A tool for the synthesis of cryptographic orchestrators

Vincenzo Ciancia[♯], Antonio Martín[♮], Fabio Martinelli[♯],
Ilaria Matteucci[♯], Marinella Petrocchi[♯], Ernesto Pimentel[♮]

[♯]IIT-CNR, Pisa, Italy
[♮]E.T.S. Ingeniería Informática, Universidad de Málaga, Spain

Sept 19, 2012

Contents

Abstract

Security is one of the main challenges of service oriented computing. Services need to be loosely coupled, easily accessible and yet provide tight security guarantees enforced by cryptographic protocols. In this paper, we address how to automatically synthesize an orchestrator process able to guarantee the secure composition of electronic services, supporting different communication and cryptographic protocols. We present a theoretical model based on process algebra, partial model checking and logical satisfiability, plus an automated tool implementing the proposed theory.

Keywords: Synthesis of Functional and Secure Processes, Secure Service Composition, Partial Model Checking, Process Algebras, Temporal Logic.

Chapter 1

Introduction

Security concerns over Web services are regarded as a major research challenge for service oriented computing. The loose coupling of services, which enables higher reuse and interoperability, is often constrained by their security requirements. This fact complicates *orchestration*, that is, the design of an entity which coordinates some services in order to achieve a higher goal from their composition. In this paper, we address how to automatically synthesize an orchestrator for composing services with incompatible signature, behaviour and cryptographic capabilities. In particular, we deal with the problem of how to assure that the global service obtained by composition of the orchestrator, and the services, satisfies both functional and security requirements.

We introduce a didactic example illustrating the aforementioned challenge. Consider an user willing to purchase both an airplane ticket and an insurance contract covering the trip. As expected in the service-oriented approach, we suppose that this is achieved by the orchestration of a booking service and an insurance service. The booking service is represented by a process accepting some cryptographic material from the user, such as an encryption key, and the proof of a successful payment transaction signed with the received key. After such a request, the booking service issues a ticket, which is then sent encrypted to the insurance service for further processing. The insurance service receives the encrypted ticket, and returns an insured version of it. The procedure ends correctly once the insured version of the ticket is generated. From a security point of view, it is important to protect the ticket from malicious users willing to discover its content (*e.g.,* pairing the user name with the destination could represent a privacy violation, ac-

cording to some requirements of the owner; or the ticket could be anonymous, and the attacker use it in place of the legitimate owner).

There are some communication issues in this apparently simple procedure. For example, the user might be unable to manage both operations by himself, for some reason that is immaterial here. Furthermore, the two services might not be able to communicate directly, due to firewalls, mismatching policies, or incompatibilities in their interfaces. In this case, one needs to synthesize a third service, called hereafter *orchestrator*, whose aim is to make the system *functional*, *i.e.,* fully satisfying its goal (request and successfully obtain both a ticket and the correspondent insurance). The correct execution of the whole procedure is guaranteed by the issuance and proper reception of the insured ticket. In the process, one should also care about some security properties (in our case, the secrecy of the issued ticket). Thus, the orchestrator must not only be "functional", but also "secure", since its functionality should obey to the security requirements imposed on the composed system.

In this paper, we present a theoretical model and an automated tool for the synthesis and verification of service orchestrators which are both functionally correct and secure.

The paper is structured as follows. Section 2 recalls the Crypto-CCS language used to describe cryptographic services. In Section 3, we present the theoretical framework used to synthesize cryptographic orchestrators. Section 4 describes a tool implementing the proposed theory and shows an example analysis. Section 5 recalls related work in the area. Finally, Section 6 concludes the paper.

Chapter 2

Languages for crypto-services

In this section, we recall the Crypto-CCS process calculus [8] used here for specifying cryptographic services, *i.e.,* services that are able to perform cryptographic primitives, such as encryption and decryption. We also present a variant of the logic language in [8], used to express properties of services.

2.1 Crypto-CCS

We define those parts of the syntax and semantics of Crypto-CCS needed in the sequel of this paper.

Crypto-CCS is a slightly modified version of CCS [11], including some cryptographic primitives. A model defined in Crypto-CCS consists of a set of sequential agents able to communicate by exchanging messages (*e.g.,* data manipulated by the agents). Inference systems model the possible operations on messages and therefore consist of a set of rules of the form:

$$r = \frac{m_1 \quad \cdots \quad m_n}{m_0}$$

where $m_1, \ldots, m_n$ form a set of premises (possibly empty) and m_0 is the conclusion. An instance of the application of a rule r to closed messages $m_1, \ldots, m_n$ (*i.e.,* messages without variables) is denoted as $m_1 \quad \cdots \quad m_n \vdash_r m_0$.

