Synthesis of Secure Adaptors

J.A. Martín, F. Martinelli, E. Pimentel

E.T.S. Ingeniería Informática, Universidad de Málaga, Campus de Teatinos, 29071 Málaga, Spain.
Istituto di Informatica e Telematica, National Research Council, 56124 Pisa, Italy.

Abstract

Security is considered one of the main challenges for software oriented architectures (SOA). For this reason, several standards have been developed around WS-Security. However, these security standards usually hinder interoperability, one of the main pillars of Web service technologies. Software adaptation is a sound solution where an adaptor is deployed in the middle of the communication to overcome signature, behavioural and QoS incompatibilities between services. This is particularly important when dealing with stateful services (such as Windows Workflows or WS-BPEL processes) where any mismatch in the sequence of messages might lead the orchestration to a deadlock situation. We proposed security adaptation contracts as concise and versatile specifications of how such incompatibilities must be solved. Nonetheless, synthesising an adaptor compliant with a given contract is not an easy task where concurrency issues must be kept in mind and security attacks must be analysed and prevented. In this paper, we present an adaptor synthesis, verification and refinement process based on security adaptation contracts which succeeds in overcoming incompatibilities among services and prevents secrecy attacks. We extended the ITACA toolbox for synthesis and deadlock analysis and we integrated it with a variant of CCS, called Crypto-CCS, to verify and refine adaptors based on partial model checking and logical satisfiability techniques.

Key words: model-based adaptation, Web service orchestration, security verification, adaptor synthesis, partial model checking, WS-Security

1. Introduction

Security is one of the main research challenges of SOA systems [1,2]. Several standards surged to fill this gap: XML Encryption, for performing cryptographic operations over parts of XML documents; WS-SecureConversation, to establish secure sessions among services; WS-Trust, to assert and handle trust between different parties; WS-SecurityPolicy, to express the policies that are offered and required; and WS-Security, to glue it all together, to name a few members of the WS-* specifications. However, the inclusion of these new specifications complicate the development, reuse and replaceability of SOA systems.

Even though security is usually opposed to interoperability, Web services (WS) must be both secure and interoperable. For this reason, we proposed security adaptation contracts [3] which, based on software adaptation techniques [4,5], specify in a unified and concise manner how to overcome incompatibilities in interface, behaviour, and security QoS among WSs.


Email addresses: jamartin@lcc.uma.es (J.A. Martin), Fabio.Martinelli@iit.cnr.it (F. Martinelli), ernesto@lcc.uma.es (E. Pimentel)

Preprint submitted to Elsevier October 27, 2011
Figure 1 shows an overview of the approach presented in this paper. The inputs of our synthesis process are: i) services with incompatible behaviour and security QoS encoded in Crypto-CCS [6] (CCS [7] extended with cryptographic operations); ii) a security adaptation contract, i.e., a mapping between the interfaces of the services which specifies how the incompatibilities must be solved and which security checks must be enforced; and iii) a secrecy property to preserve expressed in a logical language with knowledge operators. The synthesis process is structured in three sequential steps. First, a functionally-correct adaptor is synthesised based on the given contract and services. This adaptor is able to orchestrate the services in a way that it solves their initial incompatibilities. However, the adapted system might be insecure against secrecy attacks. For this reason, we verify if the synthesised adaptor and the given services are robust with regard to the given secrecy property in a second step. Finally, if an attack exists, we proceed to refine the initial adaptor into a secure adaptor. Hence, the output of the synthesis process presented in this paper is an adaptor encoded in Crypto-CCS, able to orchestrate the services despite their incompatibilities, compliant with the security adaptation contract (which allows a fine-grained control over the result) and robust against secrecy attacks.

1.1. Functionally-correct adaptors

Security adaptation contracts describe the adaptors that must be deployed in the middle of the communication between incompatible services to allow their successful interaction avoiding deadlocks and livelocks. Stateful services (such as Windows Workflows [8] or WS-BPEL [9] processes) are particularly prone to these undesired situations. An adaptation contract defines a mapping between the operations (and arguments) of the services, and the restrictions that must be obeyed to fulfil the functional and security requirements.

The process to obtain an adaptor process $P$ conforming to a given contract is called adaptor synthesis. The idea is to synthesise an adaptor compliant with a contract in such a way that the parallel composition (denoted with the operator ‘∥’) between the adaptor and the services $S$ always reaches a satisfactory state.

$$P∥S \text{ always reaches a final/stable state.}$$

At this point, $P$ is a security adaptor process in the sense that it successfully reads and recomposes the cryptographic messages exchanged between services. These messages contain parts where different cryptographic operations have been applied. Encryption (symmetric and asymmetric), signature and hashing complicate the adaptor task as it has to be able to decrypt, verify, and
re-encrypt messages as expected by the destination service. This message manipulation is described in the adaptation contract.

1.2. Secure adaptors

A security adaptor process satisfying (1) might not yet be secure. Although it manipulates cryptographic messages, it does not guarantee that this manipulation prevents the disclosure of private information. Therefore, in addition to being able to manipulate cryptographic messages according to the security adaptation contract, we would like the adapted system to be resilient against secrecy attacks or, in other words, the adapted system must not disclose sensitive information during its communications. This is expressed in the following equation.

$$\exists X \text{ s.t. } P \parallel S \parallel X \models p_{sec}$$

where ‘|=’ is the logical satisfiability operator, $p_{sec}$ is the logical formula which represents the secrecy attack to avoid and $X$ is the possible attacker. As an example, $p_{sec}$ can represent that the attacker $X$ is able to eavesdrop the credit card number of $S$ during its communication with $P$.

There are multiple attack scenarios and (2) denotes a general one where the network is under the control of the attacker. In this case, $P$ cannot enforce any security on it because $X$ can, among other things, completely isolate the adapted system $S$ from the adaptor process $P$. Other scenarios can be envisioned depending on the actual deployment setting and the different zones of trust (see Figure 2). In fact, we could want to restrict the power of the attacker in a way that the adaptor serves as a gateway to our system. Using the restriction operator ($\backslash L$), (2) could be concretised into the following equation.

$$\exists X \text{ s.t. } (P \parallel S) \backslash L \parallel X \models p_{sec}$$

In (3), the attacker is still able to eavesdrop on any communication in the system but it can only actively participate in those channels not present in $L$. Language $L$ does not need to be the complete
set of actions between S and P. If L is empty, we revert to (2) and, if L contains every channel of S and P, then we are modelling passive attackers, i.e., attackers which can only eavesdrop but not actively participate in the communication.

We can further restrict the power of the attacker by limiting on which actions he can eavesdrop. In this case, a hiding operator (\(\setminus L\)) is used to denote the actions which cannot be accessed nor eavesdropped by the attacker. In addition, we only need the synchronisations made within the system so we can restrict the whole expression with the complete language of actions (\(\setminus L''\)). Let us highlight the fact that neither \(\setminus L\) or \(\setminus L'\) have to cover the complete language of actions or, in other words, a single service can have different actions in each of the different layers of isolation from the attacker. This flexibility allows us to model scenarios which range from attacks where the attacker controls the network (including insider attacks) to the more limited external attackers.

Summing up all the previous points, we reach our final equation where the attacker might initially know some information \(\phi\).

\[\exists X \text{ s.t. } ((P\parallel S)\setminus L\setminus L')\setminus L'' = p_{sec} \] (4)

The goal of this paper is to synthesise an adaptor process P satisfying (4) and compliant with a given security adaptation contract.

1.3. Motivational example

Let us consider two services Q and R which send certain data to a third service S. On the one hand, service Q sends the data along with its message authentication code (mac) in a single message with the following type: \texttt{send\_hash!Data, Hash(Data, Secret)}. The mac is the hash value (i.e., the result of a hash function where it is unfeasible to obtain its inverse) of the couple consisting of the data plus a secret. Both services previously shared that secret. On the other hand, service R uses asymmetric encryption to guarantee its integrity and secrecy using the following message: \texttt{send\_enc!AEnc(Pk(Key), Data)}. The data is encrypted using the public key (\texttt{Pk(Key)}) of the destination service. Finally, service S is the intended recipient of both messages but service S lacks any security infrastructure and the only message that it understands is \texttt{receive?Data}.

In this scenario, we need an adaptor P with a secure channel with S which intercepts the messages coming from Q and R, verifies their integrity and forwards them to S. Adaptor P provides service S with a unified-integrity and destination-authenticity mechanism, as it understand both mac and public-key protocols.

Such an adaptor P satisfies (1). However, we now want to enforce a property \(p\) such that “the data received by S is confidential outside \(P\parallel S\)”. In this case, we obtain an attacker X (3) that eavesdrops channel \texttt{send\_hash}, in which the data are sent as clear text. However, if L enforces the attacker to interact only through P to reach S, the adaptor could avoid the attack by forbidding the \texttt{send\_hash} action, only allowing \texttt{send\_enc}. Such an adaptor satisfies (4).

In addition, security adaptation contracts are flexible enough to cope with a different scenario where the data sent by Q and R must be the same before relaying them to the final destination, service S. This final case would be a multi-factor authentication mechanism (i.e., information must be authenticated in several ways at the same time) supported by adaptor P on behalf S. This problem can be tackled also by security adaptors.

It is worth noting that security adaptors can be deployed either transparently or as another known participant in the communication, being provided that it can intercept the communications that need adaptation. If the adaptor has all the information required to impersonate a service (like the Secret and private Key in the previous example) the deployment can be transparent between
that service and the rest of the system. Otherwise, the services must be aware of the adaptor (which must possess its own credentials) and interact through it, handling trust and authentication accordingly. Security interceptors, used in Oracle BPEL Process Manager 10g, are good examples of transparent security adaptors. In this case, BPEL processes are defined without security and security interceptors intercept outgoing and incoming messages, applying security to the former and verifying the security and extracting the encapsulated information in the latter.

Compared to other kinds of contracts, security adaptation contracts specify requirements over an adaptor that must be synthesised and deployed to support the secure conversation between WSs that were initially incompatible. 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 runtime. 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.

**Main contributions.** We take security adaptation contracts from our earlier paper [3] (briefly introduced in Section 2) and, in this paper, we formalise how to synthesise deadlock-free adaptors satisfying (1) and compliant with a given contract (see Section 3). Then, in order to address scenarios such as (3), we are going to encode our synthesised adaptor into a Crypto-CCS process. Crypto-CCS [6] is a variant of CCS [7] which uses partial model checking and logical satisfiability techniques to verify properties such as (4). In this paper, we extend Crypto-CCS in such a way that, if an attacker exists, it returns the most general attacker for the given system in the sense of any possible attack can be done by that attacker (see Section 4). If such an attacker exists, then we proceed to complete our synthesis process and refine our initial adaptor by removing the last controllable adapted action which enabled the attack (see Section 5). In this way, we finally obtain a secure security adaptor which satisfies (4). We discuss related work in Section 6 and we finally conclude with Section 7.

2. Security adaptation

Software adaptation [4 5] is conceived as an approach to allow the proper communication of services that were initially unable to interact due to incompatibilities at signature, behavioural or QoS levels.

*Signature incompatibilities* are those when services present a different interface than expected. Different operation names or incompatible type or number of arguments fall into this category. As an example, we could think of two services that, in spite of presenting the same functionality, offer incompatible WSDL descriptions.

*Behavioural incompatibilities* arise when, even though services might offer compatible interfaces, they impose certain order in the exchange of messages that makes their orchestration unsuccessful. These kinds of incompatibilities are more evident in stateful services such as those described as Windows Workflows or WS-BPEL processes. However, RESTful and JavaScript APIs (ubiquitous nowadays) can also present these kind of incompatibilities as they have implicit control dependencies in the form of cookies or authentication tokens that require certain actions (typically, a login) to happen before some other (privileged) actions.

*Non-functional incompatibilities* are those beyond the signature and behaviour of the services. These can be further classified into *QoS incompatibilities*, e.g., when the throughput of a service is expected to be higher than it is (this could be adapted, for instance, using an adaptor to perform load-balancing with two instances of that service running in parallel), or a service which expects in clear text what another service is sending encrypted. On top of that, we have *semantic incompatibilities*...
like, for instance, implementing a list from a stack or using a search engine as a spell checker.

In this paper, we will focus on behavioural adaptation extended with security QoS. The goal is to obtain an adaptor which orchestrates the services avoiding deadlocks and livelocks, and it complies with the given contract and secrecy property.

2.1. Behavioural interfaces

We model service interfaces as state machines where actions are typed with (possibly structured) cryptographic constructors and basic types.

**Definition 2.1 (Interface).** An interface is a state machine \((\Sigma, O, s_0, F, \rightarrow)\) where \(\Sigma\) are the interface actions of the service \((\Sigma \subseteq IAct)\), \(O\) is the set of states, \(s_0 \in O\) is the initial state, \(F \subseteq O\) are final states without outgoing transitions and \(\rightarrow \subseteq (O \times \Sigma \times O)\) is the labelled transition relation.

Interface actions \((\in IAct)\) are denoted by their channel \((\in Chan)\), an exclamation mark or a question mark for output and input actions, respectively, followed by a (possibly structured) type of the message \((IAct = (Chan \times \{!, ?\} \times Types) \cup \{\tau\})\). Internal actions \((\tau)\) are used to encode internal decisions such as if activities in WS-BPEL.

Consider a set of basic types \((BT \in BT)\) denoted by capitalised words and a set of cryptographic constructors \((F \in F)\). Then, the set of Types is defined by the following grammar

\[
T(\in Types) ::= BT \mid F(T_1, \ldots, T_{ar(F)})
\]

where \(ar(F)\) is the number of arguments of constructor \(F\). Typed messages are defined analogously

\[
m(\in Msgs) ::= x \mid BM \mid F(m_1, \ldots, m_{ar(F)})
\]

where \(x\) belongs to a countable set \(V\) of message variables, \(BM\) are messages of basic type and \(m_1, \ldots, m_{ar(F)}\) are messages of type \(T_1, \ldots, T_{ar(F)}\), respectively. Typed actions \((\in Act)\) are elements of \((Chan \times \{!, ?\} \times Msgs) \cup \{\tau\}\). Note that we use the set of constructors \(F\) both for types and messages.

![Figure 3](image-url) defines a system IS able to infer typed messages based on constructors \(Hash, Enc, AEnc\) and concatenation. Symbols \(m, m_i, k\) represent messages and \(T, T_i, K\) correspond to their respective
types. The inference system $\mathcal{I}S$ models hash operations (1), symmetric encryption and decryption (5 and 6), public-key encryption and decryption (2, 7 and 8) and concatenation of messages (3 and 4).

