Contracts for Security Adaptation☆

J.A. Martína,*, E. Pimentelb

aE.T.S. Ingeniería Informática, Universidad de Málaga, Campus de Teatinos, 29071 Málaga, Spain.

Abstract

Security is considered to be one of the main challenges as regards the widespread application of Service Oriented Architectures across organisations. WS-Security, and its successive extensions, have emerged to fulfil this need, but these approaches hinder the loose-coupling among services, therefore constraining their reusability and replaceability. Software adaptation is a sound solution to overcome the incompatibilities in interface, behaviour and security constraints among stateful services. However, programming adaptors from scratch is a tedious and error-prone task where special care must be given to concurrency and security issues. In this work, we propose to use security adaptation contracts that allow us to express and adapt the security requirements of the services and their orchestration. Given a security adaptation contract and the behavioural description of the services (such as BPEL processes or Windows Workflows), we can generate the protocol of the orchestrator that complies with the security requirements (confidentiality, integrity and authenticity), while overcoming incompatibilities at the signature, behaviour and security QoS levels. The formalisation behind security adaptation contracts has other applications such as security policy negotiation and automatic security protocol verification.

Key words: model-based adaptation, Web Service orchestration, security specification, adaptation contracts, WS-Security

1. Introduction

Service Oriented Architectures (SOA) are composed of interoperable Web Services. However, Web Services (WS) are not always compatible, a fact which hinders both their reusability, and the development and maintenance of SOA systems. This is particularly important in stateful services with complex behaviour.


*Corresponding author

Email addresses: jamartin@lcc.uma.es (J.A. Martín), ernesto@lcc.uma.es (E. Pimentel)
(such as those described as BPEL processes [1] or Windows Workflows [2]) where any mismatch in the sequence of the messages exchanged may lead the composition to a deadlock situation. For instance, a missing operation in a service, a mismatch in the operation name or arguments, or an unexpected sequence of messages prevent the correct termination of the services involved.

Software adaptation [3, 4] is a sound solution which enables Web Services to interoperate despite their initial incompatibilities. This adaptation is achieved by deploying an adaptor, either as a set of wrappers or as a centric orchestrator, which is in charge of receiving, translating and rearranging the messages in the way expected by the destination service.

Adaptor design is a difficult task where the developer must take into account the behaviour of all the services and their possible interactions. In this process, subtle details may be missed, therefore resulting in an erroneous adaptation. We propose adaptation contracts as an abstract specification of the adaptation. These contracts will allow us to state clearly and concisely how to solve the incompatibilities among the services.

The adaptation of security-enabled messages, such as SOAP messages enhanced with WS-Security, comprises a new set of problems since the different parts of a message might be encrypted, signed or digested. In this case, security adaptors must be able to i) decrypt and verify some of these data on receptions, and ii) encrypt, sign and digest some other parts of the message as expected by the destination service.

In addition to single message security, security protocols such as those expressed by WS-SecureConversation or WS-Trust, impose further restrictions between the sequences of exchanged messages. These protocols might involve session keys (i.e., keys which are created, shared and used throughout the session), timestamps and nonces (which stands for number used once) to avoid replay attacks, and proof of possession mechanisms (where a service authenticates its identity by proving that it has certain information), among others. This manipulation of security-enabled messages is particularly difficult due to the sensitive nature of their contents so, depending on the particular system, security adaptors must be able to access some sensitive information (such as private keys), to generate new security tokens such as new keys and timestamps, and establish and assess trust relationships. All these new capabilities complicate even more the design of security adaptors.

In this work we present security adaptation contracts that not only address incompatibilities between services but also cover several security-related WS-* specifications in a high-level and integrated manner, hence reducing the effort required from the system architect. Security adaptation contracts enable us to specify how to adapt signature, behaviour and security incompatibilities among services; they describe the security checks that must be performed over the received
messages; and, due to their centric role in the conversation, they provide a formal framework to analyse the behaviour of the system.

Compared to other kinds of contracts, the adaptation contracts presented in this work are specifications on how the orchestration must proceed. These specification contracts are extended by security adaptation contracts, which are agreements over the security requirements between the service provider and the orchestrator. As contracts, security adaptation contracts are subject to be negotiated, but this aspect is not covered in this work. In terms of monitoring, security adaptation contracts represent the security policy that must be enforced on messages intercepted by the adaptor at run time. In the presence of security violations, the adaptor is in charge of taking the appropriate measures such as interrupt every communication with the compromised service and notify the other services in the orchestration. Security adaptation contracts can be used for dynamic adaptation in the sense that we can have several contracts previously stored in the system and, once we receive a session request, we could generate at run time the adaptor that complies with any of the contracts and the available services. However, our previous approach to dynamic adaptation does not support reconfiguration once the session has begun.

The main contribution of this paper is the inclusion of security concerns into Web Service adaptation contracts. These new contracts allow us to express the different adaptations required over the behaviour of the services for them to interoperate properly, as well as the security requirements that must be met during the communication. The purpose of the security requirements in adaptation contracts is to specify: i) the security checks that must be satisfied by every received message and their sequence in secure conversations, and ii) the transformations which must be effected on the security policy of a service so that it adapts to the security constraints of the system.

Incompatibilities among services make their composition impossible due to mismatches that disable the communication and lead the services to deadlock situations. These incompatibilities may be present at different levels:

**Signature level** is the level most covered by the literature and comprises mismatches in the name, arguments and types of the operations offered and required by service interfaces.

Example: A service which offers a different WSDL description than expected.

**Behavioural level** takes into account not only the signature of the operations but also the particular sequence in which those operations must be called or offered. Loops, internal choices (i.e., if-sentences) and external choices

\(^3\text{See }[5]\text{ for an approach to contract-based run-time adaptation.}\)
pick-sentences) are also covered by the behavioural specification of services. Incompatibilities at this level may lead the composition to undesirable states where one service expects a call from another service which is not able to comply, therefore the system reaches a deadlock.

Example: A BPEL process which waits for an invocation to operation A and then B would deadlock if composed with a service that first invokes B and then A, even though they have compatible WSDL signatures.

**Non-functional level** deals with other criteria than the specific behaviour of the service and it is often referred to as *Quality of Service (QoS)*. Some of the concerns of this level are availability, performance, reliability, scalability and security.

Example: A login message that is to be encrypted although it is sent as clear text by the other party.

**Semantic level.** At this level, services and their requirements are expressed by their meaning and not by a particular encoding. Incompatibilities at this level are present when a service designed for one purpose is used for a different one.

Example: To use a search engine service as a spell checker.

In previous work [6, 7] we covered contract-based adaptation up to the behavioural level, therefore we focus this paper on the inclusion of security QoS into such contracts.

The rest of the paper is structured as follows. In **Section 2** we present an example to motivate and illustrate how security adaptation contracts are used to orchestrate incompatible services in behaviour and security QoS. **Section 3** formalises security messages and how they are used by the behaviour of the services (Definition 2). Security specifications are composed of security primitives, and these primitives must preserve some properties to be sound with regard to security (e.g., avoid hash collisions). Sound security primitives are described in Definition 4.

Once security enabled services are defined, we proceed to formalise security adaptation in **Section 4**. Adaptation is enabled by *security adaptation contracts* (Definition 9), which are composed of contract terms (Definition 5). These contract terms are fundamental for security adaptors and will serve to match the messages sent and expected by the services (Definition 11). This matching will trigger the communication between the adaptor and any of the services, formalised in Definition 14. Definition 13 describes how the communication must pass the security checks expressed in the contract. We demonstrate how the behaviour of the adaptor is perfectly determined based on its current state and the exchanged data (Corollary 1 and Proposition 3 for input and output actions, respectively).
Then, we demonstrate the properties preserved by valid security contract terms in Theorem 1. Finally, we prove that the adaptor recognises if the security expressed in the contract is respected or violated (Theorem 2) when it communicates with any of the services.

Related work and possible applications are discussed in Section 5 and we conclude with some final remarks in Section 6. The formal demonstration of all the properties stated above is presented in Appendix A and Appendix B gives the intuition behind the automatic generation of adaptors from security adaptation contracts.

2. Motivational Example

Web services with security requirements present a tight coupling with the format, security algorithms and protocols used by the messages in their communications. In this section we illustrate a scenario where such restrictions prevent the proper communication among services and, in this way, we motivate the need for security adaptation contracts.

Example 1. In Figure 1, we present the behaviour of four services for performing secure shell operations (similar to the SSH protocol). Services \( a \) and \( a' \) try to get access to the functionality provided either by service \( b \) or \( b' \), but incompatibilities prevent proper communication. These behaviours are represented by labelled transition systems (LTS, [8]) where operation names are quoted and prefixed with ‘!’ and ‘?’ in output and input actions, respectively. Operations also have a list of arguments with security expressions. Internal operations (i.e., transitions without external communication) are represented with \( \tau \). In this example we shall focus on services \( a \) and \( b \).

On one side, service \( a \) (Figure 1(a)) can perform several requests (with its credential and request as arguments), which can be refused or followed by replies. It is important to highlight that the values for name and pass must remain constant throughout the session, so the same values must be sent in every iteration of the loop. Additionally, parameter nonce is used to correlate requests and replies while avoiding replay attacks.

On the other side, service \( b \) (Figure 1(b)) begins by notifying its availability with a proceed message. Then it must receive a login message with the credentials, which can be either accepted (another proceed) or denied. If the login is accepted, several requests can be made (with their results). Additionally, the service allows users to upload files.

These services are incompatible at signature (e.g., “refused” with “denied”), behaviour (e.g., unexpected “proceed”) and security levels (e.g., service \( a \) uses an encrypted password, requires the digest of the data, and correlates requests
Figure 1: Behaviour of services simulating different ways to perform secure shell operations
and replies with the argument *nonce*). Services *a* and *b* present complementary functionality and semantics but, due to these incompatibilities, security adaptation is required to make them to cooperate successfully.

Behavioural adaptation is achieved by deploying an adaptor in the middle of the communication with such behaviour that it receives, recomposes and forwards every messages it receives in a way that all the services can interact properly and end up in a stable state. The behaviour of such adaptors can increase exponentially with the complexity of the services involved and their design requires taking into account all the possible interleaving between the messages exchanged.

Therefore, we propose to describe adaptors with security adaptation contracts, which abstract away from concurrency issues and focus on the mapping between the operations, arguments and security of the services. This mapping is expressed as a set of vectors which correlate the operations of the services. These vectors use symbolic parameters in place of arguments and they contain security expressions to process, analyse and recompose the messages.

In addition, we might want to enforce some additional requirements over the adaptation such as “a particular message must not be sent more than *x* times” or “an operation *A* will be (un)available until the operation *B* is called”. These requirements constrain the application order of the interactions expressed by vectors. In order to represent such high-level requirements, adaptation contracts also include a vector-LTS (or VLTS, for short) which is a LTS with adaptation vectors as labels. If such restrictions are not required, the vector-LTS is considered to be a single initial and final state with all the transitions looping on it.