The control part of the language consists of compound systems, basically sequential agents running in parallel. The terms of the language are generated by the following grammar (only constructs used in the sequel are

presented):

$$\begin{array}{ll} S{:=}\ S_1 \parallel S_2 \mid S\backslash L \mid A_\phi & \textit{compound systems} \\ A{:=}\ \mathbf{0} \mid p.A \mid [m_1 \cdots m_n \vdash_r x]A; A_1 & \textit{sequential agents} \\ p{:=}\ c!m \mid c?x & \textit{prefix constructs} \end{array}$$

where $m_1, \ldots, m_n, m$ are closed messages or variables, x is a variable and c is an element of the finite set Ch of channels. Informally, the Crypto-CCS semantics used in the sequel are:

- $c!m$ denotes a message m sent over channel c;
- $c?x$ denotes a message m received over channel c which replaces the variable x;
- $\mathbf{0}$ denotes a process that does nothing;
- $p.A$ denotes a process that can perform an action according to p and then behave as A;
- $[m_1 \cdots m_n \vdash_r x]A; A_1$ denotes the inference construct: if (by applying an instance of rule r with premises $m_1, \ldots, m_n$) a message m can be inferred, then the process behaves as A (where m replaces x), otherwise it behaves as A_1. This is the message manipulating construct of the language;
- $S_1 \parallel S_2$ denotes the parallel composition of S_1 and S_2, *i.e.* $S_1 \parallel S_2$ performs an action if either S_1 or S_2 does. It may perform a synchronization or internal action, denoted by τ, whenever S_1 and S_2 can perform two complementary send and receive actions over the same channel.
- $S\backslash L$ is prevented to perform actions whose channels belong to the set L, except for synchronization.
- A_ϕ is a single sequential agent whose knowledge is described by ϕ.

The language is completely parametric with respect to the inference system used. In particular, an example inference system to model asymmetric encryption is shown below. We denote with y a key belonging to an asymmetric pair of keys, we denote as y^{-1} the correspondent complementary key.

If y is used for encryption, then y^{-1} is used for decryption, and vice versa. Given a set of messages ϕ, then message $m \in \mathcal{D}(\phi)$ if and only if m can be deduced from the rules of the following inference system (modelling public key cryptography):

$$\frac{x \quad y}{E(x,y)}\ (enc) \qquad \frac{E(x,y) \quad y^{-1}}{x}\ (dec)$$

2.2 Specification example

Consider the example of the booking and insurance services that we provided in the introduction. We give here the Crypto-CCS specification of the actors, namely a user U, the booking service BS, and the insurance service InS. Each process is parametrized by the cryptographic keys it has in its knowledge, since the beginning of the computation.

$U(k_u, k_u^{-1}, k_{book}) \doteq$
- $[k_u \quad k_{book} \vdash_{enc} E(k_u, k_{book})]$ — *encrypt user key,*
- $a!E(k_u, k_{book}).$ — *send encrypted user key,*
- $[money \quad k_u^{-1} \vdash_{enc} E(money, k_u^{-1})]$ — *encrypt money,*
- $a!E(money, k_u^{-1}).\mathbf{0}$ — *send encrypted money and stop*

$BS(k_{book}^{-1}) \doteq$
- $b?Enc\_K_U.$ — *receive an encrypted key,*
- $[Enc\_K_U \quad k_{book}^{-1} \vdash_{dec} K_U]$ — *decrypt with* k_{book}^{-1}*,*
- $b?Enc\_Money.$ — *receive encrypted money,*
- $[Enc\_Money \quad K_U \vdash_{dec} Money]$ — *decrypt with* K_U*,*
- $[ticket \quad K_U \vdash_{enc} E(ticket, K_U)]$ — *encrypt ticket with* K_U*,*
- $b!E(ticket, K_U).\mathbf{0}$ — *send encrypted ticket and stop*

$InS(k_{ins}^{-1}) \doteq$
- $c?Enc\_Ticket.$ — *receive an encrypted ticket,*
- $[Enc\_Ticket \quad k_{ins}^{-1} \vdash_{dec} Ticket]$ — *decrypt with* k_{ins}^{-1}*,*
- $c!insured_t.\mathbf{0}$ — *send insured ticket and stop*