As usual, we denote by $s \xrightarrow{\alpha_1 \cdots \alpha_i} s'$ a sequence (possibly empty) of transitions such as $s \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_i} \cdots s'$ where $\alpha_i \in \Sigma, i \in \{1, \ldots, n\}$. Analogously, $s \xrightarrow{\tau} s'$ denotes a sequence of at least a transition and $s \xrightarrow{\tau^*} s'$ represents a sequence (possibly empty) of $\tau$-labelled transitions. Finally, we will denote by $O[M]$ the set of possible transition traces starting from the initial state of the state machine $M$. We will represent either as $\tilde{t}$ or $t_1 \cdots t_n$ a trace of transitions, and $t \in \tilde{t}$ will denote that transition $t$ is in the trace $\tilde{t}$.

**Example 1.** Our running example is based on the interfaces of three services depicted in Figure 4. These services present incompatible behaviour and security. Successful termination states are filled.

Service 1 (Figure 4(a)) supports two different authentication schemas: one based on hashes and a shared secret and another based on a modification of the Needham-Schroeder public-key protocol. On the one hand, the hash authentication starts with a hash output action with the
following arguments: the identification of the originating service (i.e., \textit{Id}), the request (\textit{Req}), a nonce (which stands for number used once, typed \textit{Nonce}) and the mac value of all the previous arguments with regard to a previously shared \textit{Secret}. From that point on, service 1 expects all the subsequent messages to be encrypted with the concatenation of the previous nonce and secret. It can receive either a \textit{denied} action (with an encrypted new nonce) or a \textit{reply} with the encrypted data. On the other hand, the public-key authentication part starts with a \textit{pk-auth} output action with the \textit{Id} of service 1, the request and a nonce, all of them encrypted using the public key of service 2 (typed \textit{Pk(\textit{Key})}). This branch can also be rejected by the destination service by transmitting a \textit{denied} action symmetrically encrypted with the previous nonce. Otherwise, it expects a \textit{reply} with the previous nonce and a new nonce, both encrypted with the public key of service 1, followed by the requested data signed by service 3 and encrypted with the concatenation of both nonces. Finally, it sends an acknowledgement (via \textit{ack}) with the received nonce encrypted with the public key of service 2.

Service 2 is represented in Figure 4(b). This service is able to provide the data corresponding to the received request. However, if only accepts the public-key protocol therefore every request must be received encrypted with its public key and then it replies in clear text. Service 2 has an internal choice, represented with \textit{\tau}-labelled transitions, which allows him to decide whether to \textit{reply} or \textit{deny} the access (via \textit{no access}).

The last service (service 3 in Figure 4(c)) simply encrypts with his private key all received data. Alternatively, it ends its behaviour if it receives action \textit{exit}.

2.2. Security Adaptation Contracts

Security adaptation contracts \cite{3} are an evolution of the adaptor specifications presented in \cite{10} and extended with value-passing in \cite{11}. The contribution of security adaptation contracts are the inclusion of cryptographic structures in the actions which allows the receiving, verifying and composing of cryptographic messages. Security adaptation contracts allow contract designers to focus on matching the semantics behind arguments and operations, and forget about concurrency and secrecy issues. These problems are automatically addressed by the synthesis of secure adaptors presented in the rest of the paper. Here we proceed to briefly describe security adaptation contracts.

Security adaptation contracts need to specify how to receive, verify and recompose cryptographic messages. These operations are supported by \textit{contract terms}.

\textbf{Definition 2.2 (Contract Term).} A contract term is an expression made of cryptographic constructors and annotated symbolic parameters. Contract terms respect the following grammar

\[
T ::= BT \mid \mathcal{P} \mid \text{Pk}(T) \mid \text{Hash}(T) \mid (T_1, \ldots, T_n) \\
\quad \mid \text{Enc}(T_1, T_2) \mid \text{AEnc}_c(T_1, T_2) \mid \text{AEnc}_d(T_1, T_2)
\]

where $BT \in \mathcal{B}$ and $\mathcal{P} \in \text{Param}$ is a (possibly annotated) symbolic parameter.

For the inference system of our running example (Figure 3), the asymmetric encryption constructor needs to be annotated with whichever one of the rules for encryption (\textit{AEnc}_c, rule 2) or decryption (\textit{AEnc}_d, rules 7 and 8) needs to be applied. Hash and symmetric encryption unambiguously represent by syntax which inference rules must be applied depending on the direction of the action and the annotation of the symbolic parameters within.

Symbolic parameters can be annotated to express how they are updated, verified and used for composing cryptographic messages. Parameters that represent previously known values are annotated with "\land". These parameters are called \textit{known parameters}. Outgoing messages typically have most
of their parameters annotated as known parameters since the values corresponding to the arguments of a message must be available to the adaptor before the message is sent. Known parameters are also used when part of a received message should be compared to check if it is equal to some previously known data. In order to keep track of stored values the adaptor includes an *environment*.

There are arguments whose value must be generated by the adaptor (such as new keys, timestamps and nonces, for example), these are represented by parameters annotated with ‘∗’, we call these parameters *instantiated parameters*. Once instantiated, the newly generated value is also stored in the environment. All other unannotated parameters are called *fresh parameters* and their values will be updated in the environment when a matching message is received. Intuitively speaking, fresh parameters are those whose values are going to be received during a communication whereas instantiated parameters are those whose value must be created by the adaptor on its own (normally, before an output action).

If the same parameter occurs more than once in the same contract term, all its occurrences should represent the same value.

**Example 2.** Below there are three actions with contract terms. The first two are taken from *Section 1.3* and the latter is part of the running example (Example 1).

\[
\begin{align*}
\text{send} \ D, \text{Hash}(D, S^\land) & \quad (5) \\
\text{send}!D^\land & \quad (6) \\
\text{denied}!\text{Enc}(K^\land, N^*) & \quad (7)
\end{align*}
\]

The input action in 5 occurs on channel *send* and its arguments should be matched by the contract term \(D, \text{Hash}(D, S^\land)\). Such a contract term receives and stores under parameter \(D\) whatever is matched with the first argument. Then, it verifies that the hash of both the value received in \(D\) and some previously known information \(S^\land\) (which is annotated to be known) should be equal to whatever is received as a second argument. Remember that every occurrence of the same parameter should represent the same value. In order to be known, the value of \(S\) must have been either: i) received in a previous input action as a fresh parameter; ii) instantiated by the adaptor in a previous output action; or iii) part of the initial knowledge of the adaptor. In any case, the adaptor already knew its value in order to verify the received message.

The output action in 6 states that a new message must be composed with the previously known parameter \(D^\land\) and emitted through channel *send*.

Finally, the output action *denied* (7) encrypts with a known key \((K^\land)\) a nonce which is generated and stored by the adaptor \((N^*)\).

An environment \(E = (\theta, \kappa)\) stores the types and values represented by the parameters with two substitutions: \(\theta\) which substitutes parameters with their corresponding type and \(\kappa\) which replaces parameters with their known runtime value.

Environments are extended with the infix operator ‘⊘’. For instance, \(\kappa \circ \kappa'\) represents a new substitution where substitution \(\kappa'\) takes precedence over \(\kappa\) or, in other words, every parameter which occurs in \(\kappa'\) is replaced by the values included in \(\kappa'\), and then substitution \(\kappa\) is applied on the remaining parameters.

Symbolic parameters \((P \in \text{Param})\) are evaluated by the substitutions in the environment to obtain their corresponding types \((\theta(P))\) and run-time values \((\kappa(P))\). However, these substitutions can be restricted to annotated parameters with a superscript (e.g., \(\theta^f, \theta^\land\) and \(\theta^*\) substitute only fresh, known and instantiated parameters, respectively). Function \(\text{pm}(T)\), which returns the set
of parameters present in contract term \( T \), can also be restricted to annotated parameters in the same way. The transformation from contract terms (e.g., \( T \)) to their corresponding types in the interface is denoted by \([T]_\theta\), where substitution \( \theta \) is used to replace the parameters in \( T \) with their types (see [3] for a detailed definition of this transformation).

Contract terms can use different cryptographic constructors and inference systems provided that they unambiguously represent which inference rules need to be applied to decompose and compose the messages.

An adaptor must react to the messages it receives. Contract terms helped to receive and send individual messages, we now use security adaptation vectors to relate received messages to their corresponding replies from the adaptor.

**Definition 2.3 (Security Adaptation Vector).** A security adaptation vector (or vector, for short) is denoted as \( \langle c!T \rangle, \langle c?T \rangle \) or \( \langle c?T \triangleright c!T' \rangle \) where \( c \) and \( c' \) are channels and \( T, T' \) are contract terms.

We can unequivocally identify each side of a vector by its direction, e.g., being \( v = \langle c?T \triangleright c!T' \rangle \), then \( ?v = c?T \) and \( !v = c!T' \). In addition, we can naturally extend the previous operator ‘\([\cdot]\)’ to obtain the interface corresponding to vector components: \([c!T]_\theta \equiv c!\langle T \rangle_\theta\).

The intuition behind vectors with two actions is that, whenever the adaptor receives an action matching the left-hand side of the vector, it must eventually send the action in the right-hand side of the vector. Vectors with one action can be used as needed by the adaptor. Vectors can interleave or, in other words, we can apply additional vectors between the input and output action of a two-sided vector.

**Example 3.** Going back to our unified integrity example (Section 1.3), the vectors required for the adaptation are the following.

\[
\langle \text{send}_\text{hash}?D, \text{Hash}(D,S^\land) \triangleright \text{send}!D^\land \rangle \quad (8)
\]

\[
\langle \text{send}_\text{enc}?\text{AE}_{\text{enc}}(K^\land, D) \triangleright \text{send}!D^\land \rangle \quad (9)
\]

Vector (8) relates the actions (5) and (6) of Example 2. When the adaptor receives the former, it must reply with the latter. Vector (9) first receives action \( \text{send}_\text{enc} \) and decrypts its single argument with a previously known private key \( K^\land \) (typed \( \text{Key} \)). Rule 8 of the inference system \( \text{IS} \) (Figure 3) is used for this decryption and its result is stored under parameter \( D \). This is forwarded as expected using the second half of the vector through action \( \text{send} \).

Security adaptation contracts can enforce some 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 are state machines whose transitions are labelled with synchronisation vectors.

**Definition 2.4 (Security Adaptation Contract).** A security adaptation contract (or contract, for short) \( C \) is a state machine \( (\Sigma^C, O, s_0, F, \rightarrow_C) \) plus an environment \( (\theta, \kappa) \) where \( \Sigma^C \) is a set of security vectors, \( O \) is the set of states, \( s_0 \in O \) is the initial state, \( F \subseteq O \) is the set of final states, and \( \rightarrow_C \subseteq (O \times \Sigma^C \times O) \) is a transition relation.
Intuitively speaking, a security adaptation contract can be understood as a mapping between the different security policies of several services, along the information (e.g., keys and certificates) and restrictions required to support the secure interaction between the services, despite their initial incompatibilities.

Example 4. The adaptation required to address the multi-factor authentication of Section 1.3 is solved by the following contract and initial environment.

\[
\Sigma^C = \{\langle \text{send\_hash}\,?D,\text{Hash}(D,S^\wedge) \rangle; \langle \text{send\_enc}\,?\text{AEnc}_d(K^\wedge,D^\wedge) \triangleright \text{send}\!\!\!\!D^\wedge \rangle \}
\]

(v1) \hspace{1cm} (v2)

\[
T = \{s_0 \xrightarrow{v_1} C s_1, s_1 \xrightarrow{v_2} C s_0 \}
\]

\[
E = ([\text{Data}/D, \text{Secret}/S, \text{Key}/K], [s/S,k/K])
\]

\[
C = \left( (\Sigma^C, \{s_0, s_1\}, s_0, \{s_0\} , T) , E \right)
\]

The key difference with respect to Example 3 is that we do not send the data when we receive the first message (v1) but, instead, we wait for the second message (v2) and we check that the data received in both messages is the same to do multi-factor authentication (using the know parameter \(D^\wedge\) in v2). The transition relation of the contract (T) enforces that the vectors are applied in the order previously mentioned.

Example 5. The service interfaces of our running example can be adapted using the security adaptation contract presented in Figure 5.

The contract supports both the hash and public key protocols from service 1 (Figure 4(a)). If service 1 decides to do the request through hash, this is received by vector \(v_{\text{hs}}\). This vector allows requests coming from a service identified \(I\) and it checks that the hash of the request (\(\text{Hash}(I,R,NA,S^\wedge)\)) corresponds to what is received and the secret previously known by the adaptor. Once the reception is verified, the adaptor compliant with this contract should eventually respond by sending a request to service 2 (Figure 4(b)). Such a response is composed using the received values, i.e., the nonce of service 1 (\(NA^\wedge\)) and the request \(R^\wedge\).

Service 2 might either respond with the data \(D\) with vector \(v_{\text{rep}}\) or reject the request via \(2\,\text{no\,access}\) in \(v_{\text{dny}}\). Either way, it also provides the nonce \(NA^\wedge\) received with the request from service 1. This nonce and the secret \((s_1, 2\,\text{which substitutes parameter} S^\wedge)\) are used to encrypt and send the data to service 1 on the right-hand side of \(v_{\text{rep}}\). If, on the contrary, the request is denied, what is encrypted and sent is a new nonce instantiated by the adaptor (i.e., \(N^*\)) using vector \(v_{dny}\).

A similar process happens if service 1 goes through the public-key protocol starting with \(1\_pk\_auth\) in vector \(v_{pk}\). In this case, however, the request \(Q\) is directly understood by service 2, therefore no verification or recomposition is needed from the adaptor, hence the request is directly forwarded to service 2.

Then, service 2 might deny the request \((v_{\text{no}})\) or otherwise reply with the data using vector \(v_{\text{sign}}\). The right-hand side of this vector delegates on service 3 (Figure 4(c)) to sign it with its private key. These signed data are received in vector \(v_{ok}\) and properly encrypted and composed to reply to service 1. Notice that the nonce which has to be instantiated composing \(1\_reply\) in \(v_{ok}\) (i.e., \(NB^*\)), is the same for both occurrences of the symbolic parameter.