**Example 2.** There are several incompatibilities between services *a* and *b* which are solved by the adaptation contract *C₀* in Figure 2. First, the “proceed” messages sent from *b* are received by the adaptor due to vector *vₚ*. Operations have been prefixed with the service identifier. Vector *v₁* maps the *login* request at service *b* with the appropriate arguments coming from the first request of *a*. Symbolic parameters will be bound to the received values (e.g., *I*, *P*, *R* and *N* on the left hand side of *v₁*). Parameters with a superscript ‘∧’ will be replaced by a value already known by the adaptor, therefore *enc(K^∧, P)* indicates that the value that will be bound to parameter *P* is encrypted with a known key *K^∧*. This key, which must be known at the beginning of the session, is given in a initial adaptor environment *E₀*. This environment states that parameter *K* is a key, and *x* is its run-time value. Parameters *I* and *P* are used to compose the “*login*” message to be sent to service *b*. Vector *v₉* processes the *req* argument (in *R*) of the initial request. Once the first request has been fully received, subsequent requests can be mapped directly with vectors *v₉* and *v₁*. Arguments *name* and *pass* are checked to be always the same due to superscript ‘∧’. Nonces are updated and reused
\( V_0 = \{ \langle a:\textit{"request"}, I, \text{enc}(K^\wedge, P), R, N ; b:\textit{?"login"}, I^\wedge, P^\wedge), (v_1) \rangle, \langle b:\textit{!"proceed"}, (v_p) \rangle, \langle b:\textit{?"request"}, R^\wedge), (v_q) \rangle, \langle a:\textit{!"request"}, I^\wedge, \text{enc}(K^\wedge, P^\wedge), R, N ; b:\textit{?"login"}, I^\wedge, P^\wedge), (v_r) \rangle, \langle a:\textit{?"request"}, R^\wedge, (v_q) \rangle, \langle a:\textit{?"reply"}, D^\wedge, \text{hash}(D^\wedge), N^\wedge ; b:\textit{!"result"}, D), (v_d) \rangle, \langle a:\textit{?"refused"}, N^\wedge ; b:\textit{!"denied"}) \} (v_f) \rangle \}

(a) Set of security vectors for the contract

\[ V_0 \setminus \{ v_a, v_r \} \quad V_0 \setminus \{ v_q \} \quad C_0 = \langle V_0, \text{VLTS}_0, E_0 \rangle \]

where

\[ E_0 = \{ \{ \text{key}/K \}, \{ \text{x}/K \} \} \]

(b) \text{VLTS}_0 for the contract

(c) Adaptation contract \( C_0 \)

Figure 2: Adaptation contract for services \( a \) and \( b \)

accordingly in vectors \( v_1, v_r, v_d \) and \( v_f \). Vector \( v_r \) can conflict with \( v_q \) therefore we use the \( \text{VLTS}_0 \) (Figure 2(b)) to enforce that \( v_q \) is triggered only the first time. Vector \( v_q \) is guaranteed to be triggered first because of the need to send \( b:\textit{?"login"} \) at the beginning of \( b \). Figure 3 shows an adaptation protocol which complies with \( C_0 \) and services \( a \) and \( b \). The transition labels have been reduced to the characters underlined in \( C_0 \) and prefixed with the identification of the corresponding service.

The intuition behind contracts is that adaptation vectors are enabled by the current state of the VLTS and triggered by synchronisation of an operation of one of their sides with a service. Once a two-sided vector is triggered by one of its sides, the operation on the other side must eventually be synchronised before the behaviour of the adaptor reaches a final state. Vectors with only one operation can be used independently. Interleaving between vectors is allowed and the current state of the VLTS is updated as soon as a vector is triggered by any of its sides.

Once we have an adaptation contract and the behavioural description of the services, we use state-of-the-art techniques \([6, 9]\) for the automatic generation of the adaptor protocol. These techniques explore all the required interleaving among the messages enabled by the contract, prune those branches which lead the system to deadlock or livelock situations, and reduce the state-space of the protocol by removing duplicated and unnecessary paths. These techniques, however, do not support the adaptation of security concerns, yet they can be enabled by the security
adaptation contracts presented in this work.

2.1. The Role of Contracts

Security adaptation contracts concisely represent the mapping of the operations, arguments and security requirements among services. Adaptation contracts abstract away from concurrency issues, leaving them to the adaptor generation phase. This is especially important because adaptation protocols grow exponentially with the complexity, incompatibilities and interleaving among services. In fact, for finite service behaviours, adaptation protocols are potentially infinite in states and transitions whereas security adaptation contracts are not. For instance, Figure 3 has several dashed transitions which correspond to additional requests received before processing the previous one. These transitions represent a potentially infinite stack of pending requests, in contrast to the small size of the contract (Figure 2) and services (a and b in Figure 1) which resulted in this adaptor.

Additionally, adaptation contracts support high level restrictions expressed in vector-LTS. The level of abstraction of the contracts makes them versatile enough to cope with small changes in the behaviour of the services. Let us note that the same contract with different services can generate different adaptation protocols. For instance, any combination between services \( \{a, a'\} \times \{b, b'\} \) is supported by the same contract \( C_0 \) but not by the same adaptor (i.e., the adaptor in Figure 3 does not work for \( \{a, a'\} \times b' \)). Finally, security adaptation contracts are subject to be negotiated among service providers, but this is out of scope in this work.

2.2. Methodology for Security Adaptation

Our methodology for behavioural Web Service adaptation involves the following steps:

1. First, we abstract the behaviour of the services to our formal model. This model allows us to represent the sequence of messages sent and expected by a service and the order in which they must occur. This is done automatically from the public description of the behaviour of the services written in abstract BPEL or Windows Workflow. Security can be extracted from WS-Security messages or WS-Policy specifications.

2. Then, taking the behaviour into account, we must design (either assisted with a CASE tool [10] or automatically [11]) an adaptation contract able to solve all of the incompatibilities among the services. At this stage, we can do static validation of the contract (i.e., the contract is well defined) and, although there are not any data nor the concrete adaptor to test the adaptation,

---

2 See [7] for a discussion about the negotiation of adaptation contracts among behavioural services without security.
Figure 3: Adaptation protocol for contract $C_0$ and services $a$ and $b$
we still can perform symbolic simulation over the contract by hand-picking which adaptation vectors to apply at every moment. In addition, using this simulation, we can do symbolic model-checking based on the contract and the model of the services.

3. Using the contract and the behaviour of the services, the protocol of the adaptor is generated while taking into account all possible interleaving among messages. The resulting adaptor conforms to the given contract, it is encoded into a particular implementation language (currently BPEL), and it can finally be deployed as a single orchestrator [9] or distributed [12] in a set of wrappers over the services.

The goal of this methodology is to generate correct adaptors which are: i) secure, i.e., they enforce the security policies expressed in the security adaptation contract and security-enabled services; ii) non-intrusive, since adaptors do not alter the internal behaviour of the services and, in fact, services can be oblivious to adaptation; and iii) transparent, because adaptors should support every possible interaction which complies with the security policies of the contract and services and does not cause deadlock or livelock situations.

It is worth observing that the whole methodology is supported by our toolbox called ITACA [6], which is enhanced with hierarchical adaptation, simulation and verification capabilities over the entire composition. ITACA supports behavioural adaptation but it is yet to be extended with security concerns.

3. Web Services with Security QoS

Security concerns over Web Services are regarded as a major research challenge for service oriented computing [13]. The loose coupling feature of WSs, which enables their higher reusability and interoperability, is often constrained by new security requirements over those services. If we assume that the services we want to orchestrate are incompatible at signature or behavioural levels, and therefore the messages sent and expected are incompatible, we will have to deal with the mismatches among the security elements in those messages. For instance, a password that is to be encrypted but it is sent as clear text, some data which must be sent with their hash values in order to check their integrity but those digests are missing, a nonce that it is sent to correlate a request with its reply but that nonce is not returned in the reply, or something that must be signed but it is not.

In this section, we formalise stateful services with security requirements over their messages. This will allow us not only to adapt security-enabled services but also to express in a concise manner the security requirements for the whole orchestration and to analyse security properties over the expected behaviour of the system in [Section 4].
3.1. WS-Security

Due to the system-independent nature of Web Services, several specifications have emerged to express security requirements and properties over WSs and their orchestration. The specifications which are relevant for this work are briefly described below:

**WS-Security** describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. These mechanisms can be used to accommodate a wide variety of security models and encryption technologies. Security tokens are one of the main concepts in WS-Security. Security tokens are sets of claims (such as user names, keys or certificates) used for encryption, signature or authentication. WS-Security enables SOAP messages to describe the security transformations required to decrypt, authenticate and verify the different parts of the message by the intended recipient, therefore allowing end-to-end security. **Listing 1** presents an extract of a SOAP message with WS-Security elements.

**WS-Trust** defines how to issue, renew and validate security tokens that will be used to establish, assess and broker trust relationships among entities. They define an entity called Security Token Service that covers all the operations required for handling security tokens. The Security Token Service provides a trusted third party which asserts trust relationships among entities. Actual implementations are encouraged to make use of existing solutions that are compatible with this specification such as Kerberos or X.509 public-key certificates.

**WS-SecureConversation** is yet another extension to WS-Security which defines Security Context Tokens. These tokens, which are handled by the mechanisms described in WS-Trust, are used as shared secrets for establishing secure sessions (i.e., secure conversations). Additionally, new keys can be derived by both parties from the shared Security Context Token. With WS-SecureConversation, the security constraints described with WS-Security go beyond individual messages and it is possible relate several messages in the same conversation.

**WS-Policy** specifies an XML-based language to express the different security policies that are allowed and offered by a given entity. It is a flexible way to describe what kind of WS-Security structures are supported by the WS in the exchanged SOAP messages. For instance, it serves to specify all the different encryption algorithms that are supported by the service provider over the different parts of the SOAP message.
Listing 1: WS-Security enabled SOAP message

1  <Envelope><Header>
2     <Security>
3         <Timestamp Id="T0">... </Timestamp>
4         <BinarySecurityToken ValueType="#X509v3"
5             Id="X509Token">...
6         </BinarySecurityToken>
7         <EncryptedKey>...
8             <ReferenceList>
9                 <DataReference URI="#enc1"/>
10             </ReferenceList>
11         </EncryptedKey>
12     <Signature><SignedInfo>...
13         <Reference URI="#T0">...
14             <DigestValue>LyLsF094Pi4wP... </DigestValue>
15             </Reference>
16         <DigestValue>LyLsF094i4wPU... </DigestValue>
17             </Reference>
18     </SignedInfo>
19     <SignatureValue>HplZkmFZ/2kQ... </SignatureValue>
20     <KeyInfo>
21         <SecurityTokenReference>
22             <Reference URI="#X509Token"/>
23         </SecurityTokenReference>
24     </KeyInfo>
25     </Signature>
26  </Security>
27  </Header>
28  <Body Id="body">
29      <EncryptedData Id="enc1">... </EncryptedData>...
30  </Body></Envelope>
Example 3. Listing 1 presents a SOAP message composed of: a timestamp (line 3); an X.509 certificate (lines 4-6); an encrypted key (\texttt{#encl} at lines 7-11); the signature of the timestamp and the body of the message (lines 12-26), which is signed with the key given in the previous certificate (referenced in lines 21-25); and the body encrypted with the \texttt{#encl} key (lines 29-31).

All these WS-* specifications must be known and considered by the architect in order to design a proper orchestration that is able to comply with the security requirements of all the services involved. Moreover, the mechanisms presented in WS-* security specifications do not make the communication secure. A SOAP message with WS-Security elements can specify that certain part is encrypted but the encryption key can be also sent as clear text, for instance. WS-* security specifications do enable security concerns but they can be misused so careful design, analysis and verification are required to ensure certain security properties over the whole orchestration.

Figure 4 shows several ways of deploying Web Services which use WS-Security SOAP messages. The most common way (WS 1) involves the definition of the service without security, only its signature and behaviour are described. If the service provider offering that WS has a WS-Policy defined over the SOAP messages exchanged by that service, the provider must have WS-Security descriptions to comply with that WS-Policy. These WS-Security descriptions of the SOAP message are part of the configuration of \textit{security interceptors} that capture all SOAP messages and recompose them according to those WS-Security descriptions. These
security interceptors require security token services (either local to the service provider or as a third party service) which provide them with WS-Trust mechanisms. The first services that were deployed with WS-Security SOAP messages did not have engines with security interceptors, therefore WS-Security was hardwired within the service logic (WS 2). Finally, there are services without any security (WS 3) that might want to cooperate with security enabled engines. In this scenario, security adaptors could be used as 1) security interceptors, 2) to adapt Web services with incompatible WS-Security descriptions, and 3) to support services without security capabilities.

3.2. Service Behaviour with Security Specifications

WS-Security defines XML elements for keys, encryption, signature and certificates. This functionality can be abstracted from implementation details to a small set of security primitives. These primitives will be used in our model for service behaviour that will integrate the signature, behaviour and security properties of Web services. Therefore, WS-Security specifications over SOAP messages will be encoded into the labels of service behaviours. We refer to these security-enabled labels as security specifications.

Definition 1 (Security Specification). A security specification is an expression consisting of security constructors, atoms and types.

\[
S \in \text{SecSpec} ::= a \in \text{Atom} \quad \text{% Operation name} \\
| p \in \text{Type} \quad \text{% Argument types} \\
| \text{hash}(S) \quad \text{% Digest} \\
| \text{pk}(S) \quad \text{% Asymmetric key pair} \\
| \text{enc}(S, S) \quad \text{% Symmetric encryption} \\
| \text{penc}(S, S) \quad \text{% Asymmetric encryption} \\
| \text{cat}(S, \ldots, S) \quad \text{% List}
\]

Constructor \(\text{pk}\) represents the public key of a given private key. Constructor \(\text{hash}\) represents the digest of any given data. Symmetric encryption is covered by the \(\text{enc}\) constructor and, analogously, constructor \(\text{penc}\) is used for asymmetric encryption. Lists are also allowed through \(\text{cat}\) but this constructor will be considered implicit throughout the rest of the paper. Atoms (\(\text{Atom}\)) within a security specification are used as message selectors. They serve to express the operation name of the message but we allow them to be in several parts of the specification and nested within security constructors, hence we allow the operation name to be
subject to any security operation, such as encryption or digest. The elements of
*Type* characterise the abstract type of the run-time values they represent. *Type*
could be actual implementation types, semantic annotations or any other type
identifier.