Keys k_u, k_u^{-1} form a pair of asymmetric keys related to the user. The user requests a ticket and the corresponding insured ticket. Key k_u^{-1} is the private key of the user. Indeed, to decrypt messages encrypted with k_u^{-1}, one must know k_u. Key pair k_{book}, k_{book}^{-1} is used by the booking service to

perform some operations leading to the ticket issuance. Whereas k_{book} is public, and known also by the user, the correspondent k_{book}^{-1} is only known to the booking Service. Key k_{ins}^{-1} is the private key of the insurance service. Variables Enc_K_U, Enc_Money , and Enc_Ticket contain, respectively, an encrypted key, an encrypted payment, and an encrypted ticket. $Money$ and $Ticket$ contain the decrypted payment and ticket. Messages $money$, $ticket$ and $insured_t$ represent the actual payment, ticket, and insured ticket.

The user initiates the procedure by sending two messages: k_u, encrypted with the public key of the booking service k_{book}, and the payment for buying the airplane ticket. The payment is encrypted using the private key k_u^{-1}.

The booking service is a process ready to receive an encrypted key, decrypt it with its own private key k_{book}^{-1}, receive the encrypted payment, decrypt it with the previously received key, and produce a ticket. The booking service sends the encrypted version of the ticket and terminates.

Finally, the insurance service is a process that receives an encrypted ticket, tries to decrypt it with private key k_{ins}^{-1}, and sends an insured version of the ticket.

The three processes act in parallel, thus the full system specification is $S \doteq U \parallel BS \parallel InS$. Note that processes cannot directly communicate with each other, but rather they have to interact with the orchestrator. Indeed, U dialogues on channel a, BS on channel b, while InS on channel c. Also, whereas the booking service encrypts the ticket with k_u, the insurance service would expect to decrypt the same ticket with decryption key k_{ins}^{-1}.

Later we will see how to synthesize an orchestrator allowing these services to communicate, obtaining a composite system that is functional and secure.

2.3 A logic language for properties of services

In order to specify some properties of a composed service, we exploit a logical language $\mathcal{L}_K$ originally presented in [8]. The language is an extension of multi-modal logic (see, *e.g.,* [12]) with operators for specifying whether a message belongs to the knowledge of an agent after some computation γ. Here, γ is a sequence of actions performed by the whole system. The syntax of the logical language $\mathcal{L}_K$ is defined by the following grammar:

$$\begin{aligned} F ::= \; & \mathbf{T} | \mathbf{F} | \langle a \rangle F | [a] F | \wedge_{i \in I} F_i \, | \vee_{i \in I} F_i \, | \, m \in K^{\phi}_{A,\gamma} \, | \\ & m \notin K^{\phi}_{A,\gamma} \, | \;\; \forall \gamma : K_1, \ldots, K_n \end{aligned}$$

Table 2.1: Semantics of $\mathcal{L}_K$.

$S \models \mathbf{T}$, for every process S
$S \models \mathbf{F}$, for no process S
$S \models \wedge_{i \in I} F_i$ iff $\forall i \in I : S \models F_i$
$S \models \vee_{i \in I} F_i$ iff $\exists i \in I : S \models F_i$
$S \models \langle a \rangle F$ iff $\exists S' : S \xrightarrow{a} S'$ and $S' \models F$
$S \models [a]F$ iff $\forall S' : S \xrightarrow{a} S'$ implies $S' \models F$
$S \models m \in K^{\phi}_{A,\gamma}$ iff $\forall S' : \text{s.t.}(S \overset{\gamma}{\mapsto} S') \downarrow_A = \gamma'$ and $S' \not\xrightarrow{a}$ then $m \in \mathcal{D}(\phi \cup msgs(\gamma'))$
$S \models m \notin K^{\phi}_{A,\gamma}$ iff $\forall S' : \text{s.t.}(S \overset{\gamma}{\mapsto} S') \downarrow_A = \gamma'$ and $S' \not\xrightarrow{a}$ then $m \notin \mathcal{D}(\phi \cup msgs(\gamma'))$
$S \models \forall \gamma : K_1, \ldots, K_n$ iff $\forall_{i=1,\ldots,n} \quad S \models K_i$

where $a \in Act$, m is a closed message, A is an agent identifier, ϕ a finite set of closed messages, and K_i, for i in some index set I, is of the kind $m \in K^{\phi}_{A,\gamma}$ or $m \notin K^{\phi}_{A,\gamma}$.