Finally, service 1 will acknowledge the response with \(ACK\) in \(v_{\text{ack}}\). By using simply \(ACK\) to match the acknowledgement we just receive it and ignore it. Alternatively, this message could have been
\(\Sigma^C = \{\langle 1\_hash?I,R,NA,\text{Hash}(I,R,NA,S^\wedge) \triangleright 2\_request!\text{AEnc}_c((KB^\wedge,I^\wedge,NA^\wedge,R^\wedge))\rangle; (v_{hs})\,
\langle 2\_reply?NA^\wedge,D \triangleright 1\_reply!\text{Enc}((NA^\wedge,S^\wedge),D^\wedge)\rangle; (v_{rep})\,
\langle 2\_no\_access?NA^\wedge \triangleright 1\_denied!\text{Enc}((NA^\wedge,S^\wedge),N^\wedge)\rangle; (v_{deny})\,
\langle 1\_pk\_auth?Q \triangleright 2\_request!Q^\wedge\rangle; (v_{pk})\,
\langle 2\_reply?NA^\wedge,D \triangleright 3\_sign!D^\wedge\rangle; (v_{sgn})\,
\langle 2\_no\_access?NA^\wedge \triangleright 1\_denied!\text{Enc}(NA^\wedge,N^\wedge)\rangle; (v_{no})\,
\langle 3\_signed?P \triangleright 1\_reply!\text{AEnc}_c((KA^\wedge,NA^\wedge,NB^*),\text{Enc}((NA^\wedge,NB^*),P^\wedge))\rangle; (v_{ok})\,
\langle 1\_ack?ACK\rangle; (v_{ack})\,
\langle 3\_exit!\rangle; (v_{exit})\} \}

(a) Set of security vectors \(\Sigma^C\) for the contract

\[ C = \left(\left(\Sigma^C,\{s_0\},s_0,\{s_0\},\{s_0 \xrightarrow{\theta} C s_0 | v \in \Sigma^C\}\right), (\theta, \kappa)\right) \]

where

\[ \theta = [\text{PKB}/\text{KA}, \text{PKB}/\text{KB}, \text{Secret}/S, I/d/I, \text{Req}/R, \text{Data}/D, \text{AEnc}(\text{PKB}/\text{KA}, \text{Id}/\text{Req}/\text{Nonce})/Q, \text{AEnc}(\text{PKB}/\text{KB}, \text{Nonce})/\text{ACK}, \text{Nonce}/\text{NA}, \text{Nonce}/\text{NB}] \]

\[ \kappa = [\text{PKB}/\text{KA}, \text{PKB}/\text{KB}, s_{1,2}/S] \]

(b) Adaptation contract \(C\)

Figure 5: Adaptation contract and initial environment for services 1, 2 and 3

verified to correspond to the previously sent nonce \(NB\) using a contract term \(\text{AEnc}_c((KB^\wedge,NB^\wedge))\). Such a contract term encrypts and compares because the contract environment \((\theta, \kappa)\) does not possess the private key of service 2 to be able to decrypt the message with \(\text{AEnc}_d((PKB^\wedge,NB^\wedge))\) (where \(\text{PKB}\) should represent the private key of service 2 in the environment). Vector \(v_{exit}\) allows the adaptor to finish the behaviour of service 3 if its signature is not needed because the orchestration went through the hash-based schema.

The adaptor compliant with the contract must intercept every communication between the adapted services. Formally, this is modelled in service interfaces by renaming and prefixing every channel with the unique identifier of the service. For instance, \(3\_exit!\) meaning that message \(exit\) must be sent to service 3.

The state machine of security adaptation contracts imposes the allowed sequences in which the vectors can be applied. On any given state in the contract, a process compliant with that contract can only perform: i) the actions of single-sided vectors (see rule \(\text{SING}\) in Figure 6), ii) input actions of two-sided vectors and then enqueue the output action for its eventual execution (\(\text{ENQ}\)), and iii) previously enqueued output actions (\(\text{DEQ}\)). These restrictions imposed by contract \(C\) define another state machine \(A^C_0\) with a transition relation \(\rightarrow_0\) using the rules \(\text{SING}, \text{ENQ}\) and \(\text{DEQ}\) in Figure 6 where \(Q\) is a multiset of queued actions.

However, the transition system of \(A^C_0\) might be non-deterministic, i.e., two (or more) states
with different outgoing transitions can be reached using the same trace. We need a state machine

\[ A \stackrel{\text{Simulation}}{\preceq} B \]

Definition 2.5 (Simulation). A simulation relation between the states of two given state machines

\[ A_1 = (\Sigma_1, O_1, s_0^1, F_1, T_1) \text{ and } A_2 = (\Sigma_2, O_2, s_0^2, F_2, T_2) \]

is defined as \( R_{A_1,A_2} \subseteq O_1 \times O_2 \) such that the following conditions hold for all \((s_1, s_2) \in R_{A_1,A_2}\):

1. For every transition \((s_1 \xrightarrow{\alpha} s'_1) \in T_1\) it must exist a transition \((s_2 \xrightarrow{\alpha} s'_2) \in T_2\) such that \((s'_1, s'_2) \in R_{A_1,A_2}\).
2. If \(s_1 \in F_1\) then \(s_2 \in F_2\).

From the previous notion, a simulation relation can be naturally derived for state machines.

\[ A_1 \preceq A_2 \text{ iff it exists a } R_{A_1,A_2} \text{ such that } (s_0^1, s_0^2) \in R_{A_1,A_2} \]

where \(s_0^1\) and \(s_0^2\) are the initial states of \(A_1\) and \(A_2\), respectively.

It is trivial that \(A^C\) accepts the same traces as \(A^C_0\) and \(A^C_0 \preceq A^C\).

2.3. Adaptors

Adaptors are the realisation of contracts for a given set of services. Adaptors are equipped with
an environment which is updated at run time to perform the exchange of messages and its behaviour
is customised to avoid deadlocks among the services while complying with the contract at the same
time.

Definition 2.6 (Adaptor). An adaptor \(A\) compliant with a contract \(C\) is a deterministic state ma-
chine such that \(A \preceq A^C\).
We will omit the contract C when it is clear from the context.

By substituting every contract term by its corresponding type in every transition of the adaptor we obtain the *adaptor interface*. Because both the adaptor and its interface share the same set of states and both are deterministic, it is straightforward to obtain an *equivalence relation* between their states and transitions.

Given an adaptor \( A = (\Sigma^A, O^A, s_0^A, F^A, \sim) \) compliant with a contract whose type substitution is \( \theta \), its *interface* is given by \( ia(A) \), defined as follows.

\[
ia(A) \triangleq (\Sigma, O, s_0, F, T)
\]

where

\[
\Sigma = \{ [\alpha]_\theta \mid \alpha \in \Sigma^A \} \quad \text{and} \quad T = \{ s \xrightarrow{[\alpha]_\theta} s' \mid s \xleftarrow{\alpha} s' \}
\]

Given an adaptor \( A \) as above and an interface \( I = (\Sigma^I, O^I, s_0^I, F^I, \sim) \), we can define the *inverse* function of \( ia \) as follows:

\[
ia^{-1}_A(I) \triangleq (\Sigma^A, O^I, s_0^I, F^I, T')
\]

where

\[
T' = \{ s \xleftarrow{\alpha} s' \mid s \xrightarrow{\alpha} s' \land s \xrightarrow{[\alpha]_\theta} s' \}
\]

As we will discuss later, the synthesis process presented in this paper depends on these two mappings. In fact, we need \( ia \) to preserve the determinism of adaptors, keeping a tree structure on the generated interfaces.

**Proposition 2.1.** Given an adaptor \( A \) compliant with a contract \( C \), \( ia(A) \) is a deterministic interface. In addition, if the transition relation of \( C \) presents a tree structure, then \( ia(A) \) also presents a tree structure.

On the other hand, the complementary relationship between \( ia \) and \( ia^{-1}_A \) (as inverse mappings) is justified by following result.

**Proposition 2.2.** Given an adaptor \( A \) compliant with a contract whose type substitution is \( \theta \) then

\[
I \preceq ia(A) \Rightarrow ia^{-1}_A(I) \preceq A
\]

We have not yet defined the operational semantics among services and adaptors to really understand what their parallel composition is. In order to do that we require an operational semantics which allows the carrying out of security checks and cryptographic operations specified in the adaptation contracts and finally verify that certain security properties are preserved by the orchestration. We will properly define the operational semantics in Section 4 when we need all its details for the verification but, for the time being, we can understand the parallel composition between adaptor and service interfaces similarly as in CCS [7].

\[
\begin{align*}
\frac{s_1 \xrightarrow{\alpha} s'_1}{s_1 \otimes s_2 \xrightarrow{\alpha} s'_2} & \quad \frac{s \xrightarrow{\alpha} s'}{s \otimes s_1 \xrightarrow{\alpha} s' \otimes s_1}
\end{align*}
\]

We denote by \( \overline{\alpha} \) an action with the same channel and type as \( \alpha \) but with complementary direction. Communications are synchronous. We have omitted the symmetric rule for ‘\( \otimes \)’. Configuration \( s_0 \otimes
$s'$ is the initial state of the composition. A configuration $s \otimes s'$ of the transition system given by $s_0 \otimes s_0'$ is a final state of the composition if $s$ and $s'$ are final states of their respective service interfaces.

We say that an adaptor $A$ is an adaptor for services $S$ if the parallel composition between the adaptor and the services does not present deadlocks or livelocks.

**Definition 2.7 (Deadlock).** An interface presents a deadlock situation if it arrives to a state where it cannot reach a final state. More formally, there exists a deadlock state $s$ such that $s_0 \xrightarrow{\tau}^* s$ and $\exists s' \in F \text{ s.t. } s \xrightarrow{\tau}^* s'$.

The previous definition covers both deadlocks and livelocks because it guarantees that every state can eventually reach a final state through $\tau$-labelled transitions. For the rest of this paper, we will only mention deadlocks as a general term including livelocks as well.

The operator ‘$\otimes$’ is the composition operator among interfaces as opposed to the parallel composition between Crypto-CCS processes, presented in Section 4. However, for deadlock analysis purposes, we can extrapolate the results obtained using interfaces being given Lemma 4.1, which claims that two interfaces synchronise if their corresponding Crypto-CCS processes synchronise.

**Example 6.** The adaptor in Figure 7 is compliant with the security adaptation contract $C$ in Figure 5 and successfully adapts the three services in Figure 4 independently of their internal choices and supporting both authentication schemas. The actions are prefixed (and formally renamed with) the identification of the communicating service. Because contract vectors ($\Sigma^C$) contain adaptor actions (i.e., $\Sigma^C \subseteq (\Sigma^A \times \Sigma^A) \cup \Sigma^A$), we have represented adaptor actions with the direction and vector that contains them in the contract. For instance, $?v_{rep}$ represents the input action of vector $v_{rep}$, i.e., $2_{reply}\text{?}\text{NA}^{} \wedge D$. Several transitions labelled $!v_{ext}$ are dashed because they represent same interactions as solid transitions but with occurrences of $!v_{ext}$ at different points in the trace. The reason for this is that, once service 1 decides to go through the hash authentication schema, we can finish the session with service 3 using vector $v_{ext}$ at any time, thus the interleaving.

### 3. Synthesis of functionally-correct security adaptors

Cryptography is an orthogonal aspect to functionality. If there were no security attacks, malicious users and distrust, there would be no need for cryptography. For this reason, we are going to approach the synthesis process by abstracting away the security or, in other words, we forget about the cryptographic inference system, security checks and symbolic parameters presented so far and focus only on the interfaces of the services and the adaptor. We can do that to a certain level thanks to Lemma 4.1.

In this context, **functional correctness** means contract compliance and absence of deadlocks. However, deadlock freedom cannot always be preserved in the presence of active attackers. For example, an attacker could interrupt any further communication therefore leaving the system in a deadlock state. Because of this, our goal for functional correctness is to preserve deadlock freedom in the absence of active attackers. In this way, we might synthesise an insecure functionally-correct adaptor but, in Section 5, we will refine such adaptor into a secure one.

There are several approaches [5, 11, 13–15] in the literature to the synthesis of behavioural adaptors with similar properties of functional correctness. However, these approaches do not support cryptographic messages which must be cryptographically processed on receptions and emissions. Although these related papers could be extended to manipulate cryptographic messages, they still synthesise functionally-correct yet insecure adaptors. This means that the synthesised adaptors
A = P + H
P = 1_pinkauth?x:AEnc(Pk(Key),Id,Req,Nonce).P'
H = 1_hash?q:(Id,Req,Nonce,Hash(Id,Req,Nonce,Secret)).Qq \vdash i:Id
H_e = [\langle q, i, r, na, s \rangle] \vdash 1_hsc:Hash(Id,Req,Nonce,Secret)
H' = [\langle i, na, r \rangle] \vdash 3_ec:(Id,Nonce,Req)
\langle (P(kb),ec) \rangle \vdash 2_en:Eck(Pk(Key),Id,Nonce,Req)\_request!en.(R+D)
D = 2_noaccess?na':Nonce[na' = na][\langle (na,s) \rangle] \vdash 3_k:(Nonce,Secret)
R = 2_reply?z:(Nonce,Data)[\langle z \rangle] \vdash 4_d:Data[\langle (na,d) \rangle] \vdash 3_z'(Nonce,Data)[z' = z]
\langle (na,s) \rangle \vdash 3_k':(Nonce,Secret)
\langle (k',d) \rangle \vdash 5_e:Eck((Nonce,Secret),Data)\_reply!e.0
P' = ...
succeed in exchanging messages avoiding deadlocks but they might do so in a way that sensitive information remains insecure and therefore the adapted system is vulnerable to secrecy attacks.

Alternatively to the solution proposed below, it is possible to use these related approaches by first transforming the problem into an equivalent one without security, synthesise the functionally-correct adaptor using traditional approaches, transform back the result into an adaptor with security, and then proceed to the verification and refinement stages described in this paper. The ITACA toolbox supports this alternative approach and it allows the visualisation, simulation and analysis of behavioural adaptation contracts. For further details on this alternative procedure, please consult Appendix C.

Building on this previous work, we now proceed to illustrate how to synthesise functionally-correct adaptors able to manipulate cryptographic messages (whose parts might be encrypted or digested, for instance). For now, we will focus on contract compliance and deadlock freedom, therefore the adaptor synthesised at the end of this section is still insecure with regard to the secrecy property to preserve. In Section 5, we complete this process by taking advantage of the verification mechanism presented in Section 4 so that we can reuse the functions formalised in this section to refine functionally-correct adaptors into secured ones.

The synchronisations between services and an adaptor depend on three conditions:

**Signature matching.** Actions with complementary direction but identical types occurring on the same channel. This can be reduced to comparing transitions in the adaptor interface with those in the service interface and check that are complementary, i.e., same channel, same type and different direction.

**Contract compliance.** The contract allows synchronisation. In other words, each synchronisation between a service and the adaptor corresponds to one of the $A^C$ transitions. This condition can be understood as a control dependency.

**Parameter dependencies.** The adaptor has the information required to perform the synchronisation. These are the implicit dependencies among the symbolic parameters in contract terms. An example of this second sort of dependencies is that, if a certain parameter is annotated to be known in a contract term (i.e., it is needed to process that contract term) the value corresponding to that parameter should be present in the initial environment or must have been received in advance in a previous transition. The value of a parameter is obtained on input actions where the parameter is annotated to be fresh, or in an output action when the parameter is annotated to be instantiated. In the latter case the value is generated by the adaptor. These are data dependencies.

### 3.1. Data dependencies

We can avoid dealing with parameter dependencies as these can be made explicit in the contract. In order to do that, we refine any given contract into an equivalent one with the dependencies included using the rule $\text{DEP}$ in Figure 8.

