 571 bits. We also increased the number of conditions from 25 to 100 with steps of 25. Figure 3-a shows the time of generating provider proofs at services side. In case of the largest key size (i.e., 571 bits) and the highest number of conditions (i.e., 100 conditions), the time overload is 2.007s. In case of the smallest key size (i.e., 163 bits) and the lowest number of conditions (i.e., 25 conditions), the time consumption is 0.182 s.

With the provider proof validation at ATS, we measure the time overhead and present the efficiency by Figure 3-b. In the worst case with the largest key size (i.e., 571 bits) and the highest number of conditions (i.e., 100 conditions), the time is 37ms. In case if the smallest key size (i.e., 163 bit) and the lowest number of conditions (i.e., 25 conditions), the time is 5ms.

We also measure the space load required by the ATS for the provider proof validation. The results are described on Figure 3-c. In the case of the largest key size (i.e., 571 bits) and the highest number of conditions (i.e., 100 conditions),
the space load is 28.41 KB. In the case of the smallest key size (i.e., 163 bits) and the lowest number of conditions (i.e., 25 conditions), the space load is of only 2.12 KB. We consider these to be very good results, especially that, as suggest in Section 6.2, we consider that the ATS keeps records of web services that it saw collaborating already and uses these records to directly fetch audit results. That is, without having to run the provider proof test again.

Figure 4-a shows the time used to compute an execution order token through involved services. In the worst case, with 20 services, the largest ECC key size (i.e., 571 bits) and the largest SHA digest size (i.e., 512 bits), the time is 116.04 ms. In the case of the smallest ECC key size (i.e., 163 bits), the shortest digest size (160 bits) and 5 services, the time is 2.65 ms.

Concerning the space load, we note that the size of the token does not change as it moves and as it is updated from web service to the other in deploying the composition. This is because, at the level of each web service, the token is hashed after it is updated. However, the token space load changes according to the ECC key size and the SHA digest size. As on Figure 4-b, the data load is 355 bytes in case of the largest ECC key size (i.e., 571 bits) and the longest SHA digest size (i.e., 512 bits). Whereas, the data load is 107 bytes in case of the smallest ECC key size (i.e., 163 bits), and the shortest SHA digest size (i.e., 160 bits).

8. Security Properties

The aim of our audit-based system is not to prevent collusion attacks, but to make sure that they are detected if they happen. Detecting collusion attacks is followed by a reputation-based punishment scheme, creating by this a mechanism that deters entities from colluding against the requirements of the system. The first security property of our system relates to guarantying that any collusion between web services in executing a CWSC request is detected and the concerned parties are accordingly punished. The second property of the system concerns the privacy preservation of the user and provider requirements throughout the auditing process. We formalize and prove the two properties in what follows.

8.1 Collusion detection insurance

We recall that audit reports are received by the ATS either deliberately from users reporting suspected collusion in a deployed composition they requested, or as an answer to the periodic audit cycle that the ATS performs. We also recall that the periodic audits are performed in time periods of value \( \tau \), and that they target all users in the system requesting all deployed compositions they have requested since the last previous audit cycle. We rely on the EPApp interface to answer the periodic audit requests from the ATS. Our system ensures that any collusion that happens in the deployment of a requested composition is detected after a time period of maximum \( \tau \). We formalize this as follows:

**Theorem 1** Let \( u \) be a user in the system and let \( \text{compList}_u \) be the list of all deployed composition requests made by \( u \). Let \( \tau \) be the period of the periodic audit cycles performed by ATS. Let \( \text{comp}_i \in \text{compList}_u \) be a composition that contains a collusion \( \text{col} \). The system guarantees that \( \text{col} \) is detected after maximum a time period \( t_{\text{max}} = \tau \) from the deployment of \( \text{comp}_i \).

**Proof.** Let \( \text{compList}(h) \) be the list of deployed compositions requested by user \( u \) between two periodic
audit cycles, $AC_{h}$ and $AC_{h+1}$ performed by the ATS. Let $comp_i \in compList(h)$ be a composition made at time $t_i \in [t_h, t_{h+1})$. Assume $comp_i$ contains a collusion. There are two possibilities: 1) user $u$ sends an audit report concerning the suspected $comp_i$, or 2) user $u$ does not report it. In the first case, the collusion will be detected by Algorithm 6.1 in place at ATS. Let us consider the case of the user not reporting. At time $t_{h+1}$, ATS runs the periodic audit and receives an audit report $\forall comp \in compList_i(h)$. That is, $comp_i$, containing the collusion, is audited and the collusion is detected at time $t_{h+1}$. Since the periodic audit cycle is of period $\tau$, $t_{h+1} - t_h \leq (t_{h+1} - t_h) \leq \tau$. Therefore, the collusion is detected at a maximum time $t_{\text{max}} = \tau$.