We briefly explain the syntax. $\mathbf{T}$ and $\mathbf{F}$ are the true and false logical constants. $\langle a \rangle F$ is a modality expressing the *possibility* to perform an action a and then satisfy F. The modality $[a]F$ expresses the *necessity* that, after performing an action a, the system satisfies F. $\vee_{i \in I}$ ($\wedge_{i \in I}$) represents logical disjunction (conjunction). A system S satisfies a formula $m \in K^{\phi}_{A,\gamma}$ (resp., $m \notin K^{\phi}_{A,\gamma}$) if S can perform a computation γ of actions and an agent of S, identified by A, can (resp., cannot) infer the message m starting from the set of messages ϕ plus the messages it has come to know during the computation γ. The formula $\forall \gamma : K_1, \ldots, K_n$ is satisfied by a system S if and only if S satisfies each formula of the kind $\forall \gamma : m \in$ (resp., $\notin$) $K^{\phi}_{i,A,\gamma}$ with $i = 1, \ldots, n$; such base case is satisfied by a system S if for all computation γ, an agent A of S can (resp., cannot) infer m during γ starting from the set of messages ϕ.

Let $S \overset{\gamma}{\mapsto} S'$ denote the transition of a compound system S to S' through the sequence of actions γ. Then, we denote with $(S \overset{\gamma}{\mapsto} S') \downarrow_A$ the sequence of actions performed by agent A in S, contributing to the transition $S \overset{\gamma}{\mapsto} S'$.

Table 2.1 shows the formal semantics of a formula $F \in \mathcal{L}_K$.

Chapter 3

The synthesis approach

We now proceed to extend some results in the area of partial model checking, logic languages and satisfiability, in order to synthesize an orchestrator process able to i) combine several services and provide an unified interface that satisfies a client request and ii) guarantee that the composite service is secure. Hereafter, from a functional perspective, we concentrate on the *deadlock freedom* and *livelock freedom* properties, and from a security perspective, we concentrate on the *secrecy* property.

Let us consider finite services. We assume that each service in the composition is not able to communicate with the others, *i.e.,* the set of channels over which S_i is able to communicate does not intersect the set of channels over which S_j is able to communicate, for each pair S_i S_j in S.

The functional property of *deadlock/livelock freedom* is expressed by the formula $\forall\gamma(max)\ m_F \in K^{\phi_O}_{O,\gamma(max)}$, where $\gamma(max)$ is the maximal length execution trace of S, m_F is a special *final* message, signalling that the services in S successfully terminate their execution (*i.e.,* no deadlock or livelock has happened). For the sake of modelling, we assume that: 1) if m_F falls into the orchestrator's knowledge ϕ_O, then the services in S have successfully terminated their execution; 2) if m_F does not fall into the orchestrator's knowledge after a number ns of transitions performed by the orchestrator, then at least one service in S has not successfully terminated its execution. In particular, ns is defined according to the size of S, and it represents the maximal size of the orchestrator.

The *secrecy* property is expressed by the formula
$\forall\gamma\ m \notin K^{\phi_X}_{X,\gamma}$, where X is the identifier of an intruder. This means that message m, that should remain a secret, will not fall into the intruder's

knowledge ϕ_X.

Let us consider the process $(S \| O_{\phi_O} \| X_{\phi_X}) \setminus L$. No matter what the behaviour of X is, we require that this process satisfies both formulas. It is worth noticing that in this case there are two components whose behaviour is unknown: the orchestrator O and the intruder X.

One issue is to *decide* if there exists an orchestrator O such that, for all the possible behaviours of X, after the computation of maximal length $\gamma(max)$, m_F is in the knowledge of O and m is not in the knowledge of X.

$$\begin{aligned} &\exists O_{\phi_O} \; s.t. \; \forall X_{\phi_X} \; (S \| O_{\phi_O} \| X_{\phi_X}) \setminus L \models \\ &\quad \forall \gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}, m \notin K^{\phi_X}_{X,\gamma(max)} \end{aligned} \tag{3.1}$$

In the above expression, we only consider maximal behaviours $\gamma(max)$ as the knowledge of the agents is incremental: if an agent does not know m after a computation of maximal length, then it is safe to assume that m was not known at any point in time during the computation.