A contract $C = \left( \Sigma^C, O, s_0, F, \rightarrow_C \right), E \right)$, where $E = (\theta, \kappa)$, is transformed by rule $\text{DEP}$ into another contract

$$C^E = \left( \left( \Sigma^C, O', s'_0, F', \rightarrow_C \right), E \right)$$

where

$$O' = \left( O \times 2^{\text{Param}} \right); \quad s'_0 = (s_0, \text{Dom}(\kappa)); \quad F' = (F \times 2^{\text{Param}})$$

\[1\] Available at http://itaca.gisum.uma.es/.
In every reachable state of $C^E$, it is guaranteed that the adaptor knows the runtime values needed for the following transitions. We have overloaded function $pm$ to be applicable to vectors, e.g., $pm^\land(⟨c?T ⊢ c'!T'⟩) = pm^\land(T) \cup pm^\land(T')$.

**Lemma 3.1.** The contract transformation given by rule DEP satisfies that:

- Every adaptor compliant to $C^E$ is also compliant to $C$.
- For every trace of $C^E$ such as $s_0 \xrightarrow{v_0} C \ldots \xrightarrow{v_i} C \xrightarrow{v_{i+1}} C$ it holds that $pm^\land(v_{i+1}) \subseteq Dom(κ_E) \cup \bigcup_{j=0}^{i} pm^\land(v_j) \cup pm^\ast(v_j)$

### 3.2. Control dependencies

Now, we have to deal with contract compliance and interface matching. Assuming that the original contract was $C_0$ and we obtained $C = C_E$ with explicit parameter dependencies, we are now going to work on a deterministic $A_C$, which is the state machine which characterises contract $C$. We start by using $A_0 = A_C$ as an initial approximation to the adaptor. However, $A_0$ might present deadlocks when it orchestrates the services, so we are going to do a deadlock analysis on its adaptor interface and prune the controllable branches that cause those deadlocks. This pruning process only works if $ia(A_C)$ is a tree. This, however, can be easily imposed explicitly by the contract or implicitly by modifying rule DEP in a way that every generated contract state is a unique state.

Given the service interface $S$ with an initial state $s_S^0$, the deadlock analysis proceeds as follows. We define an iterative synthesis process over a candidate adaptor interface $I_i = (Σ, O, s_i^0, F, T_i)$ by selecting a transition $t = s \xrightarrow{α} s' \in T_i$ such that:

$$s_0^S \otimes s_0^S \xrightarrow{τ_i} s_1 \otimes s_1 \xrightarrow{τ_i} s_1' \otimes s_1'$$

for some $s_1'$ where $s_1' \otimes s'$ is a deadlock state and $s_1 \xrightarrow{α} s_1 \xrightarrow{τ} s_1'$, and considering a new adapter interface given by:

$$I_{i+1} = prune(S, I_i, s \xrightarrow{α} s')$$

where function $prune$ (defined below) removes the given transition $t$ from $I_i$ without creating new deadlocks. The initial candidate is $I_0 = ia(A_0)$ and we iterate while a transition $t$ exists satisfying the above mentioned conditions.

It is not possible to synthesise the adaptor if any of the $prune(S, I_i, t)$ is undefined or if $s_0^S \otimes s_0^S$ still presents deadlocks at the end of this process; in either case we return an empty adaptor. Otherwise, $A = ia_{A_0}^{-1}(I_n)$ is an *adaptor for* $S$. The reason to check deadlock absence again at the end of the process is that, in some cases, deadlock situations are inherent to service interface $S$ and, no matter
how many times we prune the adaptor interface $I$, the interface $S$ might still reach a deadlock. For example, a service interface $S$ with no final states.

Function $\text{prune}$ must remove a transition leading to the given transition in a way that it does not cause anymore deadlocks in the process. It must be a deadlock-free pruning. Because of this, we have to check that the transition to remove has a sibling and it is a controllable decision, i.e., the adaptor can control which of those branches is followed by the services. This controllability check is done by function $\text{prunable}$. Function $\text{prune}$ is formally defined over a tree-like state machine $I$ as follows.

**Definition 3.1 (Deadlock-free pruning).** Given an interface $S$ with initial state $s^S_0$, an interface $I = (\Sigma, O, s_0, F, T)$, and a transition $t \in T$, function $\text{prune}(S, I, t)$ defines a new interface $(\Sigma, O, s_0, F, T')$ where the new transition relation is given by:

$$T' = \{ u \in T \mid \exists \bar{u} \in O[I]. u \in \bar{u} \land t' \not\in \bar{u} \}$$

if there exists $t' \in P = \text{prunable}(S, I)$ such that exists a trace in $O[I]$ where $t'$ precedes $t$, and for each of these traces $\cdots \bar{t} \cdot t \cdot \bar{t} \cdots \in O[I]$, $t \cap P = \emptyset$. If it does not exist such a transition $t'$, we consider that $\text{prune}(S, I, t)$ is undefined (represented as $\bot$).

Function $\text{prunable}$ is defined as:

$$\text{prunable}(S, I) \triangleq \{ s \xrightarrow{\alpha} s' \in T \mid \forall u \cdot s^S_0 \otimes s_0 \xrightarrow{\tau}^* u \otimes s$$

$$\exists \beta \neq \alpha \cdot s \xrightarrow{\beta} \land u \xrightarrow{\tau}^* \bar{\beta} \text{ and}$$

$$\forall v \cdot u \xrightarrow{\tau}^* v \land v \not\xrightarrow{\bar{\beta}} \}$$

Function $\text{prunable}$ returns the set of transitions that can be removed from the adaptor without generating new deadlocks. Prunable transitions must have a sibling transition $(s \xrightarrow{\beta})$ so that the execution can continue through an alternative branch and, in addition, it must be a controllable choice of the adaptor, i.e., none of the services can internally require the prunable action at the parent state $(v \not\xrightarrow{\bar{\beta}})$.

It is worth noting that mapping $\text{prune}$ is well defined because if $t'$ exists, it is clearly unique. In addition, the proposed iterative process, which builds a sequence of interfaces $\{I_i\}_{i=0 \ldots n}$, is also well defined, in spite of the apparent non-determinism exhibited by the selection of the transition $t_i$ in each step. In fact, the process is independent of the pruning order. This is what the following proposition states.

**Proposition 3.1.** Function $\text{prune}$ is independent of the pruning order. More formally, given two interfaces $S$ and $I$, and two transitions $t_1$ and $t_2$ in $I$, we have:

$$\text{prune}(S, \text{prune}(S, I, t_1), t_2) = \text{prune}(S, \text{prune}(S, I, t_2), t_1)$$

This proposition allows us to univocally define the adaptor $C[S]$ (possibly empty) generated by this pruning process as follows:

$$C[S] \triangleq \left\{ \begin{array}{ll}
(0, 0, \bot, 0, 0) & \text{if } I_n = \bot \\
\text{ia}_{AC}^{-1}(I_n) & \text{otherwise}
\end{array} \right.$$
where \( I_n \) is the last interface produced by the iterative process and \( I_0 = \text{ia}(A^C) \).

In order to demonstrate that prune behaves as expected first we prove that a pruned adaptor is still and adaptor, and then the main result of this section will prove that the synthesis process returns adaptors for the given services, if it converges to a non empty adaptor.

**Lemma 3.2.** For any service interface \( S \) and any transition \( t \) of an adaptor \( A \), if \( \text{prune}(S, \text{ia}(A), t) \neq \perp \) then

\[
\text{ia}_A^{-1}(\text{prune}(S, \text{ia}(A), t)) \preceq A
\]

and \( \text{prune}(S, \text{ia}(A), t) \) is deterministic.

Now, we can prove that, given a contract, the iterative pruning process for a certain interface (representing services) returns either an empty adaptor or an adaptor for those services, compliant with the contract.

**Theorem 3.1.** Given a contract \( C \), the iterative pruning process for a certain interface \( S \), providing the sequence of interfaces \( \{I_i\}_{i=0..n} \) (with \( I_0 = \text{ia}(A^C) \)), satisfies that if \( I_n \neq \perp \) then \( \text{ia}_A^{-1}(I_n) \) is an adaptor for services \( S \) compliant with contract \( C \).

4. Verification

We synthesised a functionally-correct adaptor compliant to a contract in Section 3. However, the contract was conceived to support and mediate between the security protocols of the services and these protocols might preserve different security properties. In this section, we will analyse the security implications of including an adaptor in the system and we will specifically verify that the adapted system globally preserves a given secrecy property, even in presence of attackers.

**4.1. The attacker**

Recalling the discussion in Section 1, if we consider that the attacker controls the network, the attacker could completely bypass the adaptor and directly communicate with the services, isolating the adaptor from the system.

\[
\exists X \text{ s.t. } P \parallel S \parallel X \models \text{p}_{\text{sec}} \tag{2}
\]

In this case, the adaptor process \( P \) is in no position to enforce any security because the attacker \( X \) can intercept every communication with service \( S \), therefore the only kind of security property that the adaptor can control is that it does not makes things worse. For example, we can guarantee that the adaptor does not disclose sensitive information to an unauthorised party.

Alternatively, if we can assume that certain channels cannot be actively manipulated (with \( \backslash \mathcal{L} \)) or eavesdropped (with \( \backslash \mathcal{L}' \)) by the attacker, the adaptor can be used as a gateway or a firewall to sensitive services. In this scenario, we might want to verify security properties regarding the resilience of the interface offered by the adaptor to the attacker. These are external attackers.

\[
\exists X \text{ s.t. } ((P \parallel S) \parallel \backslash \mathcal{L}' \parallel X_\phi) \perp L'' \models \text{p}_{\text{sec}} \tag{4}
\]

The reason for the restriction on \( L'' \) is that we only care about the possible synchronisations between the different parties, and this is achieved by including every channel of the system in \( L'' \).

In between the previous two kinds of attacks, and also covered by (4), we have insider attacks. These are attacks where the intruder has certain privileged knowledge (\( \phi \)) and some of the services in the system trust it.
It is time to present the formal operational semantics which allow us to simulate and verify security-enabled services. We need to i) be able to model the knowledge of the different parties (so that we can model insider attacks), ii) reason on the communications between services and what can be inferred from them using our cryptographic inference system, and iii) specify restrictions on the power of the attacker, so we can distinguish external from network attackers.

For this purpose, we are going to encode our services and adaptors into Crypto-CCS [6] processes. The operational semantics of Crypto-CCS can be parameterized with the cryptographic inference system $IS$ in Figure 3 and support our previous assumption about action synchronisations (Lemma 4.1).

4.2. Crypto-CCS


Crypto-CCS processes are described by the following grammar:

$$S ::= S \cdot L \mid S \cdot S \mid A_\phi \mid A_0 \mid pc.A \mid A_1 + A_2 \mid [m = m']A_1; A_2\mid \langle\langle m_i \rangle \rangle_{i \in I} \vdash_{IS} x:T[A_1; A_2]\mid pc ::= \tau \mid c! m \mid c? x:T \mid \chi_{c,T}$$

where $m, m', m_1, \ldots, m_n$ are messages or variables, $x$ is a message variable, $T$ is a (possibly structured) type, $c \in Chan$ is a channel, $\phi$ is a finite set of typed messages, $L$ is a subset of $Chan$ and $i \in I \subseteq \mathbb{N}$ (the set of natural numbers).

We briefly give the informal semantics of the terms of the calculus.

- $0$ is the process that does nothing.
- $pc.A$ is the process that can perform an action according to the particular prefix construct $pc$ and then behaves as $A$:
  - $c! m$ allows the message $m$ to be sent on channel $c$.
  - $c? x:T$ allows messages $m:T$ to be received on channel $c$. The message received substitutes the variable $x$.
  - $\chi_{c,T}$ is used to eavesdrop a communication on channel $c$ which occurs in other sub-components of the system. The eavesdropped message substitutes the variable $x$.

- $A_1 + A_2$ is the process that non-deterministically decides to behave as $A_1$ or $A_2$. It is worth noting that this operator can be used to model both internal choices (e.g., $\tau.A_1 + \tau.A_2$) and external choices (e.g., $c_1?x:T_1.A_1 + c_2?x:T_2.A_2$ being $c_1 \neq c_2$).

- $[m = m']A_1; A_2$ is the matching construct. If the two messages are equal to each other, then the process behaves as $A_1$, otherwise as $A_2$.

- $\langle\langle m_i \rangle \rangle_{i \in I} \vdash_{IS} x:T[A_1; A_2]$ is the inference construct. If, applying a case of inference schema $IS$ with the premises $\langle\langle m_i:T_i \rangle \rangle_{i \in I}$, a message $m:T$ can be inferred, then the process behaves as $A_1$ (where $x$ is replaced with $m$); otherwise the process behaves as $A_2$. This is the message-manipulating construct of the calculus: we can build a new message by using the messages in $\langle\langle m_i \rangle \rangle_{i \in I}$ and the inference rule $IS$.
• The system $S \setminus L$ is prevented from performing actions whose channel belongs to the set $L$, except for synchronisation.

• The system $S \setminus L$ can perform actions not in $L$, in addition synchronisations whose channels are in $L$ are renamed into $\tau$.

• A compound system $S || S_1$ performs an action $a$ if either of its sub-components performs $a$, and a synchronisation action $(\tau_{c,m})$, if the sub-components perform complementary actions, i.e. send-receive actions. It is worth noticing that, unlike CCS, our synchronisation actions carry information about the message exchanged and the channel used. In this way, we can model eavesdropping. Indeed, the agents of one component, e.g. $S$, might know the message exchanged during the synchronisation of the other component, i.e. $S_1$, by simultaneously performing an eavesdropping action $\chi$.

Guarded actions (i.e., $[m = m'] A_1 ; A_2$ and $[\{m_i\}]_{i \in I} \vdash x : T | A_1 ; A_2$) have a second process $A_2$ which is executed when the guard does not hold. From now on, we assume that guards which do not hold are security failures in which the process must perform exceptional actions. These actions could trigger alarms, perform counter-attacks or tighten the security, among others. In this paper we will simply halt the process on security failures, therefore $A_2$ is always the empty process (0). Let us note that these security failures are not violations of the secrecy property, they only mean that some messages are not as expected (e.g., a message which has been tampered or a key which is not correct).

For the sake of clarity, we will omit message types in Crypto-CCS process when they can be easily inferred from the context.

The operational semantics of Crypto-CCS is described in Table 1. Function $Tmsgs(T)$ represents the set of all possible messages of type $T$. The auxiliary function $\text{channel}$ returns the channel of the given action.

\[
\begin{align*}
\text{channel}(\tau_{c,m}) &= \text{channel}(\tau) = \perp \\
\text{channel}(c?x:T) &= \text{channel}(c!m) = \text{channel}(\chi^x_{c,T}) = c
\end{align*}
\]

The complete description of Crypto-CCS supports actions to generate random values. This random value generation is used for instantiated parameters in adaptation contracts. However, for the sake of simplicity, we will assume without loss of generality that every random value needed is initially known by the agent. In this way, we can safely replace instantiated parameters by known parameters in our adaptation contracts.

Example 7. [Figure 9] shows possible Crypto-CCS processes for services 1, 2 and 3.

The eavesdropping action $\chi^x_{c,T}$ is not meant to be used by well-behaved services but only by the attacker. The service interface of well-behaved Crypto-CCS processes is obtained through the following definition.