The semantics behind security constructors determine whether their arguments
can be accessed or not. The *cat* constructor does not obfuscate its arguments
whereas the arguments of *pk* and *hash* cannot be accessed. We assume that knowing
a private asymmetric key/value it is possible to obtain its public pair/digest, but
not the opposite. Constructors *enc* and *penc* have their corresponding inverse
operations that, when evaluated with the right keys, return the message.

\[
\begin{align*}
[jac(k, enc(k, m))] &= m \\
[pdec(pk(k), penc(k, m))] &= m \\
[pdec(k, penc(pk(k), m))] &= m
\end{align*}
\]

In this way, \([\cdot]\) represents the value of a security expression.

The security specification which represents the pair of an asymmetric key is
given by \(pair : \text{SecSpec} \rightarrow \text{SecSpec}\).

\[
\text{pair}(S) \doteq \begin{cases} 
S' & \text{if } S = pk(S') \\
kr{pk}(S) & \text{otherwise}
\end{cases} \quad \text{(pair)}
\]

**Example 4.** The WS-Security message in [Listing 1] complies with the following
security specification.

\[
S_0 = \ tstamp, \\
\quad \text{info, pk(akey), penc(akey, hash(info, pk(akey)))}, \\
\quad \text{enc(key, key),} \\
\quad \text{hash(tstamp), hash(body),} \\
\quad \text{penc(akey, hash(hash(tstamp), hash(body))),} \\
\quad \text{enc(key, body)}
\]

In this example, no operation name (i.e., atom) is present as it was omitted in the
WS-Security code. The different elements of the list have been indented to see
their corresponding parts in [Listing 1]. It is worth noticing that the X.509 certificate
has been expanded to be able to reference its components: information about
the certificate (info), the certified public key (\(pk(akey)\)) and the signature of
the previous elements by a certification authority (\(penc(akey, hash(info, pk(akey)))\)).
This signature consists of the encryption of the hash value of signed elements (the
information and the public key) with the private key of the certification authority.
A similar structure is found in the signature of the hash values of the body and the timestamp. This notation abstracts implementation details (e.g., the specific algorithms applied) but it manages to represent the different parts of security tokens such as certificates and signatures.

We model service behaviour as a LTS \[^8\] restricted as follows.

**Definition 2 (Security-Enabled Service Behaviour).** A security-enabled service behaviour is formally defined as a LTS \(\langle \Sigma, O, s_0, F, T \rangle\) where: \(\Sigma \subseteq \{\tau\} \cup \{!, ?\} \times \text{SecSpec}\) is an alphabet which corresponds to the set of labels associated to transitions, \(O\) is a set of states, \(s_0 \in O\) is the initial state, \(F \subseteq O\) are final states, and \(T \subseteq (O \times \Sigma \times O)\) are the transitions.

The labels in \(\Sigma\) are either internal transitions (\(\tau\)) or communication transitions that contain an initial ‘!’ or ‘?’ if they are output or input actions, respectively, followed by an element of \(\text{SecSpec}\). For each service we will assume a unique service identifier \(\text{serv} \in SId\), and \((s, l, s') \in T\) will be denoted by \(s \xrightarrow{l}_{\text{serv}} s'\).

This model was chosen because it is simple, graphical, and it can be easily derived from existing implementation languages (see for instance \[^{14,17}\] where such abstractions for Web services were used for verification, composition or adaptation purposes). Alternatively, we could model services using process algebras such as CCS \[^{18}\] or a subset of the \(\pi\)-calculus \[^{19}\] as some of the authors did in previous work \[^{7,20}\]. However, we do not need their expressiveness in this work so we represent services with simpler LTSs.

**Example 5.** The services in Figure 5 are an evolution of previous examples with a more elaborated behaviour and a new service \(\text{ca}\) which plays the role of a Security Token Service that provides the system with the means to generate session keys. Services \(c\) and \(d\) maintain most of the incompatibilities mentioned in Section 2 and have some new ones related to their incompatible restrictions in security. For instance, service \(c\) uses a previously shared key \(\text{key}\) for symmetric encryption of requests and replies whereas \(d\) requires a Security Context Token (the argument of the \text{login} message), which contains an encrypted session key. Furthermore, the replies expected by \(c\) must be encrypted (even the operation name) and the integrity of the data confirmed with their digest. On the other hand, \(d\) uses nonces to avoid message duplication attacks, i.e., it expects to receive random data in every request and it replies with that same data so requests and replies can be correlated. Service \(\text{ca}\) must receive the identification of the service which wants to start a secure conversation followed by a password and the identification of the destination service, both encrypted with the public key of \(\text{ca}\). Then, it returns two tokens encrypted with its private key, one for each service, which contain the new session key encrypted with the symmetric keys of the source and destination services (keys previously known to \(\text{ca}\) ).
Figure 5: Behaviour of some security-enabled services

(a) Service c

(b) Service d

(c) Security token service ca
At run time, the values exchanged between the adaptor and the services are defined as Data, which must comply with their corresponding security specifications in the service behaviour. Data is defined as lists of binary information and atoms.

\[ D(\in Data) ::= b \in \text{Binary} \mid a \in \text{Atom} \mid (D_1, \ldots, D) \]

Let us recall that atoms are used as message selectors, e.g., operation names.

In order to compose security-enabled messages, we can evaluate any expression composed of security primitives and Data. These executable expressions are called security expressions.

**Definition 3 (Security Expression).** A security expression is a security specification with decryption primitives (dec and pdec) where all the types are replaced by data and it complies with the following grammar.

\[ X(\in SExp) ::= D \in \text{Data} \mid \text{hash}(X) \mid \text{pk}(X) \mid \text{enc}(X,X) \mid \text{cat}(X,\ldots,X) \mid \text{dec}(X,X) \text{ Symmetric decryption} \mid \text{penc}(X,X) \text{ Asymmetric encryption} \mid \text{pdec}(X,X) \text{ Asymmetric decryption} \]

Security expressions can be computed using function \([\cdot] : SExp \rightarrow Data\).

Let us note that security primitives might have collisions in the following scenarios: i) two different values which generate the same digest, ii) two private asymmetric keys which share the same public key, and iii) two different encrypted messages which correspond to the same clear text when decrypted with the same key. We refer to security primitives without collisions as sound security primitives.

**Definition 4 (Sound Security Primitives).** Security primitives \(pk\), \(penc\) and \(hash\) are considered sound security primitives if:

\[ [pk(D_1)] \neq [pk(D_2)] \quad (1) \]
\[ [hash(D_1)] \neq [hash(D_2)] \quad (2) \]
\[ [penc(D_0,D_1)] \neq [penc(D_0,D_2)] \quad (3) \]

for all \(D_0, D_1 \neq D_2 \in Data\).
4. Security Adaptation

We use security contract terms to represent the input and output actions in the adaptor of WS-Security enabled messages coming from/to the services. These terms are used to receive and process messages, or to compose and send messages from the adaptor, and they are combined in a security adaptation contract.

Security adaptation contracts allow to: i) express the security checks that must be performed by the adaptor; ii) describe how to decompose and recompose the messages that must be transformed by the adaptor while preserving the security restrictions of the services; iii) define security constraints among sequences of messages, such as secure sessions or security protocols; iv) perform analysis over the security of the resulting orchestration against several attacks; and v) retain the ability to adapt signature and behavioural incompatibilities.

Behavioural adaptation involves receiving the messages and sending them at the moment and with the structure expected by the intended recipient. This recomposition of messages is achieved by symbolic parameters, which specify how the data is received and how the data must be restructured before being sent. However, in security enabled WS, reception and emission of the WS-Security messages is more complex as these messages might need integrity, confidentiality and authentication over parts of the message or all of the message. Therefore, the adaptor must be capable of verifying that the messages received comply with the security policy of the sender; it must decrypt those parts of the messages that must be recomposed to match the structure accepted by the receiver; and finally send the messages encrypted and authenticated as expected by the partner.

Example 6. Figure 6 shows the security contract which solves the example in Figure 5. At the beginning of the contract, vector \( v_{tok} \) receives the first request from service \( c \) which contains its service identifier \( (S) \), its password \( (P) \), which is encrypted by a symmetric key \( C^{^\wedge} \) that must be previously known by the adaptor), and the actual request \( (R) \). For the adaptor to successfully interact with the Security Token Service \( ca \), it must know in advance the service identifier of service \( d \) (parameter \( T^{^\wedge} \)) and the public key of \( ca \) \( (PK^{^\wedge}) \); both of them are used in the left-hand side of \( v_{tok} \) to request context security tokens \( (O\text{ and } penc_{d}(PK^{^\wedge},enc(C^{^\wedge},K))) \). The purpose of these context security tokens is to distribute a shared key (authenticated by \( ca \)) between services \( c \) and \( d \). These context security tokens are encrypted with the private asymmetric key of \( ac \) (and therefore authenticated) and contain the shared key \( (K) \) to be used as session key. This session key is also encrypted with symmetric keys previously shared between each service and \( ca \). These context security tokens are received in the right-hand side of \( v_{log} \). One of them \( (O) \) is forwarded to service \( d \) in the other side of the vector, while the other will be used by the adaptor on behalf of service \( c \), which does not support Security Token Services, to obtain the session key \( K \).
\[ V_1 = \{ \]

\[ \langle \text{ca:}?”reqToken”, S \wedge, \text{penc}_c(PK \wedge, P \wedge, T \wedge) ; \text{c:}?”request”, S, \text{enc}(C \wedge, P), R \rangle, \] (v\text{tok})

\[ \langle \text{ca:}!”reqToken”, \text{penc}_d(PK \wedge, \text{enc}(C \wedge, K)), O \wedge; \text{d:}?”login”, O \wedge \rangle, \] (v\text{log})

\[ \langle d:!”proceed”) \rangle, \] (v\text{proc})

\[ \langle \text{d:}?”request”, R \wedge, \text{enc}(K \wedge, N \wedge) \rangle, \] (v\text{req1})

\[ \langle c:?”reply”, D \wedge, \text{hash}(D \wedge) \rangle; \text{d:}!”result”, D, \text{enc}(K \wedge, N \wedge) \rangle, \] (v\text{req2})

\[ \langle c:!”request”, S \wedge, \text{enc}(C \wedge, P \wedge), R \wedge; \text{d:}?”request”, R \wedge, \text{enc}(K \wedge, N \wedge) \rangle, \] (v\text{req})

\[ \langle c:!”refused”; \text{d:}!”denied”) \rangle, \] (v\text{ref})

\[ \langle c:!”quit”) \} \]

(a) Set of security vectors for services c, d and ca

\[ V_1 \setminus \{v_{\text{req1}}\} \quad V_1 \setminus \{v_{\text{req1}}\} \quad \{v_{\text{req1}}\} \quad \{v_{\text{req1}}\} \]

(b) \(VLTS_1\) for the security contract

\[ C_1 = \langle V_1, VLTS_1, E_1 \rangle \]

where

\[ E_1 = \langle \sigma, \theta \rangle \]

\[ \sigma = \{ \text{akey}/PK, \text{key}/C, \text{name}/T \} \]

\[ \theta = \{ x/\text{PK}, y/C, z/T \} \]

(c) Security adaptation contract \(C_1\)

Figure 6: Security adaptation contract for services \(c\) and \(d\)
Then, in $v_{req_1}$, the request contained in $R$ (which was previously received in $v_{tok}$) is sent to service $d$. This request must have an attached instantiated nonce ($N^*$ which is also encrypted with the session key ($K^\wedge$, received in the previous context security token) as expected by the destination service. This nonce must be the same as the one received in the reply ($N^\wedge$ in $v_{res'}$). The data coming from this reply ($D$) will be encrypted and digested ($enc(C^\wedge,"reply",D^\wedge,hash(D^\wedge))$) before being sent to service $c$.