Another important aspect is how to automatically *synthesize* the orchestrator. We can use *partial model checking* [2] to simplify expression 3.1 by partially evaluating the formula $\forall \gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}, m \notin K^{\phi_X}_{X,\gamma(max)}$. Partial evaluation is done with respect to some known part of the system. In our case, the known part is the behaviour of S. The derived formula is denoted as $\{\ldots\}//_{ns,S}$. Table 3.1 shows the partial evaluation function for Formula 3.1.

Proposition 3.0.1 *Let S be a system and O_{ϕ_O} and X_{ϕ_X} two sequential agents, where ϕ_O and ϕ_X are finite set representing the knowledge of O and X. Let $Sort(S \| X) \subseteq L$ be the set of channels on which S and X can communicate. If m_i $i = 1, \ldots, n$, are secret messages and m_F is the final one, we have:*

$$\begin{aligned} &(S \| O_{\phi_O} \| X_{\phi_X}) \setminus L \models \\ &\qquad \forall \gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}, m_i \notin K^{\phi_X}_{X,\gamma(max)} \\ &\text{iff} \\ &O_{\phi_O} \| X_{\phi_X} \models \\ &\qquad (\forall \gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}, m_i \notin K^{\phi_X}_{X,\gamma(max)})//_{ns,S} \end{aligned}$$

Table 3.1: Partial evaluation function for $\forall\gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}, m_i \notin K^{\phi_X}_{X,\gamma(max)}$.

$\forall\gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}, m_i \notin K^{\phi_X}_{X,\gamma(max)} /\!/_{ns,S}(\text{with } S \xrightarrow{a}) \doteq$
$\bigwedge_{(c,m',S')\in Send_O(S)} [c!m'](\forall\gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}, m_i \notin K^{\phi_X}_{X,\gamma(max)} /\!/_{ns-1,S'})$ $(sending_O) \wedge$
$\bigwedge_{(c,m',S')\in Send_X(S)} [c!m'](\forall\gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}, m_i \notin K^{\phi_X}_{X,\gamma(max)} /\!/_{ns,S'})$ $(sending_X) \wedge$
$\bigwedge_{S\xrightarrow{c!m'}S'} [c?m'](\forall\gamma(max) : m_F \in K^{\phi_O\cup\{m'\}}_{O,\gamma(max)}, m_i \notin K^{\phi_X}_{X,\gamma(max)} /\!/_{ns-1,S'})$ $(receiving_O) \wedge$
$\bigwedge_{S\xrightarrow{c!m'}S'} [c?m'](\forall\gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}, m_i \notin K^{\phi_X\cup\{m'\}}_{X,\gamma(max)} /\!/_{ns,S'})$ $(receiving_X) \wedge$
$\bigwedge_{(c,m')\in Send_{OX}} [\tau_{c!m'}](\forall\gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}, m \notin K^{\phi_X\cup\{m'\}}_{X,\gamma(max)} /\!/_{ns-1,S})$ $(synchro_{OX}) \wedge$
$\bigwedge_{(c,m')\in Send_{XO}} [\tau_{c!m'}](\forall\gamma(max) : m_F \in K^{\phi_O\cup\{m'\}}_{O,\gamma(max)}, m_i \notin K^{\phi_X}_{X,\gamma(max)} /\!/_{ns-1,S})$ $(synchro_{XO}) \wedge$
$\bigwedge_{S\xrightarrow{a}S'(a=\tau)} \forall\gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}, m_i \notin K^{\phi_X}_{X,\gamma(max)} /\!/_{ns,S'}$ $(idling)$
$\forall\gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}, m_i \notin K^{\phi_X}_{X,\gamma(max)} /\!/_{0,S} \quad (\text{with } S \not\xrightarrow{a}) \doteq m_F \in K^{\phi_O}_{O,\epsilon}, m \notin K^{\phi_X}_{X,\epsilon}$
where :
$Send_O(S) = \{(c, m', S') | S \xrightarrow{c?m'} S' \text{ and } m' \in \mathcal{D}(\phi_O)\}\&$
$Send_X(S) = \{(c, m', S') | S \xrightarrow{c?m'} S' \text{ and } m' \in \mathcal{D}(\phi_X)\}\&$
$Send_{OX} = \{(c, m') | m' \in \mathcal{D}(\phi_O)\}\&$
$Send_{XO} = \{(c, m') | m' \in \mathcal{D}(\phi_X)\}\&$

This result identifies the necessary and sufficient conditions that the orchestrator, interacting with every possible X, must satisfy in order to guarantee that the final message m_F is delivered correctly, *i.e.,* $m_F \in K^{\phi_O}_{O,\gamma(max)}$, without any disclosure of information to X, *i.e.,* $m_i \notin K^{\phi_X}_{X,\gamma(max)}$.