Definition 4.1. The interface of a Crypto-CCS process $P$ which does not eavesdrop is given by $\text{ip}(P) \triangleq (\Sigma, S, s_0, F, \vdash)$ where the states $S$ are the possible configurations of $P$, state $s_0 = P$, $F = \{0\}$ and the alphabet $\Sigma$ is obtained from the transition system $\vdash$, which is inferred as follows.

\[
\begin{align*}
P &\xrightarrow{\text{c}m} P' & m : T \\
P &\xrightarrow{\tau} P' & \tau &\xrightarrow{\tau} P
\end{align*}
\]
More formally, if two Crypto-CCS processes which do not eavesdrop synchronise, then their corresponding interfaces synchronise.

We are now in condition to formalise Lemma 4.1 which matches the synchronisations between Crypto-CCS processes to those of their corresponding interfaces. We used this lemma extensively in Section 3 where we presented deadlock analysis and the synthesis of functionally-correct adaptors based on service interfaces. The following lemma allows us to extrapolate those results to Crypto-CCS processes.

**Lemma 4.1.** If two Crypto-CCS processes which do not eavesdrop synchronise, then their corresponding interfaces also synchronise. More formally,

\[\text{ip}(P) \otimes \text{ip}(Q) \xrightarrow{\tau} \text{ip}(P') \otimes \text{ip}(Q') \quad \text{if} \quad P \parallel Q \xrightarrow{\alpha} P' \parallel Q', \alpha \in \{\tau_{c,m}, \tau\}\]

Security adaptors (e.g., A compliant with a contract whose environment is E) for service interfaces of Crypto-CCS processes can be encoded into Crypto-CCS processes (denoted \([A]_E\)
\[ S_1 = P + H \]
\[ P = \left[ \langle i_1, r_1, n_1 \rangle \vdash 3 \langle Id, Req, Nonce \rangle \right] \]
\[ \left[ \langle Pk(kb), c \rangle \vdash 2 p \cdot AE enc(Pk(Key), Id, Req, Nonce) \right] \]
\[ pk \_ auth \! \! p.(R + D) \]
\[ D = \text{denied?} x.0 \]
\[ R = \text{reply?} x.\left[ \langle x \rangle \vdash 4_i \right. y \cdot AE enc(Pk(Key), Nonce, Nonce) \]
\[ \left[ \langle ka, y \rangle \vdash 8 nn \cdot \langle Nonce, Nonce \rangle \right] \left[ \langle nn \rangle \vdash 4_f n_2 \cdot Nonce \right] \]
\[ \left[ \langle Pk(kb), n_2 \rangle \vdash 2 a \cdot AE enc(Pk(Key), Nonce) \right] \]
\[ \text{ack}a.0 \]
\[ H = \ldots \]
\[ \phi_1 = \{i_1, r_1, n_1, Pk(kb), ka, s_{1,2}\} \]

(a) Crypto-CCS process for service 1

\[ S_2 = \text{request?} x \cdot AE enc(Pk(Key), Id, Req, Nonce). \]
\[ \left[ \langle kb, x \rangle \vdash 8 c \cdot \langle Id, Req, Nonce \rangle \right] \]
\[ \left[ \langle c \rangle \vdash 4_i i' \cdot Id \right] \left[ i' = i_1 \right] \]
\[ \left[ \langle c \rangle \vdash 4_f n \cdot Nonce \right] (\tau.R + \tau.N) \]
\[ N = \text{no\_access!} n.0 \]
\[ R = \left[ \langle n, d_1 \rangle \vdash 3 y \cdot \langle Nonce, Data \rangle \right] \text{reply!} y.0 \]
\[ \phi_2 = \{i_1, d_1, kb\} \]

(b) Crypto-CCS process for service 2

\[ S_3 = \text{sign?} d \cdot Data. \]
\[ \left[ \langle kc, d \rangle \vdash 2 s \cdot AE enc(Key, Data) \right] \]
\[ \text{signed!} s.0 + \text{exit?}.0 \]
\[ \phi_3 = \{kc\} \]

(c) Crypto-CCS process for service 3

Figure 9: Crypto-CCS processes for the services of the running example

using the procedure described in Appendix B. It consists on a recursive procedure which encodes receptions, then it recomposes the messages using previously known and received values and it compares the newly composed message with what was received to check that the message was as expected. Output messages are composed and sent without such comparison.

Example 8. Figure 7(b) shows part of the Crypto-CCS process corresponding to the adaptor for the running example. The branch corresponding to the public key authentication has been omitted for the sake of clarity. All the expressions needed to perform the security checks in the adaptor are highlighted in red and bold.

4.3. Verifying security adaptors

Note that such properties as (4) look like validity statements of mathematical logic, i.e.: 
\[ \forall X \quad X \models p \] (10)
where the formula $p$ must be checked for every structure $X$. The main difference is that in (4) we check the components $X$ in combination with a system $S$ (including the adaptor process and the restrictions of (4)).

We can reduce such a verification problem as (4) to such a validity checking problem as (10). To obtain this, we apply and extend the partial model checking techniques used for the compositional verification of concurrent systems (see [17]).

Consider a system $S$ in combination with a process $X$ and try to figure out if the whole system $S\|X$ enjoys a property expressed by a formula $p$ or not. Then, partial model checking techniques can be used to find the sufficient and necessary condition on $X$, expressed by a logical formula $p_S$, so the whole system $S\|X$ satisfies $p$. In short, we have:

$$S\|X \models p \iff X \models p_S$$

(11)

Using the property (11), such verification problems as in (4) can be easily reduced to such problems as in (10).

### 4.4. A language to describe protocol properties

We illustrate a logical language ($L_K$) for the specification of the functional and security properties of a compound system. Language $L_K$ was presented in [6] and it is an extended normal multimodal logic with operators which make it possible to specify whether a message belongs to an agent’s knowledge after a computation $\gamma$ performed by the whole system, starting from a fixed initial knowledge. The syntax of the logical language $L_K$ is defined by the following grammar:

$$F ::= T | [a]F | [\exists x]F | \forall x\exists yF | \forall y\exists xF | m \in K_{X,\gamma}^\psi | \exists y . m \in K_{X,\gamma}^\psi$$

where $a \in Act$, $m$ is a message, $X$ is an agent identifier, $I$ is an index set (possibly infinite) and $\phi$ a finite set of typed messages. The language without $m \in K_{X,\gamma}^\psi$ and $\exists y . m \in K_{X,\gamma}^\psi$ (“knowledge” operators) is called $L$.

Informally, $T$ and $F$ are the true and false logical constants; the $[a]F$ modality expresses the possibility to perform an action $a$ and then satisfy $F$. The $[\exists x]F$ modality expresses the necessity that, after performing an action $a$, the system satisfies $F$; $\forall y\exists x\exists iF_i \Rightarrow F_{\exists i} \land \forall iF_i$ represents the logical disjunction (conjunction). As usual, we consider $\forall iE \land \forall iE$ as $F(T)$. A system $S$ satisfies a formula $m \in K_{X,\gamma}^\psi$ if $S$ can perform a computation $\gamma$ of actions and an agent of $S$, identified by $X$, can infer the message $m$ starting from the set of messages $\phi$ plus the messages it has come to know during the computation $\gamma$. The formula $\exists y . m \in K_{X,\gamma}^\psi$ is satisfied by a system $S$ if there exists a computation $\gamma$ and an agent $X$ of $S$ s.t. $X_\phi$ can infer $m$ during the computation $\gamma$.

We assume that a unique identifier can be assigned to every sequential agent in a compound system (e.g., the path from the root to the sequential agent term in the parsing tree of the compound system term). Then, given a sequence of transitions $S \xrightarrow{\gamma} S'$ of a compound term $S$, let $(S \xrightarrow{\gamma} S') \downarrow X$ be the sequence of actions of the agent identified by $X$ in $S$, that have contributed to the transitions of the whole system $F$. Finally, the formal semantics of a formula $F \in L_K$ w.r.t. a compound system $S$ is inductively defined in Table 2. Function $msgs(\gamma)$ returns all the messages occurring in the trace $\gamma$ and function $D(\phi)$ returns the set of typed messages which can be inferred (through the rules in Figure 3) from knowledge $\phi$.

---

2For simplification, here we leave out the technical details. We can however achieve this result by suitably adding information on the transitions, e.g., see [19].
This time we use a partial model checking function as described in [6] and recalled in Table 3.

There is no adaptor capable of avoiding that attack because the attack does not involve the adaptor.

Proposition 4.3. Given a system \( S \). Assume that an agent \( X \) following proposition holds.

Proposition 4.2. Consider the formula \( F \in m \in K^\phi_{X,\gamma} \) iff \( \exists \gamma . m \in K^\phi_{X,\gamma} \) if \( S \) is an initial message we have:

\[
(S||X_\phi)\backslash L \models \exists \gamma . m \in K^\phi_{X,\gamma} \iff X_\phi \models \exists \gamma . m \in K^\phi_{X,\gamma}/S.
\]

We use the formula \( \exists \gamma . m \in K^\phi_{X,\gamma} \) to check if there exists a possible intruder that can discover some confidential values while interacting with the services and the adaptor protocol. We proceed as in the previous case, now assuming that the adaptor protocol is also part of the system to be analysed. This time we use a partial model checking function as described in [6] and recalled in Table 3.

Proposition 4.2. Consider the formula \( F = \exists \gamma . m \in K^\phi_{X,\gamma}/S \). Then it is decidable whether or not a model \( X \) of such formula exists.

4.5. The most general attacker

The process \( X_F \) constructed by following the proof steps of Proposition 4.2 is maximal, i.e., any attack performed by the intruder can be performed by the one developed in the proof. Indeed, the following proposition holds.

Proposition 4.3. Given a system \( S \). Assume that an agent \( X_\phi \), with a finite \( \phi \) and \( \text{Sort}(S||X) \subseteq L \), then if \( m \) is an initial message and \( \left( \left( \left( (S||X) \backslash L \models \gamma \right) \models S_1 \right) \downarrow_X = \gamma \right) \) and \( m \in \mathcal{D}(\phi \cup \text{msgs}(\gamma)) \) and \( m \notin \mathcal{D}(\phi \cup \text{msgs}(\gamma')) \), for any \( \gamma' \) strict prefix of \( \gamma' \), then also \( \left( (S||X_F) \backslash L \models S_2 \right) \downarrow_{X_F} = \gamma' \) and \( m \in \mathcal{D}(\phi \cup \text{msgs}(\gamma')) \), where \( X_F \) is the process obtained in the proof of Prop. 4.2 for the formula \( F = \exists \gamma . m \in K^\phi_{X,\gamma}/S \).

Example 9. For our running example and considering that \( \phi = \{ Pk(ka), Pk(kb), i \} \) is public information, if we want the request coming from service 1 (\( r_1 \)) to be secret to third parties (secrecy property \( p_{sec} = \exists \gamma . r_1 \in K^\phi_{X,\gamma} \)), then by Proposition 4.2 we obtain an attacker \( X = \text{hash}?x + x_{\text{hash},x} \) able to violate \( p_{sec} \). Formally, \( (S||P||X) \backslash L \models p_{sec} \) being \( L \supseteq \text{Sort}(S||P||X) \). In the current setting, there is no adaptor capable of avoiding that attack because the attack does not involve the adaptor.
we can guarantee that the adaptor is resilient to attacks and, when the deployment scenario allows

with equations such as \( L \). However, in a slightly different example where we can assume that possible attackers cannot actively participate in the system (passive attackers are modelled by restricting the system communications with \( \setminus L \), we had this new formula \( (|P|X) \setminus L \models p \) for the sake of clarity.

5. Securing adaptors through refinement

Let us note that we are always interested in the possible communications between services, adaptors and attackers and not in any other external communication. Therefore, we always deal with equations such as \( (S||X) \setminus L \models p \) but we often omit this final restriction \( \setminus L \) leaving just \( S||X \models p \) for the sake of clarity.

5.1. Refinement

In some cases, a security adaptation contract can be designed to avoid certain attacks but this is limited to the extent allowed by the service interfaces. A service can be inherently insecure so, if it can be directly accessed by the attacker, there is no way the adaptor can secure it. However, we can guarantee that the adaptor is resilient to attacks and, when the deployment scenario allows it, we can use the adaptor as a firewall to the services. In these cases, we are going to refine the adaptor by removing from its behaviour traces that can be compromised by an attacker.

Given an initial adaptor \( A_0 \) compliant with a contract \( C \) whose environment is \( E \), two languages \( L \) (of actions which can be eavesdropped but cannot be actively accessed by the attacker) and \( L' \) (of actions which can neither be accessed nor eavesdropped by the attacker), the Crypto-CCS processes of

### Table 3: Partial evaluation function for \((S||X) \setminus L\), with \(\text{Sort} (S||X) \subseteq L\), and \(\exists \gamma \cdot m \in K_{X,\gamma}^p\)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\exists \gamma \cdot m \in K_{X,\gamma}^p)</td>
<td>(\setminus L \models p )</td>
</tr>
<tr>
<td>(\forall (c, m', S') \in \text{Send} (S) \cdot c \cdot m' )</td>
<td>((\exists \gamma \cdot m \in K_{X,\gamma}^p) \setminus S') (sending) (\vee)</td>
</tr>
<tr>
<td>(\forall S \cdot c \cdot m )</td>
<td>((\exists \gamma \cdot m \in K_{X,\gamma}^p) \setminus S) (receiving) (\vee)</td>
</tr>
<tr>
<td>(\forall \gamma \cdot m \in K_{X,\gamma}^p)</td>
<td>((\exists \gamma \cdot m \in K_{X,\gamma}^p) \setminus S') (eaves-dropping) (\vee)</td>
</tr>
<tr>
<td>(\forall S \cdot a \rightarrow S' (a = e_{a,m}) \cdot \exists \gamma \cdot m \in K_{X,\gamma}^p \setminus S) (idling) (\vee)</td>
<td></td>
</tr>
<tr>
<td>(m \in K_{X,\gamma}^p)</td>
<td>((\text{nothing to do}))</td>
</tr>
</tbody>
</table>

where:

\[\text{Send} (S) = \{(c, m', S') \mid \text{if } c \cdot m' \rightarrow S' \text{ and } m' \in \mathcal{D} (\phi)\}\]
the service \(S\), and an attacker \(X\) such that \((S \| [A_0]_E) \setminus L' \| L \models p_{sec}\), we define an iterative refinement process over an adaptor \(A_i\) and its interface \(R_i = (\Sigma, O, s_0, F, T_i)\) by selecting a minimal trace \(\gamma\) such that \(Q \xrightarrow{\gamma} Q'\) where \(Q = (S \| [A_i]_E) \setminus L' \| L \times 0\), and proceeding as follows:

- If \((Q \xrightarrow{\gamma} Q') \downarrow_{[A_i]_E}\) is empty, then it does not exist an adaptor capable of preserving \(p_{sec}\) with \(L\) and \(L'\) since the attack did not involve the adaptor. Therefore, the refinement returns the empty adaptor.