From this point on (due to $VLTS_1$) all further requests proceed through $v_{req_2}$ which, apart from all the re-encryptions and nonces that must be dealt with, it also checks that the service identifier ($S^\wedge$) and password ($P^\wedge$) sent by $c$ are the same as the first request that started the communication.

The environment $E_0$ in Figure 6(c) is the initial environment for the adaptor corresponding to contract $C_1$. The notation of the substitutions imply that parameters on the right-hand side of the slash are replaced by whatever is on the left-hand side. Parameters $PK$, $C$ and $T$ are the public key of $ca$, the private key of service $c$ (needed for decrypting its requests and encrypting the replies in $v_{tok}$, $v_{req_2}$ and $v_{req_3}$), and the identifier of service $d$, respectively. In substitution $\theta$, the elements $x,y,z \in Data$ represent binary data corresponding to the stored values referenced by the parameters.

Security adaptation can be achieved using a security adaptor either in a transparent way or as a visible third party in between the communication, although the former implies that the adaptor must know some sensitive information to perform its task. If the adaptor is transparent to the WSs, it must know some of their private security tokens in order to recompose messages reusing those same tokens. If the adaptor is perceived as a third party, it has its own security tokens, and trust must be handled (through WS-Trust, for instance) between the adaptor and the WSs. Either way, the adaptor must be enabled with security token generation capabilities as it might need to generate session keys, timestamps and signed data to be able to adapt the conversation among the services. The adaptor works as a proxy or central orchestrator that acts as a gateway among different WS-Policies. The security adaptors presented in this work support both transparent and opaque approaches.

Let us focus on single input or output actions. These are not straightforward when security primitives are involved since we have to use them differently depending on the direction of the message. In output actions, only constructors are required to compose the message but, in input actions, both constructors (for comparison against known data) and their inverse operations (for decomposing the message) are needed. Following the design of WS-Security and the model presented in Section 3, in order to simplify the comparison between contract terms and security specifications we are only going to use constructors in contract terms. In input actions, $pk$ and $hash$ cannot be deconstructed, $cat$ does not obfuscate its
contents and the first argument of \textit{enc} specifies unambiguously how to decrypt it, therefore the inverse operation of their constructors can be inferred. However, due to the asymmetric nature of public-key cryptography, we have to annotate whether the specified asymmetric key must be used for decryption or encryption. This will be annotated as subscripts of the asymmetric encryption constructor, \textit{penc}_d and \textit{penc}_c respectively (see Definition 5).

4.1. Security Contract Terms

Encoding WS-Security descriptions into elements of \textit{SecSpec}, we gain the types of the message (e.g., which part of the message is a key, is encrypted or is a hash value) but we lose all the references between the elements of the message (such as which part of the message contains the key which encrypts another message or where is the digest of a given argument). However, we retain this capability using symbolic parameters in security adaptation contracts. Symbolic parameters enable us not only to relate arguments among different services but also to relate arguments of the same service, even within the same message (which is useful for relating arguments with their signatures or hash values). Security specifications where all the types are replaced by symbolic parameters (or parameters, for short) are called contract terms.

\textbf{Definition 5 (Contract Term).} A \textit{contract term} is a security specification where all the types are replaced by symbolic parameters.

\[ T \in CTerm ::= a \in \text{Atom} \mid P \in \text{Param} \mid \text{hash}(T) \mid \text{pk}(T) \mid \text{enc}(T,T) \mid \text{cat}(T, \ldots, T) \mid \text{penc}_c(T,T) \mid \text{penc}_d(T,T) \]

Within contract terms, constructors for asymmetric encryption have a subscript specifying whether it is going to be used for decrypting (\textit{penc}_d) or encrypting (\textit{penc}_c) messages. Function \( \text{pm} : CTerm \rightarrow \{ \text{Param} \} \) returns the set of symbolic parameters within a given contract term.

\textbf{Example 7.} The following contract term is used for composing secured messages which match with the WS-Security example in \textbf{Listing 1}.

\[ q_0 = T, \quad I, \text{pk}(S), \text{penc}_c(A, \text{hash}(I, \text{pk}(S))), \quad \text{enc}(K,L), \quad \text{hash}(T), \text{hash}(B), \quad \text{penc}_c(S, \text{hash}(\text{hash}(T), \text{hash}(B))), \quad \text{enc}(L,B) \]
Security specifications (like $S_0$) describe the security structure and types of the message whereas contract terms (such as $T_0$) relate each part of the message while respecting the structure imposed by security specifications. For instance, $S$ in $T_0$ is the parameter for the key in the X.509 certificate and it is used three times: once to specify the actual key, the second time to refer to its digest within the certificate, and finally to specify that the key is used in the signature of the digests of the timestamp and the body. Similarly, the encrypted key $L$ is used a second time to specify that the body of the message is encrypted with such key. Symbolic parameters $T$, $B$ and $I$ are referenced several times by their digests.

4.2. The Environment of the Security Adaptor

At run time, the adaptor receives messages and matches them against contract terms. The data in the messages are matched with symbolic parameters. These parameters will be used to reference these data. The adaptor has an environment to store these pairs of parameter-data. In addition, the values sent by the services comply with their corresponding security specifications in the behaviour of the services. When a parameter is matched at run time, both the value and its corresponding security specification are stored in the environment within the adaptor. Therefore, this environment is composed of two substitutions: $\sigma$, which partially replaces parameters with their corresponding security specification; and $\theta$, which partially replaces parameters with the data they represent. Substitution $\sigma$ will be used to transform contract terms into security specifications and $\theta$ will be used to evaluate contract terms into actual messages to be exchanged at run time.

**Definition 6 (Adaptor Environment).** An adaptor environment is a pair of partial substitutions $E = \langle \sigma, \theta \rangle$ of type $\sigma : \text{Param} \rightarrow \text{SecSpec}$ and $\theta : \text{Param} \rightarrow \text{Data}$. Both substitutions in an environment have the same domain

$$\text{dom}(\sigma) = \text{dom}(\theta).$$

We will denote by $\sigma_E$ and $\theta_E$ the corresponding substitutions $\sigma$ and $\theta$ in the environment $E$. We will denote as $\text{Env}$ the set of all possible environments.

Substitution $\theta$ is used for composing messages on output actions in the adaptor, and comparing received data against previously known data, whereas $\sigma$ is used to infer which transformations are allowed on the data, and to check that the security specification of composed messages is as expected by the receiving service.

The substitutions $\sigma$ and $\theta$ of the environment are updated with the operator ‘$\otimes$’. For instance, in $\sigma \otimes \sigma'$, substitution $\sigma'$ takes precedence over $\sigma$.

It is worth observing that environments might not be empty at the beginning of the protocol of the adaptor. The reason for this is that security adaptors might

---

Notice that substitutions are partial mappings on parameters. Thus, the domain of a substitution is composed of only those parameters which are actually replaced.
need some initial data (such as public keys) in order to engage the communication properly.

Contract terms can be transformed into security expressions (\(SExp\), in Definition 3) using an environment with enough information to substitute every symbolic parameter in the contract term with its corresponding data. Therefore, with the appropriate environment, contract terms can be evaluated into data using this transformation.

**Definition 7 (Contract Term Value).** The value of a Contract term \(T\) with regard to substitution \(\theta : \text{Param} \rightarrow \text{Data}\) such that \(\text{pm}(T) \subseteq \text{dom}(\theta)\), denoted by \(|T|_\theta\), is inductively defined as follows.

\[
\begin{align*}
|a|_\theta & = a \\
|P|_\theta & = \theta(P) \\
|pk(T_1)|_\theta & = \left[ pk(|T_1|_\theta) \right] \\
|\text{hash}(T_1)|_\theta & = \left[ \text{hash}(|T_1|_\theta) \right] \\
|\text{enc}(T_1, T_2)|_\theta & = \left[ \text{enc}(|T_1|_\theta, |T_2|_\theta) \right] \\
|\text{penc}(T_1, T_2)|_\theta & = \left[ \text{penc}(|T_1|_\theta, |T_2|_\theta) \right] \\
|\text{cat}(T_1, \ldots, T_n)|_\theta & = \left( |T_1|_\theta, \ldots, |T_n|_\theta \right)
\end{align*}
\]

where \(a \in \text{Atom}, P \in \text{Param}\) and \(T_i \in \text{CTerm}, i \in \{1, \ldots, n\}\).

Let us note that contract term values are used to compose messages, therefore asymmetric decryption (\(\text{penc}_d\)) is not required in the definition.

Symbolic parameters can be annotated to express how they are affected by security restrictions and the environment. Parameters that must be compared to a previously known value by the adaptor (such as nonces) will be annotated with ‘\(^\wedge\)’. These parameters are called known parameters. Contract terms that are used for sending messages from the adaptor typically have all their parameters annotated as known parameters, since the arguments of a message must be available in the environment before the message is sent. Those arguments whose value must be generated by the adaptor (such as new session keys and timestamps) will be replaced by parameters annotated with ‘\(^\ast\)’, we call these parameters instantiated parameters. All other unannotated parameters are called fresh parameters and their values will be updated in the environment when a matching message is received. Contract terms can also contain anonymous parameters (‘\_'’) that are used when the values received in these parameters are not needed for the rest of the communication, hence they are ignored. In short, anonymous parameters are

---

Example 7 is not complete since none of its parameters are annotated accordingly. However, Example 6 makes extensive use of parameter annotations.
the same as one-time-use fresh parameters and they will be dealt with as such for the rest of the paper.

Symbolic parameters (\(P \in \text{Param}\)) are evaluated by the substitutions in the environment to obtain their corresponding security specifications (\(\sigma(P)\)) and runtime values (\(\theta(P)\)). However, these substitutions can be restricted to annotated parameters with a superscript (e.g., \(\sigma^f\), \(\sigma^k\) and \(\sigma^*\) substitute only fresh, known and instantiated parameters, respectively). Function \(pm\) (in Definition 5) can also be restricted to annotated parameters in the same way.

4.3. Security Adaptation Contracts

Definition 8 (Security Adaptation Vector). A security adaptation vector (or security vector, for short) for a set of security-enabled services is an element of \(S\text{Vector} = VText \cup (VText \times VText)\) where \(VText = SId \times \{?,!\} \times CTerm\). Such a vector is noted \(\langle s:\!T \rangle\), \(\langle s:\?T \rangle\), \(\langle s:\?T ; s':\!T' \rangle\), or \(\langle s:\!T ; s':\?T' \rangle\) where \(s \neq s'\) are service identifiers and \(T, T'\) are contract terms for any security specification in their respective interfaces.

Definition 9 (Security Adaptation Contract). A security adaptation contract (or security contract, for short) of a set of services is a triple \(\langle V, VLTS, E_0 \rangle\) where \(V\) is a set of security vectors over the services, \(VLTS\) is a vector-LTS over the vectors of \(V\), and \(E_0\) is the initial environment of the adaptor which contains the initial security tokens required to perform the adaptation.

The adaptor must compose messages according to the security specification expected by the destination service. Analogously, the adaptor will receive and decompose messages according to the security specification expected from the sender. Messages are composed/received based on a given contract term, therefore we define first how to relate contract terms with security specifications by replacing parameters with elements of \(\text{SecSpec}\), and then we define when a given contract term is able to match a security specification using the previous relation.

Definition 10 (Contract Term to Security Specification). The security specification corresponding to a contract term \(T\) with regard to a substitution \(\sigma : \text{Param} \rightarrow \)}
SecSpec, denoted by $[T]_\sigma$, is defined as follows.

$$
[a]_\sigma \doteq a \\
[P]_\sigma \doteq \sigma(P) \\
[pk(T_1)]_\sigma \doteq pk([T_1]_\sigma) \\
[hash(T_1)]_\sigma \doteq hash([T_1]_\sigma) \\
[enc(T_1, T_2)]_\sigma \doteq enc([T_1]_\sigma, [T_2]_\sigma) \\
[penc_c(T_1, T_2)]_\sigma \doteq penc([T_1]_\sigma, [T_2]_\sigma) \\
[penc_d(T_1, T_2)]_\sigma \doteq penc(pair([T_1]_\sigma, [T_2]_\sigma) \\
[cat(T_1, \ldots, T_n)]_\sigma \doteq cat([T_1]_\sigma, \ldots, [T_n]_\sigma)
$$

where $T_i \in CTerm, i \in \{1, \ldots, n\}; a \in \text{Atom}; P \in \text{Param};$ and $\sigma : \text{Param} \rightarrow \text{SecSpec}.$

Notice that we used the appropriate pair of keys to generate the security specification in asymmetric cryptography primitives due to the duality $penc_c/penc_d$ in contract terms, which is not present in security specifications.