However, the presence of the universal quantifier on X makes the formula $\forall\gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}$ not satisfiable, since X can always interfere with the normal execution of S getting the overall system stuck, so that the final message m_F is not delivered.

However, still keeping the intuition behind Formula 3.1, we can weaken the property to the conjunction of Formulas $A1$ and $A2$, where:

A1 When there is not an intruder, the orchestrator always drives the services to correct termination.

A2 When there is an intruder, no matter what actions it takes, it is not able to learn the secret m.

Now we need to determine whether it is possible to determine an orchestrator

O satisfying this weaker assumption. Decidability comes from the following proposition.

Proposition 3.0.2 *Given a system S, and two finite sets ϕ_O and ϕ_X, it is decidable if* $\exists O_{\phi_O}$ s.t. $\forall X_{\phi_X}$

$$\begin{array}{lllll} A1 & (S \| O_{\phi_O}) \setminus L & \models & \forall \gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)} \\ A2 & (S \| O_{\phi_O} \| X_{\phi_X}) \setminus L & \models & \forall \gamma(max) : m \notin K^{\phi_X}_{X,\gamma(max)} \end{array}$$

In A1, we are assuming that the attacker X is the empty process, with an empty initial knowledge ϕ_X.

According to Proposition 3.0.1, we can apply the partial model checking techniques to A1 and A2 obtaining:

$$\begin{array}{lllll} A1' & O_{\phi_O} & \models & (\forall \gamma(max) : m_F \in K^{\phi_O}_{O,\gamma(max)}) //_{ns,S} \\ A2' & (O_{\phi_O} \| X_{\phi}) & \models & (\forall \gamma(max) : m \notin K^{\phi_X}_{X,\gamma(max)}) //_{ns,S} \end{array}$$

Hence, since the formulas in A1 and A2 are finite, the application of the partial model checking, in conjunction with the usage of some satisfiability procedure for process algebra (see *e.g.,* [13]), allows us to synthesize an orchestrator, whenever it exists. We shall see an example in Section 4.1.

Chapter 4

PaMoChSA 2012: Tool description

The two formulas of Proposition 3.0.2 have been implemented [1] as an extension of the PaMoChSA tool, the PArtial MOdel CHecker Security Analyser, see [10]. The original tool implemented a partial model-checking solution to the problem of finding an attack in a security protocol. The techniques in the current paper extend the approach to address the problem of synthesizing a secure orchestrator.

The algorithm enumerates all secure orchestrators. Besides being an implementation of the formulas in Proposition 3.0.2, the algorithm can be more intuitively explained as path-finding in a state graph. In principle, the behaviour of an orchestrator is a tree. However, since the system and the orchestrator are assumed to be deterministic, such a tree has an equivalent description in terms of all its paths.

Nodes of the graph are triples (s, K_O, K_X) where s is the current continuation of the system specification (initially, the system specification itself); K_O is the knowledge of the orchestrator; and K_X is the knowledge of the intruder.

The initial values for s, K_O and K_X are user-specified, as well as a set of channels H that are considered secure against intrusion (therefore, messages sent on these channels can neither be eavesdropped nor manipulated).

Arcs of the graph are the possible steps that an orchestrator can do, that is, communications on channels that are shared with the system. An arc has a source (s, K_O, K_X) and a target (s', K'_O, K'_X). In principle, the target s' should be the continuation of s after that some communication has been

performed. The knowledge of the orchestrator K_O should accordingly be updated, in order to reflect the messages acquired from s (in case s can send messages to the orchestrator). Paths formed by such transitions obey to the requirement A1 in Proposition 3.0.2. However, also the requirement A2 has to be kept into account.

A practical way to account for possible attacks is to build the state graph in such a way that additional transitions are present, simulating eavesdropping and manipulation of messages by the intruder (with the exception of communications happening on channels in H). Thus, whenever s can receive a message of type T from the orchestrator, the continuation s' can also be instantiated with all messages of the same type that can be deduced from the knowledge of the intruder K_X. Likewise, whenever s can send a message to the orchestrator, the knowledge K'_O of the orchestrator can also be augmented with all the messages of the same type that can be deduced from K_X. This machinery implements the ability of X to interfere with communications between the orchestrator and the system.