- If \((Q \xrightarrow{\gamma} Q') \downarrow_{[A_i]_E} = P_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} P_k\), then the interface \(ip([A_i]_E)\) presents a unique trace \(P_0 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_k} P_k\), and it exists a unique trace \(s_0 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_k} s_k\) in \(ia(A_0)\) (since \(ip([A_i]_E) \searrow ia(A_0)\)). Then, we iterate considering

\[
t_i = (s_{k+1} \xrightarrow{\beta_k} s_k); \quad R_{i+1} = prune(ip(S) \times R_i \times t_i)\); \quad A_{i+1} = ia_{A_0}^{-1}(R_{i+1})
\]

The initial adaptor interface is \(R_0 = ia(A_0)\) and we iterate while a trace \(\gamma\) exists satisfying the above conditions. If any of the \(prune(ip(S) \times R_i \times t_i)\) is undefined, we return the empty adaptor. When there are no more of such \(\gamma\) traces, at step \(n\), the final result of the refinement is \(ia_{A}^{-1}(R_n)\). It is worth noting that this iterative process can be used at the same time to synthesise functionally-correct adaptors (i.e., remove deadlocks) being \(A_0 = A^C\), \(Q_i = S \| [A_i]_E\) and being \(Q_i\) any deadlock state, i.e., \(Q_i\) is such that \(\exists \gamma \cdot Q_i \xrightarrow{\gamma} Q_i'\) and \(\not\exists \gamma' \cdot Q_i' \xrightarrow{\gamma'} \emptyset\) where every label in \(\gamma\) and \(\gamma'\) is either \(\tau\) or \(\tau_{c,m}\).

### 5.2. Synthesis and refinement overview

With this refinement process we conclude the synthesis of secure security adaptors which is depicted in [Figure 10](#) and summarised as follows:

1. Convert service Crypto-CCS process \(S\) into its interface, i.e., \(ip(S)\).
2. Based on that service interface and a given contract, synthesise (syn) a functionally-correct service adaptor \(A_0 = A^C\) as in [Section 3](#).
3. Encode the adaptor into a Crypto-CCS process \(P = [A_0]_E\), where \(E\) is the environment given in the security adaptation contract.
4. Verify [Section 4](#) if there exists an attacker \(X\) to \(S\) and \(P\), i.e., \((S \| P) \setminus L' \| L \models p_{sec}\) where \(p_{sec}\) is the logic expression of the attack to avoid being restricted by \(L\) and \(L'\).
5. Refine (ref) the adaptor \(A_0\) based on a possible attacker \(X\) (Section 5).
6. The final result is the adaptor \(A_n = ia_{A}^{-1}(R_n)\) and its Crypto-CCS process is \([A_n]_E\).

Our approach synthesises, if it exists, an adaptor \(A\) for service interfaces \(ip(S)\) compliant to contract \(C\) and environment \(E\) which preserves a given secrecy property \(p_{sec}\). This is formalized as follows.

**Theorem 5.1.** *Given a contract \(C\), a process \(S\), and a secrecy property \(p_{sec}\) (restricted to alphabets \(L\) and \(L'\)), the iterative refinement procedure which provides a sequence of interfaces \(\{R_i\}_{i=0\ldots n}\) (with \(R_0 = ia(C[\|ip(S)]\)) satisfies that if \(R_n \not\models \perp\) then \(A = ia_{A}^{-1}(R_n)\) is an adaptor for services \(ip(S)\) compliant with contract \(C\) such that property \(p_{sec}\) is preserved for \(L\) and \(L'\), i.e.:

\[
\not\exists X \cdot (([A]_E \| S) \setminus L' \| L \| X_\phi) \setminus L'' \models p_{sec}
\]

where \(E\) is the environment of contract \(C\), \(L'' \supset Sort(([A]_E \| S) \setminus L' \| L \| X_\phi)\), and \(\phi\) is the initial knowledge of the attacker.*

28
Example 10. Going back to our running example, being the adaptor process $P = [A]_E$, we had seen in Example 9 that it was possible for an attacker to learn the request coming from service 1 ($r_1$) or, more formally, $(P \parallel S) \downarrow I X \parallel \text{psec}$ being $p_{\text{sec}} = \exists \gamma. r_1 \in K_{X, \gamma}$ is satisfied by an attacker $X = \mathcal{X}_{\text{hash}, \gamma}$. This attacker is the result of the verification process (see Proposition 4.2). Then, we calculate the trace of synchronisations made by the adaptor during the attack $(Q \Rightarrow Q') \downarrow P$, whose corresponding interface contains the single action:

$$\text{hash?Id, Req,Nonce,Hash(Id,Req,Nonce,Secret)}$$

During the refinement process, we prune that action from the adaptor interface and then we convert the result into an adaptor using function $ia_{A C}^{-1}$. The resulting adaptor corresponds to the blue bold transitions of Figure 7. This final adaptor succeeds in preserving $p_{\text{sec}}$ because it allows only the use of the public-key protocol where the request $r_1$ is only known between services 1 and 2. Not even the adaptor, which forwards all the messages between the services, is aware of the request.

Example 11. Blue bold transitions in the adaptor of our running example are based on the Needham-Schroeder public-key protocol. This protocol is known to have a flaw where a malicious trusted service can impersonate a third one. In order to analyze this new scenario, we have to include the attacker $X$ as a trusted insider so we consider the system to be $S = (S_{1.2} + S_{1.X}) || (S_{2.1} + S_{2.X}) || S_3$, $\phi_X = \{ i_X, i_1, Pk(ka), Pk(kb), kx, s_{X.1}, s_{X.2} \}$ where subscripts represent the legal parties in the communication, e.g., $S_{1.2}$ represents service 1 talking with service 2, $r_1$ and $d_1$ is the request and data.
for service 1, $s_{1,2}$ is the secret shared between services 1 and 2, and so on. The contract has to be extended to include allowed communications between the services and $X$ because $X$ is a trusted service or inside attacker.

Then, as expected, the verification returns an attacker $X$ such that $(S || A_{1,2}) \setminus L \models \exists \gamma \cdot \{r_1, d_1\} \subseteq K_{X, \gamma}$ where $L$ restricts every channel not prefixed with $X$. This attacker does not use the hash! action to impersonate another service as it does not know the shared secret between service 1 and 2. If the attacker tried to use action hash with a random secret, this would have resulted in a security failure recognised by the adaptor process which would have become 0, therefore stopping all communications. For this reason, and because of the flaw in the Needham-Schroeder public-key protocol, every branch of attacker $X$ requires going through pk_auth actions on the adaptor.

During the pruning, the adaptor could use actions denied to avoid the attack but the contract enforces that such actions depend on no_access actions from service 2, so it is not a controllable decision, therefore it is not prunable. The other alternative, and the solution for this example, is to prune the adaptor at the pk_auth action, therefore enforcing communications with $X$ using the hash-based schema. As in Example 9, attacker $X$ is still able to eavesdrop the request $r_1$ but it cannot learn the reply $d_1$, hence preserving the secrecy property. Such an adaptor corresponds to the non-bold black transitions of the right-hand side of Figure 7.

6. Related work

There are several other papers [19–25] that take advantage of adaptation to support and enhance security. Most of them [20–25] are compositional approaches where security is dynamically changed depending on runtime QoS parameters and attack detection. These approaches focus on the performance vs security trade off and, when certain triggering conditions are met (a mobile agent who goes to a different host, an intruder is detected, or a possible buffer overflow attack) they tighten the security. This is done by replacing the security components of the systems by another one previously known and configured. The approach presented in this paper differs radically from theirs as we focus on the synthesis whereas they do not. Our work could be used in conjunction with theirs to adapt alternative security components at design time.

Li et al. [19] presented an interesting approach for securing distributed adaptation. In their work, their main focus is on data adaptation and to synthesise and execute a plan that allows the different parties to distributively apply a set of data transformations. They present an elaborated middleware, called Conductor, which is in charge of the planning. More interestingly, it also analyses, asserts and handles the trust between the different services and the different privacy requirements of each piece of information. They built this security schema based on several security box implementations. These security boxes wrap the different Conductor-enabled services to provide them with the appropriate cryptographic capabilities. Such security boxes are pre-designed but interchangeable at run time. Conductor presents a more high-level solution to a more specific problem. Our approach lacks their dynamic planning as we require static security adaptation contracts. However, our security adaptors could be used to develop such security boxes with new security protocols. In addition, Crypto-CCS allows us to describe and verify the different deployment scenarios tackled by Conductor.

---

3Actually, the adaptor is the parallel composition of $A_{1,2,3}$ (the original adaptor mediating between the services of the running example), $A_{1,X,3}$ (the same adaptor but replacing service 2 with $X$) and $A_{X,2,3}$ (the adaptor which uses $X$ as service 1) because $X$ is an inside attacker included in the adaptation. All of these are copies of the non-bold black transitions of Figure 7 but each one has the appropriate initial environment and action prefixes to interact with their corresponding services.
The synthesis of security adaptors is analogous to the automatic composition of services with security policies [26]. In this related work, the authors propose a two-staged approach. The AVISPA tool is run in each stage: first to obtain the protocol of the composition and second to verify that it preserves the desired security properties. The latter fits the functionality of AVISPA and, for the former, the authors converted the goal of the composition into another security property, therefore using the verifier to obtain an attacker which is, instead, the actual protocol of the composition. Although our work shares the motivation and the synthesis-and-verification approach, our proposal goes beyond theirs because i) our security adaptors are synthesised to be compliant with a given contract, so we have a finer control over the result; and ii) our approach includes a novel refinement stage where, if any attacker is discovered during the verification stage, the adaptor is automatically refined to be secure against such an attack.

As far as security adaptation contracts describe the allowed interactions among the participants, controller synthesis [27, 28] is another area where security adaptors can be applied. In particular, the synthesis of security controllers is based on wrapping part of the system and only allowing certain actions to show out of that wrap. Being provided that the deployment scenario supports that the adaptor completely wrap part of the system, security adaptation contracts can serve for the same purpose.

In this work, we extended Crypto-CCS [6] for the verification and refinement stages because of its sound verification algorithm based on partial model checking. However, other verification tools could be employed such as ProVerif [29], AVISPA [30] or Spi calculus [31]. These approaches should be adapted to support the interfaces given by security adaptation contracts (via Definition 4.1 and Lemma 4.1) and must provide the most general attacker (Proposition 4.3) so that they can take advantage of the functionally-correct synthesis (Section 3) and the refinement stages (Section 5) presented in this work.

7. Conclusions and future work

We have presented an approach to the synthesis, verification and refinement of secure security adaptors. The desired interactions between the adaptor and the services, along with the cryptographic operations that must be performed by the adaptor on every intercepted message, are described in security adaptation contracts. These contracts are able to overcome incompatibilities in signature, behaviour and security QoS. Based on the contract and the behaviour of the given services, the adaptor is synthesised to avoid deadlock and livelock situations. On a second step, the synthesised behaviour of the adaptor is verified and refined using Crypto-CCS to forbid any interaction with the adaptor that can violate the secrecy properties we want to preserve. At the end of this process we obtain an adaptor robust against attacks to the given secrecy property, compliant with the given security adaptation contract and able to overcome service incompatibilities. Our approach is versatile enough to cope with a range of security protocols and deployment scenarios with different zones of trust such as an attacker which controls the network, a trusted insider or an external attacker which can participate actively or passively in the communications.

As regards future work, we plan to extend our work with other verification approaches which cover multiple attackers at the same time. In addition, we want to support adaptor synthesis by untrusted third parties and, in these cases, we plan to use Proof Carrying Code [32] techniques based on security adaptation contracts.

Acknowledgements. Work partially supported by EU-funded project FP7-231167 CONNECT and by EU-funded project FP7-256980 NESSOS, project P06-TIC-02250 funded by the Andalusian local government and project TIN2008-05932 funded by the Spanish Ministry of Education and Science (MEC) and FEDER. This document does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of its content.
References


[27] F. Martinelli, I. Matteucci, A framework for automatic generation of security controller, STVR.
Appendix A. Proofs

Proposition 2.1. Given an adaptor A compliant with a contract C, \( ia(A) \) is a deterministic interface. In addition, if the transition relation of C presents a tree structure, then \( ia(A) \) also presents a tree structure.

Proof: As A is simulated by \( AC \), we only need to prove this result for \( AC \). Indeed, by construction of \( AC \), \( ia(AC) \) is a deterministic tree (see Sections 2.2 and 3.2), and if \( st \in S^A \) progresses to two different states \( st_1 \) and \( st_2 \) by \( st \overset{\alpha}{\rightarrow} st_i \) \((i = 1, 2)\), then \( [\alpha_1]\theta \neq [\alpha_2]\theta \). Therefore, \( ia(A) \) is still deterministic. On the other hand, \( A \) may present different transitions joining two given states: \( st \overset{\alpha}{\rightarrow} st' \) \((i = 1 \ldots n)\), but in that case \( [\alpha_1]\theta = \cdots = [\alpha_n]\theta \), and then \( ia(AC) \) keeps a tree structure. \( \square \)

Proposition 2.2. Given an adaptor A compliant with a contract whose type substitution is \( \emptyset \) then

\[
I \preceq ia(A) \implies ia_A^{-1}(I) \preceq A
\]

Proof: Let us consider:

\[
I = \left( \Sigma^f, O^f, s_0^f, F^f, \rightarrow_I \right) \quad \text{and} \quad A = \left( \Sigma^A, O^A, s_0^A, F^A, \rightarrow_A \right)
\]

In order to distinguish different transition relations in \( ia(A) \) and \( ia_A^{-1}(I) \), we will denote them by \( \rightarrow \) and \( \overset{\alpha}{\rightarrow} \), respectively. We are going to prove that, given \( s_1 \in O^f \) and \( s_2 \in O^A \), if \( s_1 \in \rightarrow_I \), then \( s_1 \) is also final in \( ia(A) \) because \( s_1 \preceq_I s_2 \). But final states in \( ia(A) \) coincide with final states in A (by construction). Therefore, \( s_2 \) is also final in A, and we conclude \( s_1 \preceq_A s_2 \).

Base case. If \( s_1 \not\overset{\alpha}{\rightarrow} \) (there is no transitions from \( s_1 \)), then the only condition of simulation (Definition 2.5) to be checked is the second one: if \( s_1 \) is final in \( ia_A^{-1}(I) \), then it is also final in \( I \), and hence \( s_2 \) is final in \( ia(A) \), because \( s_1 \preceq_I s_2 \). But final states in \( ia(A) \) coincide with final states in A (by construction). Therefore, \( s_2 \) is also final in A, and we conclude \( s_1 \preceq_A s_2 \).