**Definition 11 (Contract Term Matching).** A contract term matches with a security specification, denoted by $S \vdash T$, if and only if a substitution $\sigma : \text{Param} \rightarrow \text{SecSpec}$ exists such that $[T]_\sigma = S$.

As the next result shows, if $S \vdash T$, the substitution $\sigma$ verifying $[T]_\sigma = S$ is unique.

**Proposition 1.** Given a contract term $T$ which matches a security specification ($S \vdash T$), if two substitutions $\sigma_1, \sigma_2$ enable this match ($[T]_{\sigma_1} = [T]_{\sigma_2} = S$), then

$$
\sigma_1(P) = \sigma_2(P)
$$

for all $P \in pm(T)$.

Thus, we will denote by $\sigma_{S,T}$ the unique substitution such that $S \vdash T$ and $\text{dom}(\sigma_{S,T}) = pm(T)$.

**Example 8.** Let $T_1$ and $S_1$ be a contract term and a security specification, respectively, given by:

$$
T_1 = \text{“reqToken”, penc}_d(PK,\text{enc}(C,K)), O \\
S_1 = \text{“reqToken”, penc} (\text{akey}, \text{enc}(\text{key}, \text{key})), \text{penc} (\text{akey}, \text{enc}(\text{key}, \text{key}))
$$

Then, we can easily derive that $S_1 \vdash T_1$ with the following substitution:

$$
\sigma_{S_1,T_1} = \{pk(\text{akey})/PK, key/C, key/K, penc(\text{akey}, \text{enc}(\text{key}, \text{key}))/O\}
$$
It is worth noticing how the constructor penc in \( T_1 \) is removed from its subindex, and its key \( (PK, \sigma_{S_1, T_1}(PK) = pk(akey)) \) is replaced with its private pair:

\[
pair([PK]_{\sigma_{S_1, T_1}}) = akey
\]

In this way, even though the contract term expressed which asymmetric key was necessary to decrypt the message, the same contract term is rewritten and compared with the security specification which composed that very message (\( S_1 \)).

4.4. Synchronisation between Security Adaptors and Web Services

Prior to the output of messages from the adaptor, it must be checked that all the parameters required to compose the message are available within the environment \( (E = \langle \sigma, \theta \rangle) \) and that their corresponding security specifications \( (\sigma) \) comply with those expected by the recipient. This comparison between a security specification and a contract term with a given environment is called synchronisation trigger as it will pose the condition to trigger security vectors.

**Definition 12 (Synchronisation Trigger).** A contract term \( T \) can be triggered by a security specification \( S \) in a given environment \( E = \langle \sigma, \theta \rangle \) when the security specifications in \( \sigma \) are in such state that all the known parameters of \( T \) can be substituted to match \( S \). That is,

\[
\sigma(P) = \sigma_{S, T}(P)
\]

for all \( P \in pm^\wedge(T) \).

Notice that a contract term \( T \) can only be triggered when it matches a security specification \( S \vdash T \). We will denote as \( S \triangleright_{\sigma} T \) such a trigger condition.

**Example 9.** Let us assume \( S_2 \in \text{SecSpec} \) and \( T_2 \in \text{CTerm} \) given by:

\[
S_2 = \text{"request", } req, \text{enc(key, nonce)} \quad \sigma = \{\}
\]

\[
T_2 = \text{"request", } _, \text{enc}(K^\wedge, N) \quad \sigma' = \{\text{key}/K\}
\]

then

\[
S_2 \not\vdash_{\sigma} T_2 \quad S_2 \triangleright_{\sigma'} T_2
\]

Receptions at the adaptor are triggered by [Definition 12] but, in addition, once all the data in the message are received, the adaptor must check that these data satisfy the restrictions imposed by the contract term that allowed the communication. If this security check fails, then the adaptor must evolve to an exceptional behaviour where specific actions should be taken to solve the problem (e.g., ignore every other communication with that service and restart its own behaviour). This check is performed recomposing the message with the expected data (atoms and known parameters) filling the gaps with the values received in fresh parameters and comparing this recomposed message to the one received.
**Definition 13 (Security Check).** Some data $D$ comply with the security checks expressed in a contract term $T$ with a substitution $\theta$ (denoted by $D \models_\theta T$) when $T$ can be evaluated to have the same value as $D$ using $\theta$. Formally,

$$a \models_\theta a \tag{=}$$

$$D \models_\theta P \quad \text{iff} \quad D = \theta(P)$$

$$D \models_\theta \text{pk}(T_1) \quad \text{iff} \quad [\text{pk}(T_1)]_{\theta} = D$$

$$D \models_\theta \text{hash}(T_1) \quad \text{iff} \quad [\text{hash}(T_1)]_{\theta} = D$$

$$D \models_\theta \text{enc}(T_1, T_2) \quad \text{iff} \quad [\text{dec}([|T_1|_\theta, D])]_{\theta} = T_2$$

$$D \models_\theta \text{penc}_c(T_1, T_2) \quad \text{iff} \quad [\text{penc}_c(T_1, T_2)]_{\theta} = D$$

$$(D_1 \ldots D_n) \models_\theta \text{cat}(T_1 \ldots T_n) \quad \text{iff} \quad D_i \models_\theta T_i, i \in \{1, \ldots, n\}$$

where $T_i \in \text{CTerm}, i \in \{1, \ldots, n\}; a \in \text{Atom}; P \in \text{Param}$.

When data $D$ pass the security checks of contract $T$, the substitution $\theta$ such that $D \models_\theta T$ is univocally determined by security primitives, if they do not present any collision. This is what is stated by next proposition.

**Proposition 2.** Given a fixed set of keys and sound security primitives, if $D \models_\theta T$ then $\theta$ is unique on $\text{pm}(T)$.

The environment of the adaptor evolves after every communication with any service. When messages are received and successfully validated, the environment is updated with the data received in fresh parameters. When messages are sent from the adaptor, any instantiated parameter must be stored in the environment as well. Any communication (input or output actions) between the adaptor and the services is called *synchronisation*.

An adaptor able to engage a security conversation must be able to generate some security tokens such as timestamps, signatures and session keys. This scenario is covered by WS-Trust and WS-SecureConversation where the adaptor could be a Security Token Service or it could use another service in the orchestration as its Security Token Service. Either way, *instantiated parameters* (i.e. parameters annotated with ‘∗’) in a contract term $T$ fall within this category of tokens. This instantiation process ($\text{inst} : \text{SecSpec} \times \text{CTerm} \to \{\text{Param} \to \text{Data}\}$) proceeds by associating new values to every instantiated parameter by means of new data substitutions ($\theta : \text{Param} \to \text{Data}$). Thus, for all $\theta \in \text{inst}(S, T)$, $\text{dom}(\theta) = \text{pm}^*(T)$. Only parameters referring to types can be instantiated, i.e., for all $P \in \text{dom}(\theta)$, $\sigma_{S, T}(P) \in \text{Type}$. It does not make sense to instantiate parameters which refer to structured security specifications (e.g., $P$ such that $\sigma_{S, T}(P) = \text{hash}(\text{info})$). If there are no instantiated parameters, function $\text{inst}$ returns a set with a single empty substitution.
**Definition 14 (Synchronisation).** A contract term $T$ triggered by specification $S$ transforms the current environment of the adaptor $\langle \sigma, \theta \rangle \in Env$ on output actions with $\Psi_{S,T} : Env \rightarrow \{(Data, Env)\}$, and input actions with $\Gamma_{S,T} : Data \times Env \rightarrow \{Env\}$.

$$ \Psi_{S,T}(\langle \sigma, \theta \rangle) = \{(|T|_{\theta^\prime} \circ \theta^\prime}, \langle \sigma \otimes \sigma_{S,T}, \theta \otimes \theta^\prime \rangle) \mid \theta^\prime \in inst(S,T) \} \quad (\Psi) $$

$$ \Gamma_{S,T}(D, \langle \sigma, \theta \rangle) = \left\{ \sigma \otimes \sigma_{S,T}, \theta \otimes \theta^\prime \right\} \left| D \models \theta^\prime \otimes \theta^\prime \langle T \rangle, \right. \quad \text{dom}(\theta^\prime) = pm^\prime(D) \quad (\Gamma) $$

**Example 10.** Given the following environment, security specification and contract terms, and being $x,y,y',z \in Data$:

$$ S_3 = \text{``request'', req, enc(key, nonce)} \quad T_3 = \text{``request'', R\^, enc(K\^, N\^)} \quad \text{dom}(\theta^\prime) = pm^\prime(D) \quad \text{iff } D \models \theta^\prime \otimes \theta^\prime \langle T \rangle $$

$$ \sigma = \{key/K, req/R\} \quad \theta = \{x/K, y/R\} $$

then

$$ (\langle T_3 |_{\theta^\prime} \circ \theta^\prime}, \langle \sigma', \theta^\prime \rangle \rangle) \in \Psi_{S_3,T_3}(\langle \sigma, \theta \rangle) $$

$$ (\langle \sigma', \theta^\prime \rangle) \in \Gamma_{S_3,T_3}(D, \langle \sigma', \theta' \rangle) \quad \text{iff } D \models \theta^\prime \otimes \theta^\prime \langle T_3 \rangle $$

where

$$ \sigma' = \sigma \otimes \{key/K, req/R, nonce/N\} \quad \theta' = \sigma \otimes \{z/N\} \quad \theta'' = \sigma \otimes \{y'/R\} $$

As it can be observed in the previous example, the environment transformation made by $\Gamma$ depends on how $\theta''$ satisfies the security check on $T_3$. Actually, this transformation is fully determined by data $D$ when security primitives behave as expected. In fact, in this situation the transformation $\Gamma$ is exactly determined for the received data and the environment, that is, the adaptor evolves to at most one environment. This is formalised in the following result, a direct consequence of Proposition 2.

**Corollary 1.** If the security primitives are sound then $|\Gamma_{S,T}(D, E)| \leq 1$ for a fixed set of keys.

In the case of output actions, the environment transformation $\Psi_{S,T}(E)$ returns as many environments as possible instantiations can be made over the parameters of $T$. That implies that, if there are no instantiated parameters in $T$, transformation $\Psi$ is exactly determined by a given environment $E$, that is, there is only one possible transformation, at most. This is stated by the following proposition.

**Proposition 3.** If a given contract term $T$ does not have any instantiated parameters (i.e., $pm^*(T) = \emptyset$), then $|\Psi_{S,T}(E)| \leq 1$. 

30
4.5. Validation of Contract Terms

Adaptation contracts are easier to design than programming adaptors from scratch. Adaptation contracts must be written with care [10], otherwise they might be invalid or result in empty adaptors. This is particularly important with security adaptation contracts since we are able to write contract terms that do not match their corresponding security specification (covered in Definition 11), or contract terms which contain parameters that cannot be matched on input actions because their values might be obfuscated by security. In order to prevent the latter we define the reachability of parameters and, at the same time, we define valid contract terms.

**Definition 15 (Parameter Reachability).** The function \( \text{reach} : \text{Param} \times \text{CTerm} \times \text{CTerm} \to \text{Boolean} \) returns whether a parameter \( P \) can be obtained from a message that complies with the given contract term or not.

\[
\text{reach}(P, T, T') =
\begin{cases} 
  \text{True} & \text{if } T = P \\
  \text{reach}(P, T_2, T') \land \forall P' \in \text{pm}^f(T_1), \text{reach}(P', T', T') & \text{if } T = \text{enc}(T_1, T_2) \\
  \text{reach}(P, T_2, T') \land \forall P' \in \text{pm}^f(T_1), \text{reach}(P', T', T') & \text{if } T = \text{penc}_d(T_1, T_2) \\
  \bigvee_{i=1}^n \text{reach}(P, T_i, T') & \text{if } T = \text{cat}(T_1, \ldots, T_n) \\
  \text{False} & \text{otherwise}
\end{cases}
\]

where \( T, T' \in \text{CTerm} \).

Function \( \text{reach} \) is used on the fresh parameters in input actions, therefore there is no need for including \( \text{penc}_c \) in the definition.