Finally, the knowledge of the intruder is always augmented with the messages that are exchanged between the orchestrator and the system, unless the used channel is in H. Rationale is that the intruder can eavesdrop such communications in order to acquire new information.

4.1 Example

In this section, we first present the PaMoChSA 2012 specification of the example introduced in Section 2.2. Then, we show the output from a run of the tool, presenting the synthesis of a secure orchestrator able to coordinate the user, the booking service, and the insurance service in a functional and secure way.

In our running example, the *functional* goal of the orchestrator is to deliver an insured ticket, while its *secure* goal is to let the ticket a secret not falling into the intruder's knowledge. Below, we show an input file describing our running example. The file format comprises tags delimiting the input parameters. The specification of the system is written in a slight variant of Crypto-CCS. The language is typed. In particular, the type EKey for messages representing cryptographic keys denotes public keys, while the type DKey denotes private keys. Even if the language is quite intuitive, the interested reader can find more notions on this

variant on the first version of the PaMoChSA manual, available online at http://www.iit.cnr.it/node/16284.

```
<GOAL> insured_ticket : Insured </GOAL>

<ORCH_KNOWLEDGE> k_ins : EKey ;
k_u : DKey; k_u : EKey </ORCH_KNOWLEDGE>

<FORMULA> ticket : Ticket </FORMULA>

<KNOWLEDGE> k_book : EKey; k_ins : EKey;
k_attack : EKey; k_attack : DKey </KNOWLEDGE>

<SPEC>
Parallel

(* User *)
Send(a, Encrypt(k_u: EKey, k_book : EKey)).
Send(a, Encrypt(money : Money, k_u : DKey).0

And

(* Booking service *)
Recv(c,ENC_K_U : Enc(EKey * EKey)).
If Deduce (K_U = Decrypt(ENC_K_U,k_book : DKey)) Then
 Recv(c,ENC_MONEY : Enc(Money * DKey)).
 If Deduce (MONEY = Decrypt(ENC_MONEY,K_U)) Then
    Send(c,Encrypt(ticket : Ticket, K_U)).0
 End Deduce
End Deduce

And

(* Insurance service *)
Recv(d, ENC_TICKET : Enc(Ticket * EKey)).
If Deduce (TICKET =
           Decrypt(ENC_TICKET,k_ins : DKey)) Then
 Send(d,insured_ticket : Insured).0
End Deduce
```

```
End Parallel
</SPEC>
```

The tags in the input file describe the information necessary to synthesize an orchestrator. The tag *formula* introduces the typed message that should not be learned by the intruder (in our case, the ticket). The tag *goal* contains the final "success" message (in this case, the insured ticket). The terms in the *Orch_Knowledge* tag represent the typed messages that the orchestrator knows initially. In our running example, the initial knowledge of the orchestrator contains the public key of the insurance service, k_{ins}, and both the public and private key of the user. Especially since it knows said private key, in this example the orchestrator is considered a trusted party that acts on behalf of the user. Indeed, this is not necessarily the case. The terms in the *knowledge* tag represent the typed messages that the intruder knowns at the beginning of the computation, *i.e.,* some public cryptographic keys (k_{book} and k_{ins}), plus a pair of keys (k_{attack}, k^{-1}_{attack}) owned by the intruder. Finally, the *spec* tag introduces the specification of the known part of the system, *i.e.,* the parallel composition of the user, plus the booking and insurance services.

Figure 4.1 shows the results of the elaboration. The tool correctly synthesizes two secure orchestrators (basically, they represent the same process, up-to a different order of execution). The orchestrator is a process that communicates with the user on channel a, forwarding messages to the booking service on channel b. Then, it receives an encrypted ticket from the booking service on channel b, decrypts the obtained message, encrypts the ticket with the public key k_{ins}, and sends the result to the insurance service over channel c. Thus, the insurance service is able to decrypt the ticket using key (k^{-1}_{ins}) and to successfully issue the insured ticket, therefore fulfilling the functional goal. The synthesised orchestrators are secure, in the sense that their operations does not let a potential intruder learn the secret message "ticket".

```
*** 2 orchestrators found:
Receive(a, Enc[booking_k](user_k) : Enc(EKey*EKey))
Receive(a, Enc[user_k](money) : Enc(Money*DKey))
Send(b, Enc[booking_k](user_k))
Send(b, Enc[user_k](money))
Receive(b, Enc[user_k](ticket) : Enc(Ticket*EKey))
Send(c, Enc[insurance_k](ticket))
Receive(c, insured_ticket : Insured)

Receive(a, Enc[booking_k](user_k) : Enc(EKey*EKey))
Send(b, Enc[booking_k](user_k))
Receive(a, Enc[user_k](money) : Enc(Money*DKey))
Send(b, Enc[user_k](money))
Receive(b, Enc[user_k](ticket) : Enc(Ticket*EKey))
Send(c, Enc[insurance_k](ticket))
Receive(c, insured_ticket : Insured)

*** Elapsed time: 0.008
```

Figure 4.1: Results: two secure orchestrators found.