Inductive case. Let us assume that \( s_1 \preceq_A s_2 \) \( (\text{given } s_1 \preceq_I s_2) \) holds for every \( s_1' \) such that \( s_1 \overset{\alpha}{\rightarrow} s_1' \) for some \( \alpha \in \Sigma^A \). Then, by construction of \( ia_A^{-1}(I) \), we have that both \( s_1 \overset{\alpha}{\rightarrow} s_1' \) and \( s_1 \overset{[\alpha]\theta}{\rightarrow} s_1' \). As \( s_1 \preceq_I s_2 \), \( s_2 \overset{[\alpha]\theta}{\rightarrow} s_2' \) for some \( s_2' \) such that \( s_1 \preceq_I s_2' \). By induction hypothesis \( s_1 \preceq_A s_2' \), and also \( s_2 \overset{\alpha}{\rightarrow} s_2' \), because \( s_2 \overset{[\alpha]\theta}{\rightarrow} s_2' \) in \( ia(A) \).

Lemma 3.1. The contract transformation given by rule DEP satisfies that:

- Every adaptor compliant to \( CE \) is also compliant to C.
- For every trace of \( CE \) such as \( s_0 \overset{v_0}{\rightarrow}_C \ldots s_i \overset{v_i}{\rightarrow}_C s_{i+1} \overset{v_{i+1}}{\rightarrow}_C \) it holds that
  \[
  pm^\land(v_{i+1}) \subseteq \text{Dom}(k_E) \cup \bigcup_{j=0}^{j=i} pm^f(v_j) \cup pm^*(v_j)
  \]
Proof: The proof is trivial following the rules in Figure 8 and the definition of adaptors (Definition 2.6).

Proposition 3.1. Function prune is independent of the pruning order. More formally, given two interfaces $S$ and $I$, and two transitions $t_1$ and $t_2$ in $I$, we have:

$$\text{prune}(S, \text{prune}(S,I,t_1), t_2) = \text{prune}(S, \text{prune}(S,I,t_2), t_1)$$

Proof: Let $I_1 = \text{prune}(S,I,t_1)$ and $I_2 = \text{prune}(S,I,t_2)$, $T_1$ and $T_2$ their corresponding transition relations, and $P = \text{prunable}(S,I)$. If $I_1$ is undefined ($\perp$), then the result is trivial, if we consider $\text{prune}(S,\perp,t) = \perp$, because if there is no $t' \in P$ satisfying the pruning condition for $t_1$ in $I$ will neither exist a such transition in $\text{prunable}(S,I_2)$, since $\text{prunable}(S,I_2) \subseteq P$. The symmetrical reasoning could be made if $I_2 = \perp$. Thus, we can suppose both $I_1$ and $I_2$ are undefined.

Let us consider $t \in \text{prune}(S,I_1,t_2)$. If $t \not\in T_2$, it would exist $t' \in P$ such that a trace exists, $\bar{u} = \cdots \cdot t' \cdot t_2 \cdots \in O[I]$, where $\bar{t} \cap P = \emptyset$ and $t$ is included in $\bar{t}$ or after $t_2$. As $t \in T_1$, $\bar{u} \in O[I_1]$. Now, if we consider $P_1 = \text{prunable}(S,I_1)$, we have two alternatives:

(i) $t' \in P_1$. Then, $t \not\in \text{prune}(S,I_1,t_2)$.

(ii) $t' \not\in P_1$. Clearly, $\bar{t} \cap P_1 = \emptyset$ because $P_1 \subseteq P$, and $\bar{t} \cap P = \emptyset$. Therefore, one of the following conditions holds:

(a) There exists $t''$ preceding $t'$ in $\bar{u}$ such that $t'' \in P_1$, and then transition $t \not\in \text{prune}(S,I_1,t_2)$.

(b) Otherwise, $\text{prune}(S,I_1,t_2)$ is undefined, and trivially, $t \not\in \text{prune}(S,I_1,t_2)$.

Thus, we have proved that $t \in \text{prune}(S,I_1,t_2)$ implies $t \in T_2$.

Let us suppose that $t \not\in \text{prune}(S,I_2,t_1)$. Then, there exists a transition $t' \in P_2 = \text{prunable}(S,I_2)$ such that a trace exists, $\bar{u} = \cdots \cdot t' \cdot t_1 \cdots \in O[I_2]$, where $\bar{t} \cap P_2 = \emptyset$ and $t$ is after $t'$ (i.e., it has been pruned). If $t$ is after $t_1$, then $t \not\in T_1$ because $t' \in P_1 \subseteq P$ (that is, it was already pruned by $I_1$), and $t \not\in \text{prune}(S,I_1,t_2)$, such as it was assumed initially. Therefore, $\bar{t} = \bar{u} \cdot t_1$. But, as we have already mentioned, by hypothesis, $t \in T_2$, and then $\bar{t} \cap P = \emptyset$. Thus, to prune $t_1$ in $I_2$, it must exist $t'' \in P$ after $t$, i.e. such that $\bar{u}_2 = \bar{u}_2 \cdot t'' \cdot \bar{t}_2''$, with $\bar{u}_2'' \cap P = \emptyset$. But, $\bar{t} \cap P_2 = \emptyset$, hence $t'' \not\in P_2$, and the only way to have a transition $t''$ prunable in $I$ but not prunable in $I_2$ is because (see definition of prunable mapping) some branch previous to $t''$, and reaching $t_2$ has been pruned in $T_2$. As $t$ is still in $T_2$, we can find $t''$ in $P$ in $\bar{u}_1 \cdot t \cdot \bar{u}_2'$ such that a trace $\bar{u}' = \cdots \cdot t' \cdot \bar{t}_1 \cdot t'' \cdot \bar{t}_2' \cdots \in O[I]$ exists, coinciding with $\bar{u}$ until $\bar{t}_1$, satisfying $\bar{t}_2 \cap P = \emptyset$. If $\bar{u}' \not\in T_1$, $t \not\in \text{prune}(S,I_1,t_2)$; then $\bar{u}' \in T_1$. Taking into account that transitions after $t''$ in $\bar{u}$ are not in $T_1$, we have that $t'$ (or some previous transition) is in $P_1$ and $\bar{t} \cap P_1 = \emptyset$ (if they are not, by considering definition of prunable mapping, $t' \not\in P_2$ or $\bar{t} \cap P_2 \neq \emptyset$, which are the case). Thus, we conclude that there exists $t' \in P_1$ such that $\bar{u}' = \cdots \cdot t' \cdot \bar{t}_2' \cdots \in O[I_1]$ with $\bar{t} \cap P_1 = \emptyset$, and $t$ is included in $\bar{t}_1$. In fact, this is true for $\bar{t}' = \bar{u}_1 \cdot t \cdot \bar{t}_1 \cdot t'' \cdot \bar{t}_2'$, because $\bar{u}_1 \cdot t \cdot \bar{t}_1 \cap P_1 = \emptyset$ (since $\bar{u}_1 \cdot t \cdot \bar{t}_1$ is a subtrace of $\bar{t}_1$), and also $\bar{t}_2 \cap P_1 = \emptyset$ (since $P_1 \subseteq P$ and $\bar{t}_2 \cap P = \emptyset$). However, at this point, we get a contradiction because $t \in \text{prune}(S,I_1,t_2)$. □

Lemma 3.2. For any service interface $S$ and any transition $t$ of an adaptor $A$, if $\text{prune}(S,\text{ia}(A),t) \neq \perp$ then

$$\text{ia}_A^{-1}(\text{prune}(S,\text{ia}(A),t)) \preceq A$$

and $\text{prune}(S,\text{ia}(A),t)$ is deterministic.

35
Proof: We know that for every $S$, $I$ and $t$ such that $\text{prune}(S,I,t) \neq \perp$ it happens that $\text{prune}(S,I,t) \preceq I$ because function $\text{prune}$ only removes transitions from $I$. Then, by applying Proposition 2.2 we obtain $\text{ia}_{A}^{-1}(\text{prune}(S,\text{ia}(A),t)) \preceq A$. For the same reason (transitions in the pruned interface are a subset of transitions in $\text{ia}(A)$), $\text{prune}(S,\text{ia}(A),t)$ is also deterministic because $\text{ia}(A)$ is deterministic. □

Theorem 3.1. Given a contract $C$, the iterative pruning process for a certain interface $S$, providing the sequence of interfaces $\{I_{i}\}_{i=0..n}$ (with $I_{0} = \text{ia}(A^{C})$), satisfies that if $I_{n} \neq \perp$ then $\text{ia}_{A}^{-1}(I_{n})$ is an adaptor for services $S$ compliant with contract $C$.

Proof: Given an interface $S$ and a contract $C$, if we consider the interface generated in each step $i$ of the pruning process $I_{i} = \text{prune}(S,I_{i-1},t)$, we can prove that it satisfies

$$\text{ia}_{A}^{-1}(\text{prune}(S,I_{i},t)) \preceq A^{C} \text{ and } I_{i} \preceq \text{ia}(A^{C})$$

if $I_{i} \neq \perp$ ($i = 0,\ldots,n-1$), being $I_{0} = \text{ia}(A^{C})$. In fact, by Lemma 3.2 this is true for $n = 0$. If we proceed by induction on $i$, and we assume as inductive hypothesis $\text{ia}_{A}^{-1}(\text{prune}(S,I_{i-1},t)) \preceq A^{C}$ and $I_{i-1} \preceq \text{ia}(A^{C})$, then we can derive the result by applying Proposition 2.2. Thus, the resulting adaptor (if it is not empty) at the end of the process is still compliant with the contract $C$. Additionally, every deadlock avoidable by the adaptor is removed by the pruning process, which does not generate new deadlocks, and this process converges because $\text{prune}$ is a monotonically decreasing function (w.r.t. transitions in the adaptor). □

Lemma 4.1. If two Crypto-CCS processes which do not eavesdrop synchronise, then their corresponding interfaces also synchronise. More formally,

$$\text{ip}(P) \otimes \text{ip}(Q) \xrightarrow{\alpha} \text{ip}(P') \otimes \text{ip}(Q') \quad \text{if} \quad P \parallel Q \xrightarrow{\alpha} P' \parallel Q', \alpha \in \{\tau, \nu, \tau\}$$

Proof: Synchronisation among services which does not eavesdrop occur by rule ‘$\parallel$’ (described in Table 1). The condition of this rule requires the transitions of the two services to be labelled with the same message, thus the transitions have the same type and by extension, they present the same interface. □

Proposition 4.1. Given a system $S$ and an agent $X_{\phi}$ where $\phi$ is finite and $\text{Sort}(S\parallel X) \subseteq L$, then if $m$ is an initial message we have:

$$(S\parallel X_{\phi})\parallel L \models \exists t. \ m \in K^{X_{\phi}}_{X,\gamma} \quad \text{iff} \quad X_{\phi} \models \exists t. \ m \in K^{X_{\phi}}_{X,\gamma}/S.$$

Proof: This result corresponds to Proposition 4.3 in [6]. □

Proposition 4.2. Consider the formula $F = \exists t. \ m \in K^{X_{\phi}}_{X,\gamma}/S$. Then it is decidable whether or not a model $X$ of such formula exists.

Proof: We prove the thesis by structural induction on $S$ and $F$; furthermore, if $F$ is satisfiable, we construct a model $X_{F}$ for such a formula.
\(F = \mathbf{F}\). Then, no sequential agent models \(F\), and thus \(F\) is not satisfiable.

\(F = \mathbf{T}\). Then, every sequential agent models \(F\). Let \((X_F)_\emptyset\) be \((0)_\emptyset\)

Then \(F\) is the disjunction of several formulas, say \(F = F_1 \lor F_2 \lor F_3 \lor F_4 \lor F_5\). Each of these formulas corresponds to a behaviour of \(X\) as it follows from the partial model checking table. At least one of these formulas must be satisfiable, if \(F\) is satisfiable. For each satisfiable formula \(F_i\), with \(i = 1..5\), we built a process \(X_i\) and as \(X_F\) we consider the summation of these processes.

\[- F_1 = \bigvee_{(c,m',S')} \langle c!m' \rangle (\exists \gamma \cdot m \in K_{X_{\gamma}}^{\phi}) / / S'\]

* Consider the set of formulas \(F_{1,S'} = (\exists \gamma \cdot m \in K_{X_{\gamma}}^{\phi}) / / S'\) that are satisfiable and consider for each of them the corresponding synthesised process \(X_{F_{1,S'}}\) (by structural induction on \(S\) this must hold). This set cannot be empty otherwise \(F_1\) is not satisfiable. For any \(c,m'\) there is at most one \(S'\) s.t. \((c,m',S') \in \text{Send}(S)\) s.t. \(X_{F_{1,S'}}\) is a synthesised model. Then let \(X_{1,S'}\) be \((p.c!x.X_{F_{1,S'}})_\emptyset\), where: 1) \(p\) is a proof of \(m\) from \(\phi\) whose root is an assignment to the variable \(x\); 2) \(x\) is a variable that does not appear in \(X_{F_{1,S'}}\); 3) \(X_{F_{1,S'}}\) is the term \(X_{F_{1,S'}}\) where \(m\) is replaced with \(x\). We then consider as \(X_1\) the summation of all these processes.

\[- F_2 = \bigvee_{S' \vdash \gamma} \langle c!m' \rangle (\exists \gamma \cdot m \in K_{X_{\gamma}}^{\phi}) / / S'\]

* Fix \(c\) s.t. exists \(m',S'\) with \((\exists \gamma \cdot m \in K_{X_{\gamma}}^{\phi}) / / S'\) satisfiable. By induction we can find \(X_{F_2,m',S'}\) that satisfy that formula. Consider a variable \(x\) that is not present in any of these processes. Then, let \(X_{2,c}\) be \(c?x:T.Y\), where \(Y\) is the summation of summands of the form \([x = m']X_{F_2,m',S'}\), where \(X_{2,m',S'}\) is the term \(X_{F_1,S'}\) where \(m'\) is replaced with \(x\) (assuming \(x\) is fresh). Eventually, \(X_2\) is the summation on all \(c\) for which a satisfiable formula exists.

\[- F_3 = \bigvee_{S' \vdash \gamma} (\exists \gamma \cdot m \in K_{X_{\gamma}}^{\phi}) / / S'\]

* This case proceeds analogously to \(F_2\).

\[- F_4 = \bigvee_{S' \vdash \gamma} (\exists \gamma \cdot m \in K_{X_{\gamma}}^{\phi}) / / S'\]

* This case is subsumed by \(F_3\) where the eavesdropped message is actually included in the knowledge of \(X\).

\[- F_5 = m \in K_{X_{\gamma}}^{\phi} / / S.\] This is the base case.

\[\square\]

**Proposition 4.3.** Given a system \(S\). Assume that an agent \(X_\emptyset\), with a finite \(\phi\) and \(\text{Sort}(S|X) \subseteq L\), then if \(m\) is an initial message and \((S||X) \not\vdash \left[ S_1 \right] \downarrow X = \gamma'\) and \(m \in \mathcal{D}(\phi \cup \text{msgs}(\gamma'))\) and \(m \not\in \mathcal{D}(\phi \cup \text{msgs}(\gamma'))\), for any \(\gamma'\) strict prefix of \(\gamma\), then also \((S||X_F) \not\vdash \left[ S_2 \right] \downarrow X_F = \gamma'\) and \(m \in \mathcal{D}(\phi \cup \text{msgs}(\gamma'))\), where \(X_F\) is the process obtained in the proof of Prop. 4.2 for the formula \(F = \exists \gamma \cdot m \in K_{X_{\gamma}}^{\phi} / / S\).
Proof: The proof is done by induction on the length of $\gamma$.