It is worth noting the role played by the third argument \( (T') \) which represents the whole contract term of which the second argument is a part. This argument allows us to look for fresh parameters in other parts of the contract term. In this way, even though keys might be represented by contract terms with fresh parameters, these keys can still be obtained (therefore allowing us to keep looking inside the encrypted message) if those fresh parameters can be reached in any other part of the message. Thus, we will define \( \text{reachable}(P, T) = \text{reach}(P, T, T) \).

**Example 11.** Being \( P \in \text{Param}, T \in \text{CTerm} \) then:

\[
\begin{align*}
\text{reachable}(P, \text{enc}(K, P)) &= \text{False} \\
\text{reachable}(P, \text{cat}(\text{enc}(K, P), K)) &= \text{True} \\
\text{reachable}(P, \text{hash}(P)) &= \text{False} \\
\text{reachable}(P, \text{cat}((\text{hash}(P), K, \text{penc}_d(pk(K), P))) &= \text{True}
\end{align*}
\]

31
In (4), parameter $P$ is within an encrypted message with an unreachable key, therefore the message cannot be decrypted and the parameter is unreachable. If the key is given in any other part of the message (5), parameter $P$ can be reached. Digests cannot be decomposed so $P$ cannot be reached in (6). Finally in (7), we can reach $K$ therefore we can obtain its public key and then we reach $P$.

We now define two notions of valid contract term over a given security specification $S$ depending on whether the given contract term $T$ is used for output (validSend) or input (validRec) actions in the adaptor. Contract term $T$ is considered valid only if $S \vdash T$. Additionally, on input actions, every fresh parameter must be reachable and, on output actions, every parameter must be annotated to be known or instantiated. Instantiations must assign values to parameters according to their types.

**Definition 16 (Valid Contract Term).** A contract term $T$ is considered valid with regard to a security specification $S$ and the direction of the action if:

- **validSend** ($T, S$) iff:
  \[
  S \vdash T, 
  \begin{align*}
  pm^f(T) &= \emptyset, \\
  \forall P \in pm^*(T). \sigma_{S,T}(P) &\in \text{Type}
  \end{align*}
  \] (validSend)

- **validRec** ($T, S$) iff:
  \[
  S \vdash T, 
  \begin{align*}
  pm^*(T) &= \emptyset, \\
  \forall P \in pm^f(T). \text{reachable}(P, T)
  \end{align*}
  \] (validRec)

**Example 12.** Given

- $S_4 = \text{enc(key, “reply”, data, hash(data))}$
- $T_4 = \text{enc(C, “reply”, D, hash(D))}$
- $T_5 = \text{enc(C, “reply”, D, hash(D))}$
- $T_6 = \text{enc(C, “reply”, D, hash(D))}$
- $T_7 = \text{enc(C, “reply”, D, hash(D))}$
- $T_8 = \text{enc(C, “reply”, D, hash(D))}$
- $T_9 = \text{hash(D)}$
then

\[
\begin{align*}
\text{validRec}(T_4, S_4) &= \text{True} & \text{validSend}(T_4, S_4) &= \text{True} \\
\text{validRec}(T_5, S_4) &= \text{False} & \text{validSend}(T_5, S_4) &= \text{False} \\
\text{validRec}(T_6, S_4) &= \text{False} & \text{validSend}(T_6, S_4) &= \text{False} \\
\text{validRec}(T_7, S_4) &= \text{True} & \text{validSend}(T_7, S_4) &= \text{False} \\
\text{validRec}(T_8, S_4) &= \text{False} & \text{validSend}(T_8, S_4) &= \text{True} \\
\text{validRec}(T_9, S_4) &= \text{False} & \text{validSend}(T_9, S_4) &= \text{False}
\end{align*}
\]

4.6. Behaviour of Adapted Systems

Being given a security contract term and a set of services we can generate the behaviour of the adaptor which complies with the contract and makes the orchestration to avoid deadlocks. The automatic generation of such adaptation behaviour has been addressed in related work ([4, 6, 7, 9]) but the approaches presented in these papers did not consider security adaptation. We argue that these approaches could be extended to support security adaptation contracts and we present the intuition behind this extension in the algorithm presented in Appendix B. This algorithm proceeds by generating every possible service communication allowed by the security adaptation contract, and then it selectively prune those controllable branches which present deadlocks or livelocks. Therefore, the adaptor is guaranteed to be transparent due to the exhaustive nature of the generation algorithm.

Definition 17 (Adaptor Behaviour). The behaviour of the adaptor is defined as a LTS \( \langle \Sigma_a, O_a, s_{t_0}^a, F_a, \rightarrow_{\text{adapt}} \rangle \) where \( \Sigma_a \subseteq V\text{Term} \) is the alphabet, \( O_a \) is the set of states, \( s_{t_0}^a \in O_a \) is the initial state, \( F_a \subseteq O_a \) are the final states, and \( \rightarrow_{\text{adapt}} \subseteq (O_a \times \Sigma_a \times O_a) \) is the labelled transition relation.

An adaptor behaviour is compliant with a security adaptation contract \( C = \langle V, VLTS, E_0 \rangle \) where \( V \subseteq \{ V\text{Term} \cup (V\text{Term} \times V\text{Term}) \} \) if: the labels of the adaptor behaviour are the vector terms (\( V\text{Term} \)) of \( C \); the sequences of transitions respect the restrictions imposed by the vectors (\( V \)) and \( VLTS \) in \( C \); the initial environment of the adaptor is \( E_0 \); and the orchestration among services and the adaptor do not present deadlocks or livelocks.

Being given an adaptor behaviour as defined above and a set of services, the orchestration behaviour (i.e., the behaviour of the adapted system) is characterised by a labelled transition system defined as follows.

Definition 18 (Orchestration Behaviour). The behaviour of the orchestration among a set of services \( \{ \text{serv}_i = \langle \Sigma_i, O_i, s_{t_0}^i, F_i, \rightarrow_i \rangle, i \in \{1, \ldots, n\} \} \) and an adaptor \( \langle \Sigma_a, O_a, s_{t_0}^a, F_a, \rightarrow_{\text{adapt}} \rangle \) is a LTS \( \langle \Sigma, O, s_{t_0}, F, T \rangle \) where the labels are \( \Sigma \subseteq \)
\begin{figure*}[t]
\centering
\begin{align*}
&\frac{s \xrightarrow{\text{serv}} s'}{s\langle a,E \rangle \xrightarrow{\tau} s'\langle a,E \rangle} \quad \text{(TAU)} \\
&\frac{s \xrightarrow{?\text{serv}} s', a \xrightarrow{\text{serv}!:\tau} \text{adapt} a', S \triangleright_{\sigma} \langle T, (D, E') \rangle \in \Psi_{S,T}(E)}{s\langle a,E \rangle \xrightarrow{D} s'\langle a', E' \rangle} \quad \text{(SEND)} \\
&\frac{s \xrightarrow{1\text{serv}} s', a \xrightarrow{\text{serv}?:\tau} \text{adapt} a', D \in \text{Data}, S \triangleright_{\sigma} \langle T, \forall P \in \text{pm}^{f}(T), \text{reachable}(P, T), E' \in \Gamma_{S,T}(D, E) \rangle}{s\langle a,E \rangle \xrightarrow{D} s'\langle a', E' \rangle} \quad \text{(REC)} \\
&\frac{s \xrightarrow{1\text{serv}} s', a \xrightarrow{\text{serv}?:\tau} \text{adapt} a', D \in \text{Data}, S \triangleright_{\sigma} \langle T, \forall P \in \text{pm}^{f}(T), \text{reachable}(P, T), \Gamma_{S,T}(D, E) = \emptyset \rangle}{s\langle a,E \rangle \xrightarrow{\text{fail}} s'\langle \text{except}(T, E, s, S, D, s', a) \rangle} \quad \text{(FAIL)} \\
&\begin{align*}
i &\in \{1, \ldots, n\}, \\
&s_i\langle a,E \rangle \xrightarrow{\alpha} s'_i\langle a', E' \rangle \\
\{s_1| \ldots |s_i| \ldots |s_n\langle a,E \rangle \} &\xrightarrow{\alpha} \{s_1| \ldots |s'_i| \ldots |s_n\langle a', E' \rangle \} \end{align*} \quad \text{(PAR)}
\end{align*}
\caption{Rules which define the orchestration behaviour of adapted systems}
\end{figure*}

34
The set of states is $O \subseteq O_1 \times \cdots \times O_n \times O_a \times Env$; the final states are $F \subseteq F_1 \times \cdots \times F_n \times F_a \times Env$; and the transition relation $T \subseteq O \times \Sigma \times O$ is inductively defined by the rules in Figure 7.

Let us note that the transitions of the orchestration are labelled with the exchanged data and the contract term and security specification which allowed the synchronisation. On $\tau$-transitions, the labels are $\tau, \varepsilon$ and $\varepsilon$, respectively. Security exceptions are labelled with $fail$ and the initial environment $E_0$ is given in the contract which generated the adaptor. Services are orchestrated with the adaptor as-they-are, therefore the adaptation is non-intrusive.

The transition relation of the orchestration behaviour is described in two stages: first we define the feasible interactions between a single service and the adaptor in rules $TAU$, $SEND$, $REC$ and $FAIL$, and then we define the orchestration among $n$ services in PAR rule. By abuse of notation we denote both interactions using the same symbol for the transition relation $\rightarrow$.

During synchronisations through rules $SEND$, $REC$ and $FAIL$, both the service and the adaptor must be in states $s$ and $a$ respectively, where one of the outgoing transitions satisfies the synchronisation trigger (Definition 12). Once the trigger condition is met, the synchronisation continues differently depending on the direction of the communication. In output actions (SEND) the security restrictions are guaranteed by construction ($S \triangleleft_\sigma E^T$ verifies that the security requirements of the message are covered by the contract term) so the environment is updated with instantiated parameters ($\Psi$) and the parallel composition of the service and the adaptor progresses. If the synchronisation is the result of a message being received at the adaptor (REC), the transformation of the environment ($\Gamma$) might fail if the received data do not comply with the security specification (FAIL). If the synchronisation fails, the adaptor evolves to an exceptional state (except), otherwise its environment is updated with the received data and the system progresses.

If the security primitives are sound, by Corollary 1, $\Gamma$ in REC returns only one environment for the received data. In output actions (SEND), as a consequence of Proposition 3, we could send as many messages as all the possible instantiations of the symbolic parameters in $pm^*(T)$.

**Proposition 4.** If $\{s_1|\ldots|s_i|\ldots|s_n\langle a, E \rangle\} \xrightarrow{D} \{s_1|\ldots|s_i'|\ldots|s_n\langle a', E' \rangle\}$ as a consequence of applying rule $SEND$ on $s \xrightarrow{serv} s'$ and $a \xrightarrow{serv=?T} adapt a'$, then $\sigma_{E'}(P)$ is a subexpression of $S$ for every $P \in pm^*(T)$.

**Proposition 4** implies that the security specification of the parameters used to compose messages, comply with the security expected by the destination service.
It is worth noticing that the behaviour defined in Figure 7 only allows successful communications between the services and the adaptor on valid contract terms. We understand as successful communications those where the adaptor has not reached an exceptional state, that is, except in rule FAIL. This is stated by the following theorem.

**Theorem 1.** The premises of rules REC and SEND imply that the contract term \( T \) governing the transition of the adaptor is a valid contract term.

Because of Theorem 1, every contract term that allows successful communications matches the security specification of the service.

In the following theorem we demonstrate that the environment of the adaptor is updated on successful communications (not FAIL) following the contract term matching for security specifications, and the security check for data.

**Theorem 2.** If \( \{s_1|\ldots|s_i|\ldots|s_n|\langle a,E \rangle \} \xrightarrow{T} \{s_1|\ldots|s'_i|\ldots|s_n|\langle d',E' \rangle \} \) then:

\[
\sigma_{E'} = \sigma_E \circ \sigma_{S,T} \\
\theta_{E'} = \theta_E \circ \theta_{E'}
\]

where \( \theta_{E'} \) satisfies

\[
D \models_{\theta_{E'} \circ \theta^k} T \quad \text{and} \quad \text{dom}(\theta_{E'}) = \text{pm}(T) - \text{pm}^\wedge(T)
\]

The intuition behind the previous theorem is that the data exchanged through the adaptor are verified with regard to the security checks expressed in the contract term which allowed the communication, and comply with the security specification expected by the service. Therefore, the adaptor is secure.