Chapter 5

Related work

In literature, there are some papers proposing compositional approaches to the synthesis of controllers, able to dynamically enhance security, depending on some runtime behaviour of a possible attacker, *e.g.,* [9, 5].

The current work extends the existing research line on the synthesis of secure controller programs [9] with the introduction of cryptographic primitives. Also, it tries to simplify the approach in [6] for the synthesis of deadlock-free orchestrators that are compliant with security adaptation contracts [7]. The current method differs from [6], since the synthesis happens in one step.

Similarly, our approach to synthesis differs from the one in [3], where automatic composition of services under security policies is investigated. Work in [3] uses the AVISPA tool [14] and acts in two stages: first, it derives a protocol allowing composition of some services; then, some desired security properties are implemented. The latter step uses the functionality of AVISPA and, for the former step, the desired composition is turned into a security property, so that AVISPA itself can be used to derive an "attacker" which actually is the orchestrator.

In [4], Li et al. present an approach for securing distributed adaptation. A plan is synthesized and executed, allowing the different parties to apply a set of data transformations in a distributed fashion. In particular, the authors synthesize "security boxes" that wrap services, providing them with the appropriate cryptographic capabilities. Security boxes are pre-designed, but interchangeable at run time. Our approach, in contrast, lacks dynamic planning. However, in our case the orchestrator is synthesized at run time and is able to cryptographically arrange secure service composition.

Chapter 6

Conclusions

We have presented a theoretical approach and a tool for the synthesis of cryptographic orchestrators that i) allows all the services to reach the expected end of their operations without falling into a deadlock (livelock) status, and ii) guarantee secrecy properties. The adoption of partial model checking and satisfiability techniques, to automatically synthesize cryptographic orchestrators, extends the recent research line of [9] and [6]. The tool that we implemented is an extension of the PaMoChSA system. In its original version, the tool is a partial model checker able to find security attacks on cryptographic protocols. The new version automatically synthesizes orchestrators which are secure by construction. It will be interesting to adopt the tool in real-world usage scenarios, by using concrete specification and programming languages (e.g. BPEL) in place of Crypto-CCS.

Bibliography

[1] PaMoChSA 2012. `http://www.iit.cnr.it/staff/vincenzo.ciancia/tools.html`.

[2] H. R. Andersen. Partial model checking. In *LICS*, page 398. IEEE, 1995.

[3] Y. Chevalier, M. A. Mekki, and M. Rusinowitch. Automatic composition of services with security policies. In *SERVICES'08 - Part I*, pages 529–537. IEEE, 2008.

[4] J. Li, M. Yarvis, and P. Reiher. Securing distributed adaptation. *Computer Networks*, 38(3), 2002.

[5] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS*, volume 900 of *LNCS*, pages 229–242. Springer, 2005.

[6] J. A. Martín, F. Martinelli, and E. Pimentel. Synthesis of secure adaptors. *J. Log. Algebr. Program.*, 81(2):99–126, 2012.

[7] J. A. Martín and E. Pimentel. Contracts for security adaptation. *J. Log. Algebr. Program.*, 80(3-5):154–179, 2011.

[8] F. Martinelli. Analysis of security protocols as open systems. *TCS*, 290(1):1057–1106, 2003.

[9] F. Martinelli and I. Matteucci. A framework for automatic generation of security controller. *STVR*, 2010.

[10] F. Martinelli, M. Petrocchi, and A. Vaccarelli. Automated analysis of some security mechanisms of SCEP. In *ISC*, pages 414–427. Springer, 2002.

[11] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.

[12] C. Stirling. Modal and temporal logics for processes. In *Logics for Concurrency: Structures versus Automata*, pages 149–237, 1996.

[13] R. S. Streett and E. A. Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Information and Computation*, 81(3):249–264, June 1989.

[14] L. Viganò. Automated security protocol analysis with the AVISPA tool. *ENTCS*, 155:69–86, 2006.