### 8.2 USER AND PROVIDER REQUIREMENTS PRIVACY PRESERVATION

Although ATS is considered as a trusted auditor, user requirements and provider requirements might contain private information and are not supposed to be revealed to it. As such, the system needs to guarantee the privacy preservation of these requirements at the ATS side when it is performing its audit. In addition to that, providers might be malicious adversaries, as they can intervene the information they receive to achieve as many benefits for themselves as possible. The framework in [8] addresses guaranteeing the privacy preservation of user requirements at providers side during the deployment of a composition request. As such, below we analyze the privacy preservation of both user and provider requirements at the ATS side.

Our system guarantees that the ATS cannot read user requirements and provider requirements when it is performing its audit. Regarding user requirement audit (i.e., UK double proof test), a temporary pair of public key and secret key are generated by the user, via the EPApp, and all the elements communicated to the ATS are secured with it. More precisely, user requirements are encrypted by the secret key $SK$ and several secret parameters based on [6], that are held by the user only. Even given the public key $PK$, providers and ATS cannot decrypt the secured user requirements. This ensures that user requirements are unreadable to ATS. Regarding provider requirements, all elements communicated by providers during the PR proof test to the ATS are encrypted using the user’s temporary public key. As such, ATS cannot learn the provider requirements as well. Besides, the adopted encryption algorithms are implemented with powerful key sizes that guarantee the security of the suggested protocols against attacks.

### 9. CONCLUSION

In this paper, we have suggested a privacy preserving audit-based system for aposteriori correctness control of CWSCs. More precisely, an auditor (i.e., the ATS) performs post control over compliance with the constraints imposed on a composition request, required by either users or involved providers, and over its alignment with the required workflow. We have adopted the framework we previously suggested in [8] as the underlying deployment protocol for choreographing CWSCs in a secure way. We have first suggested an extension of this framework to account for the verification of the execution order of a composition w.r.t its predefined workflow. Then, we have designed the newly suggested audit process, by exploiting and adapting the elements of the framework in [8], to offer a-posteriori control by which collusions against imposed requirements are detected. This audit process comprises both a periodic exhaustive audit and a mechanism to listen and react to reported complaints from end-users. The system punishes dishonest entities following a proper punishment-based reputation system that manages and maintains providers and users trust in its environment. We have proved that our proposal guarantees the detection of all dishonest behavior in the system, and that all the operations are assured in a secure way. Our experimental results also demonstrate that the suggested techniques perform well under varying conditions.

As future extensions of the work, we plan to consider other forms of workflows, other than sequential ones. Moreover, we plan to address the security of the workflow itself, such that it is followed without being fully revealed to intermediate entities.

### REFERENCES


Elena Ferrari is a full professor of Computer Science at the university of Insubria, Italy, where she leads the STRICT SociaLab and is the scientific director of the K&SM Research Center. Her research interests are related to various aspects of data management, including social networks and the social web, security, privacy, trust, and cloud computing.

Ngoc obtained BSc. degree (2004) and MSc. degree (2008) in Computer Science from University of Science, Ho Chi Minh City – Vietnam National University (HCMC-VNU), and PhD degree in Computer Science (2016) from University of Insubria, Italy. She has worked for Faculty of Information Technology (HCMC-VNU) since 2004. Her research interests are applied-cryptography, security and privacy in social network, mobile network, web/cloud service composition, and IoT.

Authors

Leila Bahri is a senior Ph.D. student at the university of Insubria, Italy. Her main work is on decentralized privacy preserving services for online social networks. She is a fellow of the iSocial Marie Curie ITN project.

Barbara Carminati is an associate professor in Computer Science at the university of Insubria, Italy. She visited several foreign universities as visiting researcher, among which: National University of Singapore, University of Texas at Dallas, Aristotle University of Thessaloniki and Tsinghua University, Beijing. Her main research interests are related to security and privacy for innovative applications, like XML data sources, semantic web, data outsourcing, web service, data streams and online social networks.