**Example 13.** Figure 8 contains the protocol with the successful transitions of the adaptor corresponding to contract \( C_1 \) (Figure 6) and services \( c, d \) and \( ca \) (Figure 5). At the beginning, the adaptor waits for either the quit or the request messages from service \( c \) which trigger vectors \( v_{\text{quit}} \) and \( v_{\text{tok}} \), respectively. On this initial request, vector \( v_{\text{req2}} \) is not trigger because of the restrictions imposed by \( \text{VLTS}_1 \). Then, the service \( c \) awaits a response and the adaptor processes the other half of vector \( v_{\text{tok}} \) and transforms the previously received request into a token request for \( ca \). Service \( ca \) responses and triggers \( v_{\text{log}} \) which results in a login message being sent from the adaptor to \( d \). Now the adaptor is waiting for service \( d \) which can either deny the service (this triggers \( v_{\text{ref}} \) and sends a refused message to \( c \)) or proceed, which is handled by \( v_{\text{proc}} \). In the latter case, the only option available due to the current
Figure 8: Adaptation protocol for contract $C_1$ and services $c$ and $d$ and $ca$
states of all the services and the adaptor is to use \( v_{req1} \) and to recompose the initial request received from \( c \) into a request understandable by \( d \) using a newly generated nonce \( (N^*) \) and the key \( K \) inside the token received from \( ca \). From that point on, the adaptor enters a loop where requests and responses between \( c \) and \( d \) are adapted using previously known information by means of vectors \( v_{req2} \) and \( v_{res} \) (nonces and data integrity are checked with \( N^\wedge \) and \( D^\wedge, hash(D^\wedge) \) and messages are encrypted appropriately) until service \( c \) decides to finish by triggering vector \( v_{quit} \).

5. Related Work

Several proposals \[9\, 21\, 22\] focus on solving behavioural incompatibilities among services using adaptation contracts. In \[21\], the authors presented an approach to behavioural adaptation based on a set of adaptation operations which defined basic relation patterns between message names. They also presented a visual notation for describing the mappings between services. The authors of \[22\] focused on dynamic adaptation of BPEL processes using semantic rules. These rules could be considered as semantic adaptation contracts which, upon the reception of new requests, generate the appropriate orchestration of services to attend to these requests. Mateescu et al. \[9\] established the foundations of this work as they propose a generative approach to adaptors based on regular expressions of vectors with the advantage of using a process algebra encoding and on-the-fly generation techniques. However, all the papers mentioned above lack the support of security concerns which is the main contribution of this paper.

The problems faced by behavioural Web Service adaptation are similar to those present in the automatic generation of controllers for composite systems. The authors of \[23\] aimed at the automatic composition of distributed business processes. Given a set of BPEL processes and an abstract description of the composition, expressed in their own goal-oriented language called EAGLE, they are able to automatically generate a controller implemented in BPEL. This controller plays an identical role to our adaptor. However, there are several differences due to the fact that they specify the composition with its goal whereas we pay particular attention to the detailed mapping between services expressed in adaptation contracts. This additional information has allowed us to express and adapt security requirements, something which was not addressed in \[23\]. In addition, in order to track and match the information exchanged, they use information about the internal operations of the services and they use knowledge-level planning techniques \[23\] to obtain better scalability by abstracting the actual values. Likewise, our approach handles symbolic values but no internal information of the services is required apart from their public behaviour.

The security adaptation contracts presented in this work allow us to verify at run time the security requirements over the messages exchanged and to adapt
services with different security policies. As it is stated in WS-Security, messages with security requirements are not necessarily secure in the face of certain attacks so further mechanisms to prove its security are required. Security adaptation does not make the orchestration secure against attacks which were not foreseen by the policies of the services orchestrated and, in fact, it multiplies the points of attack by the amount of services orchestrated. Therefore, extensive security analysis is required to prove that the orchestration is safe against some attacks. The authors of [24] proposed two approaches to express and prove secrecy properties of security protocols, one based on type systems and another based on logic programming. The security constructors used by the security adaptation contracts presented in this work are taken from theirs, so both approaches perform a generic treatment of cryptographic operations and the orchestrations obtained by our approach could be easily verified by their work.

Deployment of security adaptors is a particularly important issue because, depending on the location where the adaptor is deployed, one or several of the service security policies will prevail in the medium. One application for the work presented in this paper is to generate security adaptors as wrappers [12] for services without security capabilities in order to orchestrate them within a security enabled system. In this scenario, wrappers have been imposed by the orchestrator to make the communication secure but, on the service provider side, they must verify that the wrappers to be deployed do not interfere with their system or pose any security risk. In this situation, Proof Carrying Code [25] is a promising solution where an untrusted third-party can generate the adaptation code corresponding to a given security adaptation contract. This security contract serves as the safety policy that must be proved by the adaptor provider, and finally, service providers will be able to verify the proof against the adaptation contract and the received code before deployment.

Security adaptation contracts are intended to be used at design time to increase the interoperability with legacy services or services under different security policies by including security capabilities to behavioural adaptors. Sánchez-Cid et al. [26] propose a radically different approach where security is not tightly coupled with services but defined as other compositional entities of the system, therefore allowing the recomposition of the system to achieve different requirements in security. These security requirements, called S&D Properties, must be addressed throughout every software engineering stage and are expressed as i) S&D Classes, highest level of abstraction which deals with concepts such as “use a confidential channel” or “this information must be authenticated”; ii) S&D Patterns, which specify classes using a precise semantic description of the mechanisms that enable a particular S&D Class; and iii) S&D Implementations, which are specific algorithms that implement those mechanisms. There are two main advantages of this work that are worth highlighting: they achieve interoperability through the use of S&D Classes
at design time, hence allowing developers to dynamically choose different S&D Patterns and S&D Implementation at run time; and S&D Patterns allow both static and dynamic analysis over the security mechanisms they represent. Security adaptation could also be integrated within the [26] since adaptors are still needed to cover the gap between the interfaces of S&D Classes and their corresponding set of S&D Patterns. In addition, services must be S&D-aware whereas our approach does not require such intrusion in the services.

6. Conclusions and Future Work

In this work we presented an approach to adapt Web Services with stateful behaviour and security requirements in order to make them cooperate within an orchestration while overcoming their initial incompatibilities at signature, behaviour and security QoS levels. This adaptation is achieved through security adaptation contracts which allow the security requirements of the services to be concisely represented and managed. The formal model applied to security adaptation contracts is based on a set of basic security primitives that support a wide range of security protocols and can be analysed by automatic cryptographic protocol verifiers. Security adaptation contracts allow us to combine several WS-* security specifications in a single abstract notation and, at the same time, to solve possible incompatibilities among services. Such incompatibilities are common due to the tight coupling of WS-Security enabled services and their security restrictions.

As regards future work, we plan to work on the possible applications of our approach as security adaptors could be deployed either as single orchestrators adapting different security policies, or as service wrappers that act as security interceptors (local to the service provider) applying security properties to unsecured Web services. Furthermore, we plan to tackle situations in which adaptors are generated by untrusted parties. In these cases, we intend to use Proof Carrying Code techniques to use security adaptation contracts as formal specifications of the security requirements expected from the adaptor.

Acknowledgements. We thank the interesting comments made by Gwen Salaün, Javier Cámara and Javier Cubo to previous versions of this paper. This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Education and Science (MEC), FEDER and project P06-TIC-02250 funded by the Andalusian local Government.

References


Appendix A. Proofs

**Proposition 1.** Given a contract term $T$ which matches a security specification ($S \vdash T$), if two substitutions $\sigma_1, \sigma_2$ enable this match ($[T]_{\sigma_1} = [T]_{\sigma_2} = S$), then

$$\sigma_1(P) = \sigma_2(P)$$

for all $P \in \text{pm}(T)$.

**Proof.** This proposition is demonstrated trivially by induction on the structure of $T$—see Definition 11.

**Proposition 2.** Given a fixed set of keys and sound security primitives, if $D \models_{\theta} T$ then $\theta$ is unique on $\text{pm}(T)$.

**Proof.** We may assume (without loss of generality) that $\text{dom}(\theta_1) = \text{dom}(\theta_2) = \text{pm}(T)$ because the statement is made on $\text{pm}(T)$. We will prove the result by reductio ad absurdum, by supposing $\theta_1 \neq \theta_2$, and then proceeding by induction on the structure of $T$.

**Base cases.** If $T$ is an atom, then the result is trivially satisfied. In fact, $\text{dom}(\theta_i) = \text{pm}(T) = \emptyset$, and therefore $\theta_1 = \theta_2$. Analogously, if $T$ is a parameter $P$, as $D \models_{\theta_1} P$ and $D \models_{\theta_2} P$, then $\theta_1(P) = D = \theta_2(P)$. As $P$ is the only parameter in $\text{dom}(\theta_i)$, we have that $\theta_1 = \theta_2$.

**Inductive hypothesis.** Let us assume that the proposition holds for $T_1, T_2, \ldots, T_n$. We have to prove the result for $T = \{\text{enc}(T_1, T_2), \text{penc}_d(T_1, T_2), \text{penc}_c(T_1, T_2), \text{hash}(T_1), \text{pk}(T_1), \text{cat}(T_1, T_2, \ldots, T_n)\}$.

**General cases.** We have to consider different cases:

- $T = \text{enc}(T_1, T_2)$. As $D \models_{\theta_i} \text{enc}(T_1, T_2)$ ($i = 1, 2$), then $\llbracket \text{dec}(\{T_1 \mid \theta_i, D\}) \rrbracket \models_{\theta_i} T_2$. As $\theta_i$ ($i = 1, 2$) use the appropriate keys, $[T_1]_{\theta_i} = [T_1]_{\theta_2}$. Then, for $T_i = [T_1]_{\theta_i}$, we have $D_1 \models_{\theta_i} T_1$, and $[\text{dec}(D_1, D)] \models_{\theta_i} T_2$ ($i = 1, 2$). Therefore, we can apply the inductive hypothesis on $T_1$ and $T_2$, and the proposition will be satisfied for $\theta_1$ and $\theta_2$ on $\text{pm}(T_1) \cup \text{pm}(T_2)$.

- $T = \text{penc}_d(T_1, T_2)$. To prove this case we would proceed in a similar way as the previous case.

- $T = \text{penc}_c(T_1, T_2)$. As $D \models_{\theta_i} \text{penc}_c(T_1, T_2)$ ($i = 1, 2$), then $[\text{penc}_c(T_1, T_2)]_{\theta_i} = D$. That is, $[\text{penc}_c(T_1 \mid \theta_i, T_2 \mid \theta_i)] = D$. Again, as $\theta_i$ ($i = 1, 2$) use the appropriate keys, $D_1 [T_1 \mid \theta_i] = [T_1 \mid \theta_i]$. By condition 3, $D_2 = [T_2 \mid \theta_i] = [T_2 \mid \theta_2]$. Then, we can apply the inductive hypothesis on $T_1$ and $T_2$, and the proposition $\theta_1 = \theta_2$ on $\text{pm}(T_1) \cup \text{pm}(T_2)$.
• $T = pk(T_1)$, $T = hash(T_1)$. These two cases can be proved analogously, by applying conditions 1 and 2.

• $T = cat(T_1, T_2, \ldots, T_n)$. Finally, to prove the proposition for a sequence of contract terms we only have to apply the inductive hypothesis to $T_i$ such that $\theta_1$ and $\theta_2$ are different on at least one of the $pm(T_i)$.

**Corollary 1.** If the security primitives are sound then $|\Gamma_{S,T}(D,E)| \leq 1$ for a fixed set of keys.

**Proof.** Given $T \in CTerm$, $S \in SecSpec$, $D \in Data$ and $E \in Env$, if $\Gamma_{S,T}(D,E)$ has more than one element is because of there exist two different $\theta_1$ and $\theta_2$ such that $D \models \theta_1 \land \theta_2 \land \theta_3 \land \theta_4 \land \theta_5 \land \theta_6$ –see Definition 14–. However, $\theta_1 = \theta_2$ by Proposition 2, since security primitives are sound.

**Proposition 3.** If a given contract term $T$ does not have any instantiated parameters (i.e., $pm^*(T) = \emptyset$), then $|\Psi_{S,T}(E)| \leq 1$.

**Proof.** By definition, $inst(S,T)$ returns a set with a single element (an empty substitution) when $pm^*(T) = \emptyset$. Therefore, for a given environment $E \in Env$, the transformation $\Psi_{S,T}(E)$ –see Definition 14– returns a set with a single element, at most.