- $\gamma$ is empty. Then also $(X_F)_\phi$ is s.t. $m \in D(\phi)$.
- $\gamma = \alpha \gamma_1$. We can then investigate on the nature of the action $\alpha$.
  - $\alpha = \tau$. Then, $((S||X) \setminus L \xrightarrow{\tau} (S'||X) \setminus L \xrightarrow{\gamma_1} S_1) \downarrow X = \gamma'$ and $m \in D(\phi \cup \text{msgs}(\gamma'))$ and $m \not\in D(\phi \cup \text{msgs}(\gamma''))$, for any $\gamma''$ strict prefix of $\gamma'$. By structural induction on $\gamma_1$ we know that also $X_{F'}$ can perform the same sequence $\gamma'$, where $F' = \exists \gamma'. m \in K_{X,F}^\phi/S'$. Since $X_{F'}$ would be one summand of $X_F$, the result follows.
  - $\alpha = \tau_{e,m'}$. We can have four cases:
    * The action is due to a sending from $X$ and a reception from $S$. Then it must be that $m' \in D(\phi)$ and $(S,m',S') \in \text{Send}(S)$. It means that $((S||X) \setminus L \xrightarrow{\tau_{e,m'}} (S'||X') \setminus L \xrightarrow{\gamma_1} S_1) \downarrow X = (e,m')\gamma'$ and $m \in D(\phi \cup \text{msgs}(\gamma'))$ and $m \not\in D(\phi \cup \text{msgs}(\gamma''))$, for any $\gamma''$ strict prefix of $(e,m')\gamma'$. By structural induction on $\gamma_1$ we know that also $X_{F'}$ ($X_{F_1,S'}$ in the terminology of Prop. 4.2) can perform the same sequence $\gamma'$, where $F' = \exists \gamma'. m \in K_{X,F}^\phi//S'$. Since $(p,c!x,X_{F_1,S'})$, where: 1) $p$ is a proof of $m'$ from $\phi$ and $X_{F_1,S'}$ is the term $X_{F_1,S'}$, i.e. $X_{F'}$ where $m'$ is replaced with $x$, would be one summand of $X_F$, the result follows.
    * The action is due to a receiving form $X$ and sending from $S$. Then it means that $S \xrightarrow{c,m'} S'$ and $((S||X) \setminus L \xrightarrow{\tau_{c,m'}} (S'||X') \setminus L \xrightarrow{\gamma_1} S_1) \downarrow X = (c,m')\gamma'$ and $m \in D(\phi \cup \{m'\}\text{msgs}(\gamma'))$ and $m \not\in D(\phi \cup \text{msgs}(\gamma''))$, for any $\gamma''$ strict prefix of $(c,m')\gamma'$. By structural induction on $\gamma_1$ we know that also $X_{F'}$ ($X_{F_2,m',S'}$ in the terminology of Prop. 4.2) can perform the same sequence $\gamma'$, where $F' = \exists \gamma'. m \in K_{X,F}^\phi//S'$. Since $c!x:T.(\ldots + ([x = m']X'_{F_2,m',S'}) + \ldots)$, where $X'_{2,m',S'}$ is $X_{2,m',S'}$ where $m'$ is replaced with $x$ (assuming $x$ is fresh) would be one summand of $X_F$, the result follows.
    * The action is internal synchronisation of $S$ and it is eavesdropped by $X$. This is similar to the previous case.
    * The action is internal synchronisation of $S$ and it is not eavesdropped by $X$. Similar to the case with $\tau$.

\[ \square \]

Theorem 5.1. Given a contract $C$, a process $S$, and a secrecy property $p_{sec}$ (restricted to alphabets $L$ and $L'$), the iterative refinement procedure which provides a sequence of interfaces $\{R_i\}_{i=0...n}$ (with $R_0 = ia(C[\phi(S)])$), satisfies that if $R_n \neq \bot$, then $A = ia_C^{-1}(R_n)$ is an adaptor for services $ip(S)$ compliant with contract $C$ such that property $p_{sec}$ is preserved for $L$ and $L'$, i.e.:

\[ \exists X . \ (\langle [A]_E \| S \rangle \setminus L \langle L' \| L X_0 \rangle \setminus L'' = p_{sec} \]

where $E$ is the environment of contract $C$, $L'' \supset \text{Sort}([A]_E \| S) \setminus L' \| L X_0)$, and $\phi$ is the initial knowledge of the attacker.
Proof: \( C[ip(S)] \) is the adaptor resulting of applying the iterative pruning process in Section 3. Then, by Theorem 3.1 we have that \( R_0 \preceq A^C \). Thus, if we proceed as in Theorem 3.1 we can prove that \( A = ia^{-1}_A(R_n) \) is still an adaptor for services \( ip(S) \) compliant with contract \( C \). This is because the nature of the transitions pruned by mapping \( \text{prune} \) was not used to reason about the resulting interfaces; that is, it does not matter if the pruned transition exhibited a deadlock situation when interacted with \( ip(S) \) or permitted an attack from some \( X \). On the other hand, by construction \([A]_E\) preserves the property \( p_{sec} \) such as it is stated; in fact, for each step of the refinement process, if \(([ia^{-1}_A(R_i)]_E|S)\setminus L\setminus L \) presents a trace which allows an attack, the transition in \( R_i \) corresponding to the last interaction is pruned. In addition, by Proposition 4.3 we know that the avoided attacker is the most general one; and therefore, by disabling it (by successively pruning vulnerable traces), we disable any other possible attack.

Appendix B. From adaptors to Crypto-CCS

We now proceed to give some definitions to be able to encode adaptors into Crypto-CCS processes. The inference rules of \( IS \) needed to compose and decompose the value of a contract term are unambiguously given by the contract term. This assumptions boils down to two functions dependant on \( IS: ev_{IS} \) and \( out_{IS} \).

Function \( ev_{IS} \) is in charge of replacing every symbolic parameters using substitution \( \sigma \) and evaluate the resulting expression. In this section, substitutions \( \sigma, \rho \) and \( \kappa \) replace parameters with the Crypto-CCS variables which represent their values.

For \( IS \) in Figure 3 it is necessary to define the following \( ev_{IS} \) function.

\[
ev_{IS}(T, \sigma, \theta) = \begin{cases} 
(0, \sigma(P)) & \text{if } T = P \\
(Q[\langle y \rangle] \triangleright_1 \ux : [T]_0, x) & \text{if } T = \text{Hash}(T_1), \\
(Q_1 \ldots Q_n \mid \langle y_1 \ldots y_n \rangle) \triangleright_3 \ux : [T]_0, x) & \text{if } T = (T_1 \ldots T_n), \\
(Q_1, Q_2 \mid \langle y_1, y_2 \rangle) \triangleright_5 \ux : [T]_0, x) & \text{if } T = \text{Enc}(T_1, T_2), \\
(Q_1, Q_2 \mid \langle y_1, y_2 \rangle) \triangleright_2 \ux : [T]_0, x) & \text{if } T = \text{AEnc}(T_1, T_2), \\
\end{cases}
\]

We denote by \( \ux \) a new variable \( x \) not used in the rest of the process.

Function \( out_{IS} \) is dependant on the inference system used. It plays a complementary role to \( ev_{IS} \) in the sense that it obtains what is inside a constructor (the cleartext of some encrypted message or the elements of a list, for instance) as opposed to evaluating a structured contract term. Being given a symbolic parameter \( P \), a composite contract term \( T \), a variable \( x \) which contains the value corresponding to \( T \) and substitutions \( \sigma \) and \( \theta \), \( out_{IS} \) returns the process able to obtain the value of what is contained within \( T \), the variable that will be replaced with that value, and the contract term corresponding to the content.
For IS in Figure 3 function $\text{out}_{IS}$ is defined as

$$
\text{out}_{IS}(P, T, x, \sigma, \theta) \triangleq \begin{cases} 
(\langle \langle x \rangle \rangle, T, \sigma, y) & \text{if } \langle \langle x \rangle \rangle \vdash_{4} \text{vy}:[T_{i}][\sigma], T, y) \\
(\langle \langle z, x \rangle \rangle, T, \sigma, y) & \text{if } \langle \langle z, x \rangle \rangle \vdash_{6} \text{vy}:[T_{2}][\sigma], T, y) \\
(\langle \langle z, x \rangle \rangle, T, \sigma, y) & \text{if } \langle \langle z, x \rangle \rangle \vdash_{7} \text{vy}:[T_{2}][\sigma], T, y) \\
(\langle \langle z, x \rangle \rangle, T, \sigma, y) & \text{if } \langle \langle z, x \rangle \rangle \vdash_{8} \text{vy}:[T_{2}][\sigma], T, y) \\
\bot & \text{otherwise}
\end{cases}
$$

Function $\text{reach}$ (see [3]) is true iff the value of the given parameter can be obtained from the given contract term.

Being given the value of a contract term and the value of its known parameters, the value of all its fresh parameters can be obtained using Crypto-CCS. The value of the fresh parameters in the contract term can be obtained using function $\text{get}$ defined as follows:

$$
\text{get}(\langle \langle P_{1}, \ldots, P_{n} \rangle \rangle, x, \rho_{0}, \kappa, \theta) \triangleq (Q_{1} \ldots Q_{n}, \rho_{n})
$$

where $(Q_{i}, \rho_{i}) = \text{get}(P_{i}, x, \rho_{i-1}, \kappa, \theta)$

Function $\text{get}$ is defined as follows.

$$
\text{get}(P, T, x, \rho, \kappa, \theta) \triangleq \begin{cases} 
(0, \rho \odot [x/\emptyset]) & \text{if } T = P \\
(Q, \rho', P') & \text{if } (Q, T', y) = \text{out}_{IS}(P, T, x, \rho' \odot \kappa, \theta) \\
\bot & \text{otherwise}
\end{cases}
$$

The security check expressed in contract terms can be performed using Crypto-CCS. Contract terms of input actions have known parameters to represent that, whatever is received in that action, should partially match with what is contained in those parameters. If we have a variable $x$ which is bound to the message received through contract term $T$, using function $\text{get}$, we can obtain the value of fresh symbolic parameters in $T$. Then, we can evaluate $T$ (through function $\text{ev}_{IS}$) but using the known expected parameters filling the gaps with the received fresh parameters. With this evaluation we obtain the expected value $x'$. Finally, we only have to compare the expected value against the actual received value using Crypto-CCS guards, i.e., $[x = x']$.

Security adaptors for sequential agents can be encoded into Crypto-CCS processes. Using the previous definitions, the Crypto-CCS process corresponding to a security adaptor $(\Sigma, S, S_{0}, F, \rightarrow_{A})$ compliant with a contract whose environment is $(\theta, \kappa)$, is given by $[s_{0}]_{(\theta, \kappa)}$ defined as follows.
\[ [s]_{(\theta, \kappa)} \triangleq \begin{cases} 0 & \text{if } s \in F. \\ \sum_{P \in B} P & \text{if } s \notin F. \end{cases} \]

\[ [s \xrightarrow{c?T} A s']_{(\theta, \kappa)} \triangleq P.e!x.Q \text{ where} \]
- \((P, x) = \text{eval}_{IS}(T, \kappa^\wedge, \theta)\).
- \(Q = [s']_{(\theta, \kappa)}\).

\[ [s \xrightarrow{c!T} A s']_{(\theta, \kappa)} \triangleq c?u.x:[T]_0.P.Q[x = y].R \text{ where} \]
- \((P, \rho) = \text{gets}(pmf(T), x, \varepsilon, \kappa)\).
- \((Q, y) = \text{eval}_{IS}(T', \rho^f \land \kappa^\wedge, \theta)\).
- \(R = [s']_{(\theta, \kappa \land \rho)}\).

The empty substitution is denoted by \(\varepsilon\).

**Appendix C. Convert security contracts into deadlock-equivalent behavioural contracts**

**Appendix C.1. Abstracting security away**

As seen in Section 3.1, the data dependencies among the symbolic parameters in a security contract \(C\) can be explicitly included in the contract \(C^E\). Additionally, by Lemma 4.1, we can avoid altogether contract terms in order to do the synthesis of functionally-correct adaptors. So, if we want to use traditional approaches to adaptor synthesis (where no security or symbolic parameters are supported), we can include the interface information in the channel of the actions. Therefore, we have to transform the contract and the service interfaces to another contract and service interfaces without security in a way that the transformation can be reversed once the adaptor is synthesised.

**Figure C.11: Rules to remove and include security into adaptation contracts.**

\[ (a) \text{ Contract transformations.} \]
\[ (b) \text{ Rule to remove contract terms from security adaptation contracts.} \]
\[ (c) \text{ Rule to include the contract terms back into the synthesised adaptor.} \]
More formally, the contract transformation is defined in Figure C.11(b). Single-sided vector transformation is omitted. Symbol ‘?’ represents either ‘?’ or ‘!’.

We use the calligraphic letter \( \mathcal{C} \) to denote the equivalent contract without security and, similarly, for the state machine which it imposes to the adaptor, i.e., \( \mathcal{A}^C \) being the synthesised adaptor \( \mathcal{A} \). We proceed analogously for services, i.e., for every \( s_1 \xrightarrow{c,s} s_2 \) we have \( s_1 \xrightarrow{c,s} s_2 \). Let us highlight that the obtained services and contract without security are just particular cases of our service interfaces and contracts, therefore we can use our previous results.

We can now use any compatible approach to the synthesis of behavioural adaptors \cite{5,11,13-15} to generate an adaptor without security \( \mathcal{A} \) such that it complies with \( \mathcal{C} \) and the security-less services. We will follow the procedure depicted in Figure C.11(a).

### Appendix C.2. Including security back into the adaptor

The transformation from \( \mathcal{C} \) to \( \mathcal{C} \) (\( \mathcal{A}^C \) to \( \mathcal{A}^C \), analogously) is reversible because both state machines are deterministic and use the same set of states. However, the synthesised secure-less adaptor \( \mathcal{A} \) might not use the same set of states so, in order to obtain its corresponding adaptor with security (\( \mathcal{A}^C \)), we need to define a procedure to undo the transformation.

Figure C.11(c) returns adaptor \( \mathcal{A} \) using the equivalence relation between \( \mathcal{A}^C \) and \( \mathcal{A}^C \) while simulating \( \mathcal{A} \). The initial state of \( \mathcal{A} \) is the one which corresponds to the initial state of \( \mathcal{A}^C \) and \( \mathcal{A}^C \). The final states of \( \mathcal{A} \) are those which correspond to final states of \( \mathcal{A} \).