**Proposition 4.** If $\{s_1| \ldots |s_i| \ldots |s_n|\langle a,E \rangle\} \xrightarrow{D} \{s_1| \ldots |s'_i| \ldots |s_n|\langle a',E' \rangle\}$ as a consequence of applying rule SEND on $s \xrightarrow{serv} s'$ and a $\xrightarrow{serv;?T \Rightarrow adapt} d'$, then $\sigma_{E'}(P)$ is a subexpression of $S$ for every $P \in pm(T)$.

**Proof.** In the premise of rule SEND we have that $(D,E') \in \Psi_{S,T}(E)$. Thus, we know that $\sigma_{E'} = \sigma_E \circ \sigma_{S,T}$ by Definition 14, therefore $\sigma_{E'}(P) = \sigma_{S,T}(P)$ for every $P \in pm(T)$. Thus, by Definition 11, $\sigma_{S,T}(P)$ is a subexpression of $S$, hence $\sigma_{E'}(P)$ is the same subexpression.

**Theorem 1.** The premises of rules REC and SEND imply that the contract term $T$ governing the transition of the adaptor is a valid contract term.

**Proof.** To prove this theorem, we only have to check that rules REC and SEND cannot be applied over invalid contract terms. Rules REC and SEND are based on a pair of $S$ and $T$. If $T$ is not valid because $S \not\models T$ –see Definition 16–, then it does not exist any $\sigma_{S,T}$ and therefore none of the preconditions of REC and SEND holds. Now we will distinguish between input actions (REC) and output actions (SEND), by considering the other two conditions in validRec and validSend, respectively:
REC. If \( pm^*(T) \neq \emptyset \) then it does not exist any \( \theta' \) such that \( D \models_{\theta' \circ \theta_E^*} T \), since \( \theta' \circ \theta_E^* \) only valuates fresh and known parameters, and therefore \( \Gamma_{S,T}(D,E) = \emptyset \) and the precondition fails. If any of the free parameters in \( T \) is not reachable, the precondition of REC does not hold.

SEND. If there are some fresh parameters in \( T \), \( |T|_{\theta' \circ \theta_E^*} \) is undefined since fresh parameters cannot be evaluated, and therefore \( \Psi_{S,T}(E) = \emptyset \) and the precondition fails. If the third condition of \textit{validSend} is not met, then \( \text{inst}(S,T) = \emptyset \), \( \Psi_{S,T}(E) = \emptyset \), and the precondition of SEND does not hold.

**Theorem 2.** If \( \{s_1|\ldots|s_i|\ldots|s_n|\langle a,E \rangle\} \xrightarrow[D \vdash T]{S} \{s_1|\ldots|s_i'|\ldots|s_n|\langle a',E' \rangle\} \) then:

\[
\sigma_{E'} = \sigma_E \circ \sigma_{S,T} \\
\theta_{E'} = \theta_E \circ \theta_{E'}
\]

where \( \theta_{E'} \) satisfies

\[
D \models_{\theta_{E'} \circ \theta_{E}^*} T \quad \text{and} \quad \text{dom}(\theta_{E'}) = pm(T) - pm^*(T)
\]

**Proof.** The only rules which allowed data exchange between the adaptor and a service are PAR, REC and SEND. The latter two have the synchronisation trigger condition \( [\text{Definition 12}] \) therefore \( \text{dom}(E_{\emptyset}) = pm^*(T) \). Rule PAR delegates the communication in REC and SEND so we proceed to demonstrate this theorem based on those two cases:

**REC** This case is a consequence of \( E' \in \Gamma(D,E) \), which is a premise of rule REC in \cite{Fig 7}. In fact, \( D \models_{\theta' \circ \theta_E^*} T \) upon \( [\text{Definition 14}] \) where \( \theta = \theta_E \) and \( \theta' = \theta_{E'} \) due to the restrictions imposed by rule REC. Regarding the domain restriction, by \( \Gamma \) we know that \( \text{dom}(\theta_{E'}) = pm^!(T) \) and that \( T \) does not have any instantiated parameter because it is a valid contract term for input actions –see \( [\text{Theorem 1}] \).

**SEND** Because of the definition of \( \Psi \), we have that \( \sigma_{E'} = \sigma_E \circ \sigma_{S,T} \). Additionally, \( \text{dom}(\theta_{E'}) = pm^*(T) \) by definition of \( \text{inst}(S,T) \). Contract term \( T \) does not have any fresh parameter because it is a valid contract term for output actions. As \( (D,E') \in \Psi_{S,T}(E) \) we have that \( D = |T|_{\theta_{E'} \circ \theta_{E}^*} \) –see \( [\text{Definition 7}] \), which are the actual data sent by the adaptor. Thus, we only have to prove that \( |T|_{\theta_{E'} \circ \theta_{E}^*} \models_{\theta_{E'} \circ \theta_{E}^*} T \), but this is directly derived from \( [\text{Definition 13}] \). In fact, the value of \( T \) and the security check –\( [\text{Definition 13}] \)– are inductively analogous for all the possible structures of \( T \) but pencd. However, pencd cannot be present in \( T \) because such security primitive is only used on input actions, therefore not applicable to rule SEND.
Appendix B. Generation of Security Adaptors

Considering the transition relation behind orchestrations through security adaptors (Figure 7), we generate the behaviour of the adaptor (gen_security_adaptor in Algorithm 1) by first generating every possible successful synchronisation supported by the contract between the adaptor and the services with Algorithm 2 (gen_all_possible_synchronisations) and then remove every branch that does not finish in a stable state or presents deadlock situations. We restrict the size of adaptor with constant LIMIT to avoid infinite adaptors. Function vterms_in returns all the contract terms in the set of security vectors of the contract, and function process_τ_transitions simulates the evolution of the system on internal transitions in the services. If there is no feasible adaptor within the size limit and the given contract and services, gen_security_adaptor returns an empty behaviour.

Algorithm 1 gen_security_adaptor
Generates the behaviour of a security adaptor.

inputs Security-enabled Service interfaces serv_i = ⟨Σ_i, O_i, F_i, T_i⟩, i ∈ {1,...,n}; the security adaptation contract C = ⟨V, VLTS, E_0⟩ where E_0 = ⟨σ_0, θ_0⟩ and the security vector-LTS VLTS has an initial state st_vlts_0, and a LIMIT as a boundary for possible infinite adaptors

output Behaviour of the security adaptor adapt = ⟨Σ_a, O_a, F_a, T_a⟩ where Σ_a ⊆ VTerm. This behaviour is empty if no security adaptor can be generated for the given inputs.

1: st_a_0 = ⟨σ_0, σ_vlts_0, 0, {st_1, ..., st_i, ..., st_n}⟩
2: O_a, F_a, T_a = {σ_0}, 0, 0
3: adapt = ⟨vterms_in(V), O_a, σ_0, F_a, T_a⟩
4: services = {serv_1, ..., serv_n}
5: iter = 0
6: {Generate every possible state and transition}
7: while adapt does change and iter < LIMIT do
8:    iter = iter + 1
9:    adapt = process_τ_transitions(adapt, services)
10: for all a = ⟨σ, v, q, ST⟩ ∈ O_a do [For every state in the adaptor]
11:    adapt = gen_all_possible_synchronisations(adapt, services, VLTS, a)
12: end for
13: end while
14: adapt = prune_deadlock_branches(adapt)
15: return adapt

It is worth observing that, in gen_all_possible_synchronisations, when the security vectors have two contract terms (lines 1-7), the algorithm needs to queue the output actions from the adaptor (line 3) which can be executed in any moment afterwards (lines 20-26). This allows several security vectors to be executed in parallel if required.
Algorithm 2 gen_all_possible_synchronisations

It explores every possible synchronisation at a given state of the adaptor and the services.

inputs A partial behaviour of a security adaptor adapt = \( < \Sigma_v, O_a, \sigma_o, F_a, T_a > \); the behaviour of the services serv_i = \( < \Sigma_i, \Omega_i, \sigma_i^0, F_i, T_i > \), \( i \in \{ 1, \ldots, n \} \); the security vector-LTS VLTS = \( < \Sigma_vlt, \Omega_vlt, \sigma_{o_vlt}, F_{vlt}, T_{vlt} > \) with \( \Sigma_v \subseteq VTerm \) and \( \Sigma_vlt \subseteq SVector \); and an adaptor state \( \sigma = < \sigma, v, q, \sigma_o > \) where \( \sigma \) is part of the actual environment of the adaptor, \( v \in \Omega_vlt \) is the current state in the vlts, \( q \) is a queue of pending synchronisations, and \( ST \) is the set of the current states of the services \( o_i \in O_a, i \in \{ 1, \ldots, n \} \).

output Extended behaviour of the security adaptor.

1: for all \( (v, \langle j ; ? T \rangle, q, \sigma_o, \tau \rangle) \rightarrow v' \), \( (v, \langle j ; ? T \rangle, q, \sigma_o, \tau \rangle) \rightarrow v' \) \( \in T_{vlt} : s_i \in ST, s_i \mapsto s'_i \in T_i, \sigma_o \in \sigma_o \tau \), \( \langle \sigma', \_ \rangle \in \Gamma_{\Sigma, \tau} (\langle \sigma, \_ \rangle) \) do [For every enabled vector with two sides]
2: \( ST' = (ST - \{ \sigma \}) \cup \{ \sigma'_i \} \)
3: \( q' = enqueue(j; ? T, q) \) [Enqueue the output action]
4: \( a' = (\sigma', v', q', ST') \)
5: \( O_a = O_a \cup \{ a' \} \)
6: \( T_o = T_o \cup (a \xrightarrow{E_T} a') \)
7: end for
8: for all \( (v, \langle j ; ? T \rangle, q, \sigma_o, \tau \rangle) \rightarrow v' \), \( (v, \langle j ; ? T \rangle, q, \sigma_o, \tau \rangle) \rightarrow v' \) \( \in T_{vlt} : s_i \in ST, s_i \mapsto s'_i \in T_i, \sigma_o \in \sigma_o \tau \), \( \langle \sigma', \_ \rangle \in \Gamma_{\Sigma, \tau} (\langle \sigma, \_ \rangle) \) do [For every enabled vector with one output action]
9: \( ST' = (ST - \{ \sigma \}) \cup \{ \sigma'_i \} \)
10: \( a' = (\sigma', v', q', ST') \)
11: \( O_a = O_a \cup \{ a' \} \)
12: \( T_o = T_o \cup (a \xrightarrow{E_T} a') \)
13: end for
14: for all \( (v, \langle j ; ? T \rangle, q, \sigma_o, \tau \rangle) \rightarrow v' \), \( (v, \langle j ; ? T \rangle, q, \sigma_o, \tau \rangle) \rightarrow v' \) \( \in T_{vlt} : s_i \in ST, s_i \mapsto s'_i \in T_i, \sigma_o \in \sigma_o \tau \), \( \langle \sigma', \_ \rangle \in \Gamma_{\Sigma, \tau} (\langle \sigma, \_ \rangle) \) do [For every enabled vector with one input action]
15: \( ST' = (ST - \{ \sigma \}) \cup \{ \sigma'_i \} \)
16: \( a' = (\sigma, v', q, ST') \)
17: \( O_a = O_a \cup \{ a' \} \)
18: \( T_o = T_o \cup (a \xrightarrow{E_T} a') \)
19: end for
20: for all \( i ; ? T \in q : s_i \in ST, s_i \mapsto s'_i \in T_i, \sigma_o \in \sigma_o \tau \), \( \langle \sigma', \_ \rangle \in \Psi_{\Sigma, \tau} (\langle \sigma, \_ \rangle) \) do [Apply every possible queued action]
21: \( ST' = (ST - \{ \sigma \}) \cup \{ \sigma'_i \} \)
22: \( q' = dequeue(i; ? T, q) \)
23: \( a' = (\sigma', v', \sigma, \tau) \)
24: \( O_a = O_a \cup \{ a' \} \)
25: \( T_o = T_o \cup (a \xrightarrow{E_T} a') \)
26: end for
27: if \( v \in F_vlt \) and \( \forall s_i \in ST, s_i \in F_i \) and is_empty(q) then [Mark global final states]
28: \( F_o = F_o \cup \{ a \} \)
29: end if
30: return adapt = \( < \Sigma_v, O_a, I_o, F_a, T_o > \)